Patrons de conception

Ingéniérie des systèmes d'information

Paul Brunet

1) Abstraction

Animal est abstrait. Zoo contient des animaux.

2) Modularité

Construction d'un système complexe par assemblage de modules plus simples.

3) Encapsulation

Protection des attributs de l'objet.
Contrôle des accès, isolation de l'implémentation.

4) Héritage

Redéfinition de comportement par héritage.

5) Polymorphisme

Signatures polymorphes, résolution des invocations.

6) Composition

Obtenir des nouvelles fonctionnalités en combinant les services de plusieurs objets.

Introduction
Conception raisonnée

- **Objectif :** produire des conceptions extensibles, maintenables et réutilisables
- **Problème :** la conception relève fortement de l'artisanat, cf. software craftsmanship
- **Solution :** il faut s'aider du savoir-faire.

> « *Le meilleur outil de conception pour le développement de logiciels est un esprit bien éduqué sur les principes de conception. Ce n'est pas UML ou toute autre technologie.* »
>
> Craig Larman

Le savoir-faire est formalisé sous forme de :

- **Principes de conception :** notions importantes desquelles dépend la qualité d'une conception
- **Règles de conception :** ensemble de prescriptions de conception à respecter
- **Patrons de conception :** modèles de solutions à des problèmes récurrents

Principes et règles de conception

Principes de conception

- **S**ingle Responsibility Principle ..... Principe de responsabilité unique
- **O**pen-Closed Principle ..... Principe d'ouverture / fermeture
- **L**iskov Substitution Principle ..... Principe de substitution de Liskov
- **I**nterface Segregation Principle ..... Principe de ségrégation des interfaces
- **D**ependency Inversion Principle ..... Principes d'inversion des dépendances

Règles de conception

- **Règle 1.** Réduire l'accessibilité des membres de classe
- **Règle 2.** Encapsuler ce qui varie
- **Règle 3.** Programmer pour une interface, non pour une implémentation
- **Règle 4.** Privilégier la composition à l'héritage

Patrons de conception

- Solutions « **prototypiques** » à des problèmes objets.
- **Réutilisables** à des problèmes récurrents.
- Peu d'algorithmique, plus des schéma orientés-objet.
- Façons d'organiser le code pour augmenter :
  - Flexibilité
  - Maintenabilité
  - Extensibilité
  - Configurabilité
  - ...
- Le plus souvent basé sur des **interfaces** et des **abstractions**.

Patrons de conception
Intérêt

- Un vocabulaire commun et puissant
- Les patterns aident à concevoir plus facilement des systèmes :
  - Réutilisables : Responsabilités isolées, dépendances maitrisées ;
  - Extensibles : Ouverts aux enrichissements futurs ;
  - Limiter la modification de l'existant ;
  - Maintenables par faible couplage.
- Les patterns reflètent l'expérience de développeurs objets.

  Solutions éprouvées et solides.
- Les patterns ne sont pas du code mais des cadres de solutions générales à adapter à son problème particulier.
- Les patterns aident à maîtriser les changements.

  Les solutions plus triviales sont souvent moins extensibles.
- Attention à l'overkill ! Utilisez les patterns intelligemment.

Patrons de conception
Classification

Patrons créateurs

Ciblent la construction des objets (« aider » new, clone)

Patrons Factory, AbstractFactory, Singleton...

Patrons structuraux

Travaillent sur des aspects statiques, à « l'extérieur » des classes (notamment extensibilité)

Patrons Façade, Adapter, Decorator, Proxy, Composite ...

Patrons comportementaux

Travaillent sur des aspects dynamiques, à « l'intérieur » des classes (parfois même des instances)

Patrons Strategy, Iterator, Observer, Visitor

Patrons de conception
Plan

Patrons de conception
Plan

1. Patrons structurels
   - 1.1. Composite
   - 1.2. Procuration
   - 1.3. Autres patrons structurels

2. Patrons de création
   - 2.1. Fabrique
   - 2.2. Singleton

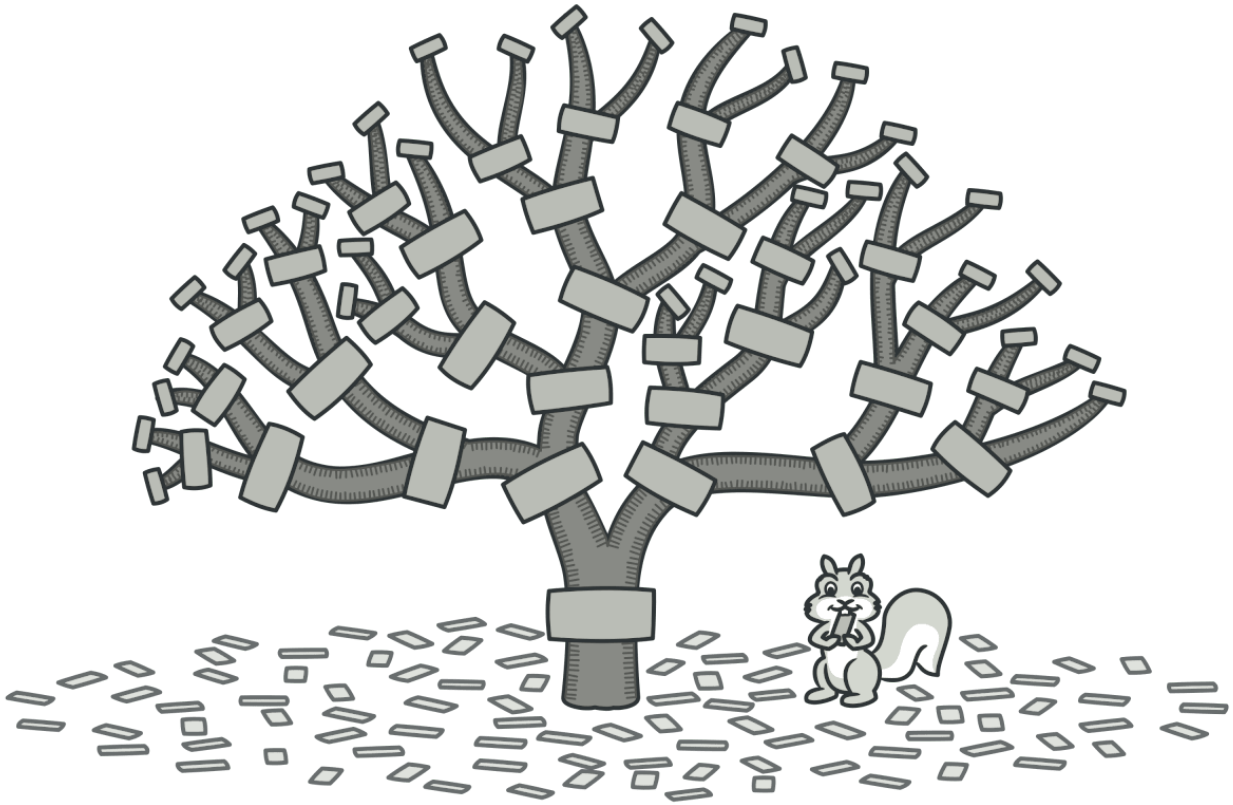
3. Patrons comportementaux
   - 3.1. Patron de méthode
   - 3.2. Stratégie
   - 3.3. État

Patrons de conception
Plan

Patrons de conception
Plan

Intention

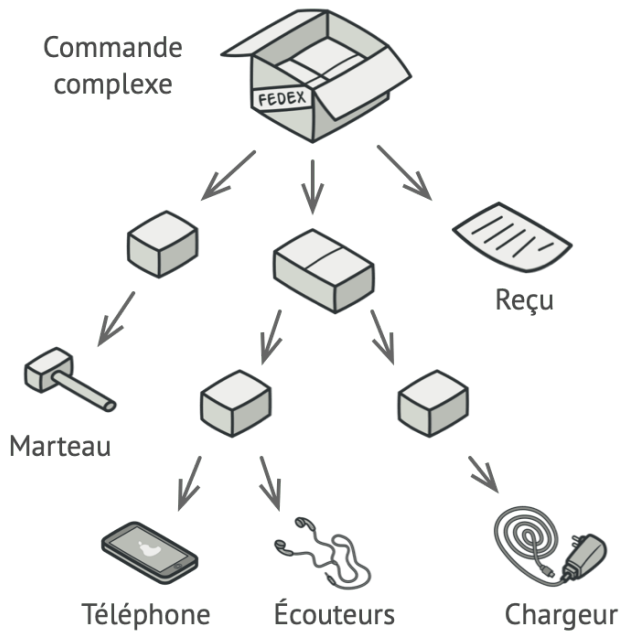
Composite est un patron de conception **structurel** qui permet d'agencer les objets dans des **arborescences** afin de pouvoir traiter celles-ci comme des objets individuels.



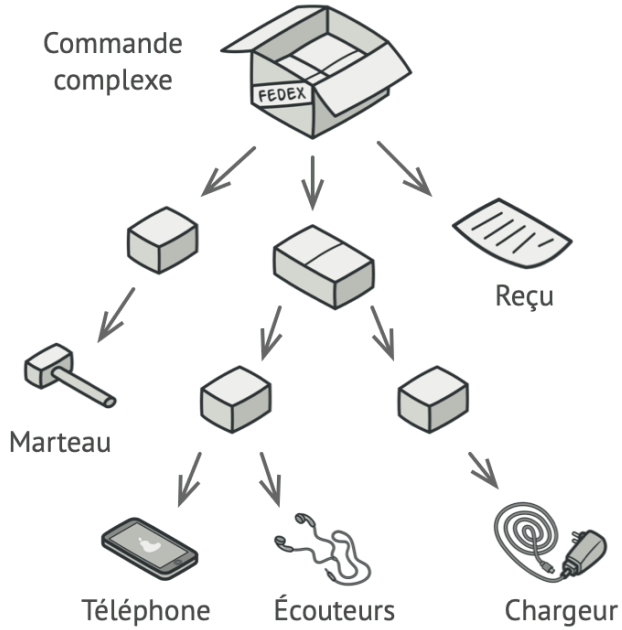
Problème

- Ce patron s'utilise pour des applications dont la structure principale se représente sous la forme d'une arborescence.
- Soit deux objets : les produits et les boîtes.
- Une boîte peut contenir plusieurs produits ainsi que des boîtes plus petites.

  Ces petites boîtes peuvent également contenir quelques produits ou même d'autres boîtes encore plus petites, et ainsi de suite.
- Vous décidez de mettre au point un système de commandes qui utilise ces classes :
  - Les commandes peuvent être composées de produits simples sans emballage,
  - ou d'autres boîtes remplies de produits… et d'autres boîtes.
- Comment allez-vous déterminer le coût total d'une telle commande ?



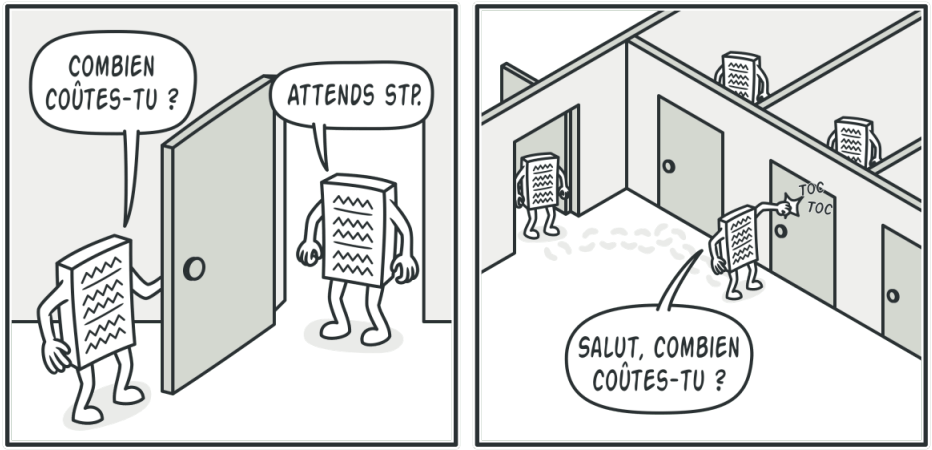
Problème



- On peut tenter l'approche directe :
  - déballer toutes les boîtes,
  - prendre chaque produit,
  - et en faire la somme pour obtenir le total.
- Facile à mettre en place dans le monde réel.
- Dans un programme : pas aussi simple ! Il faut connaître à l'avance :
  - la classe des Produits et des Boîtes que l'on parcourt,
  - le niveau d'imbrication des boîtes,
  - et d'autres détails...
- Tout ceci rend l'approche directe assez compliquée (voire impossible).

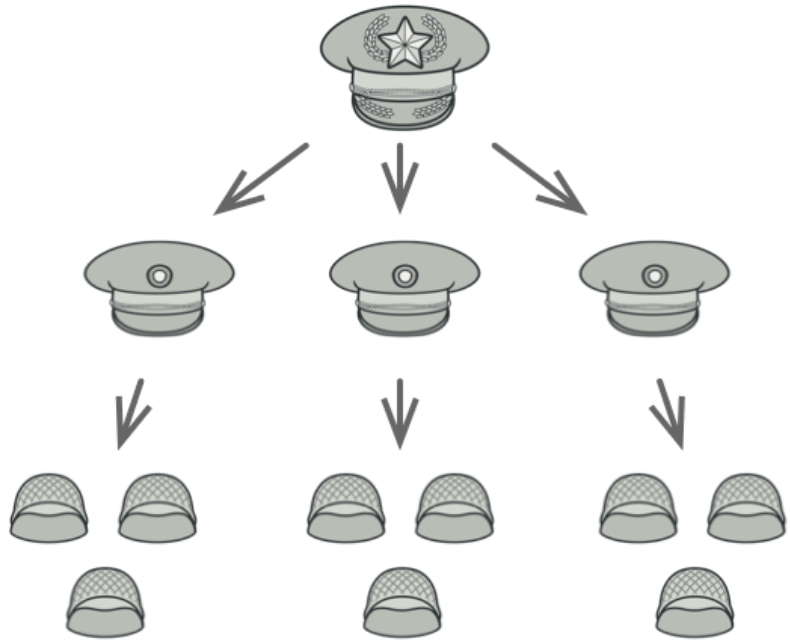
Solution

- 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, elle va aussi parcourir son contenu et ainsi de suite, jusqu'à ce que tous les prix aient été calculés.
    - Une boîte peut même ajouter des frais supplémentaires (e.g. emballage).



- Bonus : on n'a pas besoin de connaître la classe concrète des objets de l'arborescence !
  - Pas besoin de savoir si un objet est un produit ou une boîte.
  - On les manipule de la même manière grâce à une **interface** commune.
  - À l'appel d'une méthode, les objets s'occupent de propager la requête vers les feuilles de l'arbre.

Analogie



- 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.

Patrons de conception
Plan

Exemple : formes géométriques

- ☞ On souhaite réaliser une application de dessin, qui manipule des formes.
- ☞ Elle se base sur des formes à utiliser.
- ☞ Pour cela, on définit l'interface suivante :

```
1 public interface Forme {
2   public void translate (int dx, int dy);
3   public void dessine (Graphics g);
4 }
```

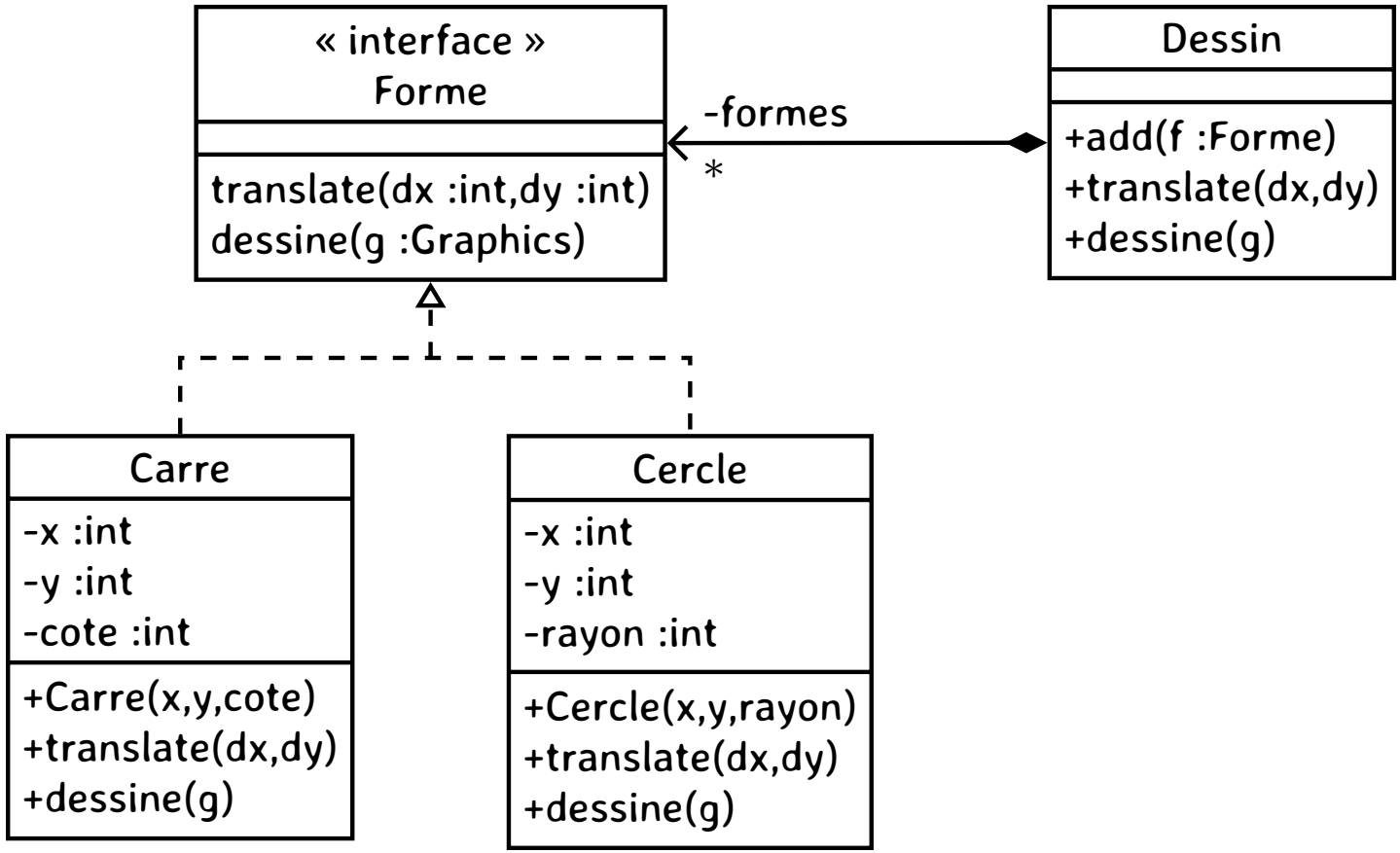
Exemple : formes géométriques
Carré, cercle, et dessin

```java
public class Carre implements Forme {
  private int x;
  private int y;
  private int cote;
  public Carre(int x,int y,int cote){
    this.x = x;
    this.y = y;
    this.cote = cote;
  }
  public void translate(int dx,int dy){
    x += dx;
    y += dy;
  }
  public void dessine(Graphics g){
    g.drawRect(x, y, cote, cote);
  }
}
```

```java
public class Cercle implements Forme {
  private int x;
  private int y;
  private int rayon;
  public Cercle(int x,int y,int rayon){
    this.x = x;
    this.y = y;
    this.rayon = rayon;
  }
  public void translate(int dx,int dy){
    x += dx;
    y += dy;
  }
  public void dessine(Graphics g){
    g.drawOval(x, y, rayon, rayon);
  }
}
```

```java
public class Dessin {
  List<Forme> formes = new ArrayList<Forme>();
  public void add (Forme f) { formes.add(f) }
  public void translate(int dx, int dy) {
    for (Forme f : formes)
      f.translate(dx, dy);
  }
  public void dessine(Graphics g) {
    for (Forme f : formes)
      f.dessine(g);
  }
}
```

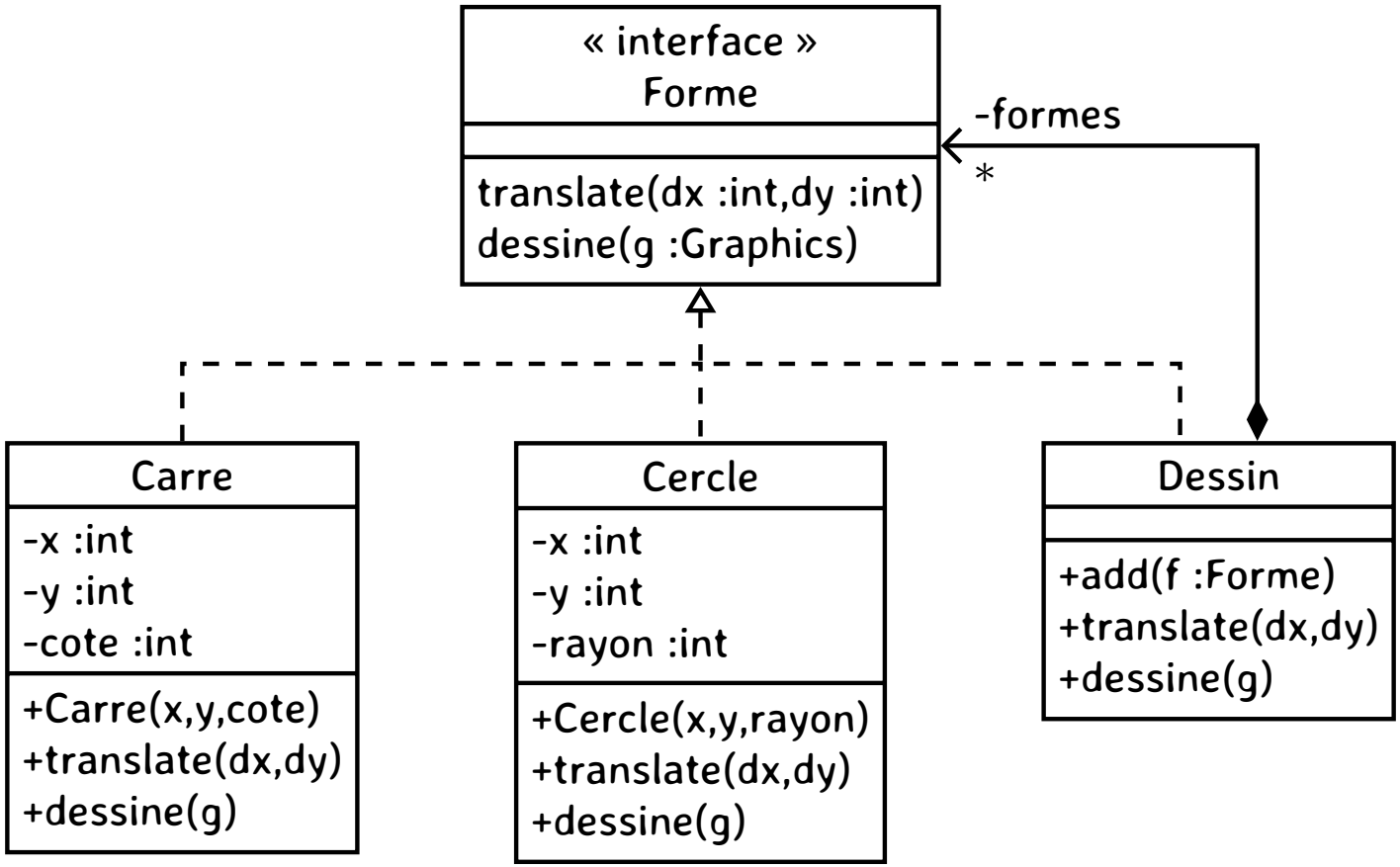
Exemple : formes géométriques
Diagramme de classes



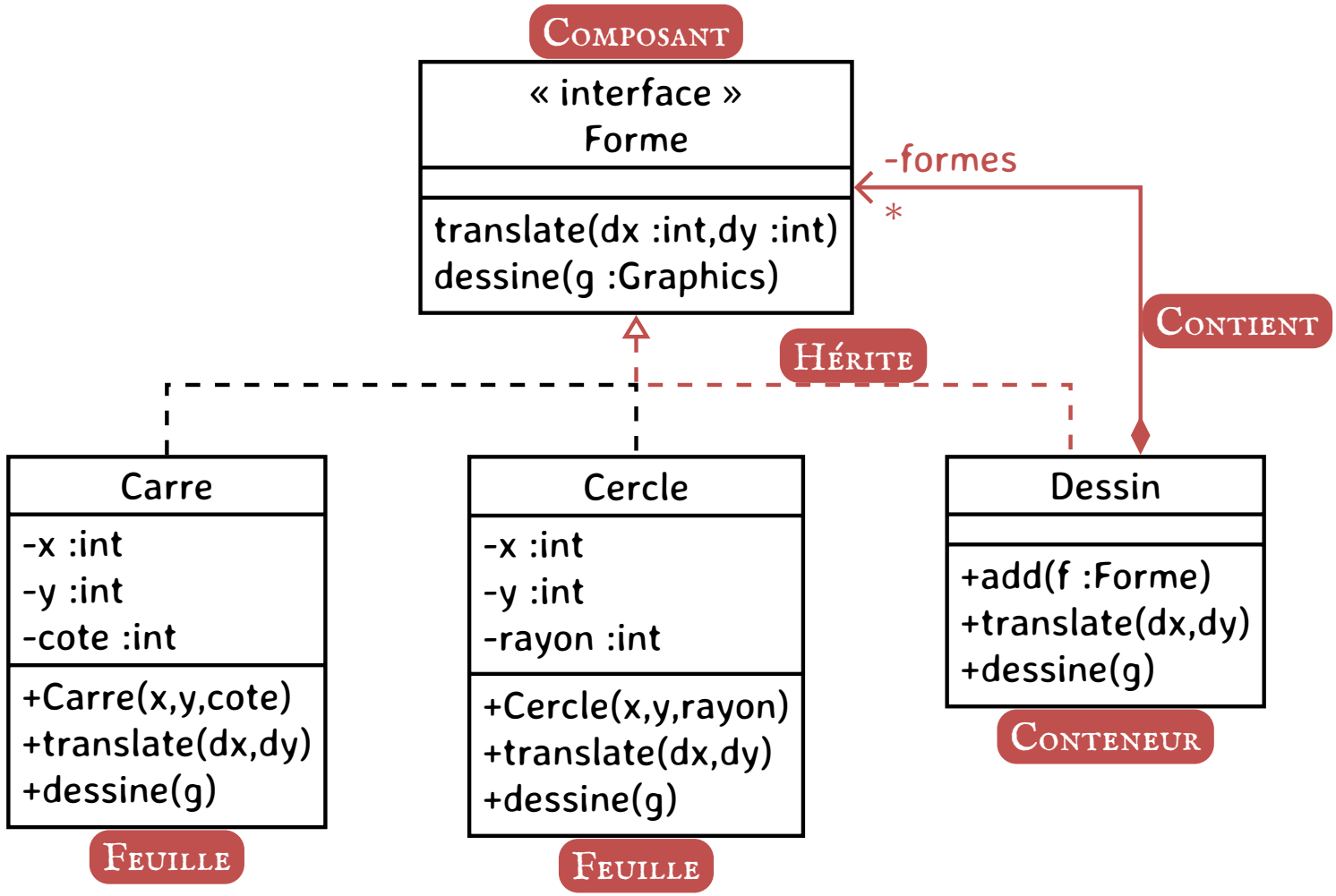
Exemple : formes géométriques
Mélanger des formes ?

- Pour étendre les possibilités de dessin, on pense aux mélanges
- Comment créer un CarréCercleConcentrique (carré contenant et cercle et contenu dans un autre) ?
  1) Implémentation directe (x,y) et longueur
     - Forte redondance dans le code
  2) Représenter par un dessin lui-même !
     - Toute forme est la composition de formes de base.
     - On peut donc représenter par une liste de formes (ie. Dessin).
     - Ici, contiendra un carré et deux cercles ...
     - Mais nécessite que **dessin soit aussi une forme**.
- L'option 2 est la définition correspondant au patron Composite.

Exemple : formes géométriques
Diagramme de classes avec Composite



Exemple : formes géométriques
Diagramme de classes avec Composite



Exemple : formes géométriques

- On modifie légèrement la définition de Dessin :
  `public class Dessin implements Forme`
- Construire notre CarreCercleConcentrique :

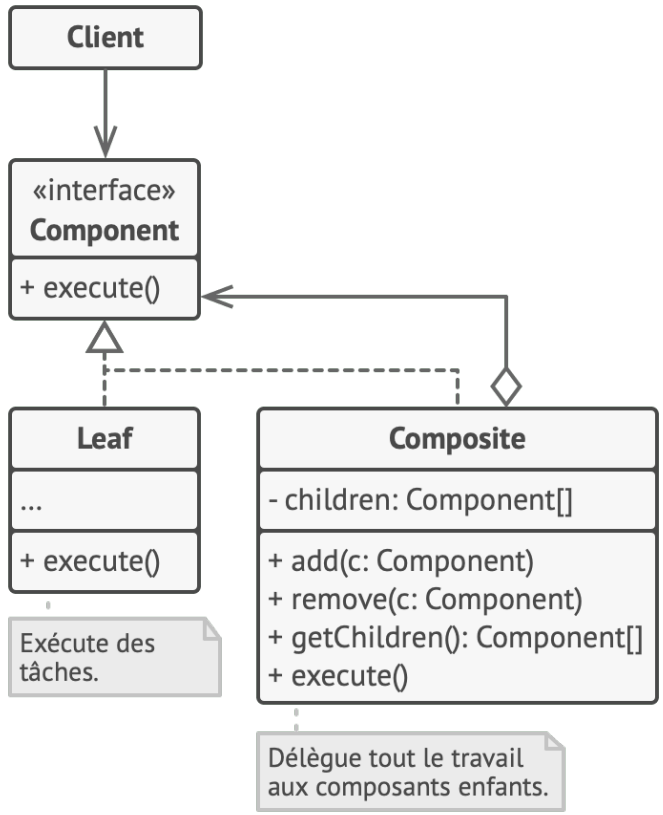
```
1 Dessin d = new Dessin();
2 d.add(new Carre(x,y,width));
3 d.add(new Cercle(x+width/2, y + width/2, width/2));
4 return d;
```

- Aucune limite sur la composition :
  - Ici un Dessin peut être la composition ...
  - ...d'un ensemble de dessins

Patrons de conception
Plan

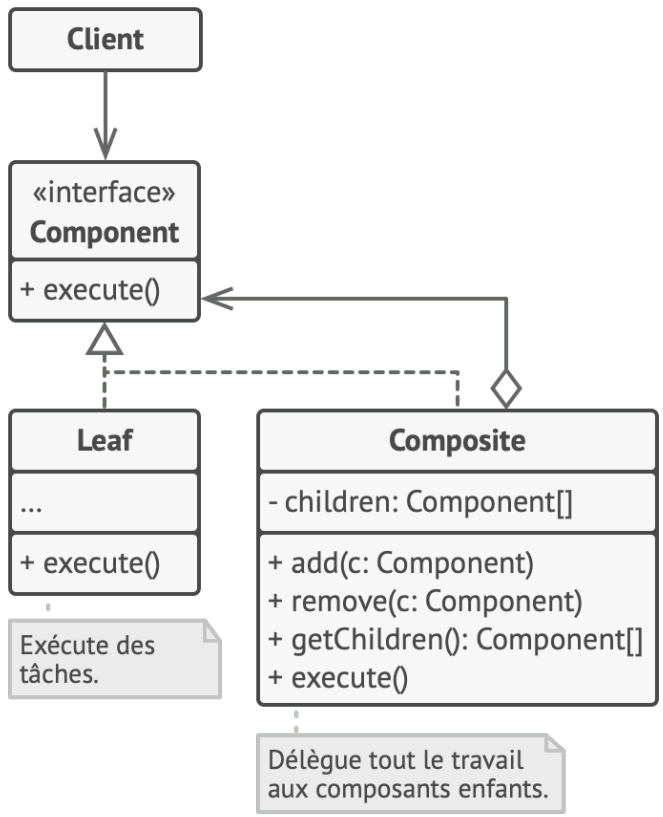
Structure

L'interface **Composant** décrit les opérations communes aux objets simples et complexes de l'arborescence.



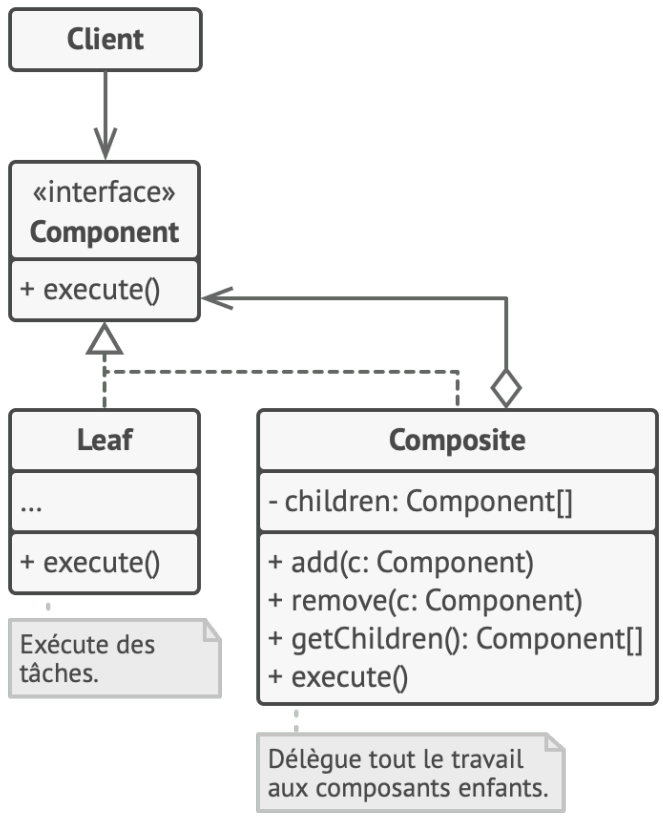
Structure

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.



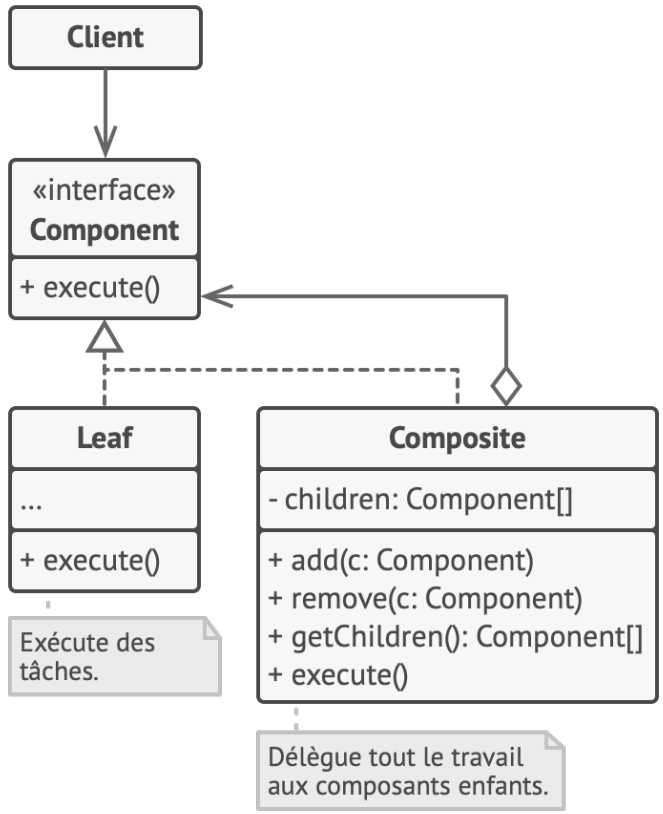
Structure

Le **Conteneur** (alias composite) est un élément composé de sous-éléments : des feuilles ou d'autres conteneurs. Un conteneur ne connait 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.



Structure

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.



Application de Composite

- Utilisez le composite si vous devez gérer une structure d'objets qui ressemble à une arborescence.
- Ce patron propose deux éléments de base qui partagent la même interface :
  1) des feuilles simples,
  2) 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.
- **Subtilité (double lien composite – composant) :** un composite est un composant et référence des composants.
- 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.*
   - *Les conteneurs peuvent accueillir à la fois 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.*
   - *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

Avantages

- On peut travailler dans des structures arborescentes complexes plus facilement en utilisant les avantages du polymorphisme et de la récursivité.
- **Principe ouvert/fermé :** on peut ajouter de nouveaux types d'éléments dans l'application, intégrables à l'arborescence, sans avoir à réécrire l'existant.

Inconvénient

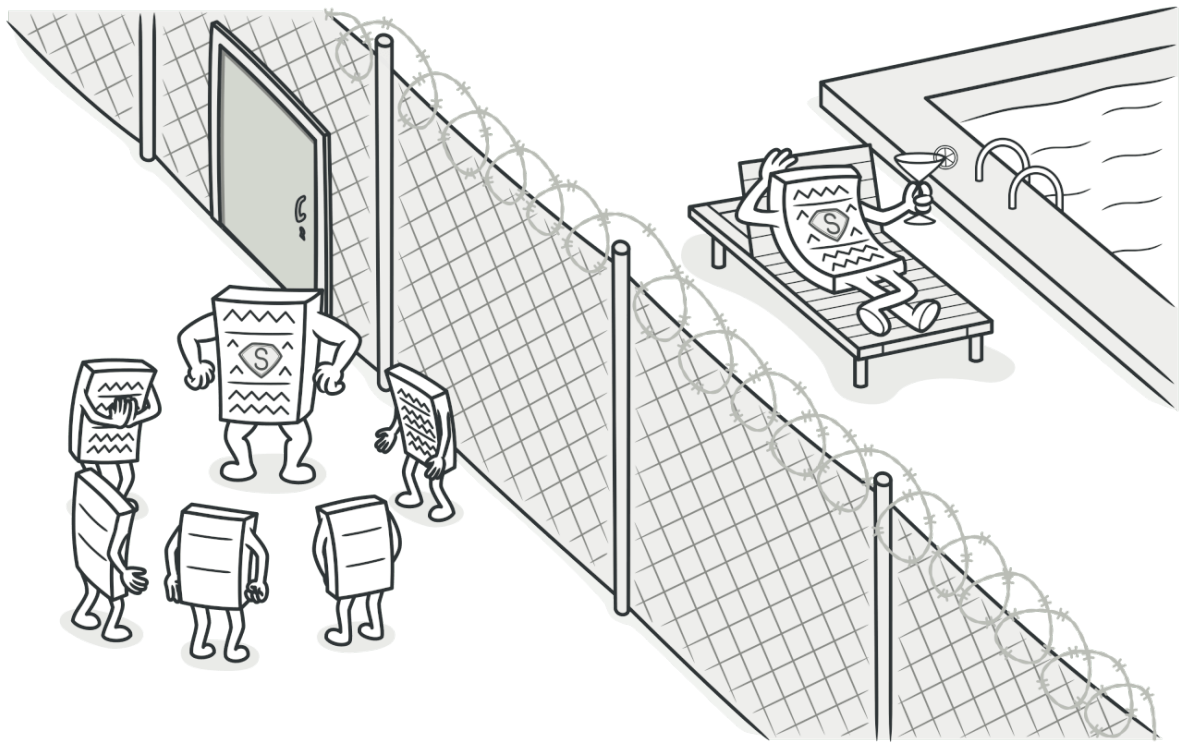
Vous rencontrerez parfois des difficultés pour définir une interface commune à certaines classes dont les fonctionnalités sont trop différentes. Dans certains scénarios, vous devez créer une interface composant bien trop générique, rendant le fonctionnement difficile à comprendre.

Patrons de conception
Plan

Patrons de conception
Plan

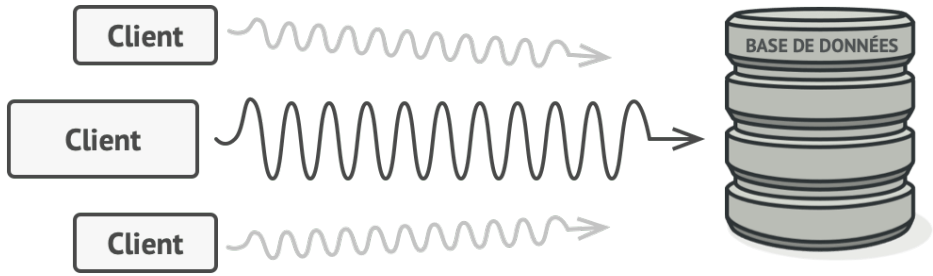
Intention

La **Procuration** (aussi appelée **Proxy**) 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

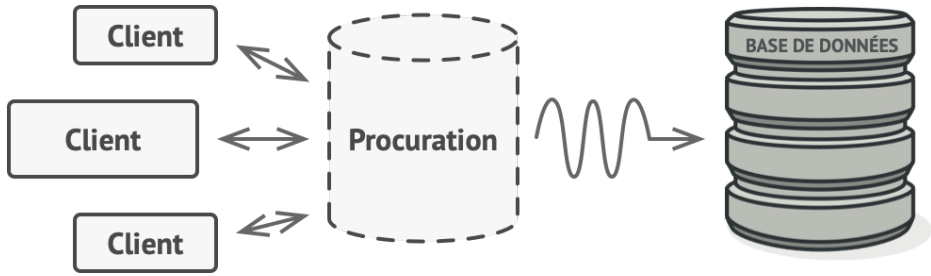
- On a un gros objet gourmand (consomme beaucoup de ressources).
- Mais on n'en a besoin que de temps en temps.



- On peut recourir à l'instanciation paresseuse :
  - On crée l'objet uniquement lorsque on en a besoin. Vous devrez
  - Il faut implémenter une initialisation différée dans tous les clients pour l'objet concerné.
  - Malheureusement, on va dupliquer beaucoup de code.
- Dans le meilleur des mondes, on voudrait 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.

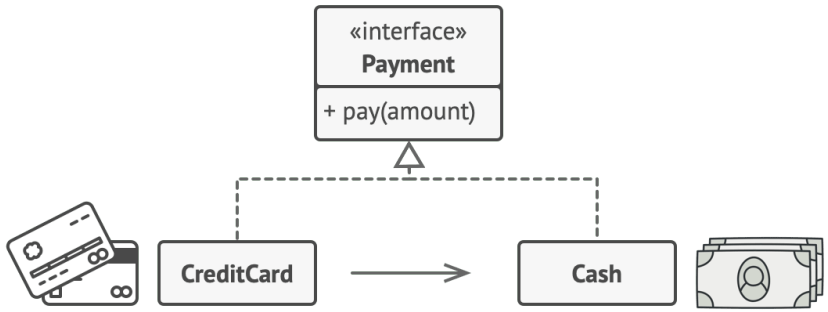
Solution : Proxy

- Proxy : objet faisant semblant d'être un autre objet



- Par exemple le proxy réseau de votre navigateur
  - Se comporte comme un gateway internet (box)
  - Mais rajoute des traitements (filtres, cache, ...)
- La classe procuration a la même interface que l'objet du service original.
- On passe 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.
- Plusieurs variantes de Proxy suivant l'usage
  - Proxy Virtuel : retarde les allocations/calculs couteux
  - Proxy de Sécurité : filtre/contrôle les accès à un objet
  - Proxy Distant : objet local se comportant comme le distant (masque le réseau)
  - Smart Reference : proxy qui compte les références (GC)

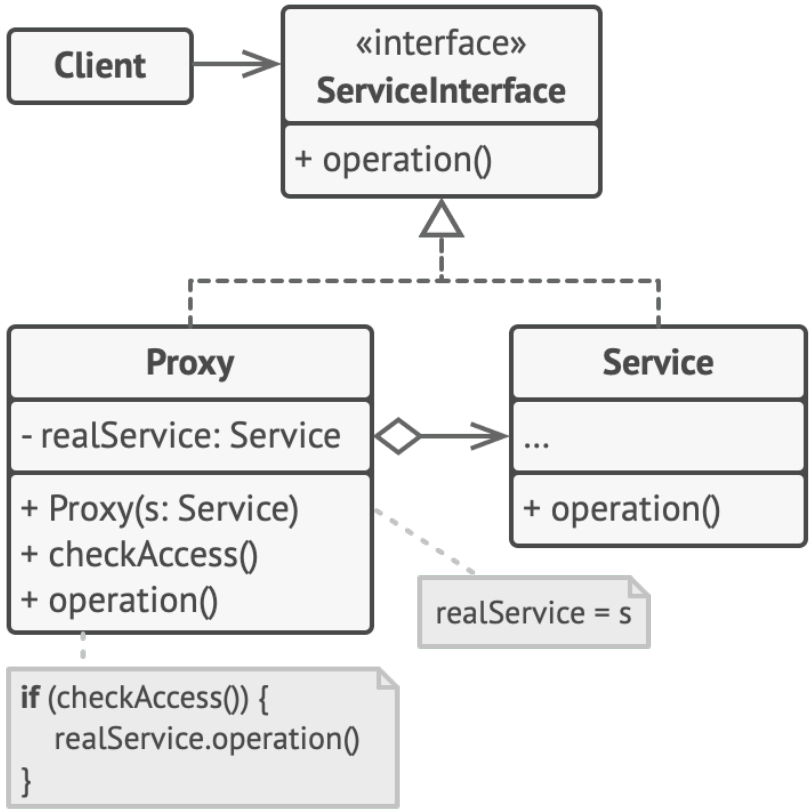
Analogie



- 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.

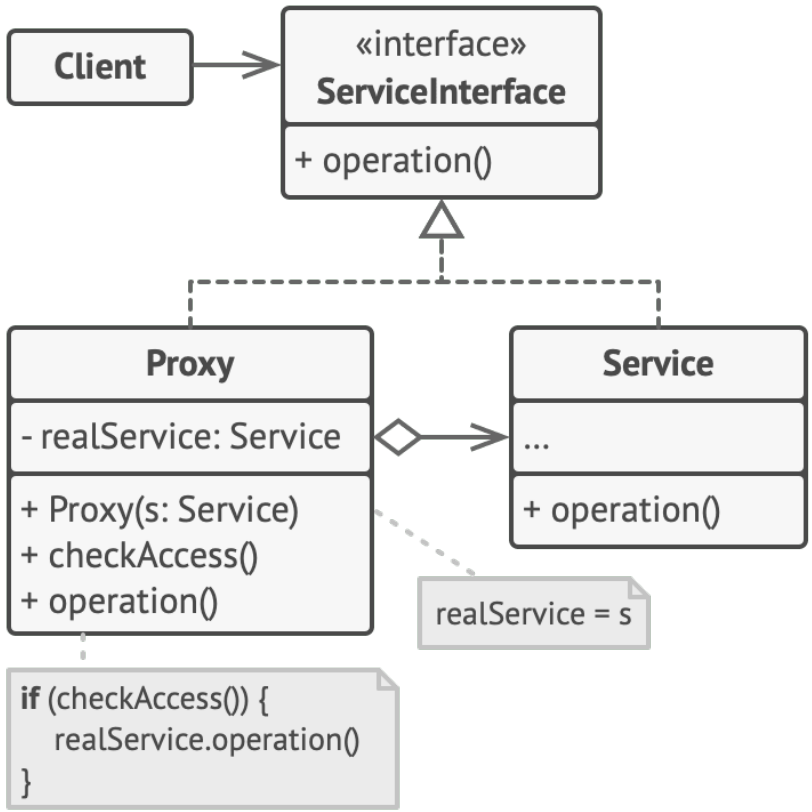
Structure

L'**Interface Service** déclare l'interface du service. La procuration doit implémenter cette interface afin de pouvoir se déguiser en objet du service.



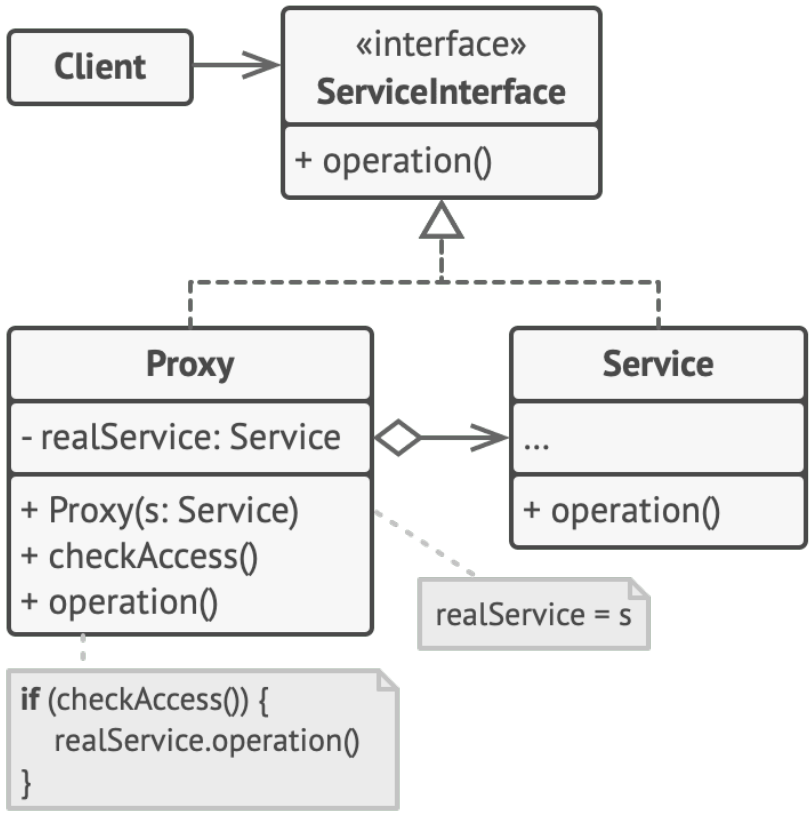
Structure

Le Service est une classe qui fournit la logique métier dont vous voulez vous servir.



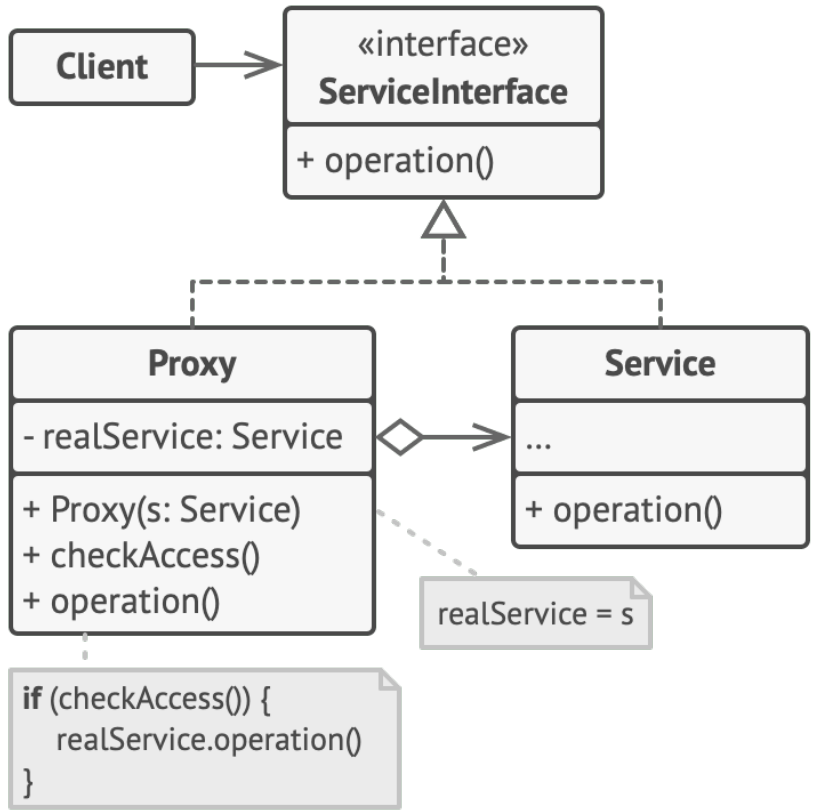
Structure

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.



Structure

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.

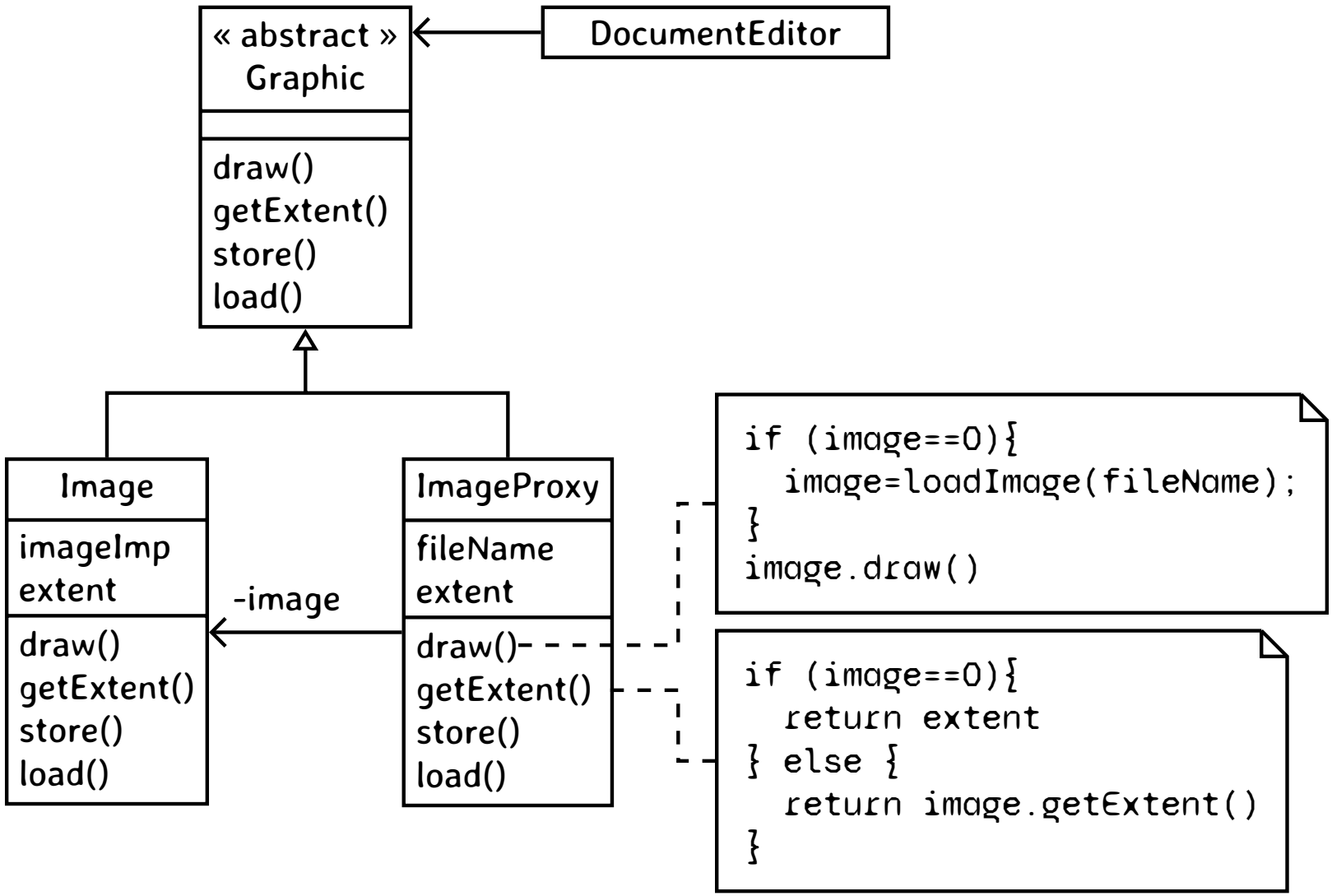


Patrons de conception
Plan

Proxy Virtuel

- Permet de retarder les opérations coûteuses
- Par exemple dans un éditeur de texte type Word
- Le document est rempli d'images « lourdes »
- Celles-ci sont stockées dans des fichiers séparés
- Quand on ouvre un document, il faut calculer la mise en page
- ...Et donc la taille des images
- ...Et donc faire le rendu des images présentes partout
- ...Quelle lenteur !
- Comment retarder le chargement des images ?

Éditeur de texte



Proxy Virtuel : Principes

- On construit initialement des ImageProxy pour chaque image.
  Par exemple le document le fait via une ImageFactory.
- Ces objets stockent et connaissent la taille de l'image
- Ce n'est que lorsqu'on affiche la page avec l'image (correspond à la prèmière invocation de draw sur le proxy) que l'image va être chargée (à la volée).
- Conclusion : le document s'ouvre rapidement et en plus le mécanisme est **transparent** pour l'utilisateur !
- Impossible de distinguer le Proxy de l'objet réel !

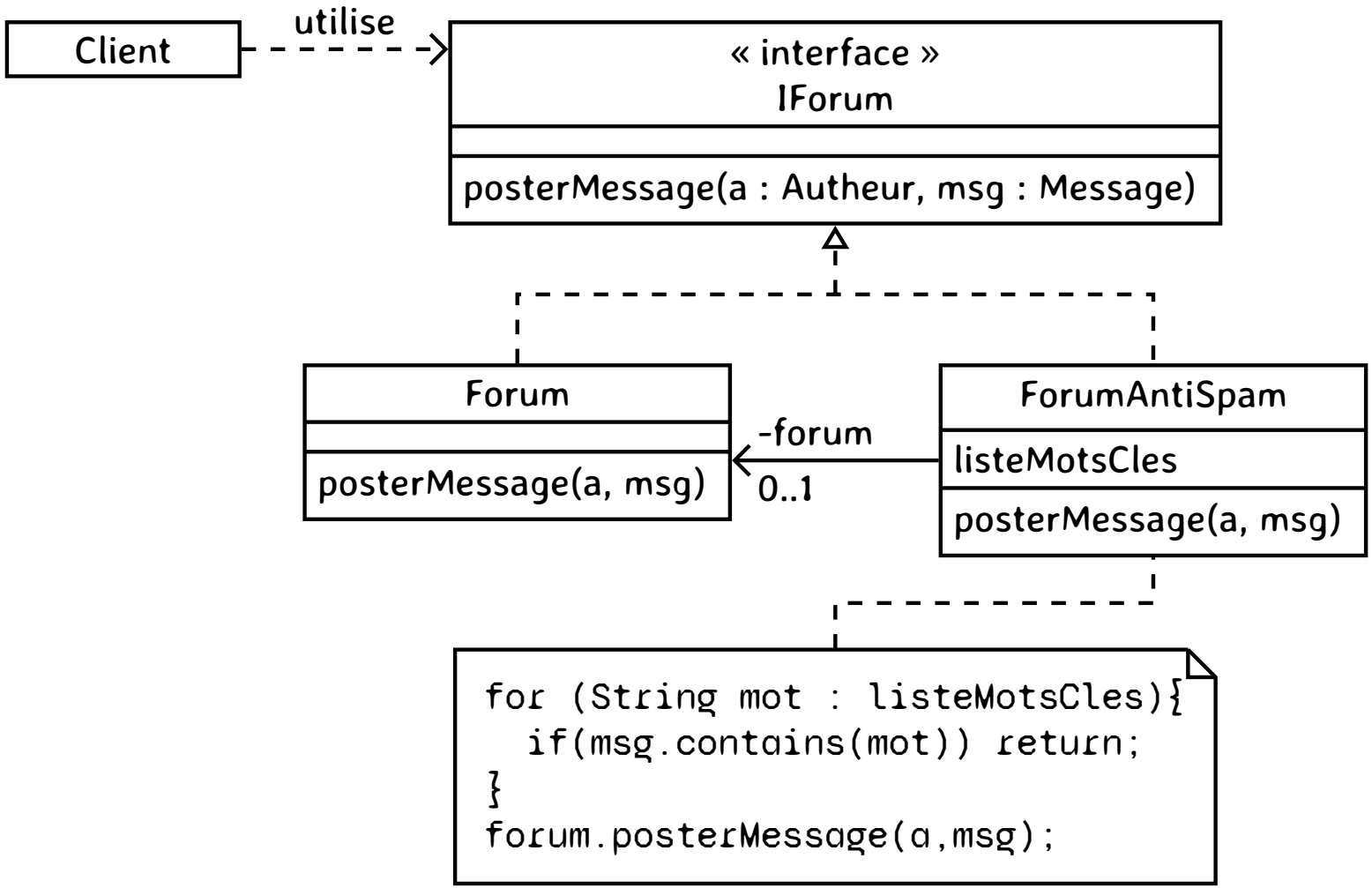
Patrons de conception
Plan

Proxy de Sécurité

- Permet de protéger ou contrôles les accès à un objet
- Exemple Forum de discussion
  - Classe Forum : munie d'une opération de post

    posterUnMessage(Auteur a, Message m)
  - La classe Forum existe, il s'agit de ne pas la modifier
- Comment bloquer des messages indésirables
- Contenant des mots clés interdits (langage SMS, Bieber)
- Cette fois un proxy de **sécurité**
- Ceci permet d'être orthogonal au **traitement protégé**
  - La sécurité est une couche supplémentaire
  - Distincte du traitement de base ...
  - Mais transparente pour l'utilisateur !

Forum anti-spam

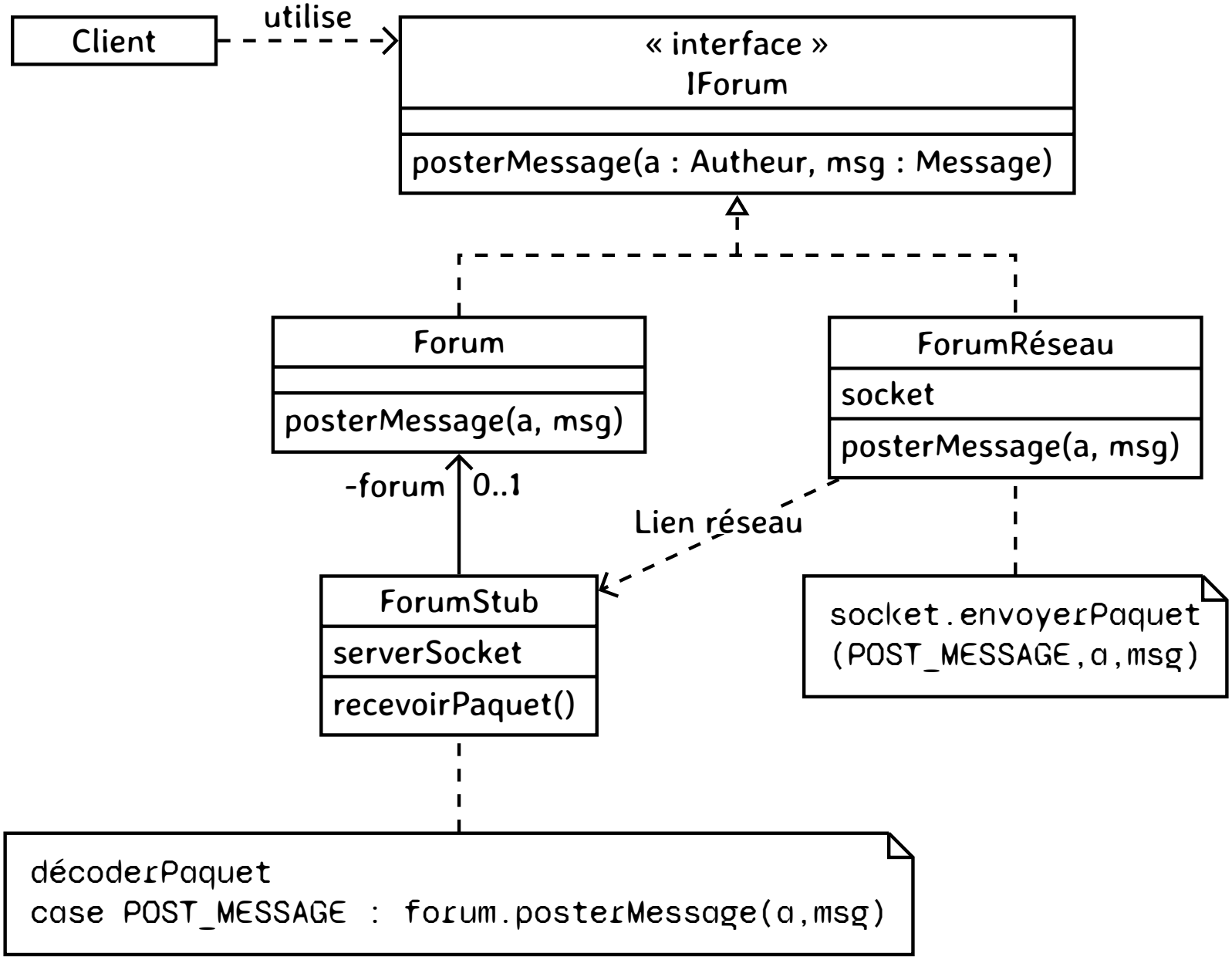


Patrons de conception
Plan

Proxy distant

- On a une application répartie sur plusieurs machines
- On voudrait développer l'application sans trop se soucier de l'endroit ou sont stockés physiquement les objets
- Proxy réseau
  - objet local à la machine
  - Se comporte comme l'objet distant
  - Répercute ses opérations sur l'objet distant via réseau
- Comportement par délégation ... mais avec le réseau interposé

Forum distant



Proxy distant

- Généralise la notion de Remote Procedure Call (RPC)
- Rend transparent la localisation des objets
- Réalisation du Proxy réseau et du stub suit une ligne standard
- De nombreux frameworks offrent de générer cette glue
- Permettent également de la cacher ou non
- Eg. Java RMI (Remote Method Invocation)

Patrons de conception
Plan

Mise en œuvre

1) Si le service n'a pas encore d'interface, créez-en une.
   - 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.
   - Plan B : transformer la procuration en sous-classe de la classe service.
   - Ainsi elle hérite de l'interface du service.
2) Créez la classe procuration.
   - Doit inclure un attribut pour la référence au service.
   - En général, les procurations créent et gèrent le cycle de vie de leurs services.
   - Parfois, un service est passé à la procuration par un constructeur du client.
3) Mettez en place les méthodes de la procuration et leur fonctionnement.
   - La procuration lance des traitements,
   - puis elle délégue le travail à l'objet du service
4) Réfléchissez à l'implémentation d'une méthode de création.
   - Décider si le 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 instanciation paresseuse pour l'objet du service.

Avantages et inconvénients

Avantages

- 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.

Inconvénients

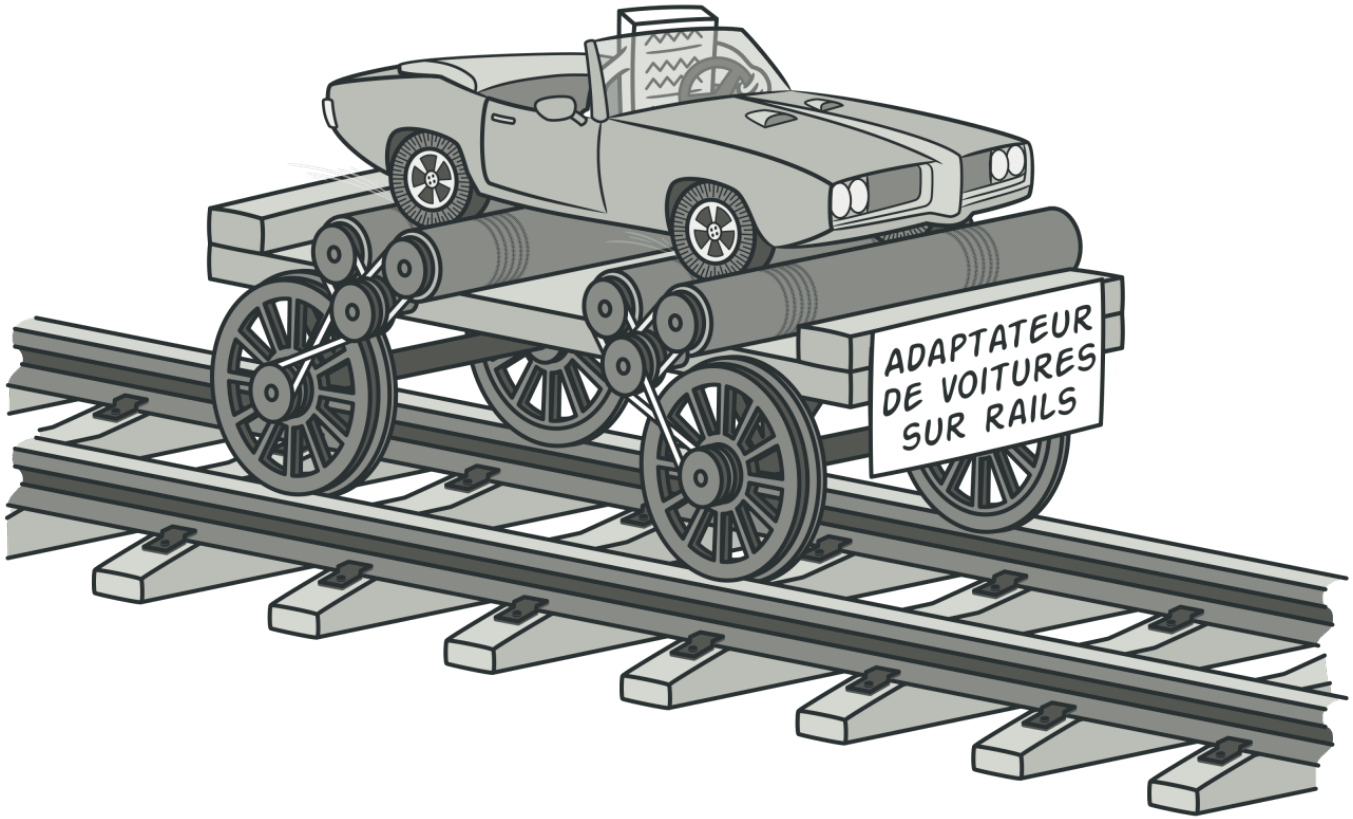
- Le code peut devenir plus complexe puisque vous devez y introduire de nombreuses classes.
- La réponse du service peut mettre plus de temps à arriver.

Patrons de conception
Plan

Patrons de conception
Plan

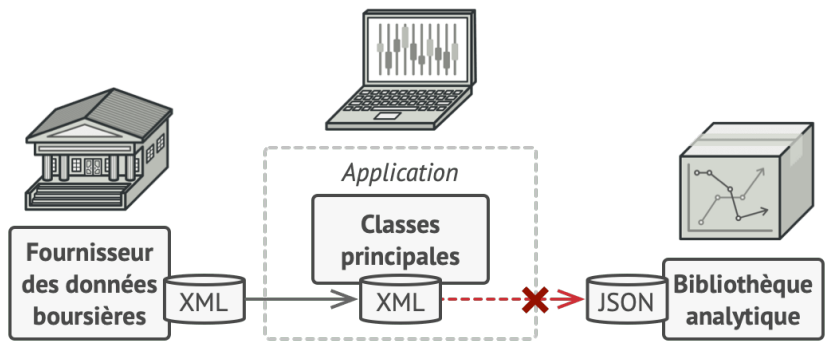
Intention

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



Problème

- Application de surveillance du marché boursier.
- L'application télécharge des données de la bourse depuis diverses sources au format XML.
- Elle 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.
- **Problème** : cette librairie ne fonctionne qu'avec des données au format JSON.

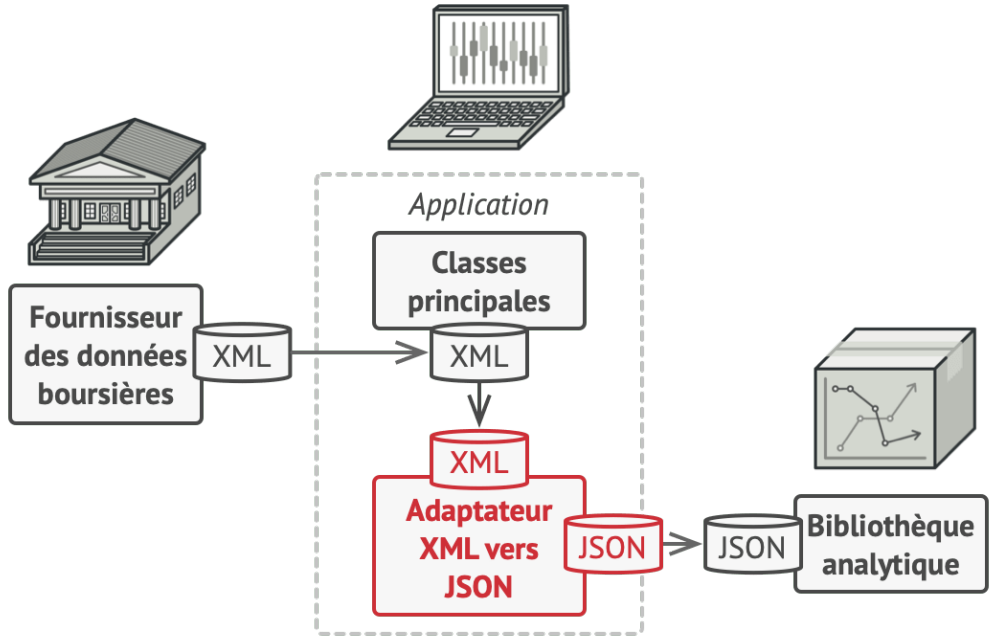


- Modifier la librairie afin qu'elle accepte du XML :
  - risque de faire planter d'autres parties de code qui utilisent déjà cette librairie ;
  - pas forcément possible, s'il s'agit d'une librairie externe.

Solution

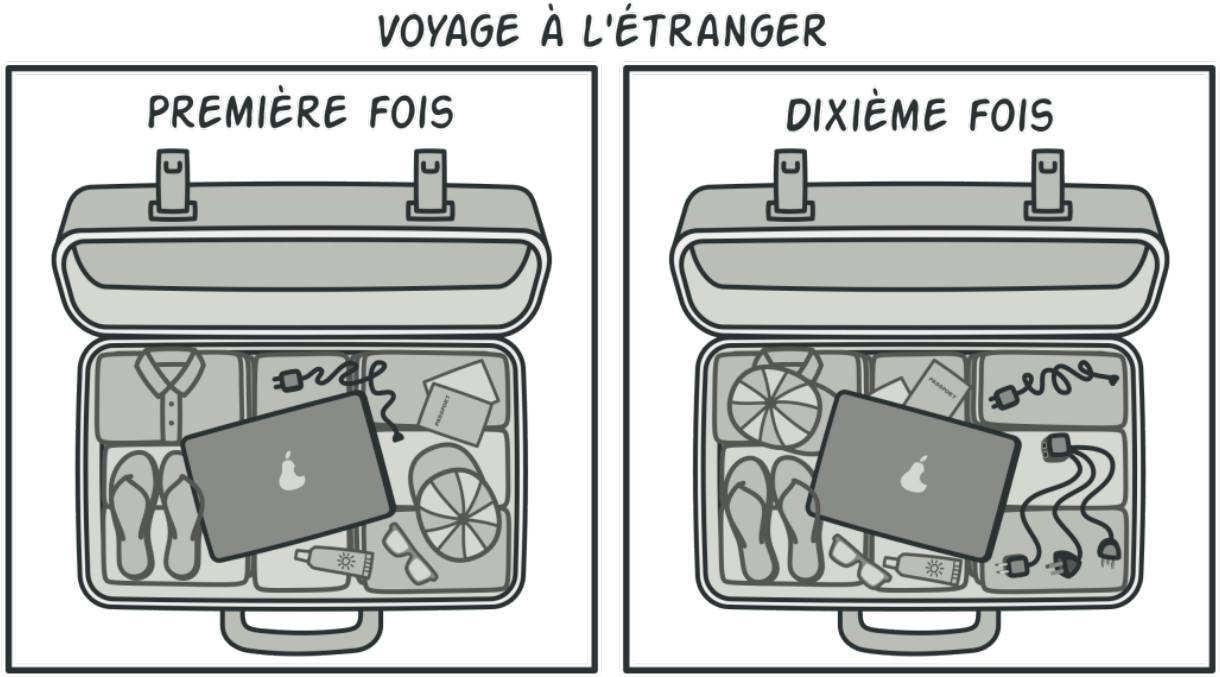
- Vous créez un adaptateur : un objet spécial qui convertit l'interface d'un objet afin qu'un autre objet puisse le comprendre.
- Un adaptateur encapsule un des objets afin de masquer la complexité de la conversion, exécutée à l'ombre des regards.
- L'objet encapsulé n'a pas conscience de ce que fait l'adaptateur.
- Exemple : conversion entre unités (métrique $\leftrightarrow$ impérial).
- 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 :
  - L'adaptateur prend une interface compatible avec un des objets existants.
  - L'objet existant peut appeler les méthodes de l'adaptateur via cette interface en toute sécurité.
  - 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 !

Solution pour notre exemple



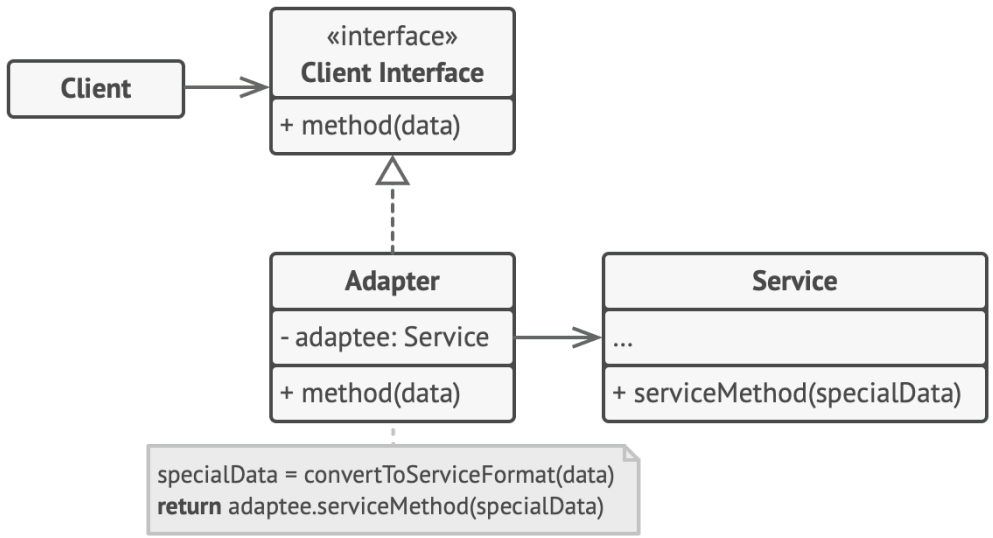
- Pour résoudre le problème des formats incompatibles, vous pouvez créer des adaptateurs XML vers JSON pour chaque classe de la librairie que notre code veut utiliser.
- Vous n'avez plus qu'à ajuster votre code pour communiquer avec la librairie à l'aide de ces adaptateurs.
- Lorsqu'un adaptateur reçoit 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 : chargeurs internationaux



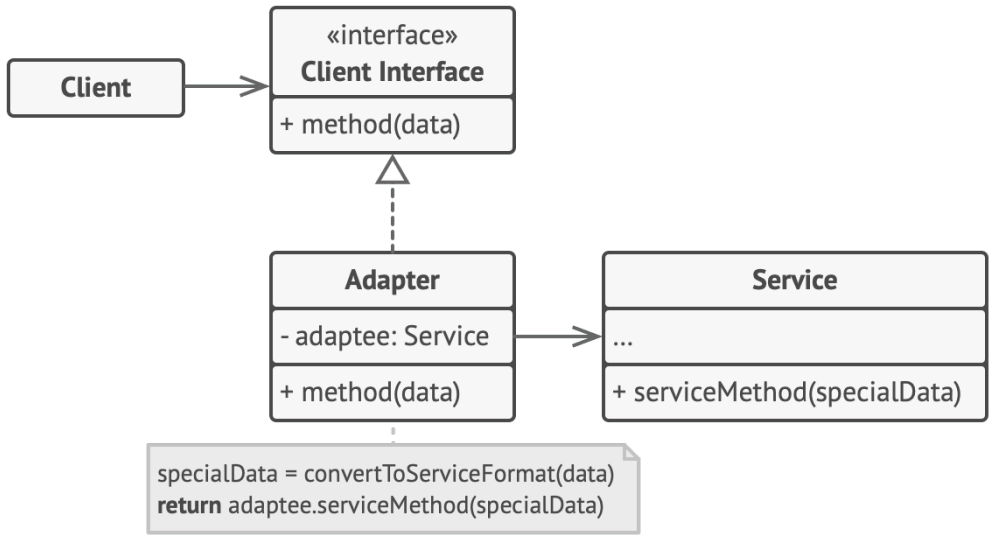
Structure

Le Client est une classe qui contient la logique métier du programme.



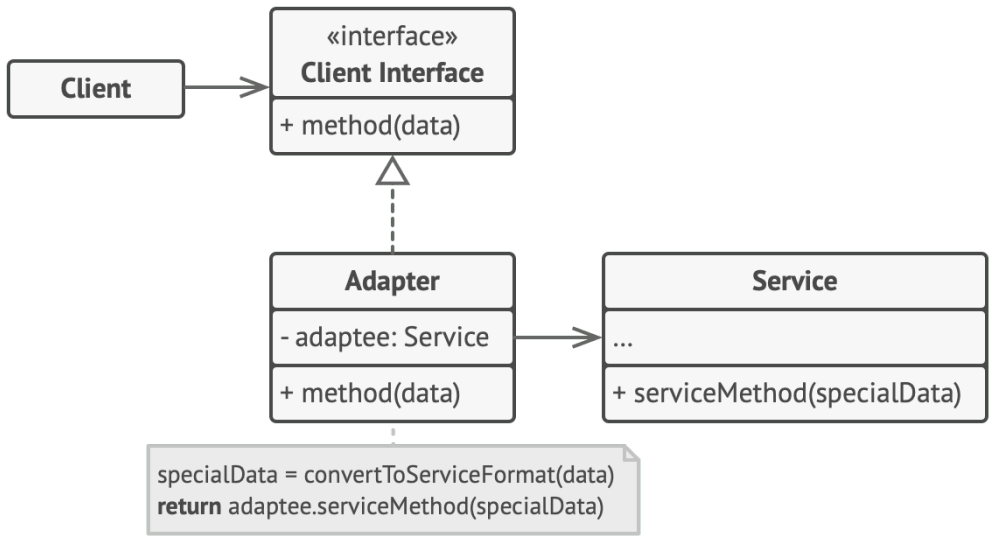
Structure

L'**Interface Client** décrit un protocole que les autres classes doivent implémenter afin de pouvoir collaborer avec le code client.



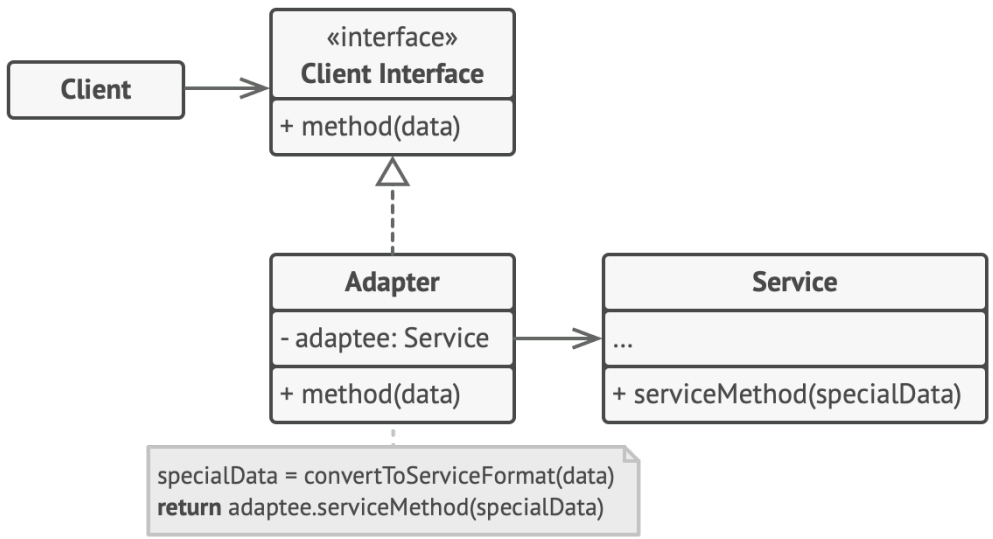
Structure

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.



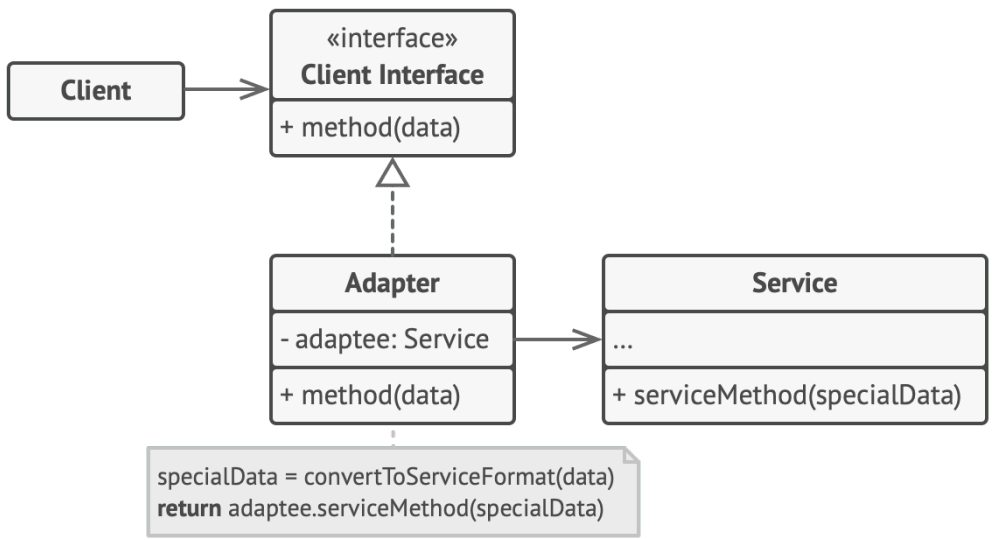
Structure

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.



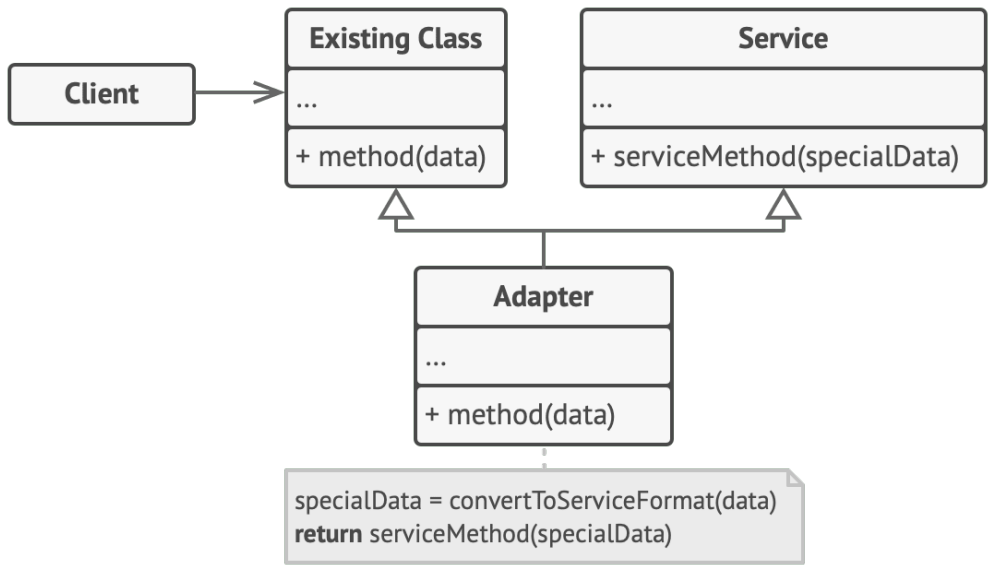
Structure

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.



Structure
Variante : adaptateur de classes

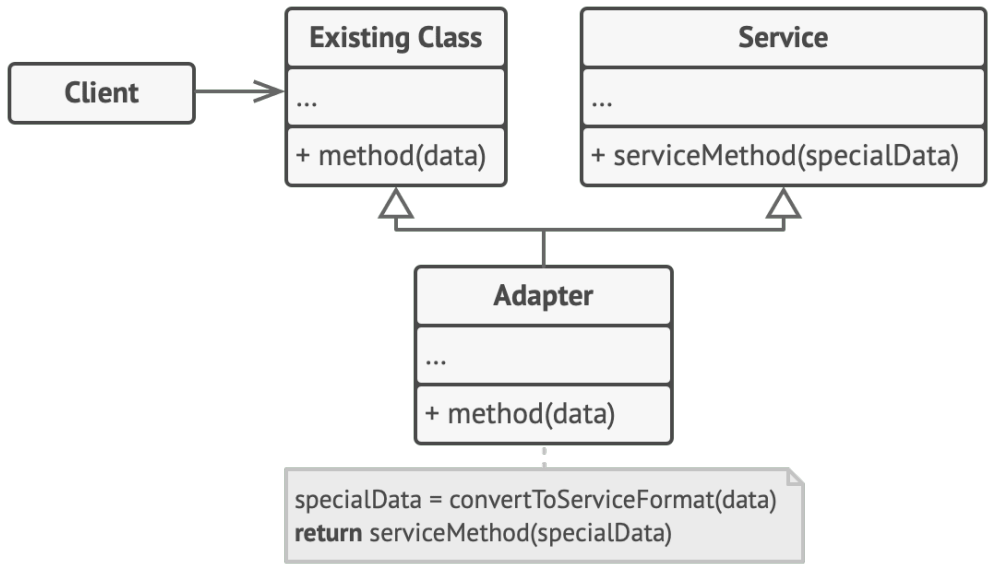
L'**Adaptateur de Classe** n'a pas besoin d'encapsuler des objets, car il hérite des comportements du client et du service. La totalité de l'adaptation se déroule à l'intérieur des méthodes redéfinies. Cet adaptateur peut remplacer une classe existante du client.



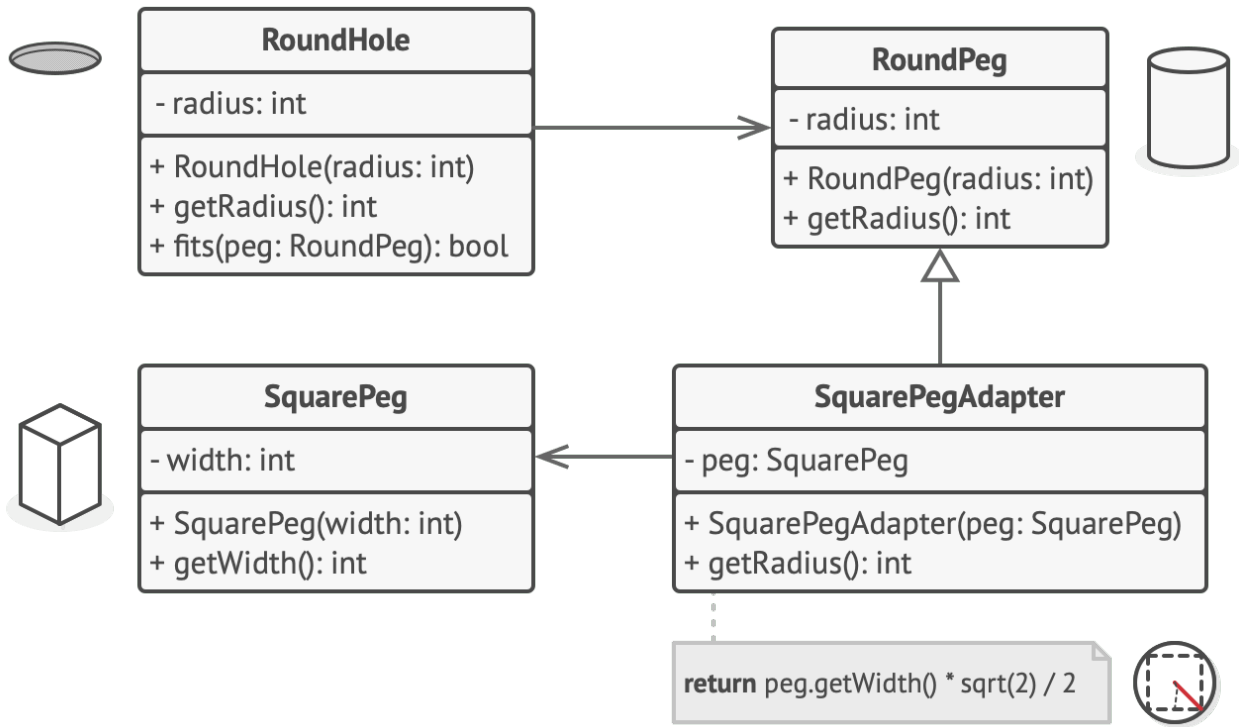
Structure
Variante : adaptateur de classes

Attention

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++.



Exemple


RoundHole
- radius: int
+ RoundHole(radius: int)
+ getRadius(): int
+ fits(peg: RoundPeg): bool
RoundPeg
- radius: int
+ RoundPeg(radius: int)
+ getRadius(): int
SquarePeg
- width: int
+ SquarePeg(width: int)
+ getWidth(): int
SquarePegAdapter
- peg: SquarePeg
+ SquarePegAdapter(peg: SquarePeg)
+ getRadius(): int
return peg.getWidth() * sqrt(2) / 2

Exemple

```
class RoundHole {
  constructor RoundHole(radius) { ... }
  method getRadius() {...}
  method fits(p: RoundPeg) {
    return (this.getRadius() >= p.radius
      ())
      }
}
class RoundPeg {
  constructor RoundPeg(radius) { ... }
  method getRadius() {...}
}

class SquarePeg {
  constructor SquarePeg(width) { ... }
  method getWidth() {...}
}

class SquarePegAdapter extends RoundPeg {
  private SquarePeg peg;

  constructor SquarePegAdapter(p:
      SquarePeg) {
    this.peg = p
      }
  method getRadius() {
    return (peg.getWidth() * Math.sqrt(2)
      / 2)
      }
}
```

```
// Quelque part dans le code client.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg)
// Ça ne compilera pas (types
      incompatibles).

small_sqpeg_adapter =
    new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter =
    new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

Possibilités d'application

- Besoin d'une classe existante, mais son interface est incompatible avec notre code.
  - L'adaptateur permet de créer une classe faisant office de couche intermédiaire.
  - Cette couche sert de convertisseur entre notre code et une classe héritée ou externe, ou n'importe quelle classe avec une interface incongrue.
- Réutiliser plusieurs sous-classes existantes à qui il manque des fonctionnalités communes qui ne peuvent pas être remontées dans la classe mère.
  - Si on étend chaque sous-classe pour y mettre la fonctionnalité manquante : duplication du code.
  - Plus élégant : mettre la fonctionnalité manquante dans une classe adaptateur.
  - Ensuite, encapsuler 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

Avantages

- **Principe de responsabilité unique :** vous découplez 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.

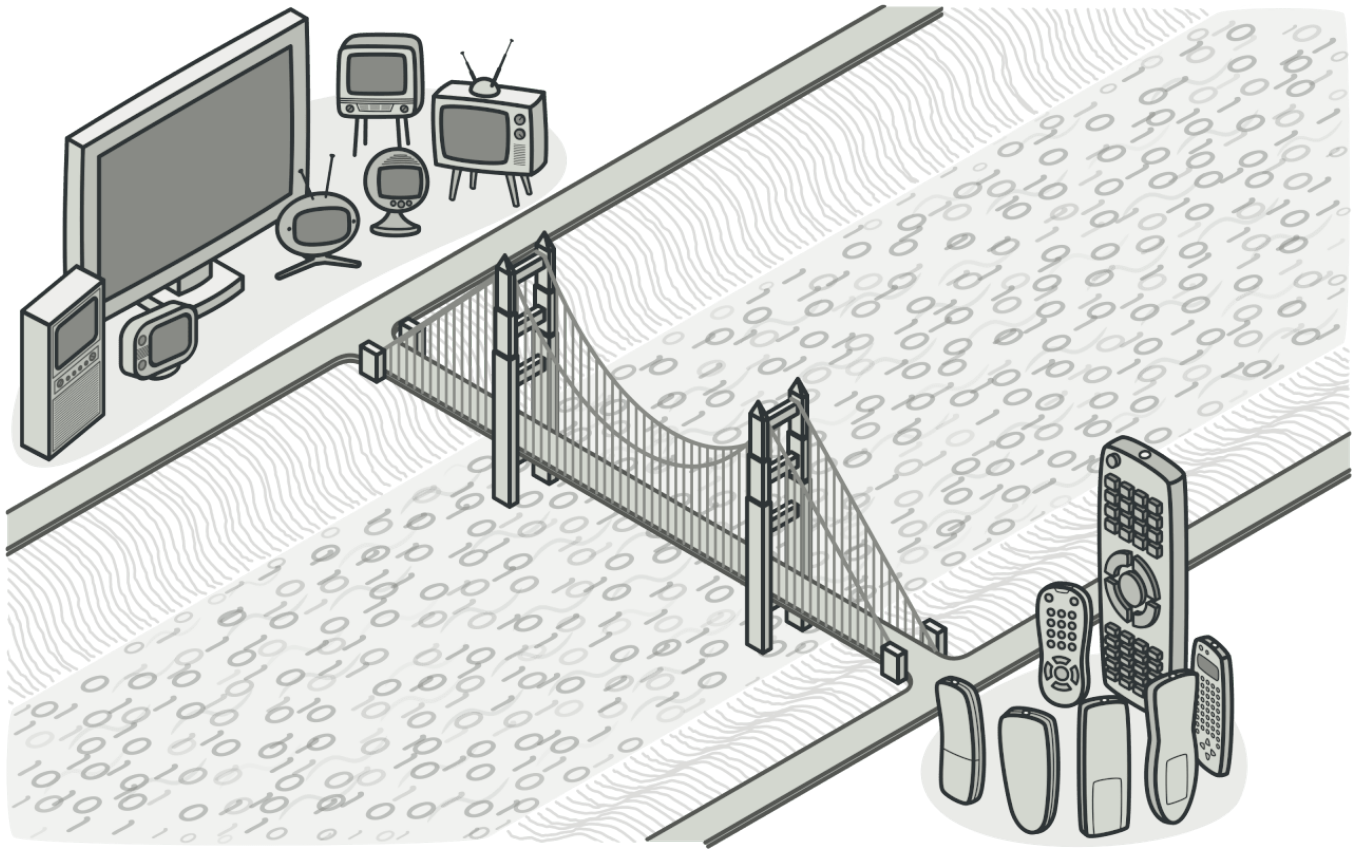
Inconvénient

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.

Patrons de conception
Plan

Intention

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

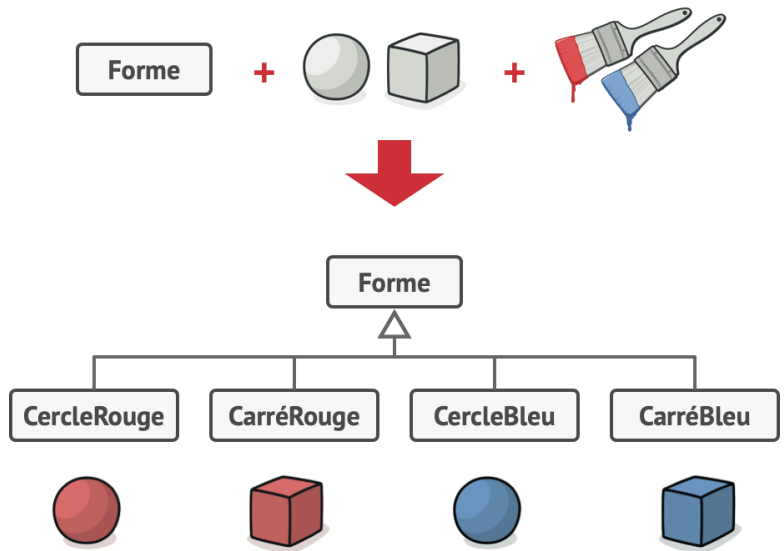
☞ Prenons une classe de Forme géométrique avec les sous-classes suivantes : Cercle et Carré.

Problème

- 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.

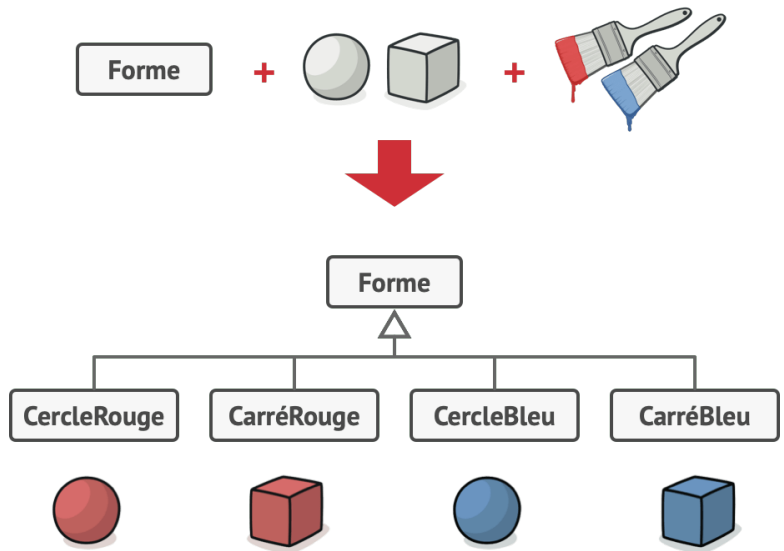
Problème

- 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.



Problème

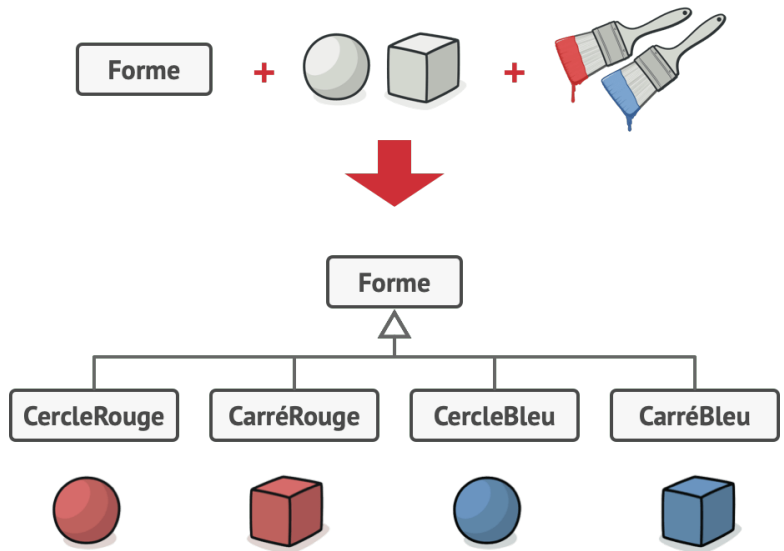
- Prenons une classe de Forme géométrique avec les sous-classes suivantes : Cercle et Carré.
- Vous voulez étendre cette hiérarchie de classes pour incorporer des couleurs, vous créez donc des sous-classes de formes : Rouge et Bleu.
- Mais vous avez déjà deux sous-classes, vous devez donc créer quatre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



- Et si on veut rajouter une forme Triangle, il faut rajouter deux classes (rouge et bleu).

Problème

- 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.

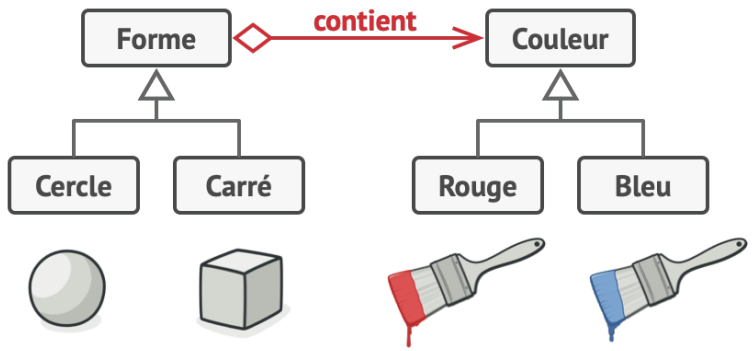


- Et si on veut rajouter une forme Triangle, il faut rajouter deux classes (rouge et bleu).
- Si on veut ensuite ajouter la couleur verte, il faut ajouter trois classes...

Solution

Le problème dans cet exemple est qu'on veut utiliser un arbre d'héritage pour représenter deux dimensions indépendantes : la forme et la couleur.

☞ Le pont résout se problème en utilisant une **composition**, pour assembler deux arbres d'héritage correspondant à des dimensions indépendantes.



Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction :**

Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction :**
   - C'est une couche de contrôle de haut niveau.

Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction :**
   - C'est une couche de contrôle de haut niveau.
   - Elle n'est pas censée effectuer de traitements toute seule.

Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction** :
   - C'est une couche de contrôle de haut niveau.
   - Elle n'est pas censée effectuer de traitements toute seule.
   - Elle doit déléguer le travail à la couche implémentation (appelée également plateforme).

Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction** :
   - C'est une couche de contrôle de haut niveau.
   - Elle n'est pas censée effectuer de traitements toute seule.
   - Elle doit déléguer le travail à la couche implémentation (appelée également plateforme).
2) **L'implémentation** :

Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction** :
   - C'est une couche de contrôle de haut niveau.
   - Elle n'est pas censée effectuer de traitements toute seule.
   - Elle doit déléguer le travail à la couche implémentation (appelée également plateforme).
2) **L'implémentation** :
   - Elle effectue le travail demandé par l'abstraction.

Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction** :
   - C'est une couche de contrôle de haut niveau.
   - Elle n'est pas censée effectuer de traitements toute seule.
   - Elle doit déléguer le travail à la couche implémentation (appelée également plateforme).
2) **L'implémentation** :
   - Elle effectue le travail demandé par l'abstraction.
   - C'est elle qui fait des appels système, qui écrit dans la mémoire...

Conceptuellement

On distingue deux parties d'une application (ou d'une entité dans une application) :

1) **L'abstraction** :
   - C'est une couche de contrôle de haut niveau.
   - Elle n'est pas censée effectuer de traitements toute seule.
   - Elle doit déléguer le travail à la couche implémentation (appelée également plateforme).
2) **L'implémentation** :
   - Elle effectue le travail demandé par l'abstraction.
   - C'est elle qui fait des appels système, qui écrit dans la mémoire...

exemple : un logiciel

☞ **Abstraction** : Interface graphique (GUI)

☞ **Implémentation** : code du système d'exploitation (API).

En réponse aux interactions de l'utilisateur, la couche GUI envoie des tâches à la couche API.

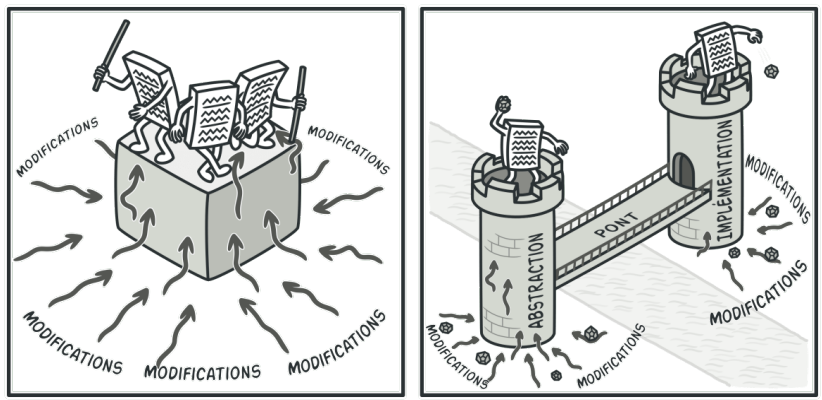
Exemple : un logiciel entier

☞ **Abstraction** : Interface graphique (GUI)

On peut souhaiter étendre un tel logiciel dans deux directions :

- ☞ Avec plusieurs GUI différentes (accessibilité, ergonomie, utilisateur novice/expérimenté...)
- ☞ avec plusieurs API différentes (Windows, Linux et macOS).

Si on ne fait pas attention, on obtient un code horrible, avec des centaines de conditions connectant différents types de GUI et d'API, réparties un peu partout à travers le code.



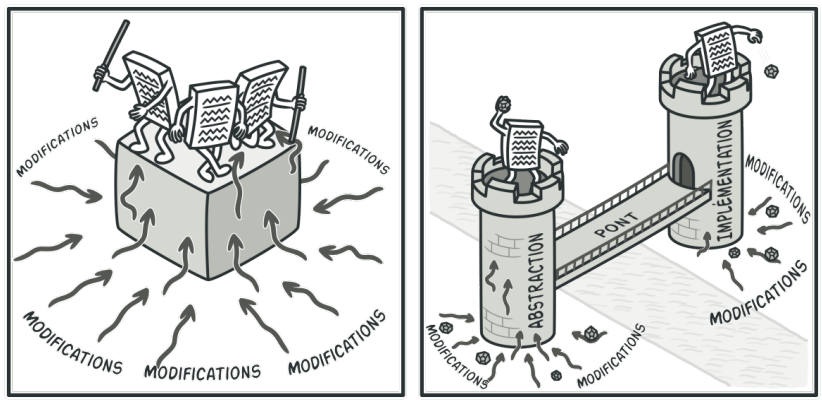
Exemple : un logiciel entier

- **Abstraction** : Interface graphique (GUI)
- **Implémentation** : code du système d'exploitation (API).

On peut souhaiter étendre un tel logiciel dans deux directions :

- Avec plusieurs GUI différentes (accessibilité, ergonomie, utilisateur novice/expérimenté...)
- avec plusieurs API différentes (Windows, Linux et macOS).

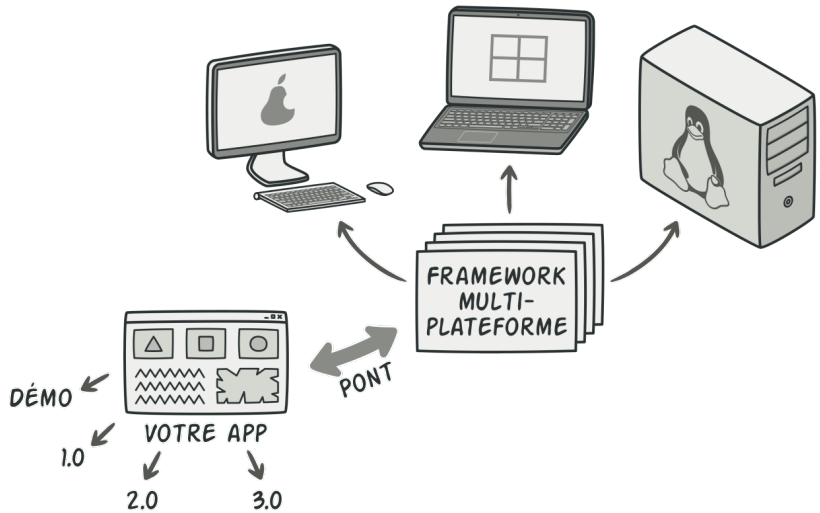
Si on ne fait pas attention, on obtient un code horrible, avec des centaines de conditions connectant différents types de GUI et d'API, réparties un peu partout à travers le code.



Solution, avec le pont

On met les classes dans deux hiérarchies :

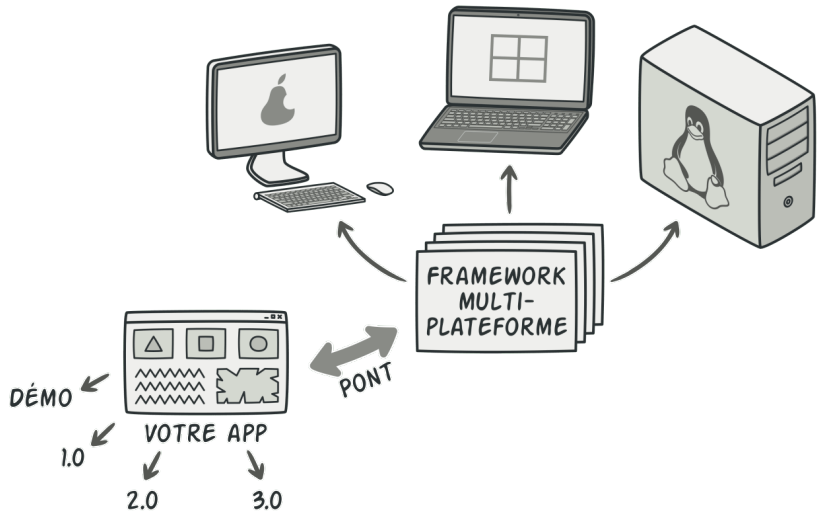
☞ Abstraction : la couche GUI de l'application.



Solution, avec le pont

On met les classes dans deux hiérarchies :

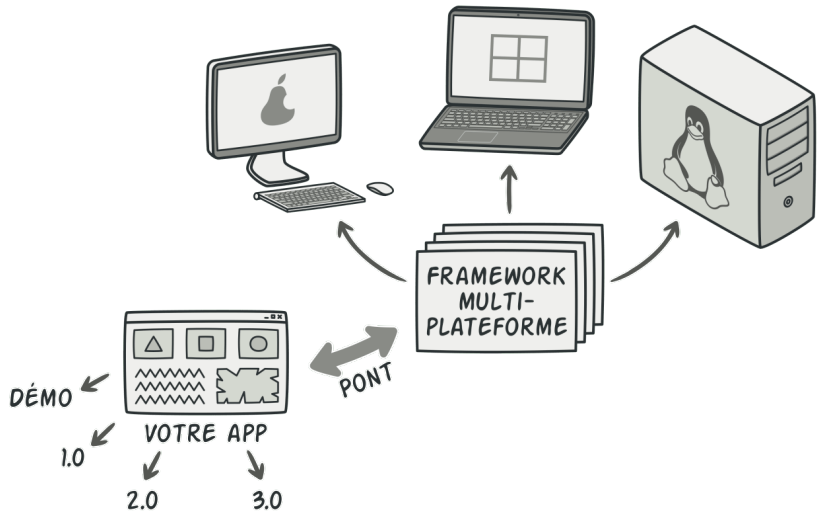
- ☞ Abstraction : la couche GUI de l'application.
- ☞ Implémentation : les API des systèmes d'exploitation.



Solution, avec le pont

On met les classes dans deux hiérarchies :

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

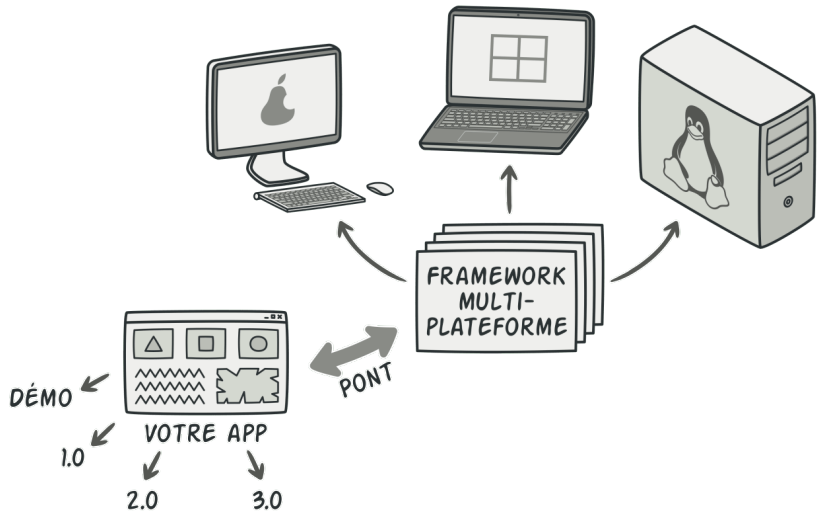


- L'objet abstraction contrôle l'apparence de l'application.

Solution, avec le pont

On met les classes dans deux hiérarchies :

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

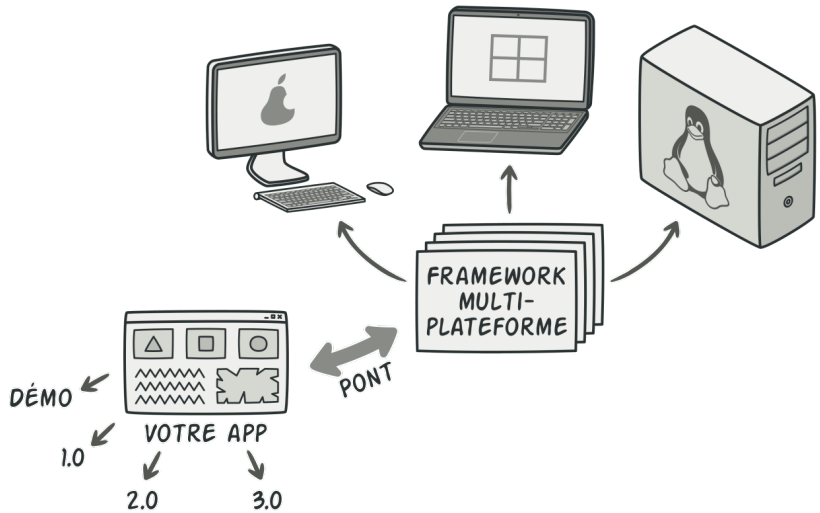


- L'objet abstraction contrôle l'apparence de l'application.
- Il délègue la partie métier à l'objet d'implémentation correspondant.

Solution, avec le pont

On met les classes dans deux hiérarchies :

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

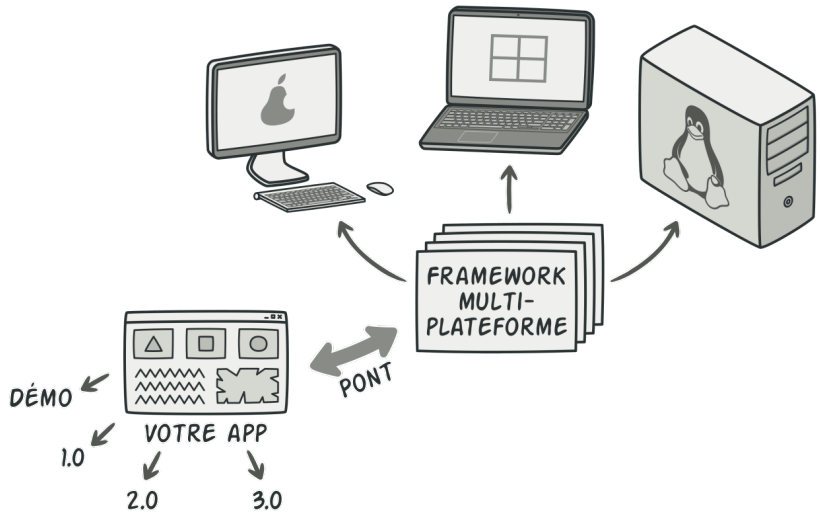


- L'objet abstraction contrôle l'apparence de l'application.
- Il délègue la partie métier à l'objet d'implémentation correspondant.
- Toutes les implémentations qui implémentent la même interface sont « équivalentes ».

Solution, avec le pont

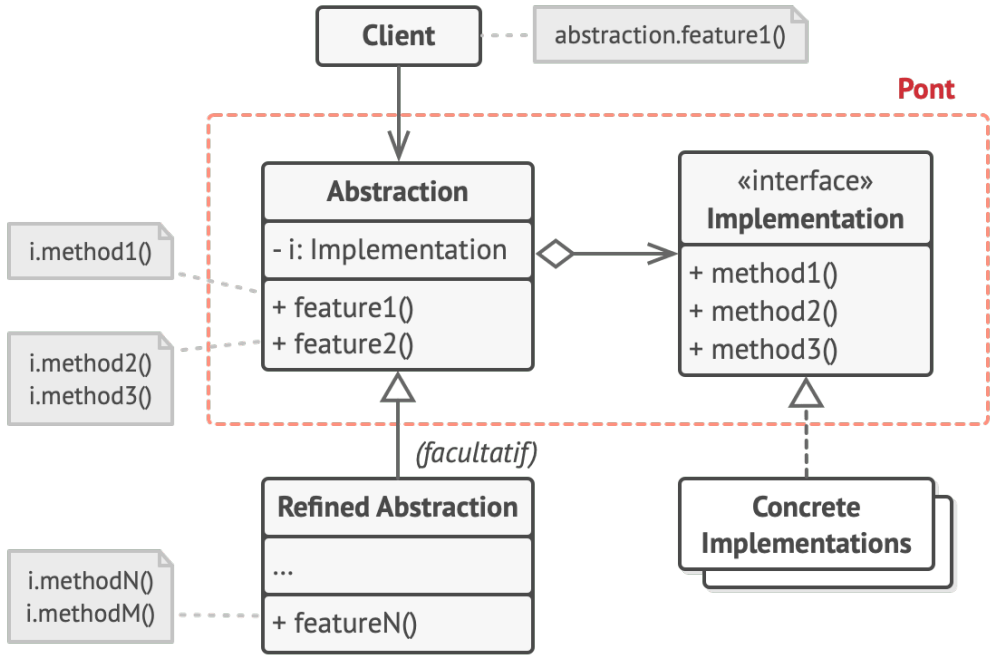
On met les classes dans deux hiérarchies :

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



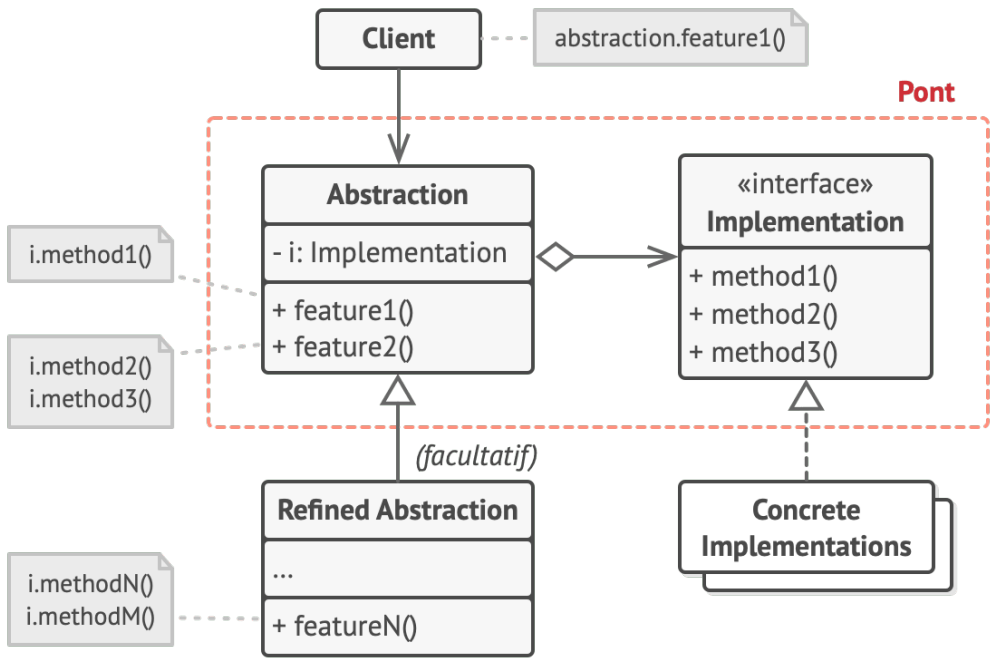
- L'objet abstraction contrôle l'apparence de l'application.
- Il délègue la partie métier à l'objet d'implémentation correspondant.
- Toutes les implémentations qui implémentent la même interface sont « équivalentes ».
- Donc la même GUI peut fonctionner aussi bien sous Windows que sous Linux.

Structure


Client
abstraction.feature1()
Pont
Abstraction
- i: Implementation
+ feature1()
+ feature2()
«interface»
Implementation
+ method1()
+ method2()
+ method3()
i.method1()
i.method2()
i.method3()
(facultatif)
Refined Abstraction
...
+ featureN()
Concrete
Implementations
i.methodN()
i.methodM()

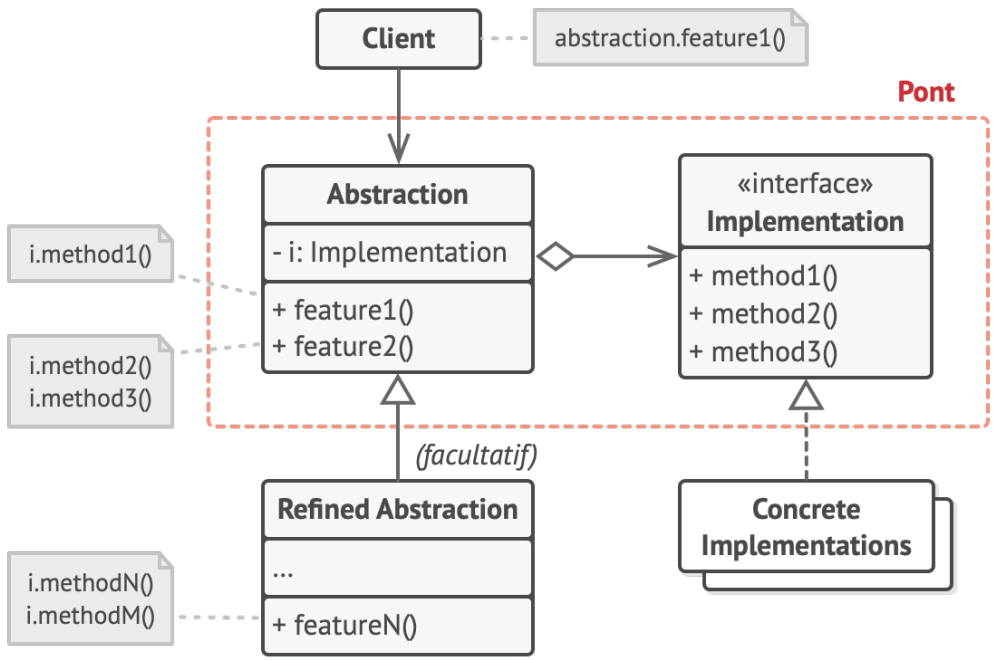
Structure

L'**Abstraction** offre une logique de contrôle de haut niveau. Elle compte sur l'implémentation pour les tâches de bas niveau.

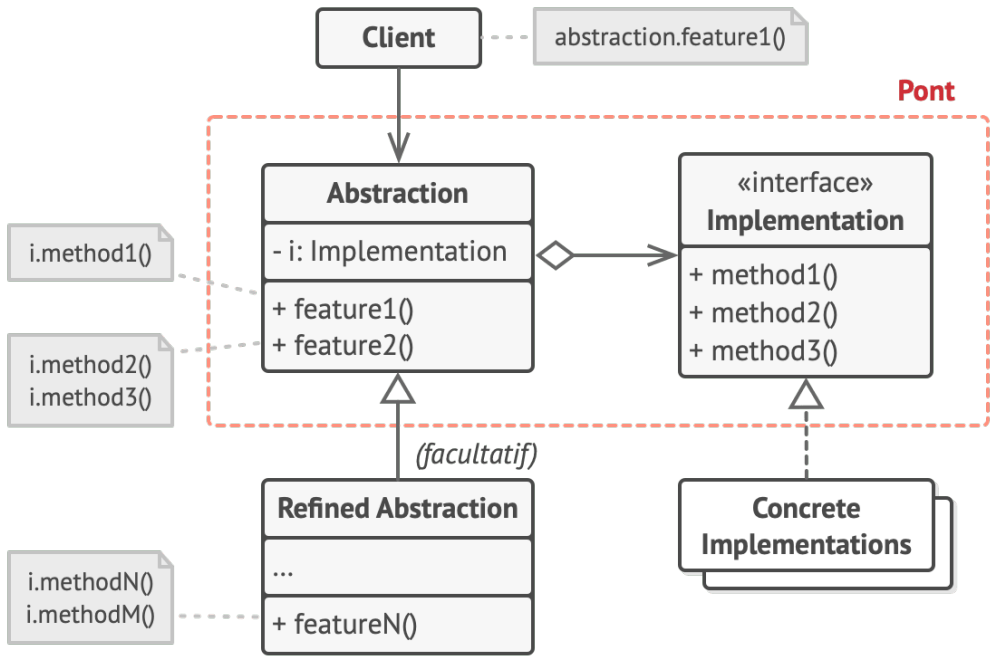


Structure

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.

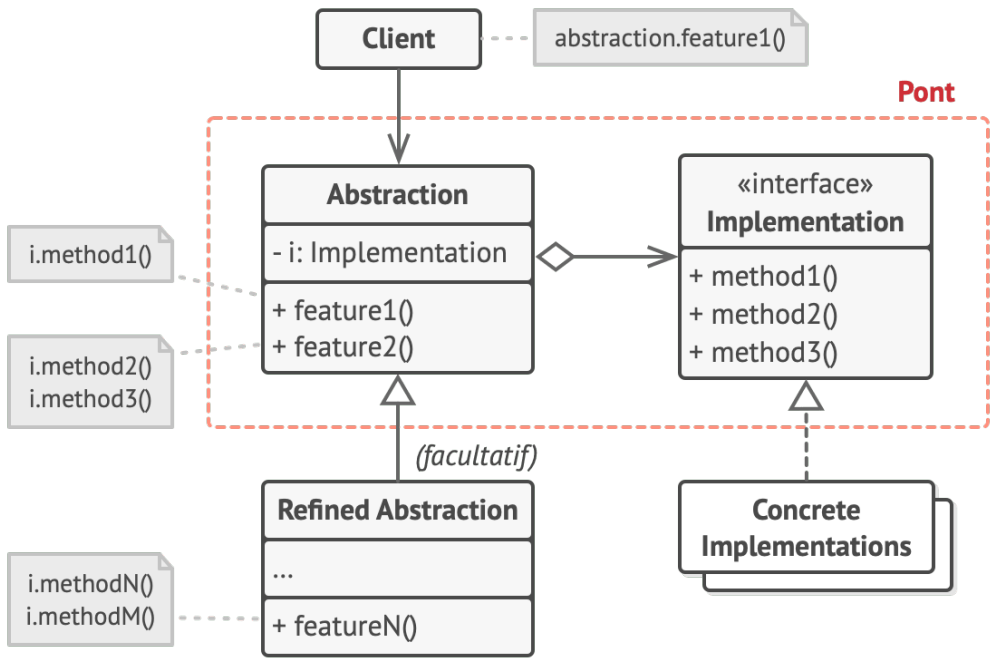


Les **Implémentations Concrètes** contiennent du code spécialisé pour les plate-formes.



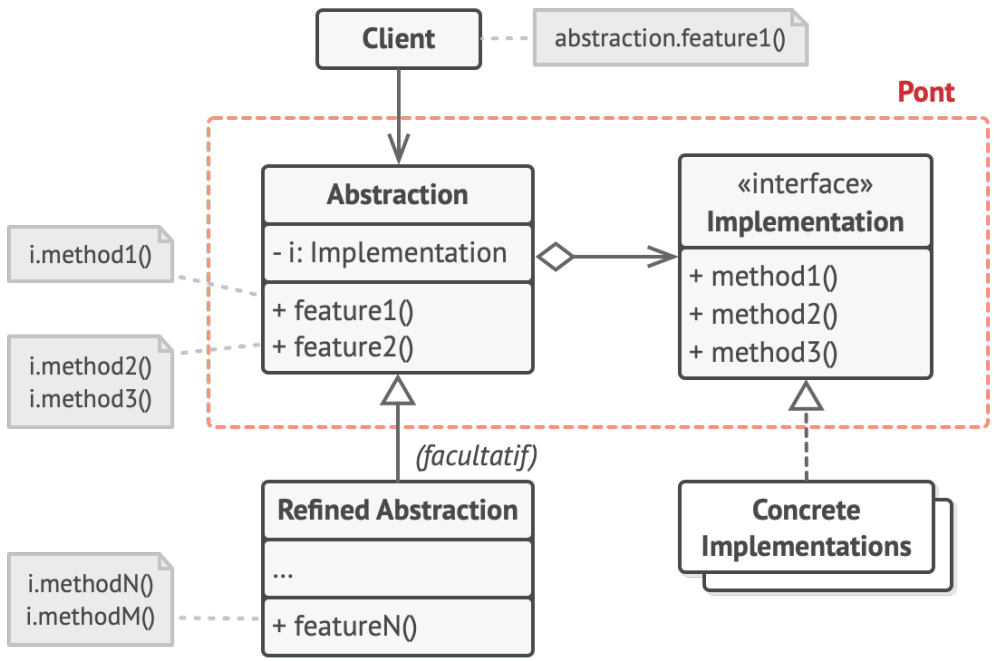
Structure

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.



Structure

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.



Applications du pont

☞ Quand on veut 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).

Applications du pont

- Quand on veut 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).
- Si on veut étendre une classe dans plusieurs dimensions orthogonales (indépendantes).

Applications du pont

- Quand on veut 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).
- Si on veut étendre une classe dans plusieurs dimensions orthogonales (indépendantes).
- Si on veut pouvoir changer d'implémentation dès le lancement de l'application.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser dans la classe d'abstraction de base.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser dans la classe d'abstraction de base.
3) Établissez la liste des opérations disponibles sur toutes les plateformes.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser 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.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser 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.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser 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.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser 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.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser 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.

Mise en oeuvre

1) Identifiez les dimensions orthogonales de vos classes.
   - abstraction/plateforme, domaine/infrastructure, front-end/back-end, interface/implémentation.
2) Définissez les opérations que le client veut utiliser 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

Avantages

☞ Vous pouvez créer des classes et des applications multiplateformes.

Inconvénients

Avantages et inconvénients

Avantages

- Vous pouvez créer des classes et des applications multiplateformes.
- Le code client manipule des abstractions de haut niveau. Il n'est pas dépendant des détails de la plateforme.

Inconvénients

Avantages et inconvénients

Avantages

- Vous pouvez créer des classes et des applications multiplateformes.
- 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.

Inconvénients

Avantages et inconvénients

Avantages

- Vous pouvez créer des classes et des applications multiplateformes.
- 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.

Inconvénients

Avantages et inconvénients

Avantages

- Vous pouvez créer des classes et des applications multiplateformes.
- 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.

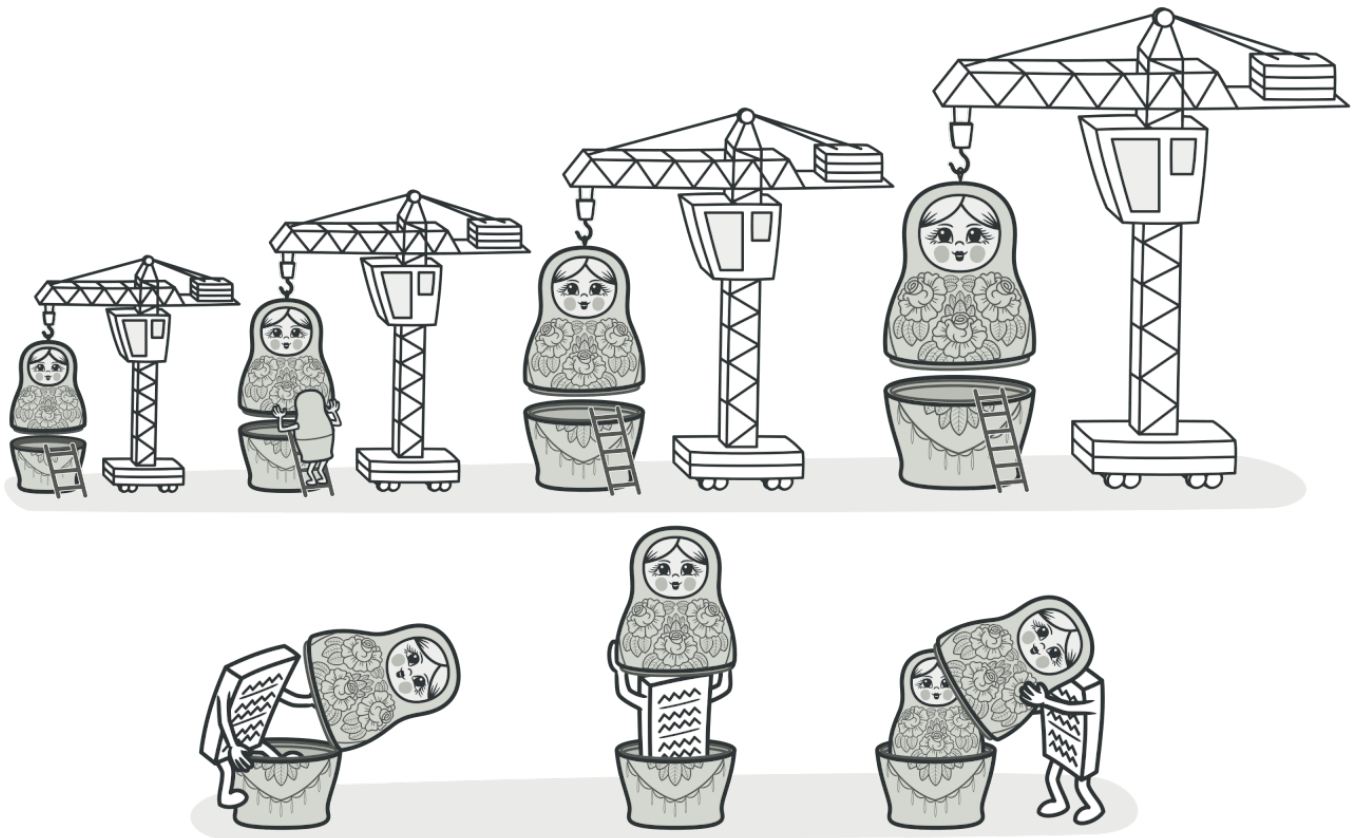
Inconvénients

- Le code va devenir plus compliqué si vous introduisez ce patron dans une classe très cohésive.

Patrons de conception
Plan

Intention

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

☞ On travaille sur une librairie qui permet aux programmes d'envoyer des notifications à leurs utilisateurs lorsque des événements importants se produisent.

Problème

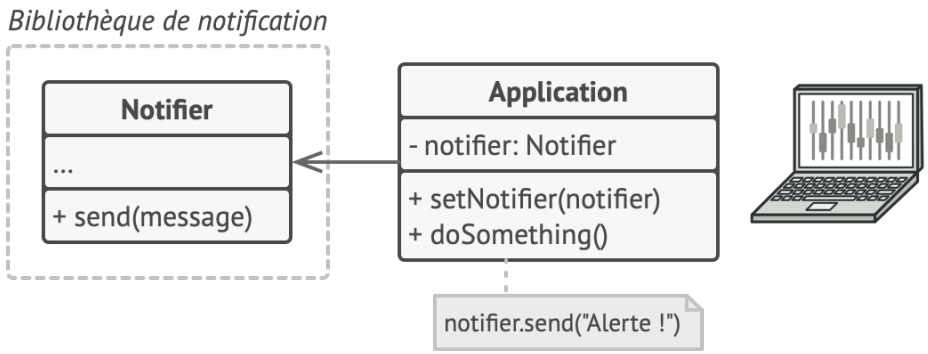
- On travaille sur une librairie qui permet aux programmes d'envoyer des notifications à leurs utilisateurs lorsque des événements importants se produisent.
- La version initiale de la librairie était basée sur la classe Notificateur qui n'avait que quelques attributs, un constructeur et une unique méthode envoyer.

Problème

- ☞ On travaille 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.

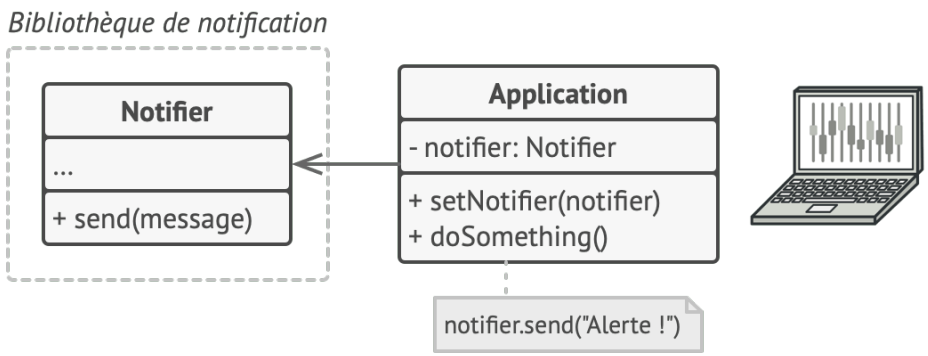
Problème

- On travaille 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.



Problème

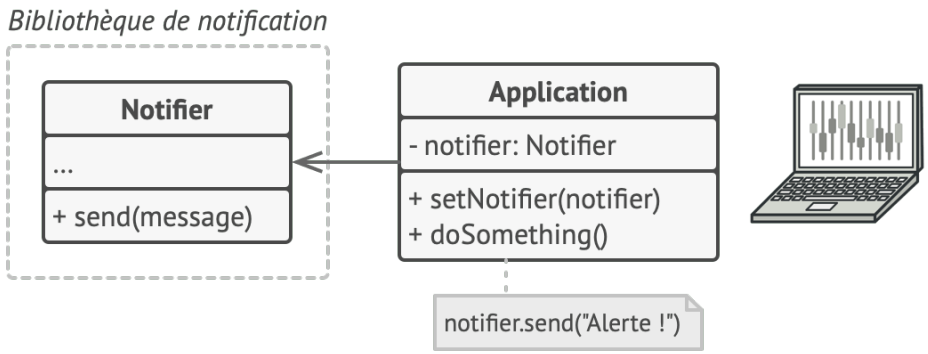
- On travaille sur une librairie qui permet aux programmes d'envoyer des notifications à leurs utilisateurs lorsque des événements importants se produisent.
- La version initiale de la librairie était basée sur la classe Notificateur qui n'avait que quelques attributs, un constructeur et une unique méthode 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.



- Au bout d'un certain temps, on se rend compte que les utilisateurs de la librairie veulent plus que les notifications qu'ils reçoivent sur leur boîte mail.

Problème

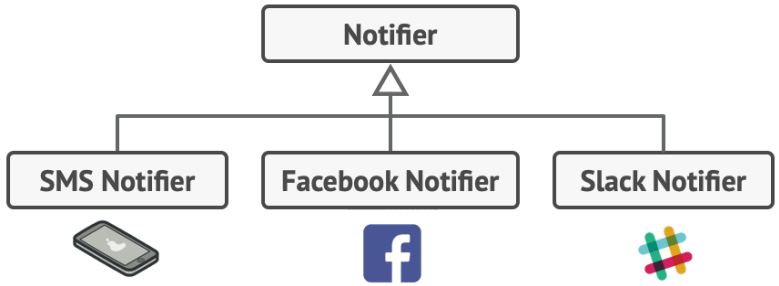
- On travaille sur une librairie qui permet aux programmes d'envoyer des notifications à leurs utilisateurs lorsque des événements importants se produisent.
- La version initiale de la librairie était basée sur la classe Notificateur qui n'avait que quelques attributs, un constructeur et une unique méthode 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.



- Au bout d'un certain temps, on se rend compte que les utilisateurs de la librairie veulent plus que les notifications qu'ils reçoivent sur leur boîte mail.
- On aimerait pouvoir gérer par exemple les SMS, ou les notifcation de Facebook ou de Slack.

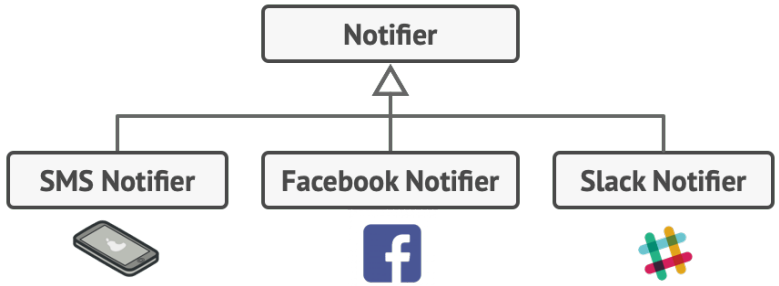
Suite du problème

☞ Solution naturelle :



Suite du problème

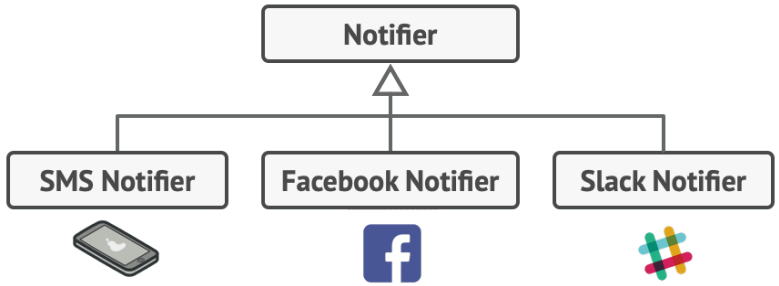
☞ Solution naturelle :



☞ Problème : si ma maison est en feu, je veux recevoir une notification à ce sujet sur TOUS les canaux disponibles.

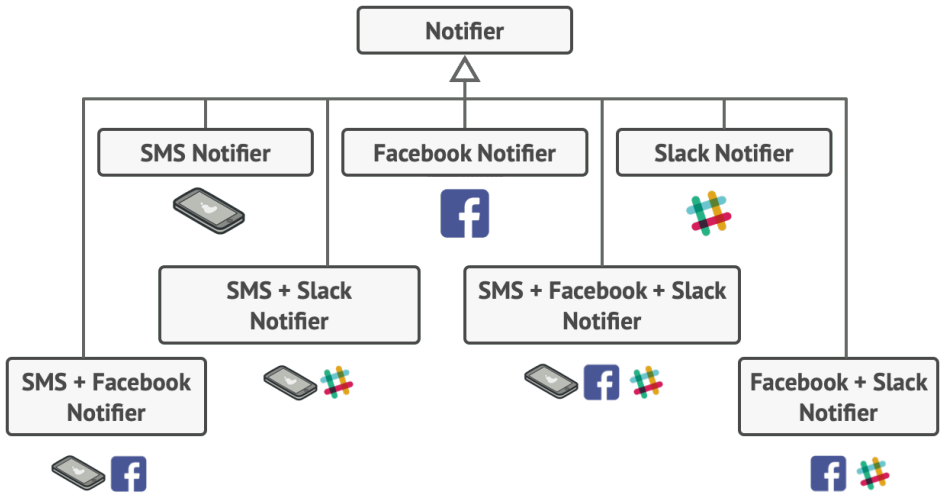
Suite du problème

☞ Solution naturelle :



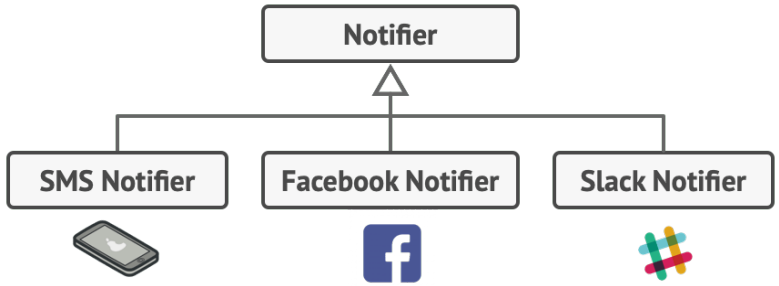
☞ Problème : si ma maison est en feu, je veux recevoir une notification à ce sujet sur TOUS les canaux disponibles.

☞ Re-solution :

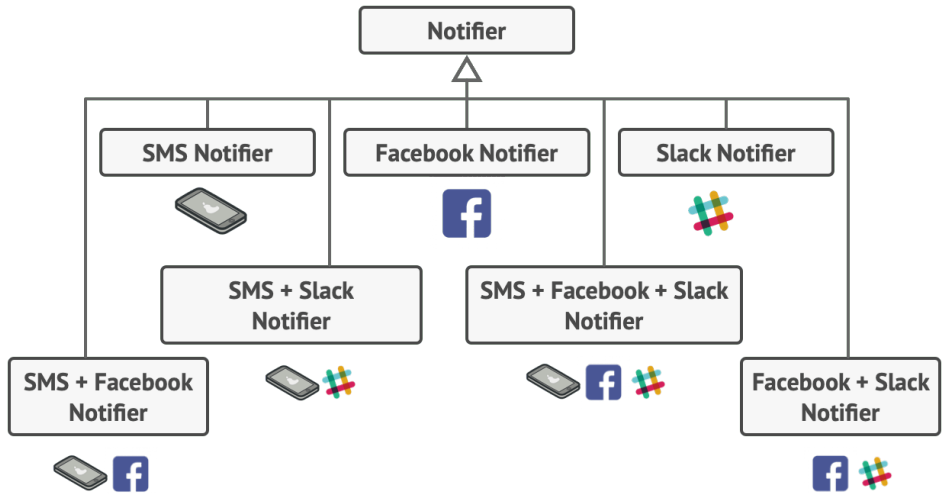


Suite du problème

- Solution naturelle :

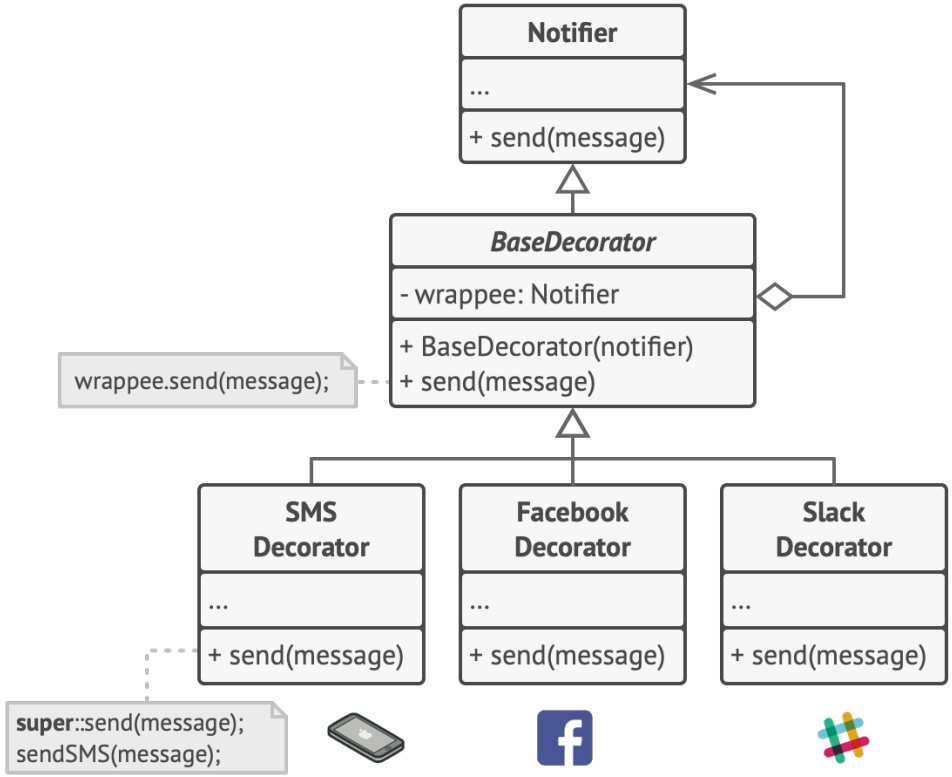


- Problème : si ma maison est en feu, je veux recevoir une notification à ce sujet sur TOUS les canaux disponibles.
- Re-solution :



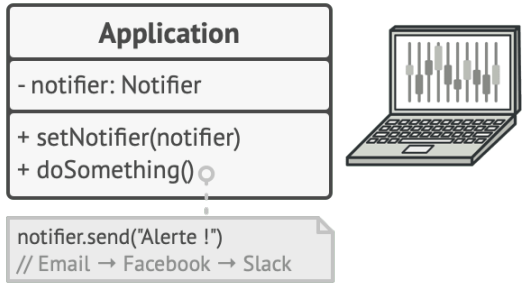
- Re-problème : explosion combinatoire !

Solution

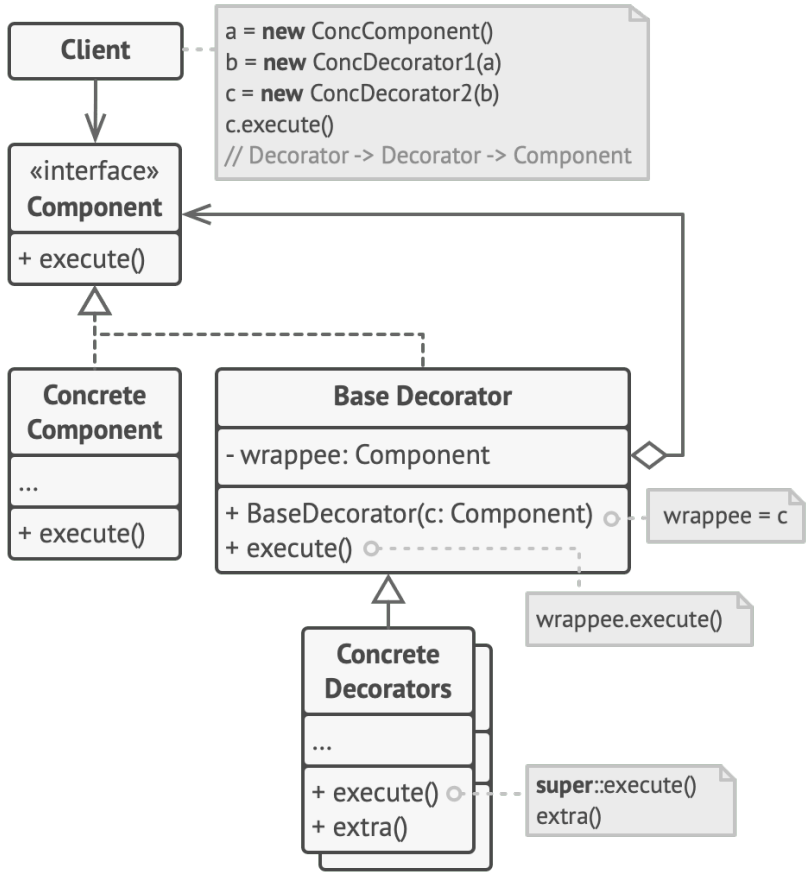



```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```

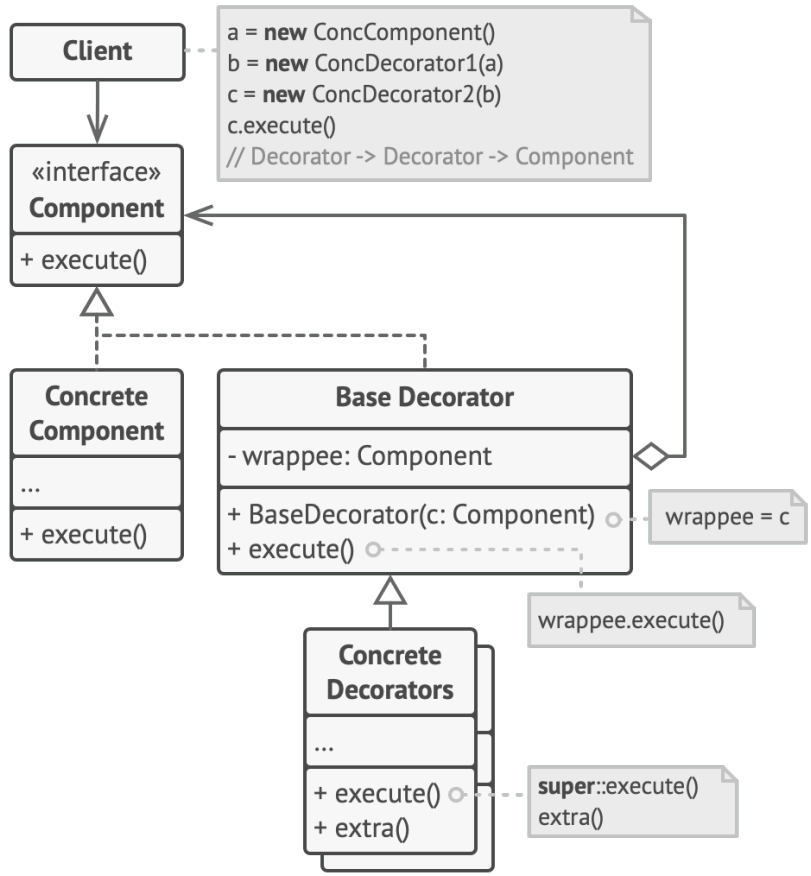


Structure du décorateur



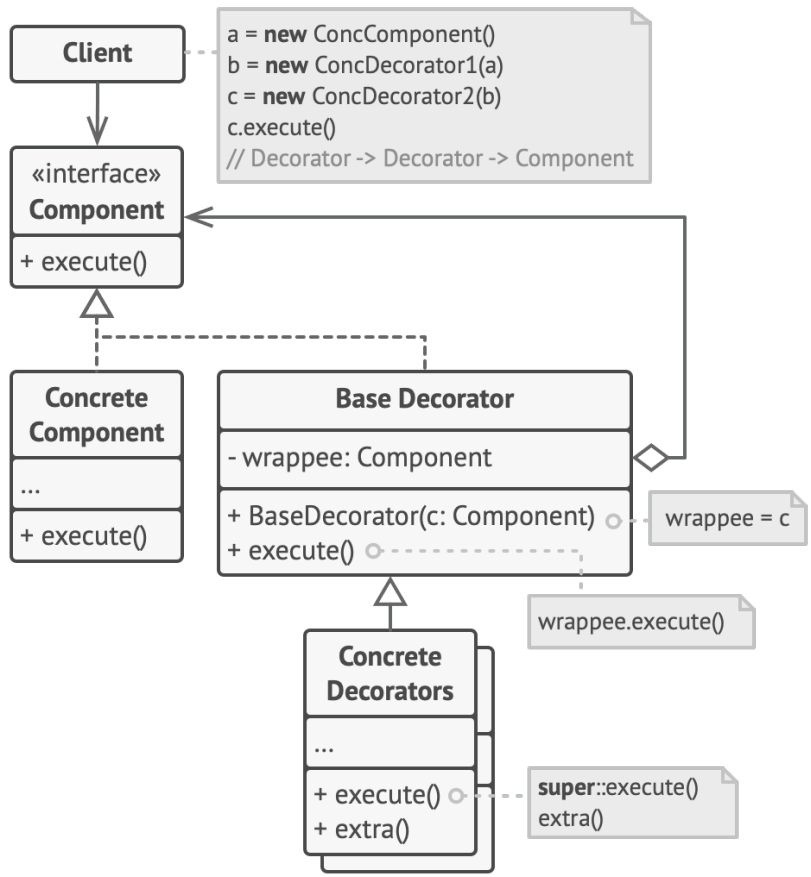
Structure du décorateur

Le Composant déclare l'interface commune aux décorateurs et aux objets décorés.



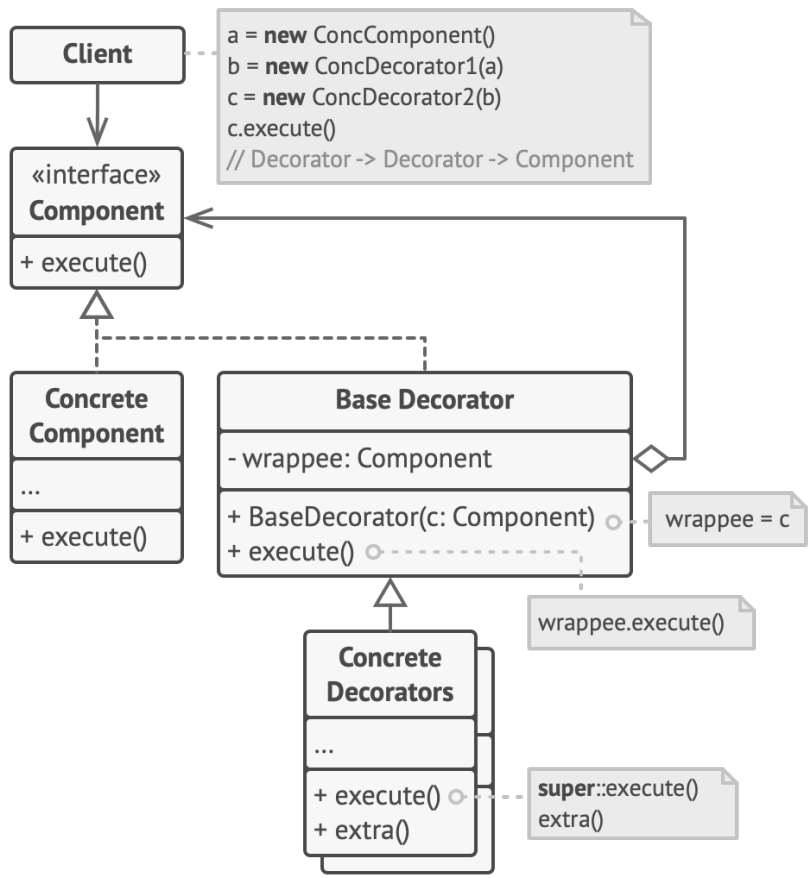
Structure du décorateur

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.



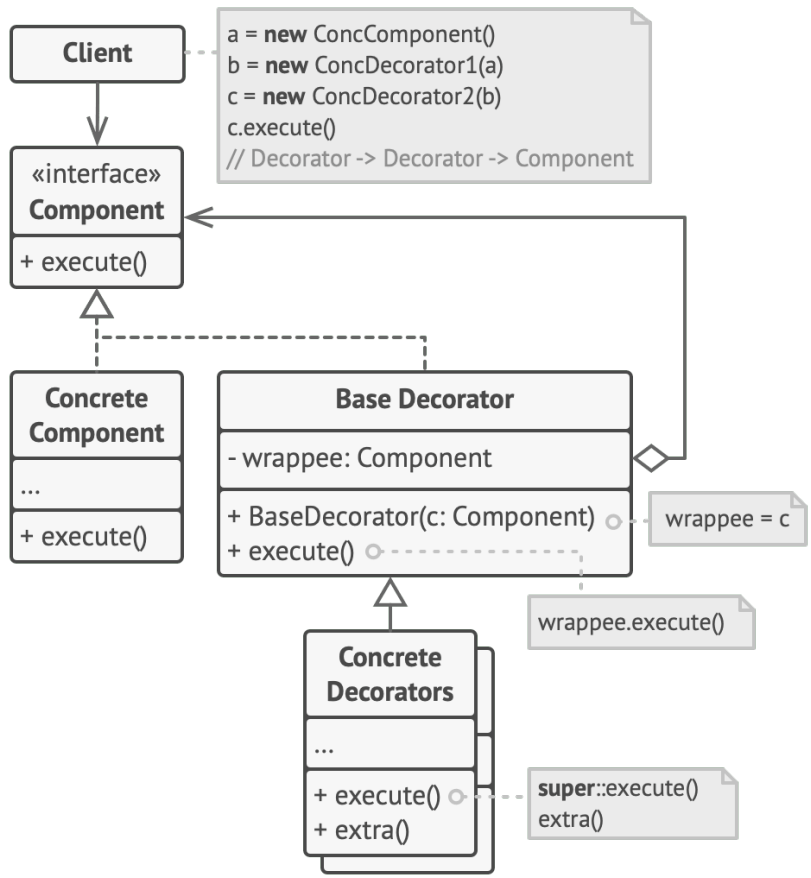
Structure du décorateur

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.



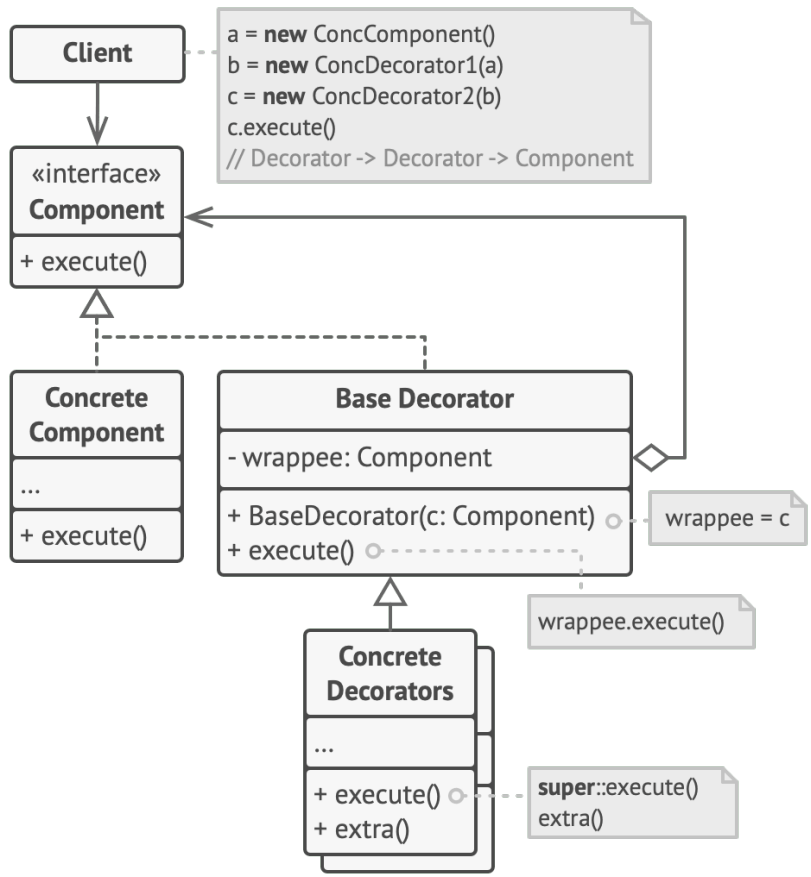
Structure du décorateur

Les Décorateurs Concrets définissent des comportements supplémentaires que l'on peut ajouter aux composants dynamiquement.
Les décorateurs concrets redéfinissent les méthodes du décorateur de base et exécutent leur traitement avant ou après l'appel à la méthode du parent.



Structure du décorateur

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.



Avantages et inconvénients

Avantages

Vous pouvez étendre le comportement d'un objet sans avoir recours à la création d'une nouvelle sous-classe.

Inconvénients

Avantages et inconvénients

Avantages

- 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.

Inconvénients

Avantages et inconvénients

Avantages

- 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.

Inconvénients

Avantages et inconvénients

Avantages

- 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.

Inconvénients

Avantages et inconvénients

Avantages

- 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.

Inconvénients

- Retirer un emballeur spécifique de la pile n'est pas chose aisée.

Avantages et inconvénients

Avantages

- 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.

Inconvénients

- 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.

Avantages et inconvénients

Avantages

- 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.

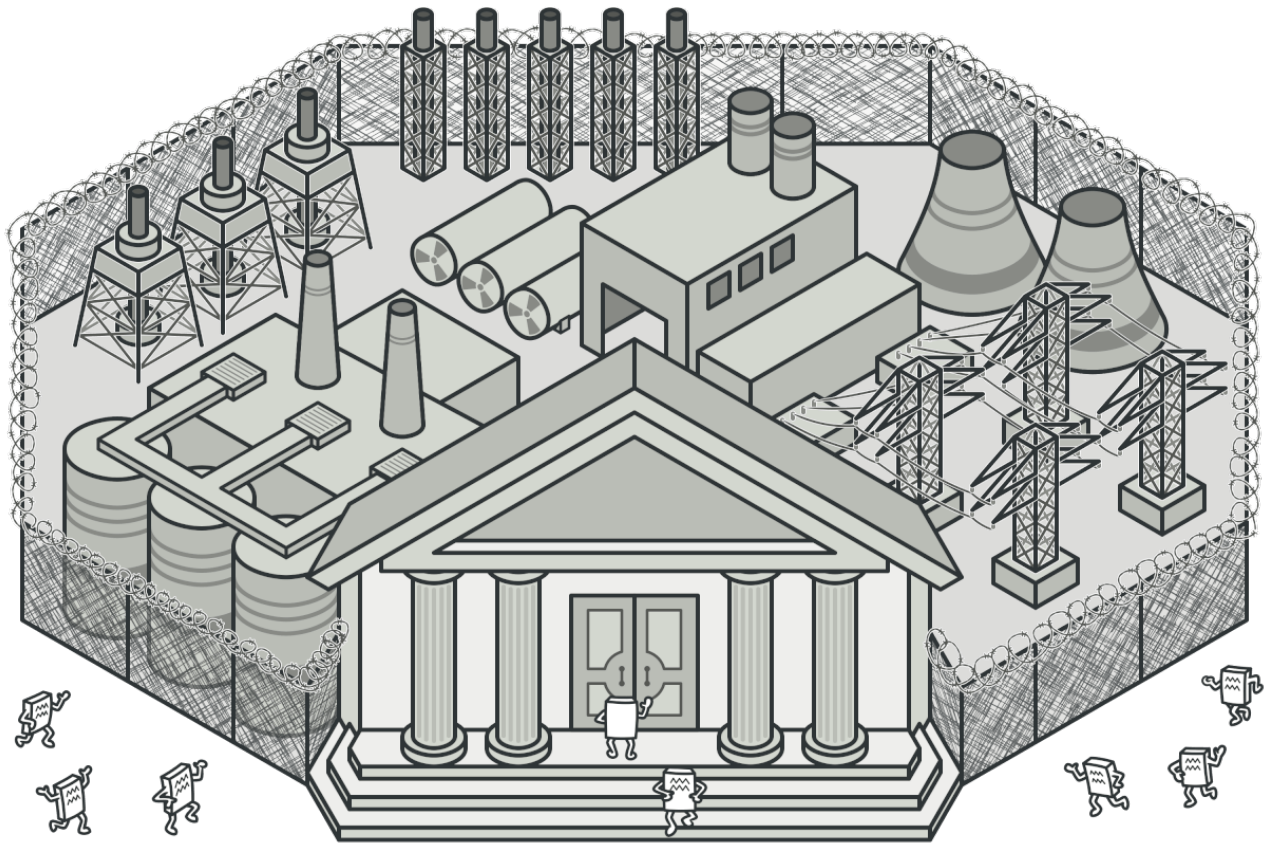
Inconvénients

- 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.

Patrons de conception
Plan

Intention

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.

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.

Problème

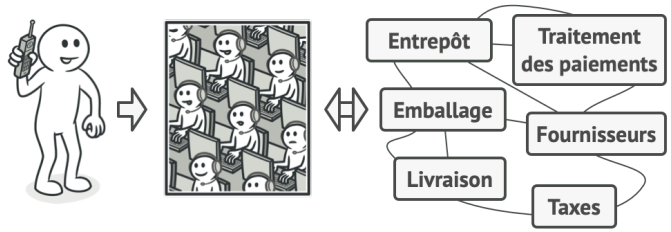
- Imaginez que vous devez adapter votre code pour manipuler un ensemble d'objets qui appartiennent à une librairie ou à un framework assez sophistiqué.
- D'ordinaire, vous initialisez tous ces objets en premier, gardez la trace des dépendances et appelez les méthodes dans le bon ordre, etc.
- Par conséquent, la logique métier de vos classes devient fortement couplée avec les détails de l'implémentation des classes externes.

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.
- Cette logique devient difficile à comprendre et à maintenir.

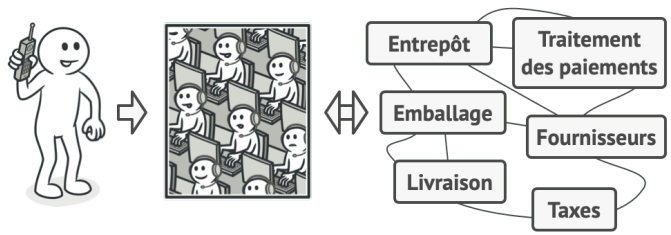
Solution

☞ Une façade est une classe qui procure une interface simple vers un sous-système complexe de parties mobiles.



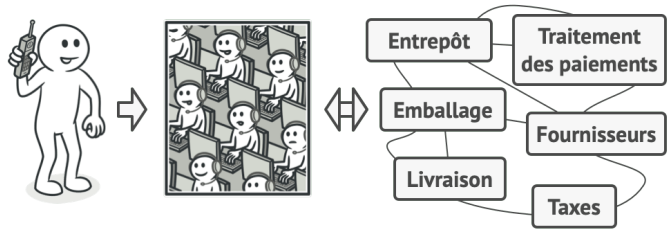
Solution

- 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.



Solution

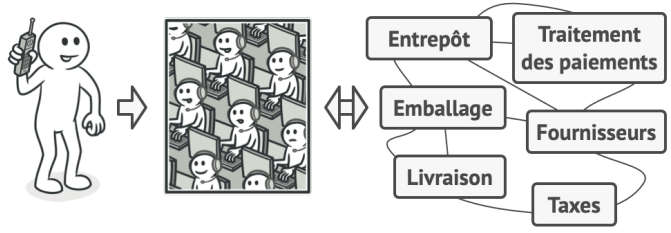
- Une façade est une classe qui procure une interface simple vers un sous-système complexe de parties mobiles.
- Les fonctionnalités proposées par la façade seront plus limitées que si vous interagissiez directement avec le sous-système, mais vous pouvez vous contenter de n'inclure que les fonctionnalités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



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.

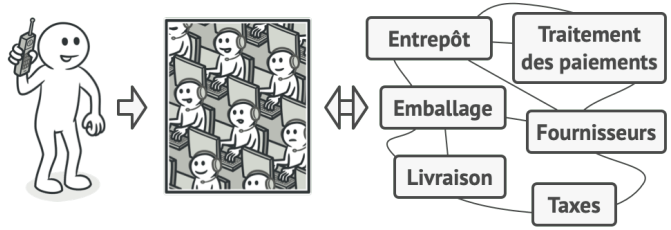
- Par exemple, une application qui envoie des petites vidéos de chats comiques sur des réseaux sociaux peut potentiellement utiliser une librairie de conversion vidéo professionnelle.



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.

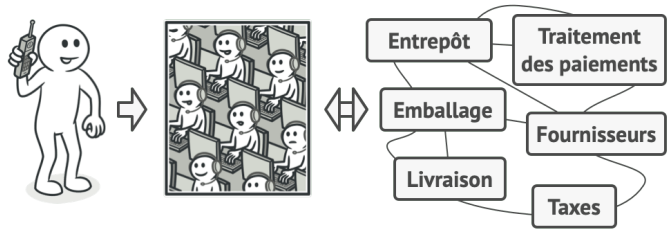
- Par exemple, une application qui envoie des petites vidéos de chats comiques sur des réseaux sociaux peut potentiellement utiliser une librairie de conversion vidéo professionnelle.
- Mais la seule chose dont vous ayez réellement besoin, c'est d'une classe dotée d'une méthode encoder(fichier, format).



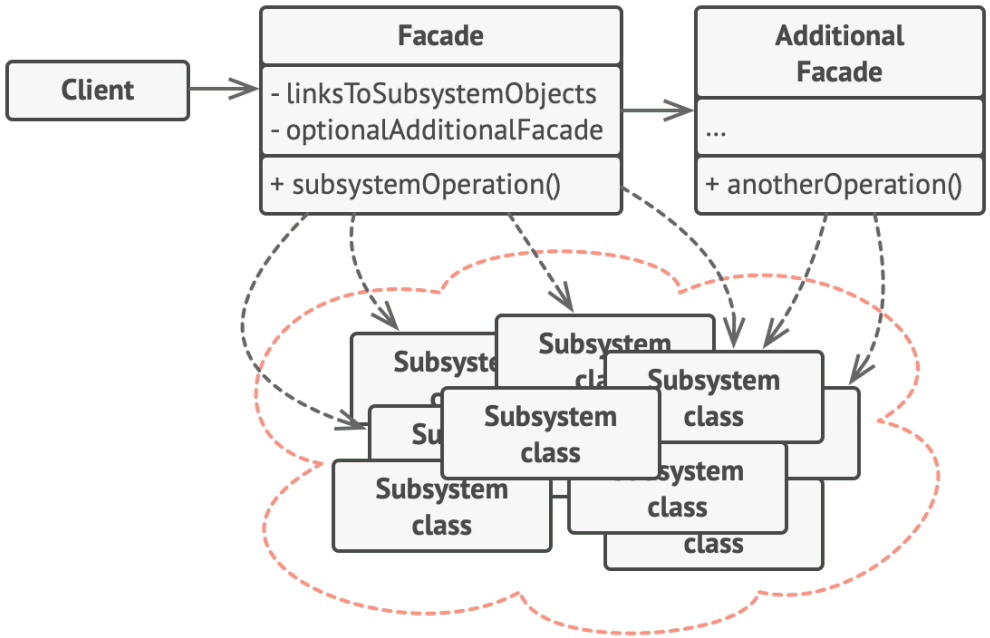
Solution

- 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.

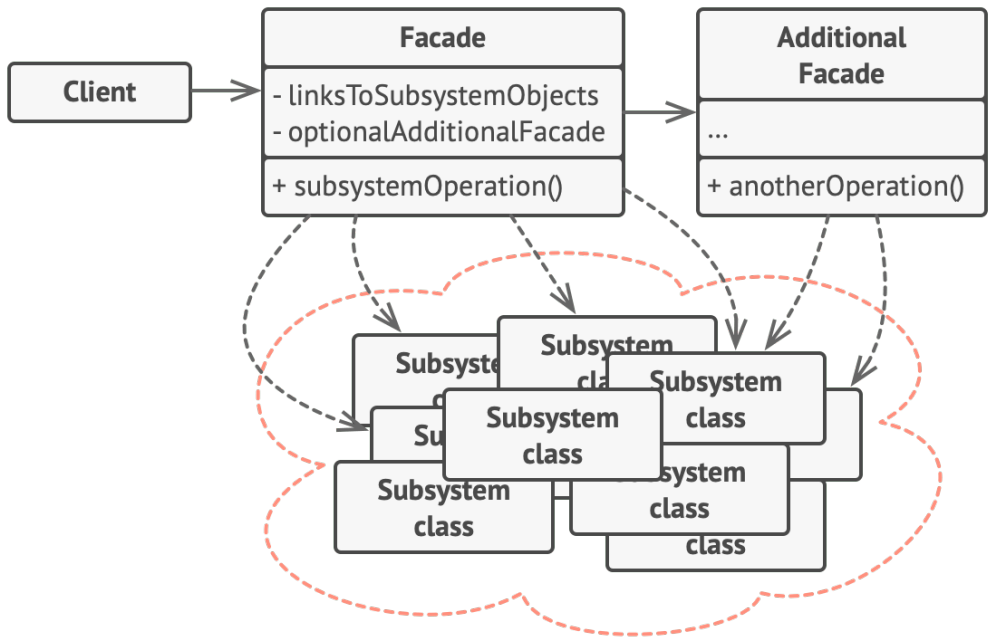


Structure


Client
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional Facade
...
+ anotherOperation()
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class

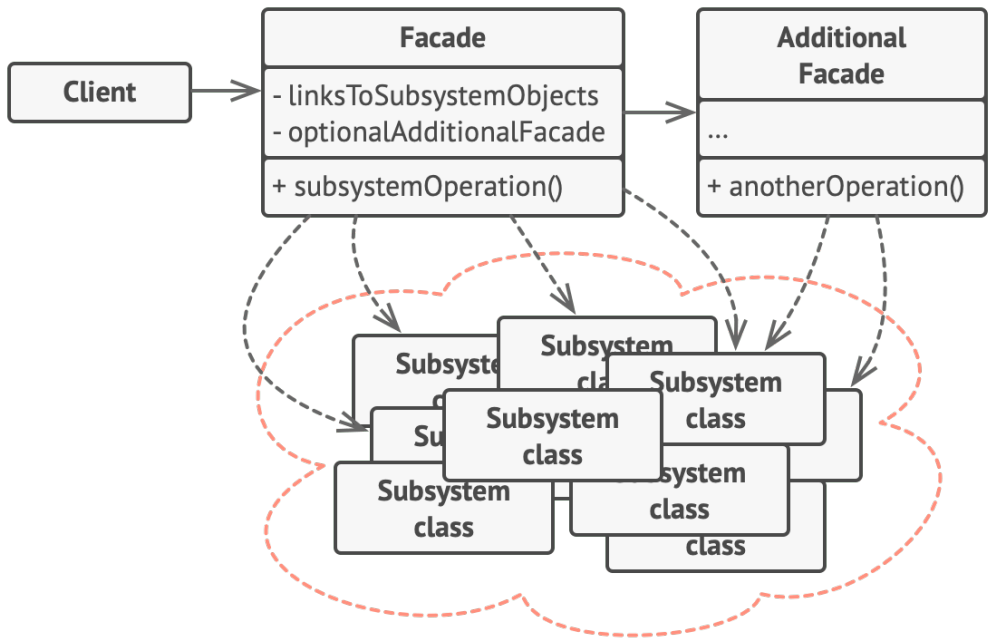
Structure

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.



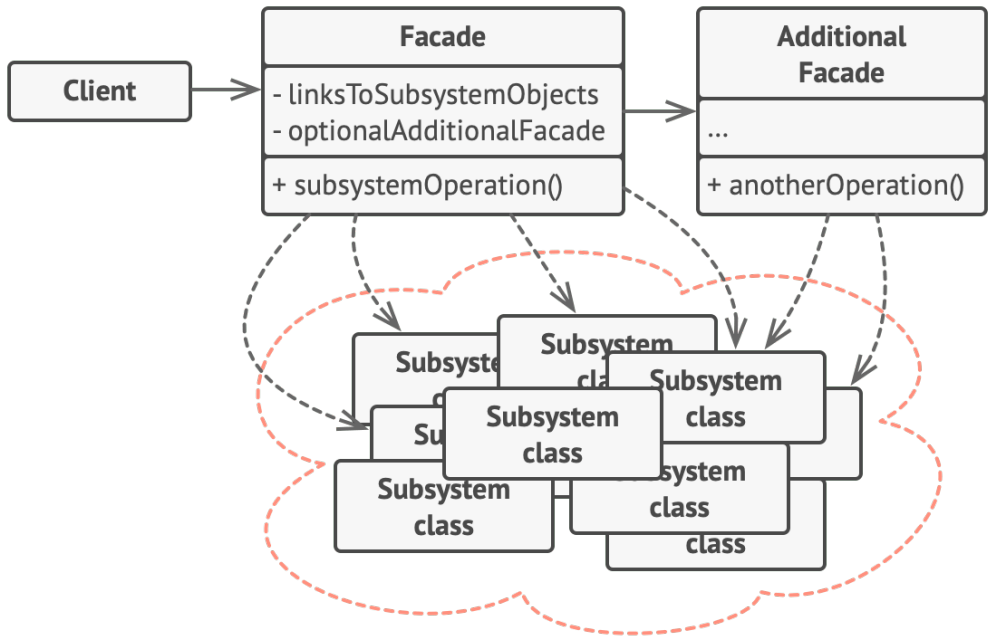
Structure

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.



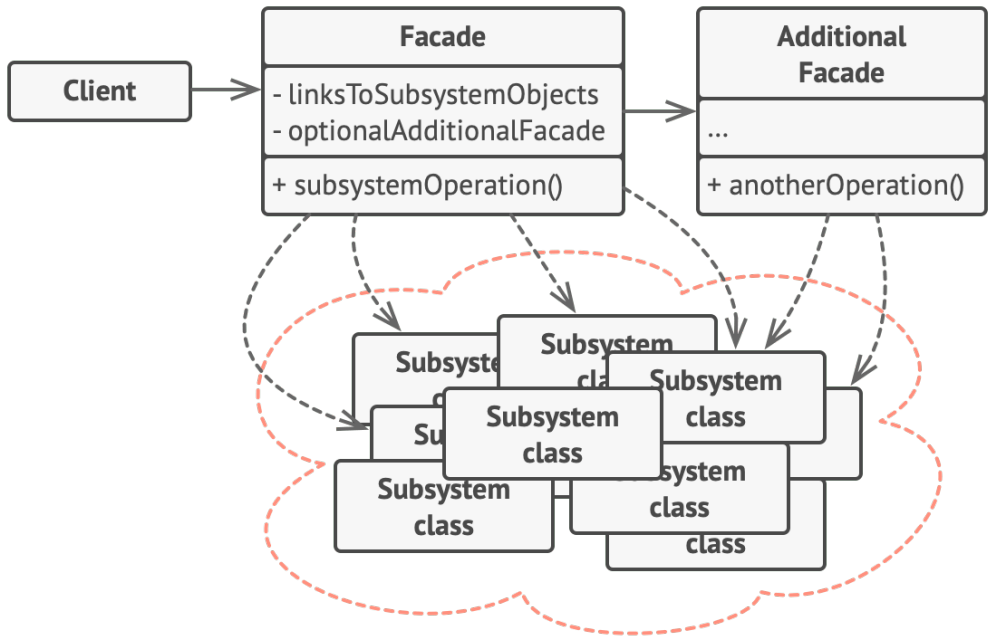
Structure

Le Sous-système Complexe 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.



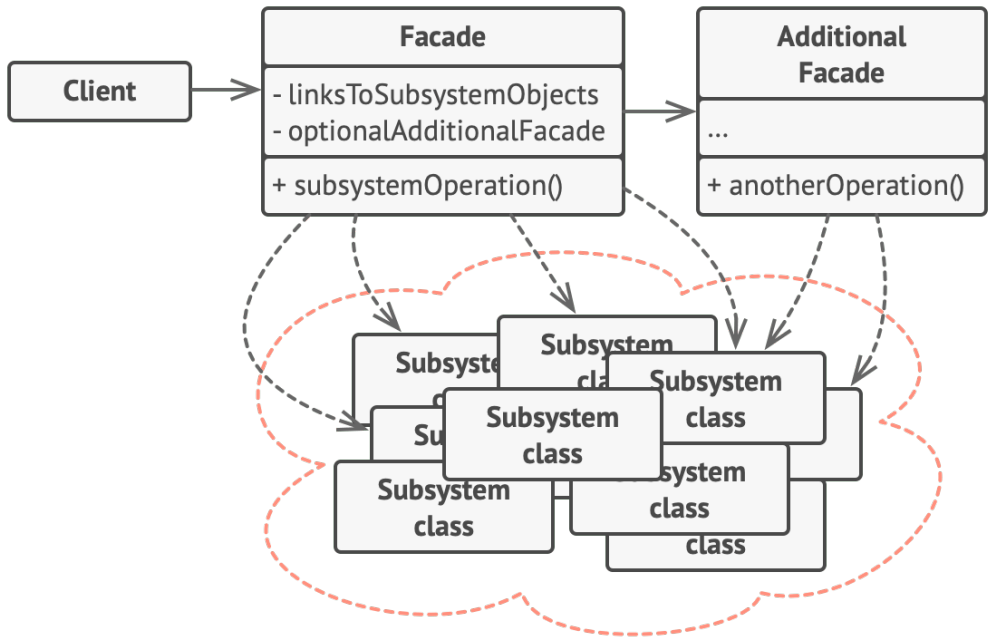
Structure

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.



Structure

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



Utiliser la façade

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

Utiliser la façade

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

Utiliser la façade

- Utilisez la façade si vous avez besoin d'une interface limitée mais directe à un sous-système complexe.
- Utilisez la façade si vous voulez structurer un sous-système en plusieurs couches.
- **Avantage :** vous pouvez isoler votre code de la complexité d'un sous-système.

Utiliser la façade

- Utilisez la façade si vous avez besoin d'une interface limitée mais directe à un sous-système complexe.
- Utilisez la façade si vous voulez structurer un sous-système en plusieurs couches.
- **Avantage :** vous pouvez isoler votre code de la complexité d'un sous-système.
- **Inconvénient :** une façade peut devenir un objet omniscient couplé à toutes les classes d'une application.

Patrons de conception
Plan

1. Patrons structurels
   1.1. Composite
   1.2. Procuration
   1.3. Autres patrons structurels

2. Patrons de création
   2.1. Fabrique
   2.2. Singleton

3. Patrons comportementaux
   3.1. Patron de méthode
   3.2. Stratégie
   3.3. État

Patrons de conception
Plan

Patrons de conception
Plan

Fabrique

Le patron Fabrique (Factory) permet d'isoler les créations d'objets.

- Par exemple, un client doit pour créer une Forme appeler :
  `Forme f = new Carre(10, 12, 20)`
- Le problème est qu'il doit donc connaître Carré, Cercle,...

- Évolution difficile : impossible de supprimer Carré pour mettre Rectangle.
- Comment gérer le passage à un Composite ?

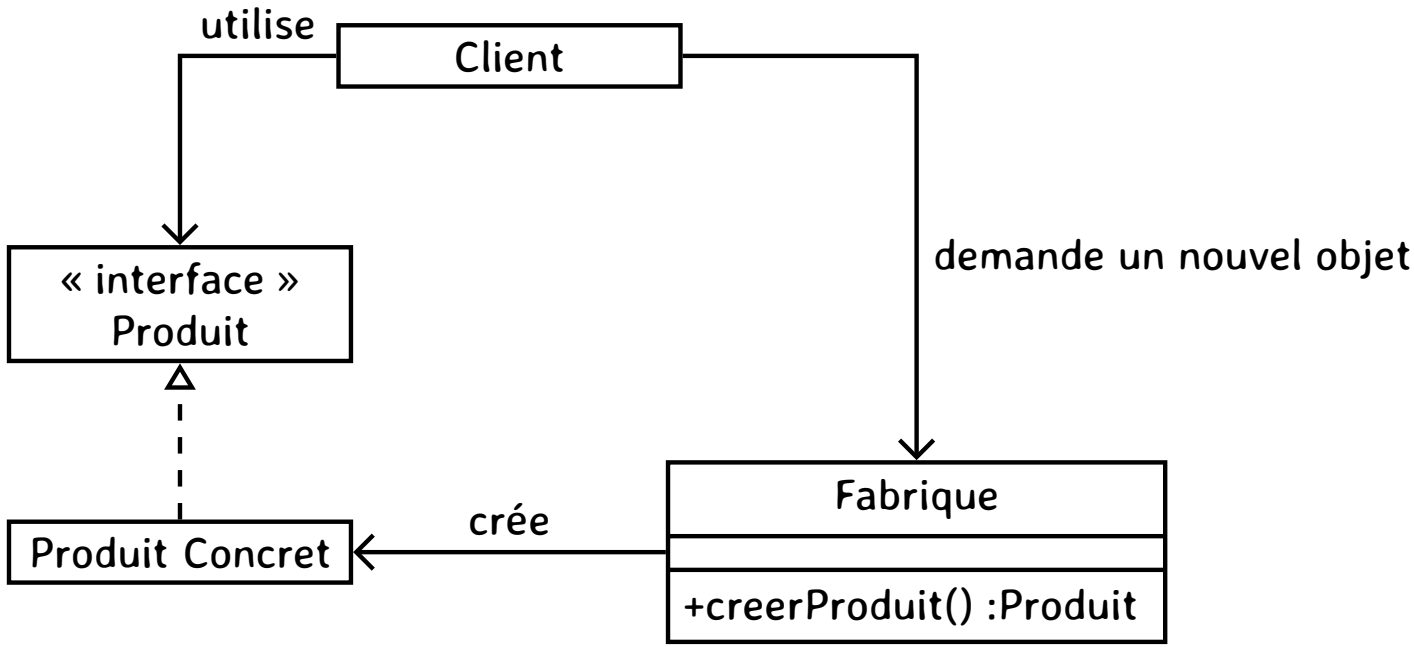
- Donc le client dépend des classes concrètes car il est forcé d'appeler leur construction par new.
- On peut retirer ces dépendances, en codant une classe qui se charge de construire les objets.

Exemple : formes géométriques

```java
public class FormeFactory {
  public static Forme createCarreCercleConcentrique
    public static Forme createCarreCercleConcentrique (int x, int y, int width) {
    Dessin d = new Dessin();
    d.add(new Carre(x,y,width));
    d.add(new Cercle(x+width/2, y+width/2, width/2));
    return d;
  }
  public static Forme createCarre (int x, int y, int width) {
    return new Carre(x,y,width);
  }
  ...
}
```

- Le client avant : `Forme f = new Carre(10, 12, 20)`
- À présent on a : `Forme f = FormeFactory.createCarre(10, 12, 20)`
- Faire évoluer l'implémentation sans modifier le client.
  - Par exemple retirer la classe Carré pour un Rectangle
  - Idem pour les classes composites etc...
- Bien sûr il faut mettre à jour le code de la Factory ...
- Mais on maîtrise la portée des modifications.

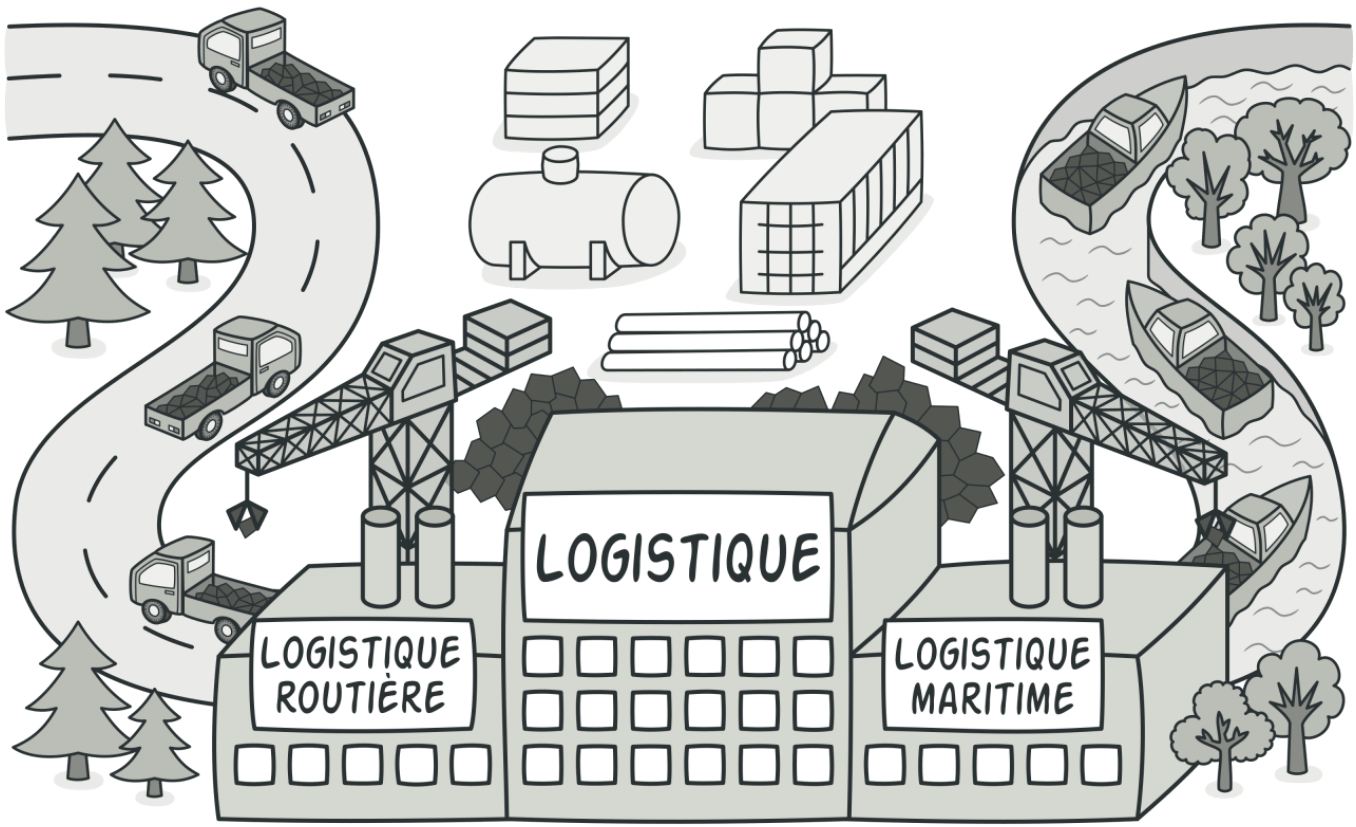
Structure de la Fabrique Simple



Patrons de conception
Plan

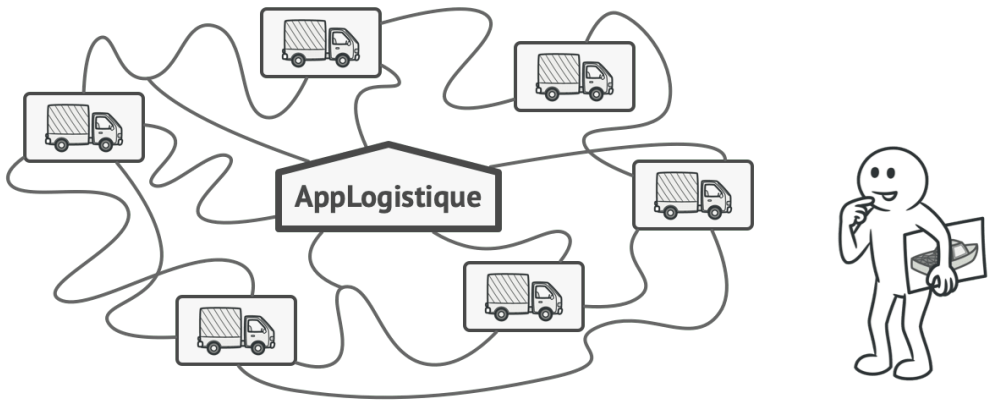
Intention

Fabrique de méthode 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

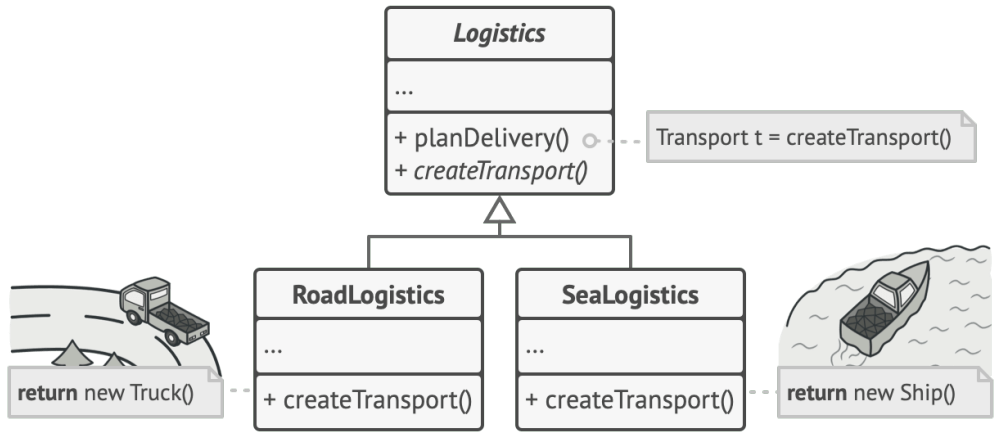
- Une application de gestion logistique.
- Première version : uniquement transport par camion.
- Essentiel du code dans la classe `Camion`.



- Au bout d'un certain temps, on veut ajouter la gestion de la logistique maritime.
- Comment faire ça dans le code ?
- La majeure partie est actuellement couplée à la classe `Camion`.
- Pour pouvoir ajouter des `Bateaux`, il faudrait revoir la base du code.
- Si on décide plus tard d'ajouter un autre type de transport, il faudra recommencer.
- Résultat : du code pas très propre, rempli de conditions qui modifient le comportement du programme en fonction de la classe des objets de transport.

Solution

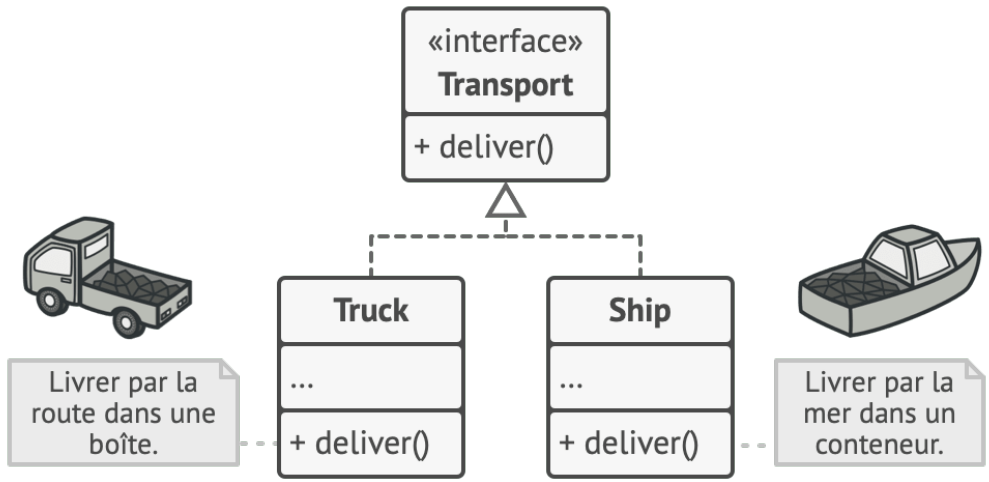
- Le patron de conception fabrique de méthodes propose :
  - remplacer les appels directs au constructeur de l'objet (avec new) en appelant une méthode fabrique spéciale.
  - les objets sont toujours créés avec l'opérateur new, mais à l'intérieur de la méthode fabrique.



- À première vue, cette modification peut sembler inutile :
  nous avons juste déplacé l'appel du constructeur dans une autre partie du programme.
- Mais maintenant, on peut redéfinir la méthode fabrique dans la sous-classe et changer la classe des produits créés par la méthode.

Solution

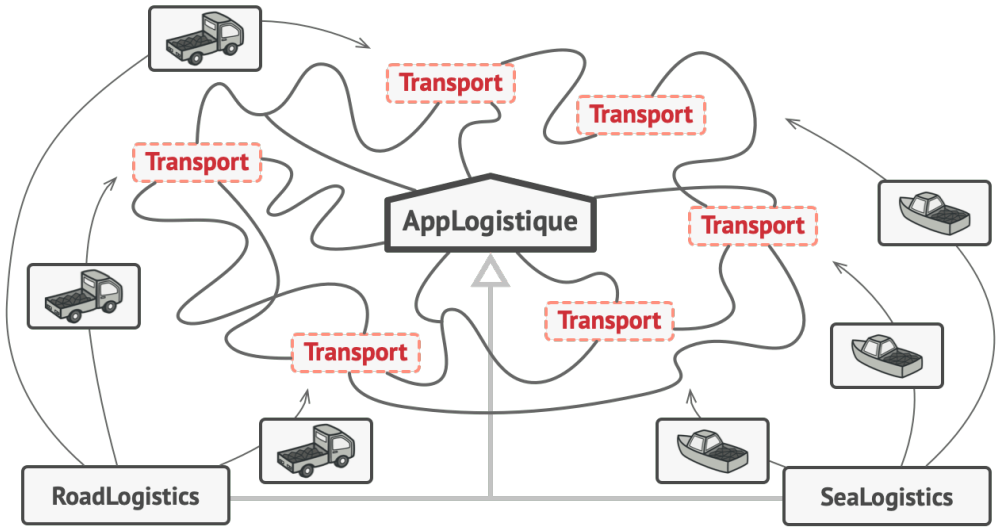
- Il y a tout de même une petite limitation :
  - les sous-classes peuvent retourner des produits différents seulement si les produits ont une classe de base ou une interface commune ;
  - cette interface doit être le type retourné par la méthode fabrique de la classe de base.



- 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.

Solution

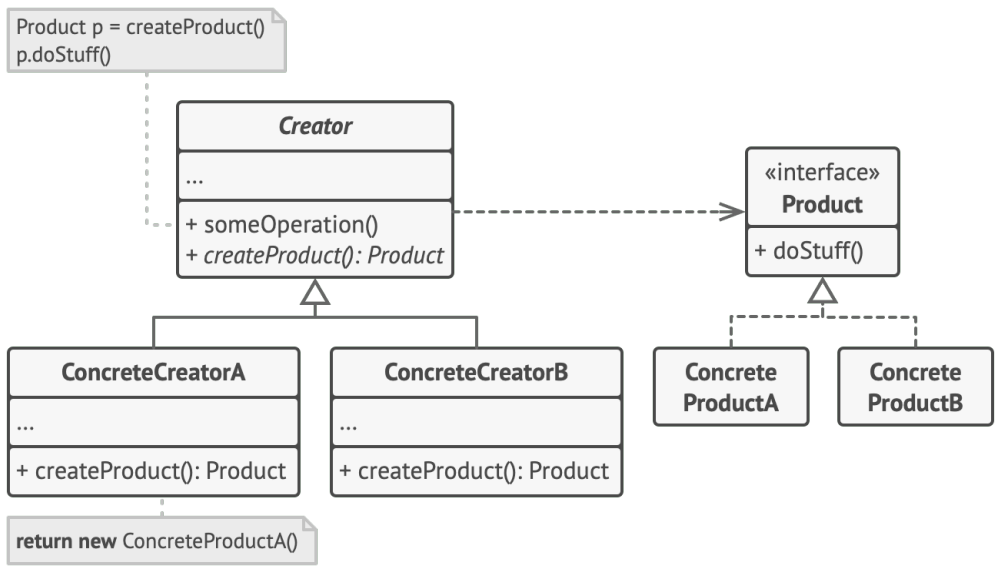
- La méthode fabrique de la classe `LogistiqueRoute` retourne des camions, alors que celle de la classe `LogistiqueMer` retourne des bateaux.



- Le code qui appelle la méthode fabrique (souvent appelé le code client) ne fait pas la distinction entre les différents produits concrets.
- 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.

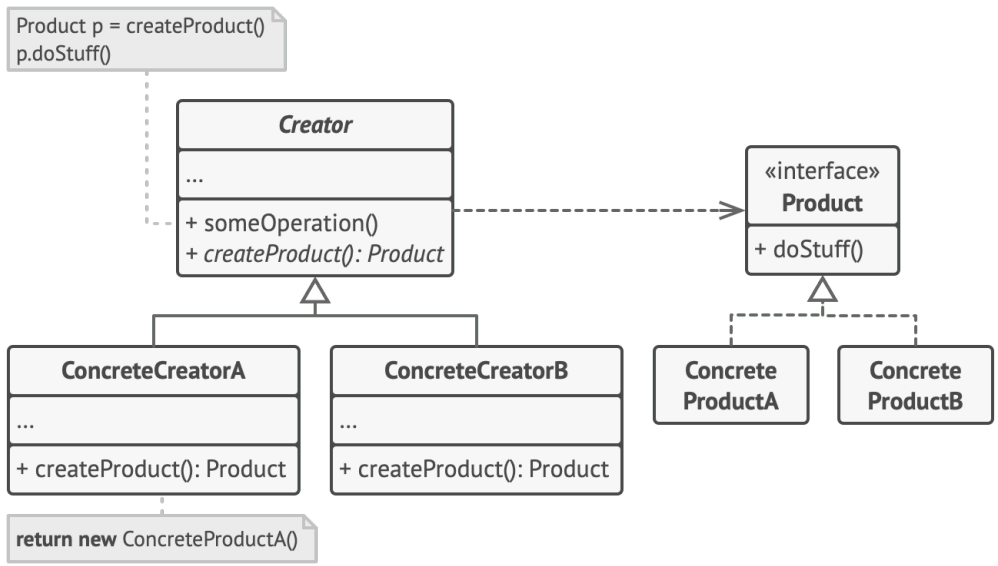
Structure

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.



Structure

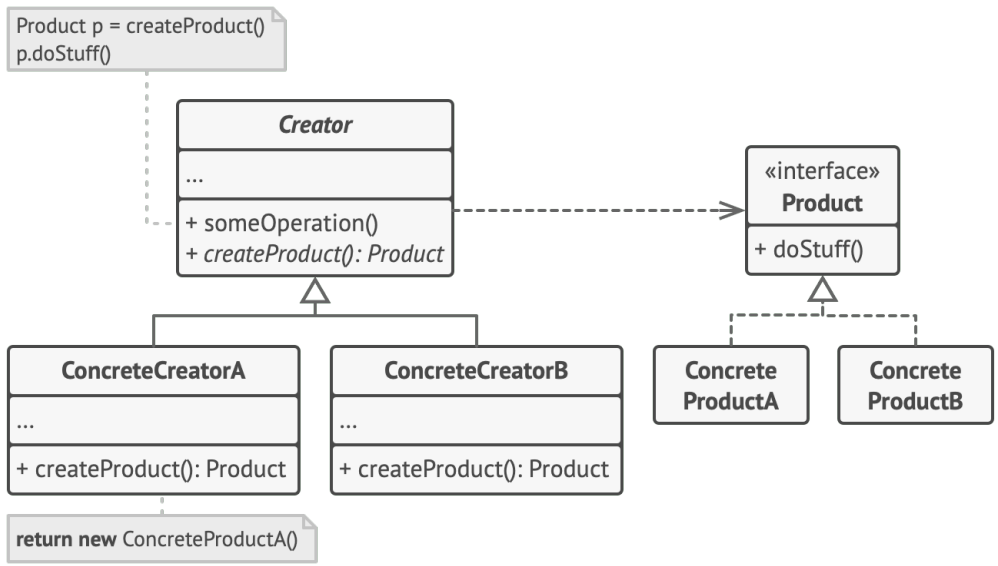
Les **Produits Concrets** sont différentes implémentations de l'interface produit.



Structure

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. On peut :

- rendre la méthode fabrique abstraite, pour obliger les sous-classes à implémenter leur propre version
- ou modifier la méthode fabrique de la classe de base afin qu'elle retourne un produit par défaut.



Structure

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. On peut :

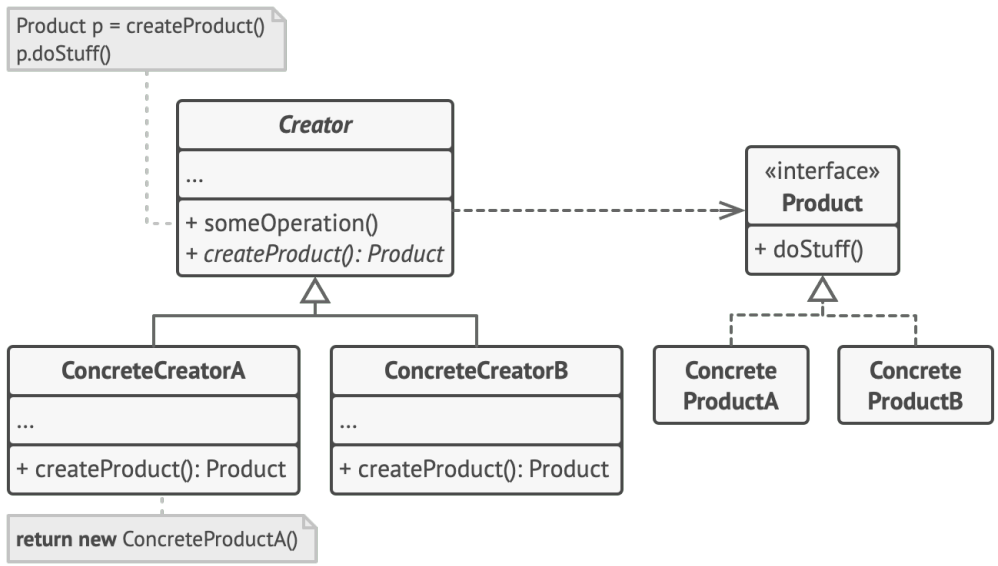
- rendre la méthode fabrique abstraite, pour obliger les sous-classes à implémenter leur propre version
- ou modifier la méthode fabrique de la classe de base afin qu'elle retourne un produit par défaut.

Attention

- Malgré son nom, la création de produits n'est pas la responsabilité principale du créateur.
- Il 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 de formation pour ses développeurs,
  - mais son activité principale reste d'écrire du code,
  - pas de produire des développeurs.

Structure

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.



Application de la fabrique de méthodes

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.

Application de la fabrique de méthodes

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 !

Application de la fabrique de méthodes

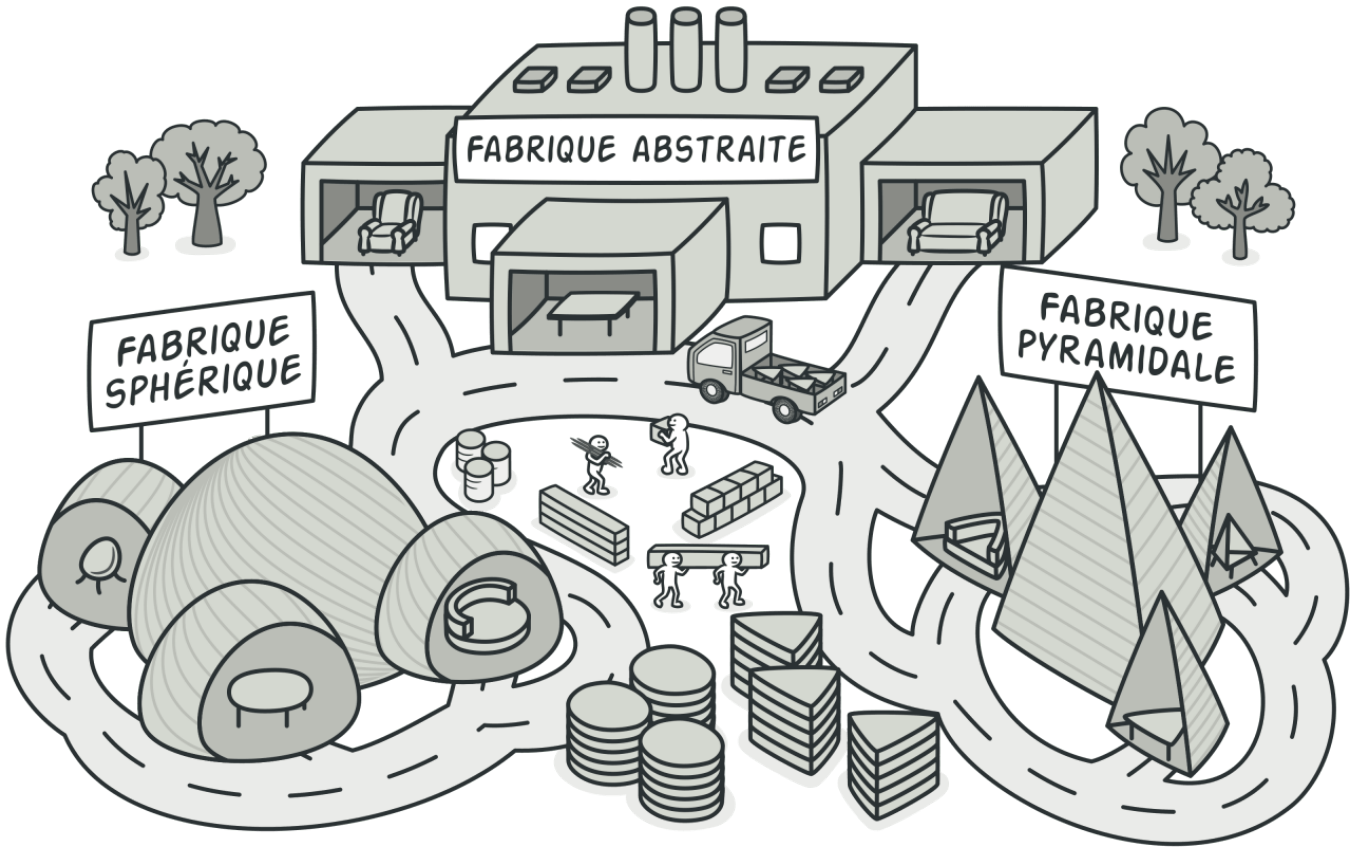
Utilisez la fabrique lorsque vous voulez économiser des ressources système en réutilisant des objets au lieu d'en construire de nouveaux.

- 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.

Patrons de conception
Plan

Intention

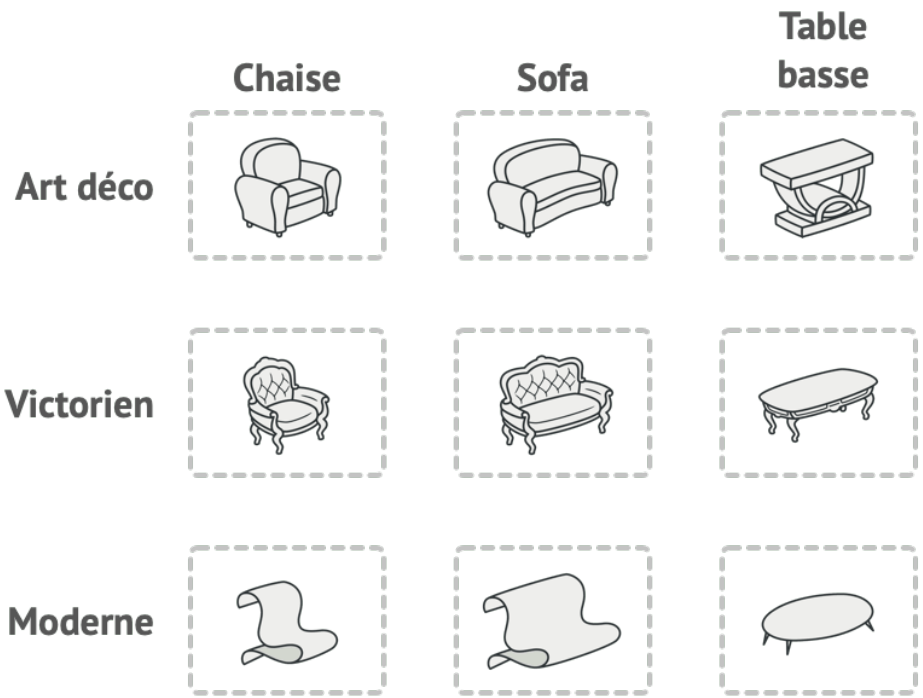
Fabrique abstraite est un patron de conception qui permet de créer des familles d'objets apparentés sans préciser leur classe concrète.



Problème

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

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



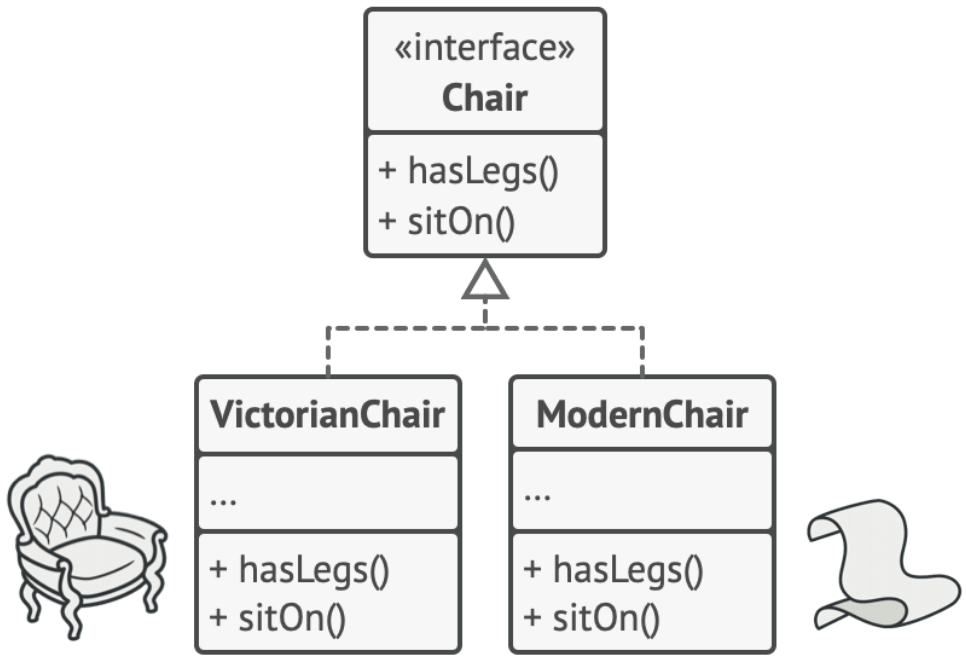
Problème

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.

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.

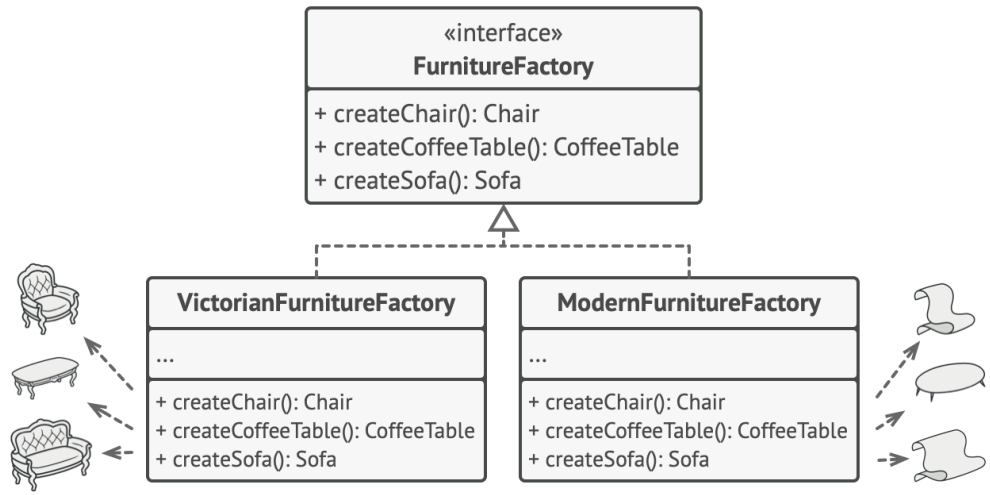
Solution

- 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`…



Solution

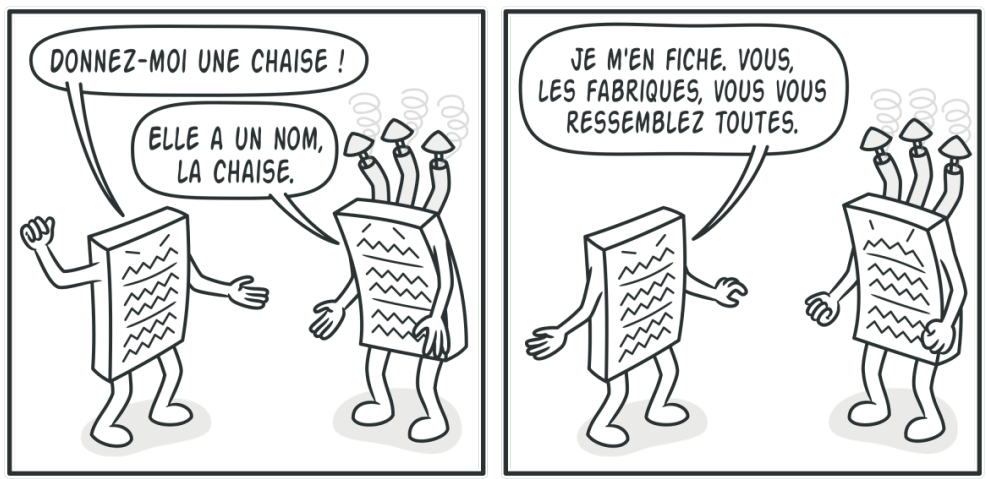
- La prochaine étape est de déclarer la fabrique abstraite
  - interface armée de méthodes de création pour toutes les familles de produits
  - (par exemple : `créerChaise`, `créerSofa` et `créerTableBasse`).
- Ces méthodes doivent renvoyer tous les types de produits abstraits des interfaces que nous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



- 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`.

Solution

- 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.
- 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.

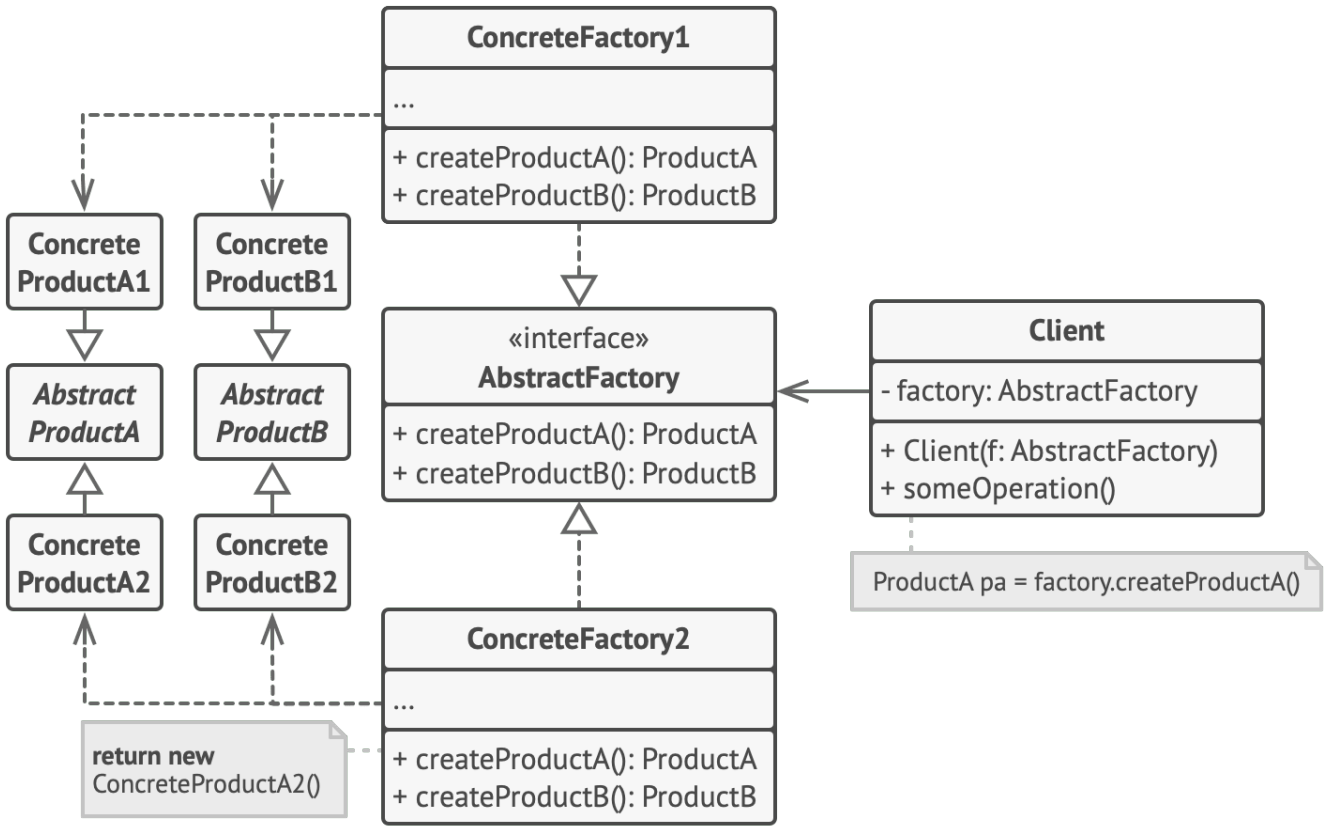


Solution

- 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.

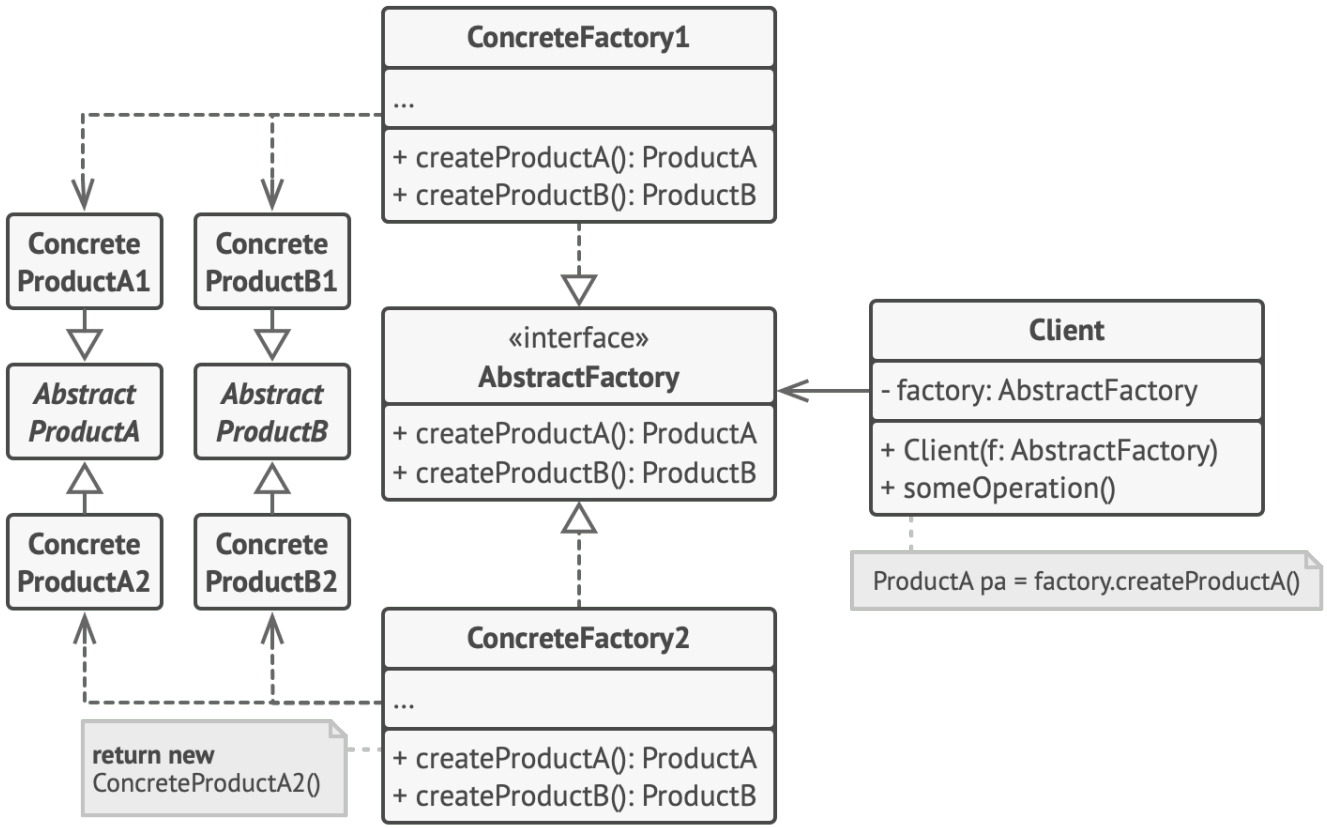
Structure

Les **Produits Abstraits** déclarent une interface pour un ensemble d'objets distincts mais apparentés, qui forment une famille de produits.



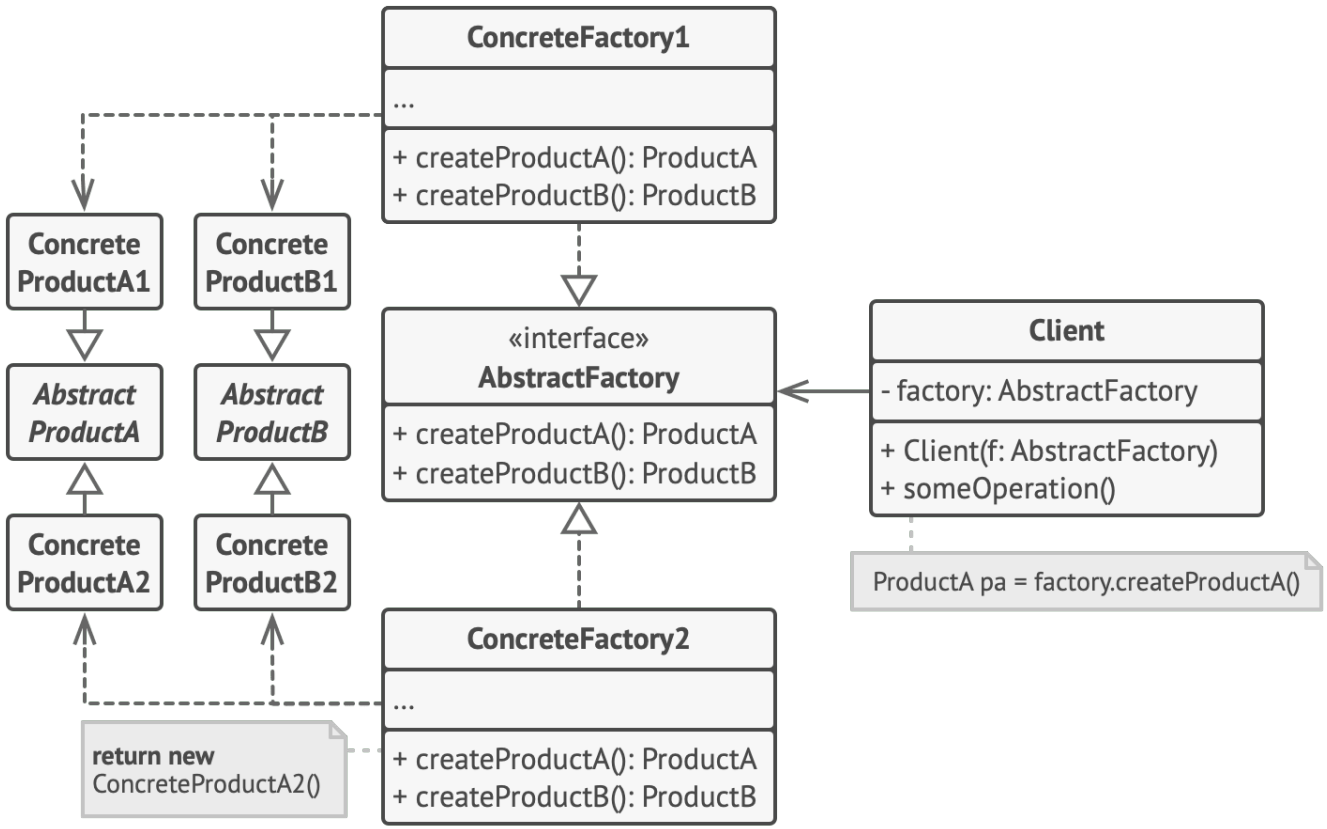
Structure

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).



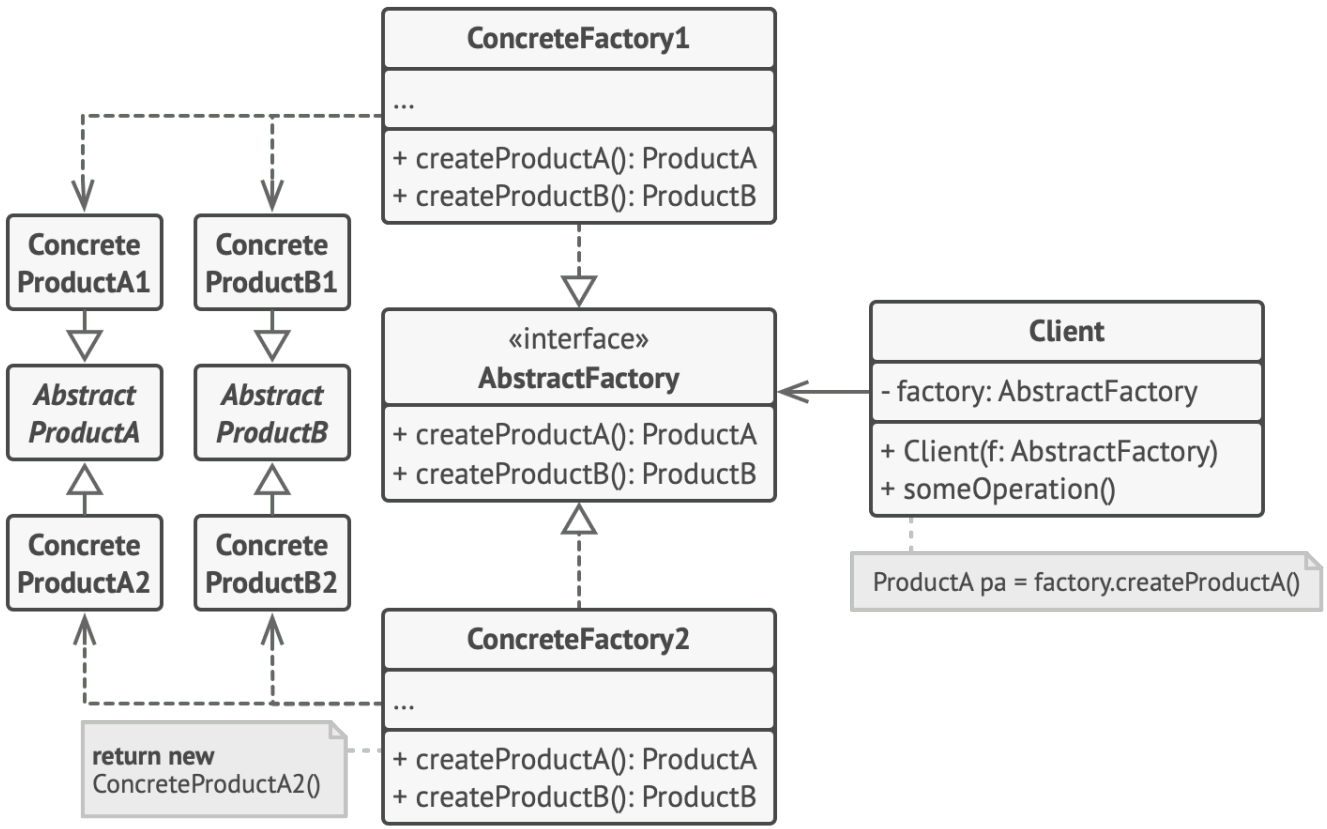
Structure

L'interface **Fabrique Abstraite** déclare un ensemble de méthodes pour créer chacun des produits abstraits.



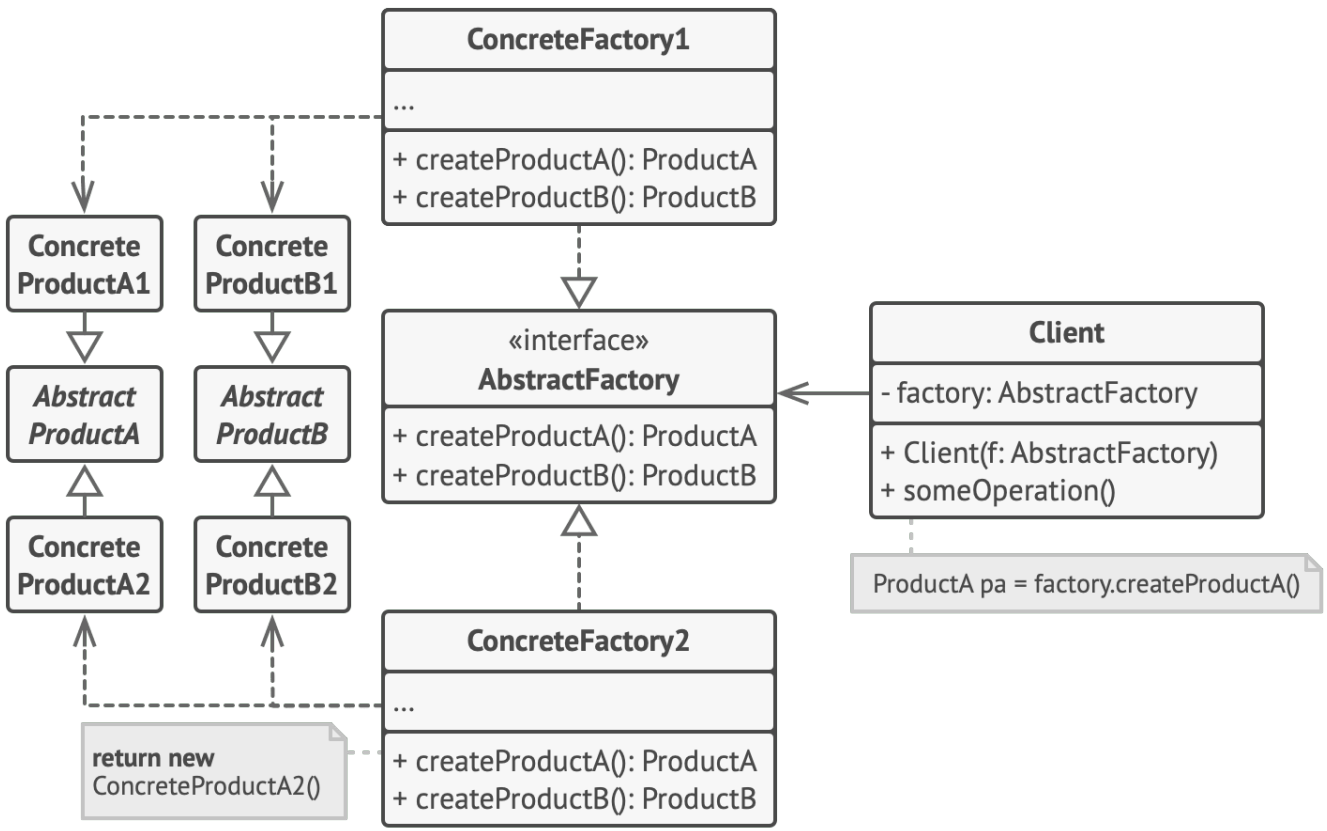
Structure

Les **Fabriques Concrètes** 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.



Structure

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.



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

Avantages

- Vous êtes assurés que les produits d'une fabrique sont compatibles entre eux.
- Vous découplez 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.

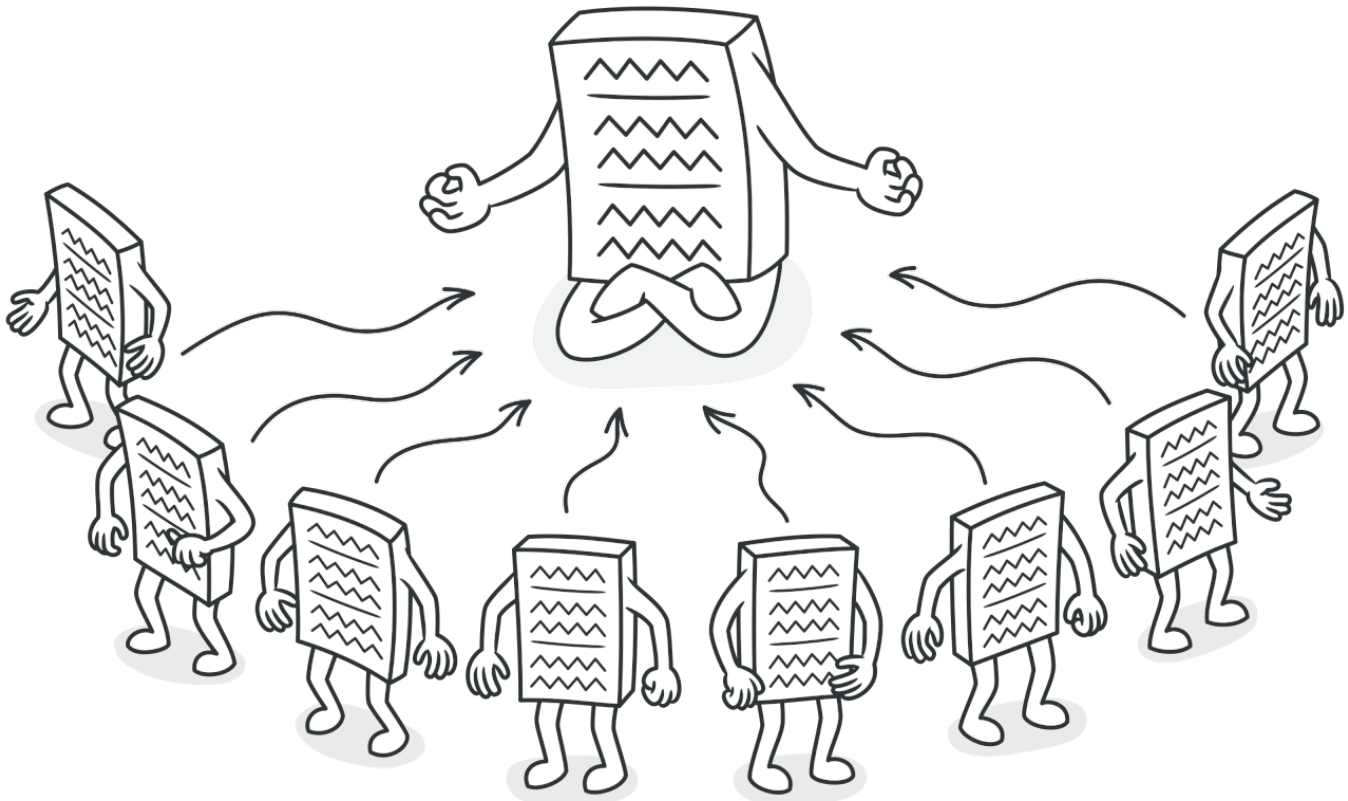
Inconvénient

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

Patrons de conception
Plan

Intention

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.

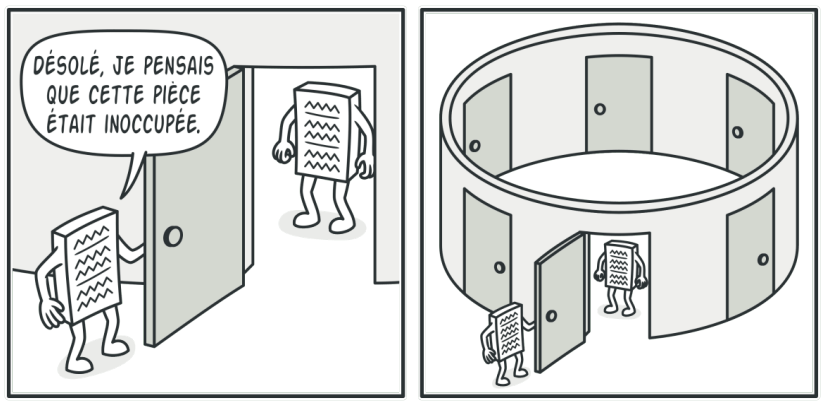


Problème

Le patron **singleton** répond à deux besoins :

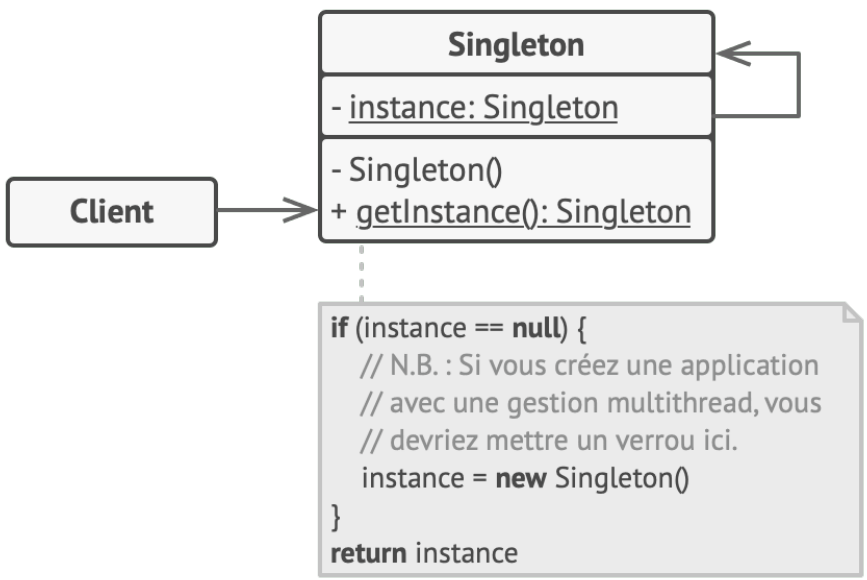
1) **s'assurer de l'unicité de l'instance d'une classe.**
   Ce patron s'applique à des classes qui doivent être instanciées par un seul objet (global). Par exemple :
   - le gouvernement d'un pays
   - les paramètres graphiques d'une application
   - la partie en cours d'un jeu
   - le contrôleur principal de l'application...

2) **fournir un accès global à cette instance.**
   Cette instance est de fait un genre de variable globale du programme : tous les éléments qui veulent accèder à la classe se réfèrent à la même instance. Cependant, on veut éviter de dupliquer le code relatif à ces manipulation partout dans l'application.



Structure

- Le constructeur de la classe Singleton est privé.
- En revanche, elle déclare une méthode statique getInstance qui retourne la même instance de sa propre classe.
- Cette méthode se comporte comme un constructeur : en coulisse, elle appelle le constructeur privé pour créer un objet et le stocke dans un attribut statique.
- Tous les appels ultérieurs à cette méthode retournent l'objet en cache.
- 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.



Exemple : base de données

```
class Database {
  private static field instance: Database;

  private constructor Database() {
    // Code d'initialisation (la connexion au serveur de la base de données par
      exemple).
  }

  public static method getInstance() {
    if (Database.instance == null) {
      acquireThreadLock();
      // Ce thread attend la levée du verrou (lock) le temps de s'assurer que l'
      instance n'a pas déjà été initialisée dans un autre thread.
      if (Database.instance == null)
        Database.instance = new Database();
    }
    return Database.instance;
  }

  public method query(sql) {
    // Toutes les requêtes sur la base de données d'une application passent par cette
       méthode. On peut y définir des limitations, ou de la mise en cache.
  }}

class Application {
  method main() {
    Database foo = Database.getInstance();
    foo.query("SELECT ...");
    // ...
    Database bar = Database.getInstance();
    bar.query("SELECT ...");
    // La variable `bar` contiendra le même objet que la variable `foo`.
  }}
```

Application du singleton

- Utilisez le singleton lorsque l'une de vos classes ne doit fournir qu'une seule instance à tous ses clients.
  - 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 « instanciation 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.

Avantages et inconvénients

Avantages

- 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.

Inconvénients

- 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.
- Attention dans un environnement multithread pour éviter que le singleton ne se retrouve en plusieurs exemplaires.
- Les tests unitaires du code client peuvent se révéler difficiles.
  - En effet, 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 qu'on ne peut pas redéfinir la méthode statique, il est difficile de produire un singleton fictif.

Patrons de conception
Plan

1. Patrons structurels
   1.1. Composite
   1.2. Procuration
   1.3. Autres patrons structurels

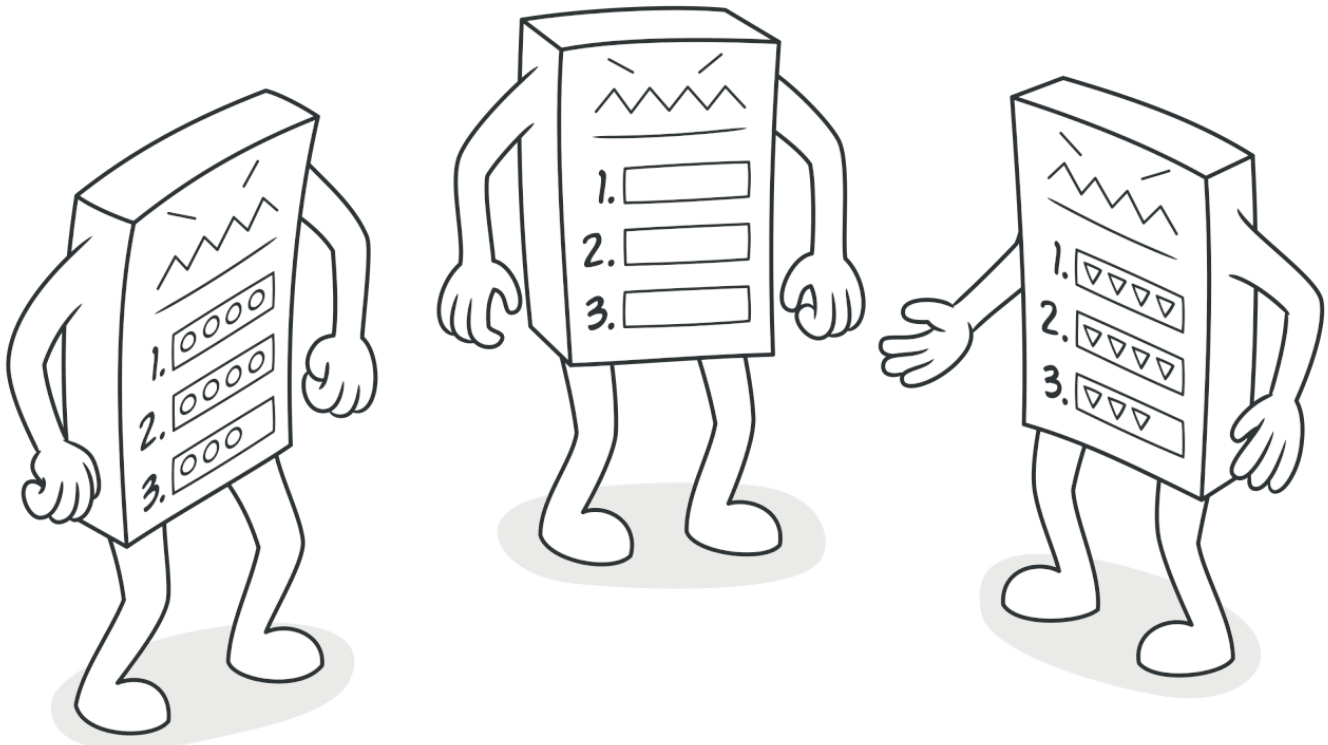
2. Patrons de création
   2.1. Fabrique
   2.2. Singleton

3. Patrons comportementaux
   3.1. Patron de méthode
   3.2. Stratégie
   3.3. État

Patrons de conception
Plan

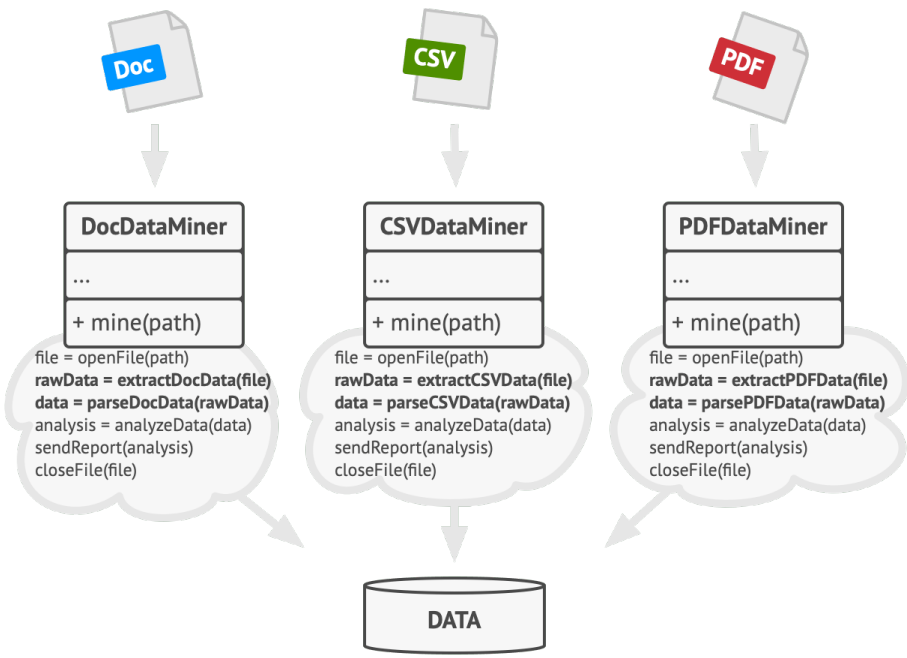
Intention

Patron de Méthode (Template Method) 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.

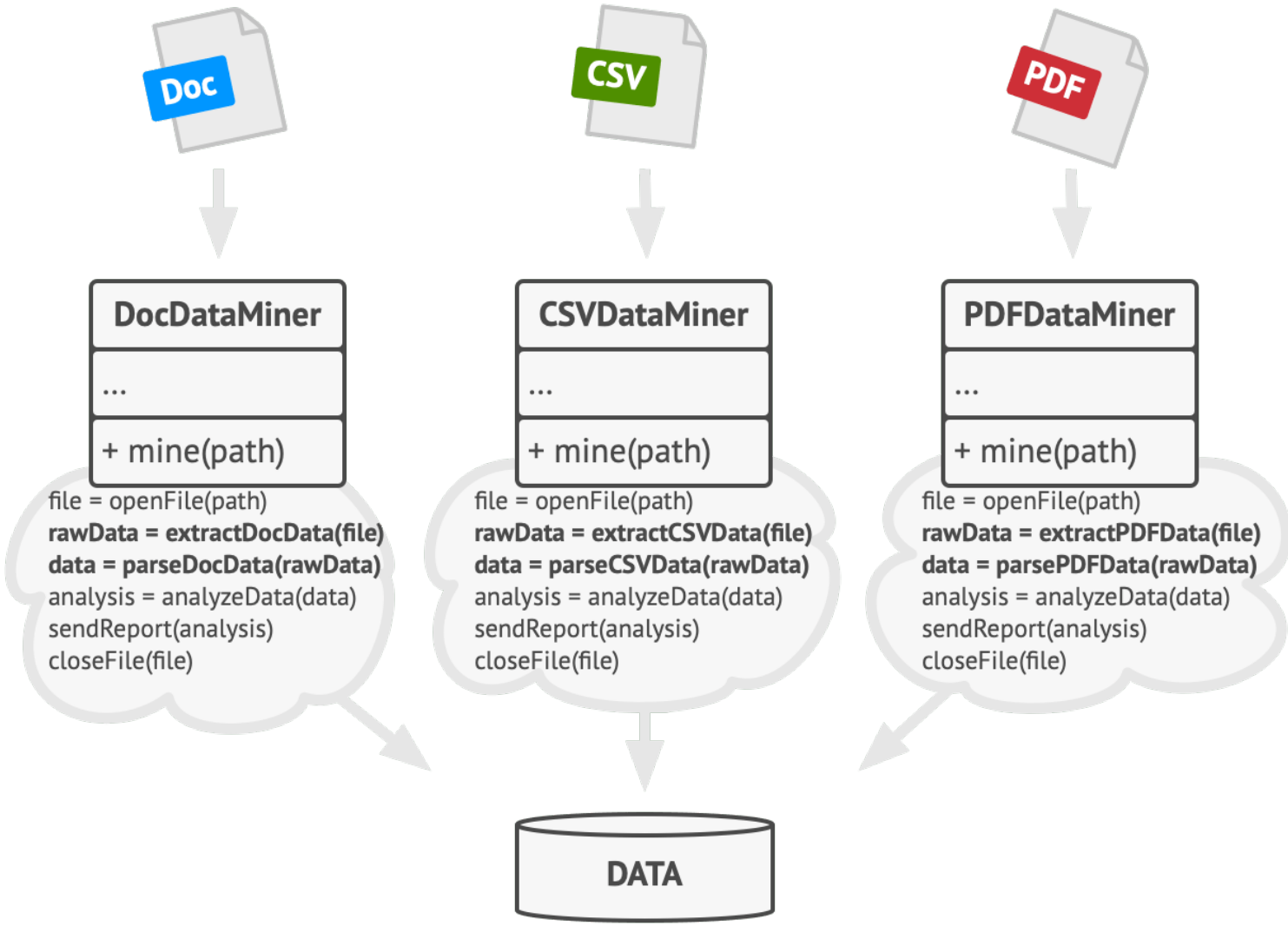


Problème

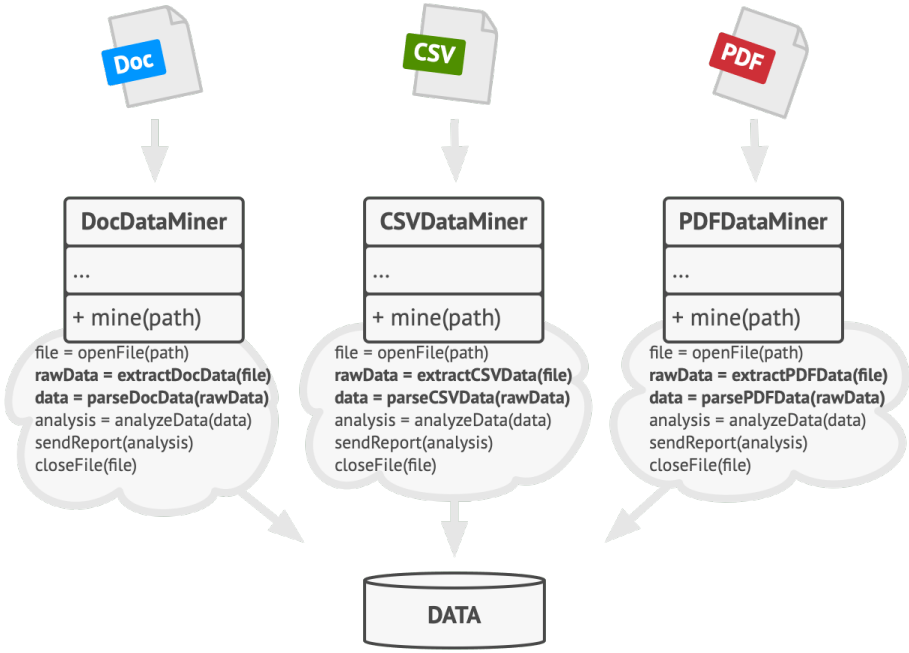
- On est en train de créer une application de data mining (fouille de données) qui analyse les documents d'une entreprise.
- Les utilisateurs alimentent l'application avec différents formats (PDF, DOC, CSV).
- On tente de récupérer les données utiles dans un format uniforme.
- La première version de l'application ne fonctionnait qu'avec les fichiers DOC.
- Dans la version suivante, les fichiers CSV étaient acceptés.
- Puis, on a ajouté les fichiers PDF.



Problème


Doc
CSV
PDF
DocDataMiner
...
+ mine(path)
CSVDataMiner
...
+ mine(path)
PDFDataMiner
...
+ mine(path)
file = openFile(path)
rawData = extractDocData(file)
data = parseDocData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
file = openFile(path)
rawData = extractCSVData(file)
data = parseCSVData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
file = openFile(path)
rawData = extractPDFData(file)
data = parsePDFData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
DATA

Problème



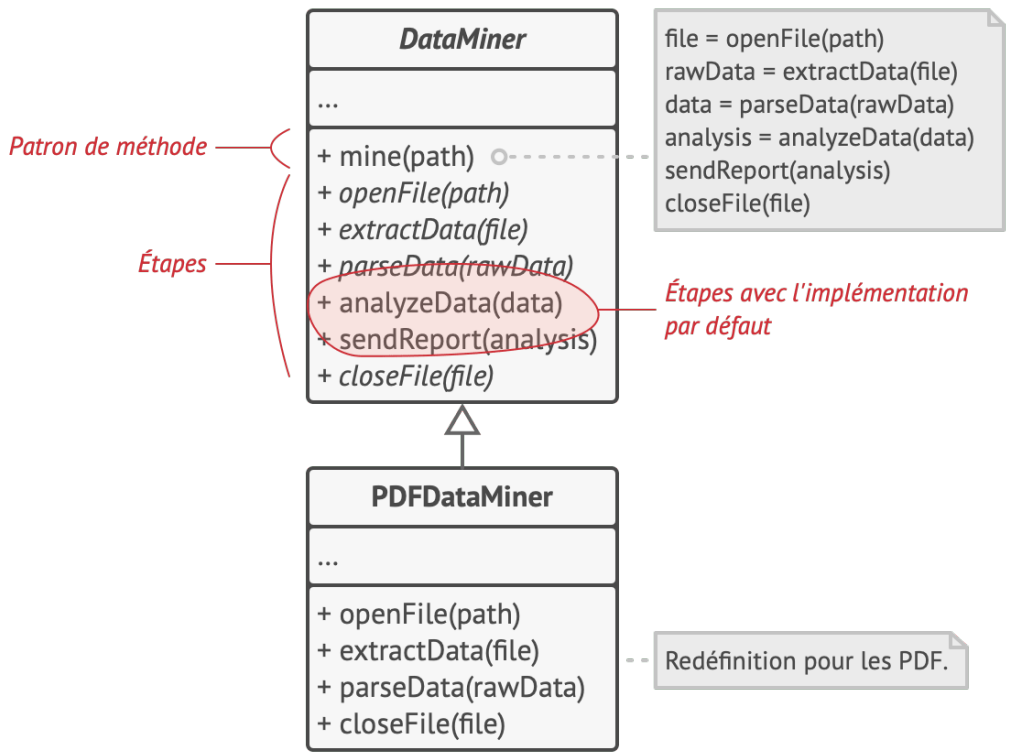
- Ces trois classes ont beaucoup de code similaire.
- Bien que le code spécifique aux formats de données soit complètement différent d'une classe à l'autre, celui qui traite et analyse les données est presque identique.
- Comment se débarrasser de tout ce code dupliqué, tout en laissant la structure de l'algorithme intacte ?
- **Problème dans le code client :**
  - 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), on pourrait simplifier le code en utilisant le polymorphisme.

Solution

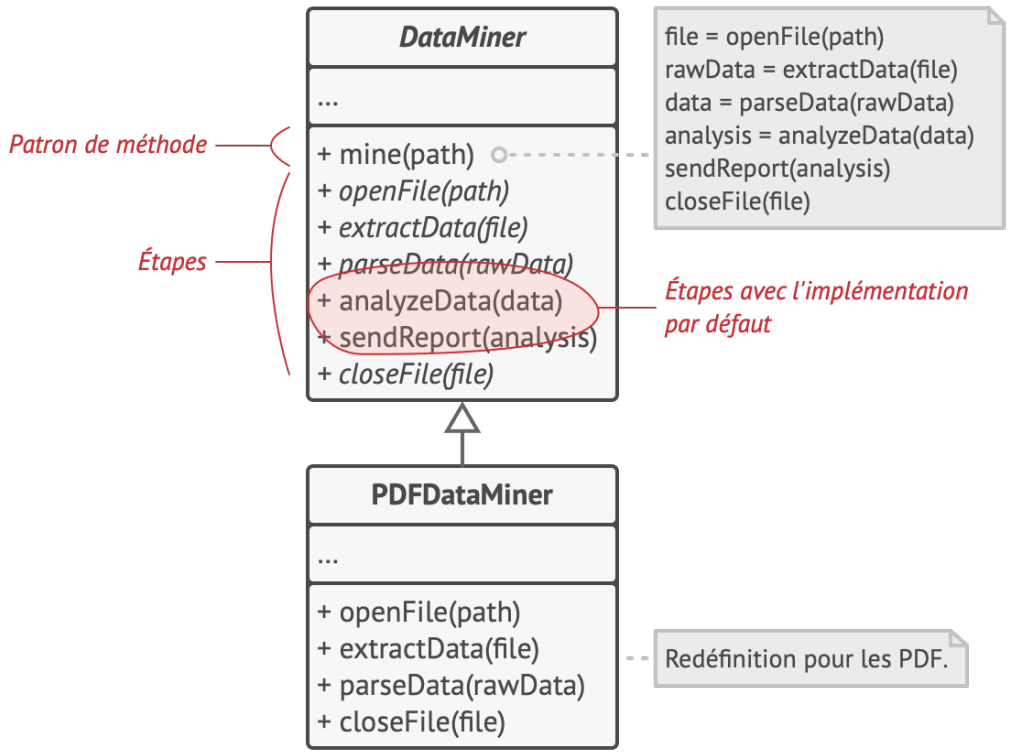
- Le **patron de méthode** consiste à :
  1) découper un algorithme en une série d'étapes,
  2) transformer ces étapes en méthodes,
  3) mettre l'ensemble des appels à ces méthodes dans une seule méthode socle, le *patron de méthode*.
- Les étapes peuvent être abstraites ou avoir une implémentation par défaut.
- Pour utiliser l'algorithme, le client doit :
  - fournir sa propre sous-classe,
  - implémenter toutes les étapes abstraites,
  - redéfinir certaines d'entre elles si besoin.
- En revanche il n'a pas besoin de redéfinir la méthode socle elle-même.

Solution

- 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.



Solution

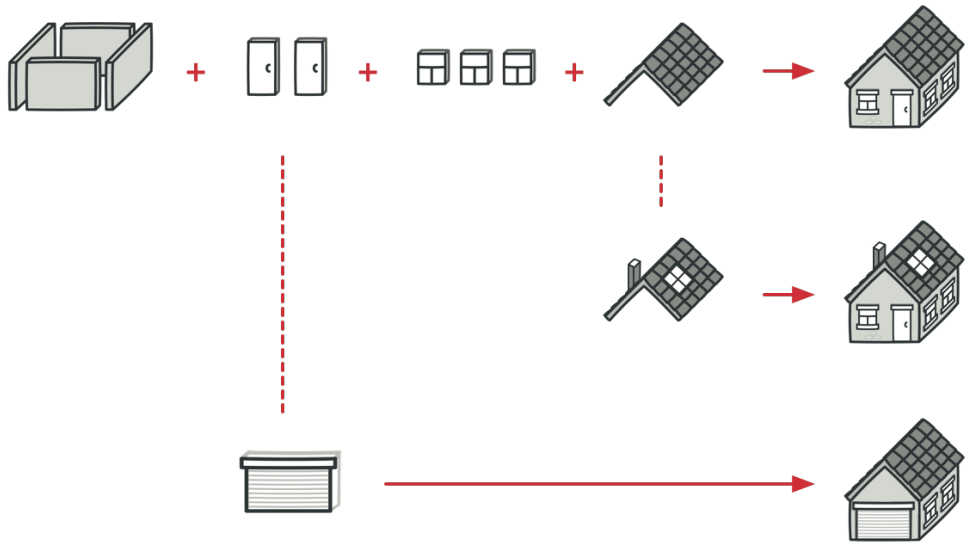


- En premier lieu, nous pouvons déclarer toutes les étapes abstraites afin de forcer les sous-classes à définir leur propre implémentation pour ces méthodes.
- Dans notre cas, les sous-classes ont déjà les implémentations nécessaires.
- Nous devons donc uniquement ajuster les signatures des méthodes en les faisant correspondre à celles de la classe mère.

Solution

- ☞ À présent, débarrassons nous du code dupliqué.
- ☞ Le code pour ouvrir/fermer les fichiers et récupérer/parser les données est différent pour chaque format de données, donc on le laisse tel quel.
- ☞ En revanche, les autres étapes (analyser les données brutes et établir les rapports) se ressemblent.
- ☞ Donc on déplace leur implémentation dans la classe de base, où les sous-classes se partagent le code.
- ☞ On voit là deux types d'étapes :
  1) les opérations **abstraites** : doivent être implémentées dans chaque sous-classe.
  2) les opérations **facultatives** : possèdent une implémentation par défaut, mais peuvent être redéfinies si besoin.
- ☞ Il existe un autre type d'étape : 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.

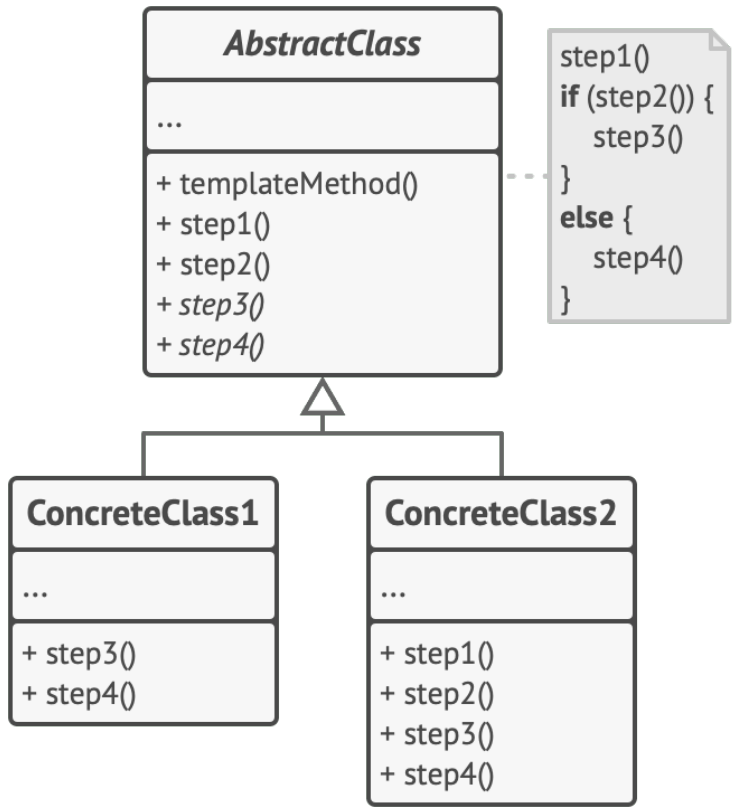
Exemple



- 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.

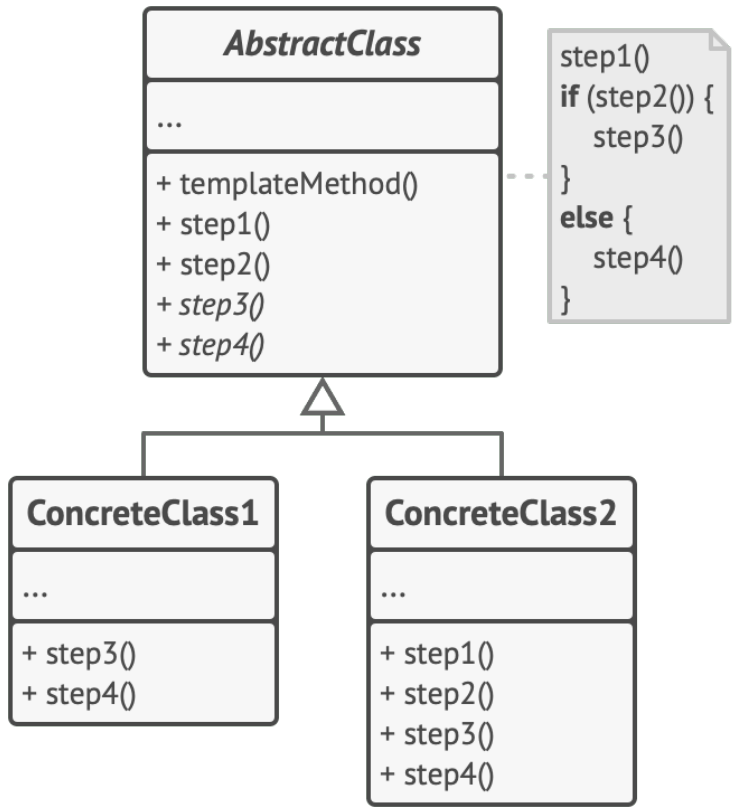
Structure

La **Classe Abstraite** déclare des méthodes (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.



Structure

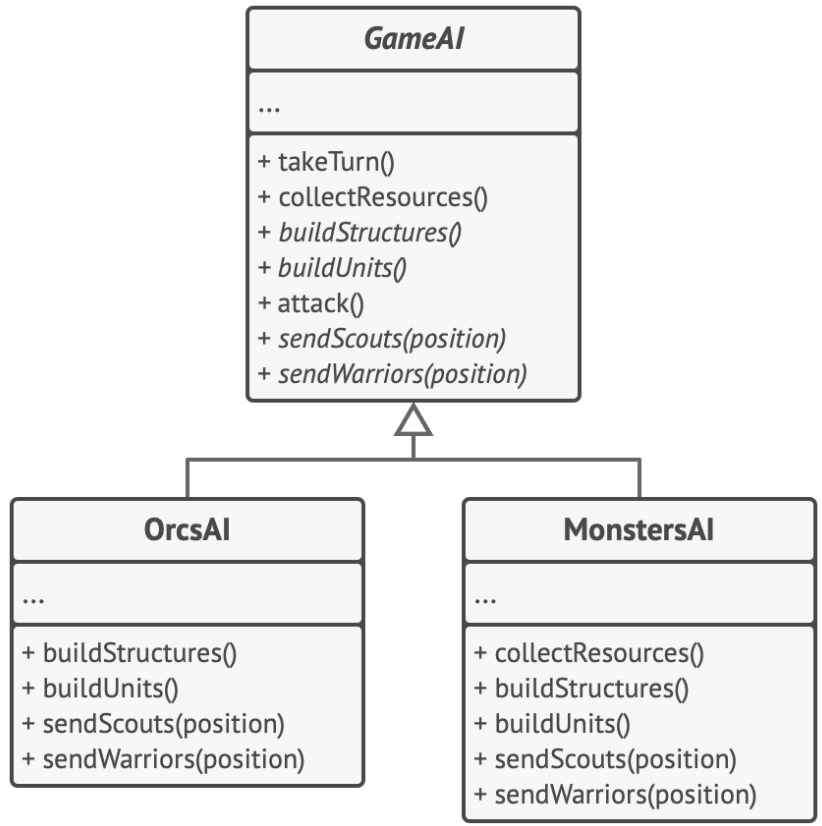
Les Classes Concrètes peuvent redéfinir toutes les étapes, mais pas patronDeMéthode.



Exemple : jeu vidéo

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.

- Dans le jeu, toutes les races ont à peu près les mêmes unités et bâtiments.
- On peut donc réutiliser la même structure d'IA pour les différentes races, tout en personnalisant certains détails.
- Ainsi, on peut 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, il suffit de créer une nouvelle sous-classe d'IA et d'y redéfinir les méthodes déclarées dans la classe de base de l'IA.



Exemple : jeu vidéo

```
class GameAI is
  method turn() is
    collectResources()
    buildStructures()
    buildUnits()
    attack()

  method collectResources() is
    foreach (s in this.builtStructures) do
      s.collect()

  abstract method buildStructures()
  abstract method buildUnits()

  method attack() is
    enemy = closestEnemy()
    if (enemy == null)
      sendScouts(map.center)
    else
      sendWarriors(enemy.position)

  abstract method sendScouts(position)
  abstract method sendWarriors(position)
```

```
class OrcsAI extends GameAI is
  method buildStructures() is
    if (there are some resources)
// Construit des fermes, des casernes,
// puis une forteresse.

  method buildUnits() is
    if (there are plenty of resources)
      if (there are no scouts)
// Construit un éclaireur.
      else
// Construit un guerrier.

  method sendScouts(position) is
    if (scouts.length > 0)
// Envoie les éclaireurs à la position.

  method sendWarriors(position) is
    if (warriors.length > 5)
// Envoie les guerriers à la position.
```

```
class MonstersAI extends GameAI is
  method collectResources() is
// Les monstres ne récoltent pas de ressources.
  method buildStructures() is
// Les monstres ne fabriquent pas de bâtiments.
  method buildUnits() is
// Les monstres ne construisent pas d'unités.
```

Application du patron de méthode

- 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.
  - Permet de transformer un algorithme monolithique en une série d'étapes.
  - Ces étapes peuvent être étendues par des sous-classes.
  - Mais elles gardent 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, on risque de devoir retoucher toutes les classes lorsqu'on modifie l'algorithme.
  - Lorsqu'on transforme un tel algorithme en un patron de méthode, on peut également remonter les étapes dotées d'implémentations similaires dans la classe mère.
  - On évite ainsi la duplication de code.
  - On peut 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 :
   - 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

Avantages

- 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.

Inconvénients

- 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.

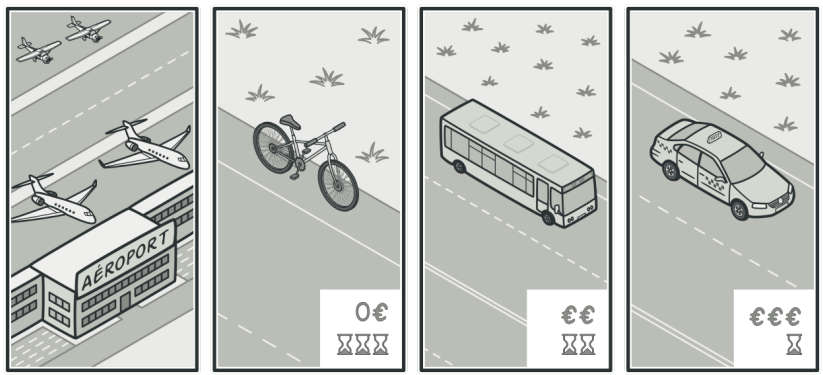
Patrons de conception
Plan

Intention

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

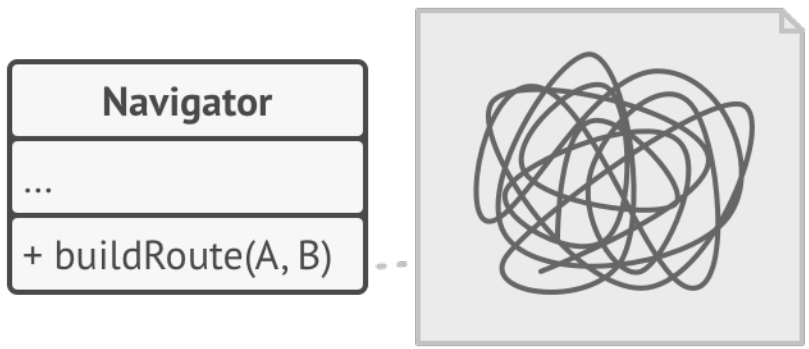
- On veut créer une application de navigation, qui propose des itinéraires.
- Une première version de l'application calcule des itinéraires routiers.
- On veut ajouter des fonctionnalités :
  - Modes de locomotion alternatifs : pédestre, vélo, transports en commun...
  - Optimisation multi-critères : temps, nombre de changements, coût...
  - Autres extensions : points de passage, accessibilité...



Problème

☞ Modifier le code existant pour ajouter des conditions permettant de paramétrer l'algo d'itinéraire serait une mauvaise idée :

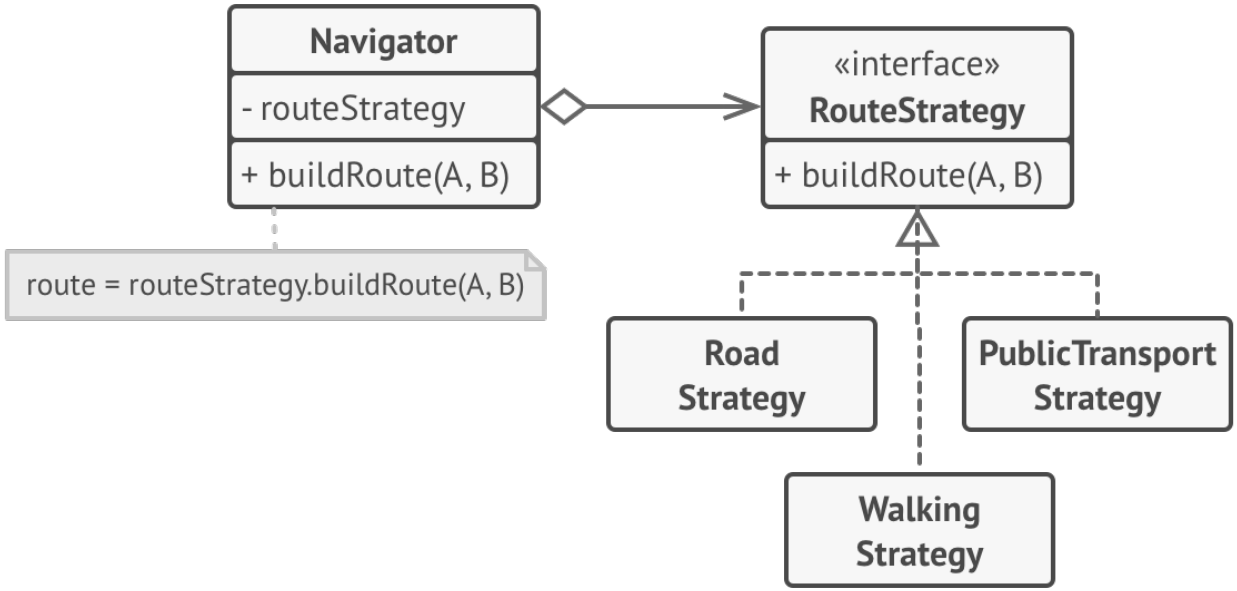
- le code en question est déjà bien compliqué ;
- le nombre de combinaisons possibles de paramètres est ingérable.



Solution

- Le patron de conception **stratégie** 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) garde une référence vers une stratégies.
- Le contexte délègue ses tâches à l'objet stratégie associé.
- Le contexte n'a pas la responsabilité de la sélection de l'algorithme : c'est le client qui lui envoie la stratégie.
- Le contexte ne connait qu'une interface de la stratégie.
- Elle n'expose qu'une seule méthode, qui déclenche l'algorithme encapsulé.
- Le contexte devient indépendant des stratégies concrètes.
- Permet de modifier les algorithmes ou en ajouter de nouveaux sans toucher au code du contexte ou aux autres stratégies.

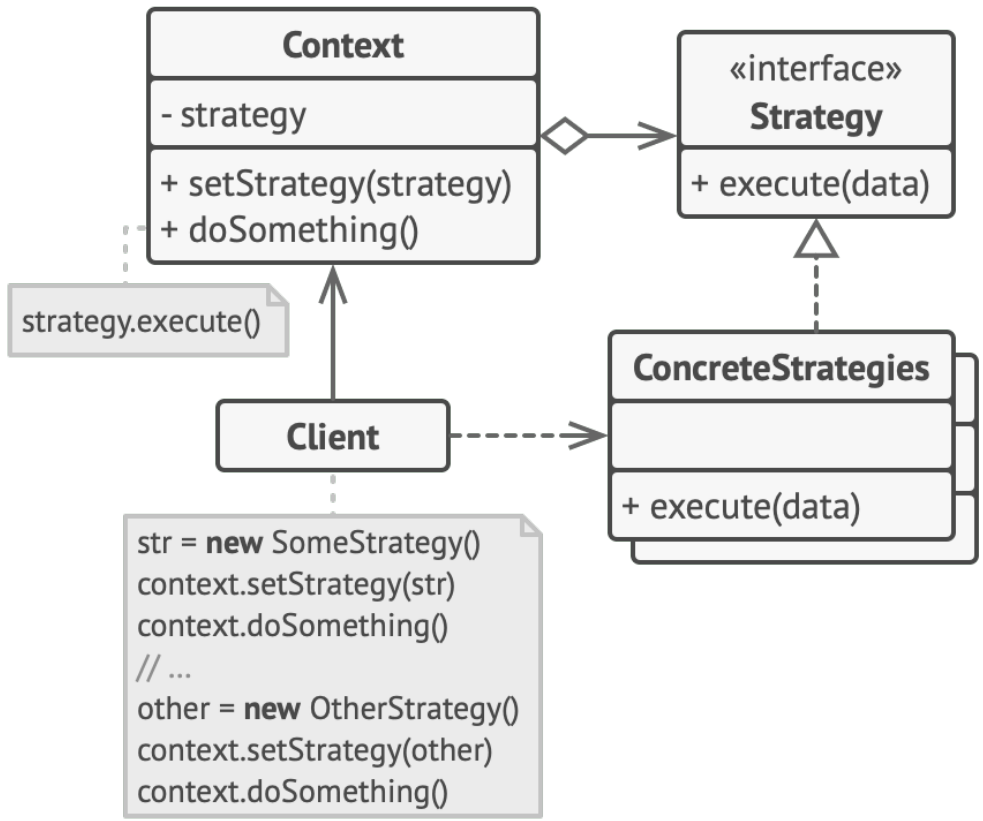
Solution



- Dans notre application de navigation : chaque algorithme d'itinéraire peut être extrait.
- Il a sa propre classe, avec une seule méthode tracerItinéraire.
- La méthode accepte une origine et une destination, puis retourne une liste de points de passage.
- 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.

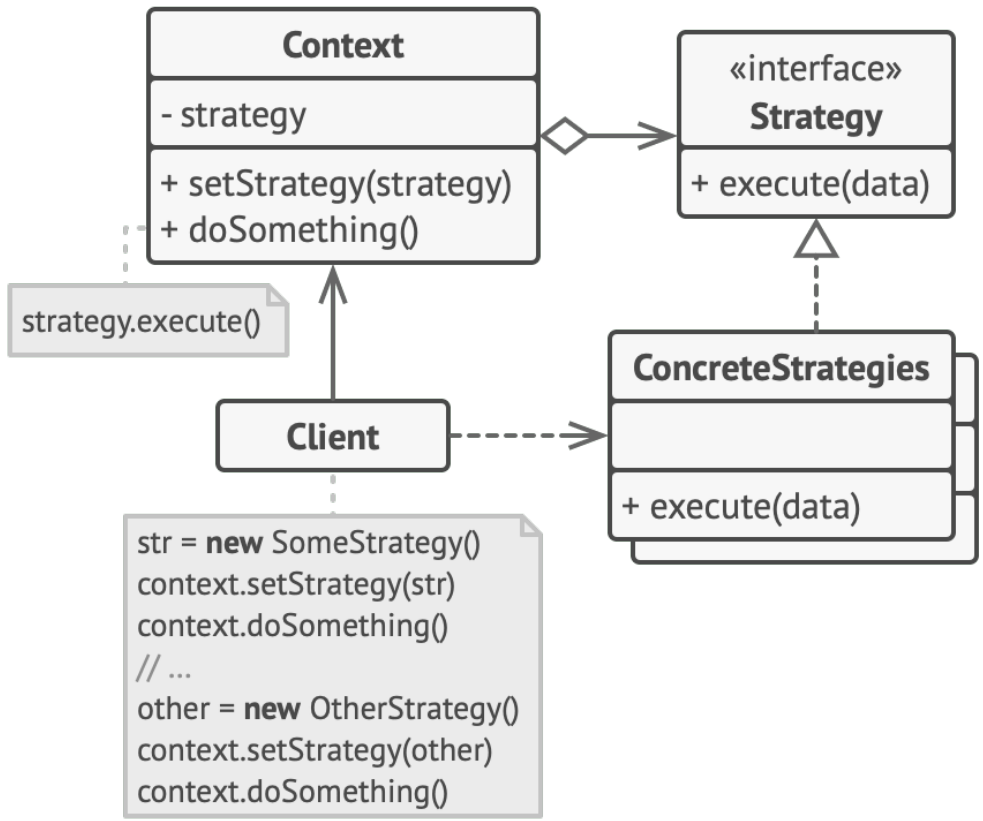
Structure

Le **Contexte** garde une référence vers une des stratégies concrètes et communique avec cet objet uniquement au travers de l'interface stratégie.



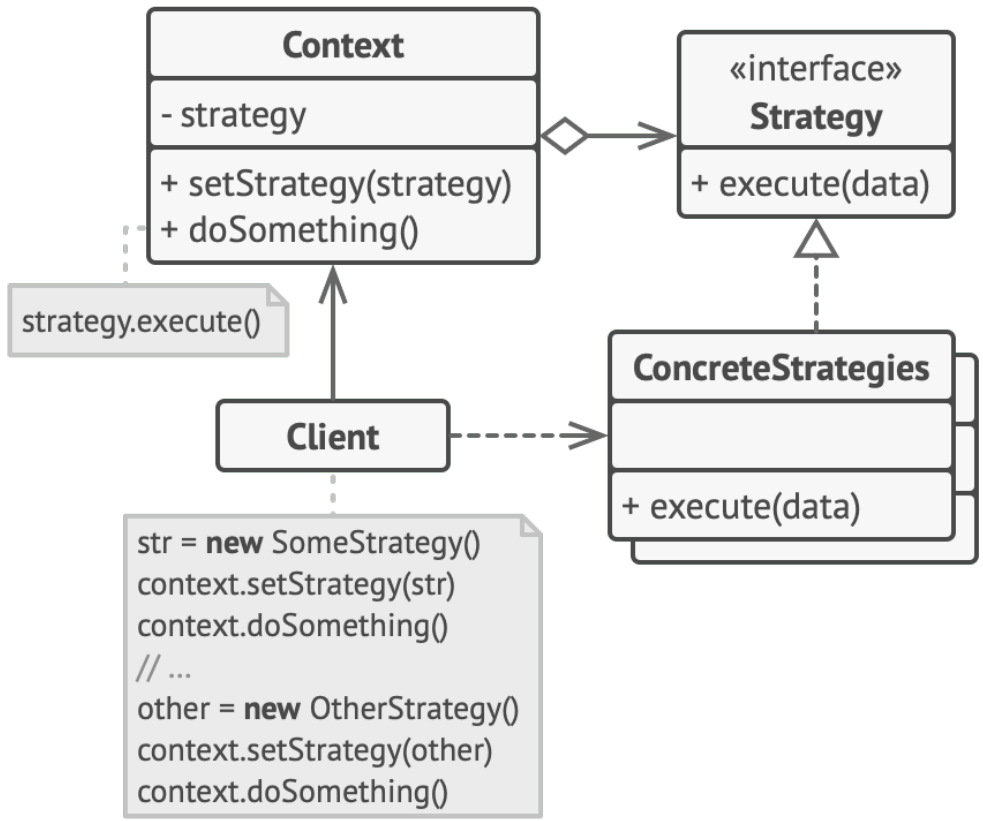
Structure

L'interface Stratégie est commune à toutes les stratégies concrètes. Elle déclare une méthode que le contexte utilise pour exécuter une stratégie.



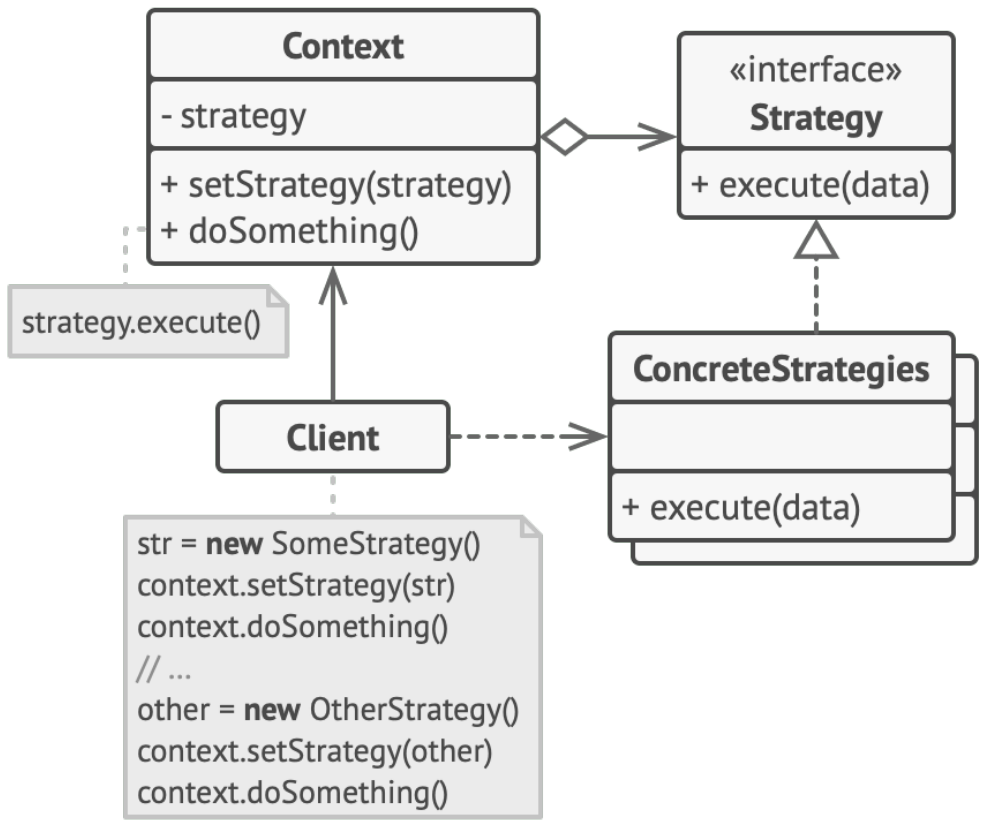
Structure

Les **Stratégies Concrètes** implémentent différentes variantes d'algorithmes utilisées par le contexte.



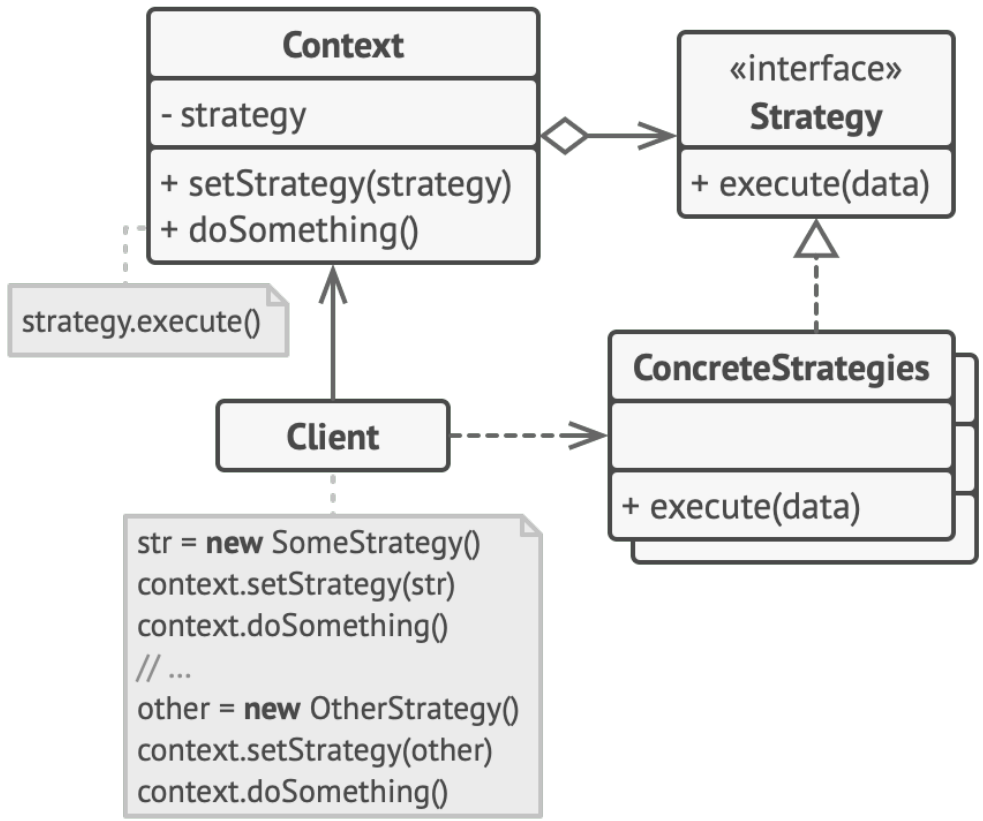
Structure

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.



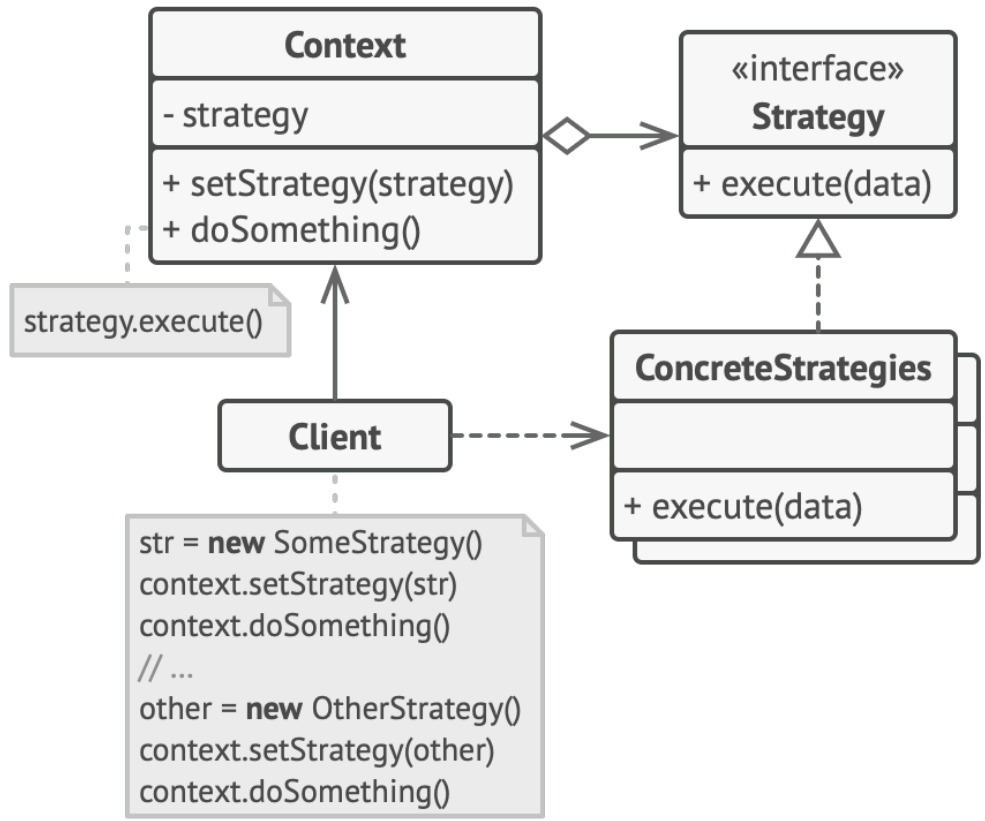
Structure

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.



Structure

Différence avec le patron de méthode : ici on utilise la composition plutôt que l'héritage pour fournir au client différents algorithmes. On peut donc changer d'algo dynamiquement, ce qui est plus délicat avec le patron de méthode, qui déclare les variantes de manière statique.



Exemple : opérations arithmétiques

```
interface Strategy is
  method execute(a, b);

// Stratégies concrètes
class Add implements Strategy is
  method execute(a, b) is
    return a + b

class Sub implements Strategy is
  method execute(a, b) is
    return a - b

class Mult implements Strategy is
  method execute(a, b) is
    return a * b

class Context is
  private strategy: Strategy

  method setStrategy(Strategy strat) is
    this.strategy = strat

  method execute(int a, int b) is
    return strategy.execute(a, b)
```

```
class ExampleApplication is
  method main() is
    Create context object.
    Read first number.
    Read last number.
    Read the desired action.

    if (action == addition)
      context.setStrategy(new Add());

    if (action == subtraction)
      context.setStrategy(new Sub());

    if (action == multiplication)
      context.setStrategy(new Mult());

    result = context.execute
        (First number,Second number);

    Print result.
```

Possibilités d'application

- Avoir différentes variantes d'un algorithme, et pouvoir en changer dynamiquement.
  - Permet de modifier indirectement le comportement de l'objet lors de l'exécution.
- Quand on a beaucoup de classes dont la seule différence est leur façon d'exécuter un comportement.
  - Permet d'extraire des variantes d'un comportement dans une hiérarchie de classes séparées
  - de combiner les classes originales dans une seule,
  - tout en évitant de dupliquer du code.
- Isoler la logique métier d'une classe, de l'implémentation des algorithmes.
  - Les détails ne sont pas forcément importants pour le contexte.
  - Permet de séparer le code, les données internes et les dépendances des divers algorithmes du reste du code.
  - Une interface simple permet aux clients d'exécuter les algorithmes et d'en changer lors de l'exécution.
- Éliminer un gros bloc conditionnel qui choisit entre différentes variantes du même algorithme.
  - Évite 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.
   - typiquement : 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

Avantages

- ☞ Changer l'algorithme utilisé à l'intérieur d'un objet à l'exécution.
- ☞ Séparer les détails de l'implémentation d'un algorithme et le code qui l'utilise.
- ☞ Remplacer l'héritage par la composition.
- ☞ **Principe ouvert/fermé :** ajouter de nouvelles stratégies sans avoir à modifier le contexte.

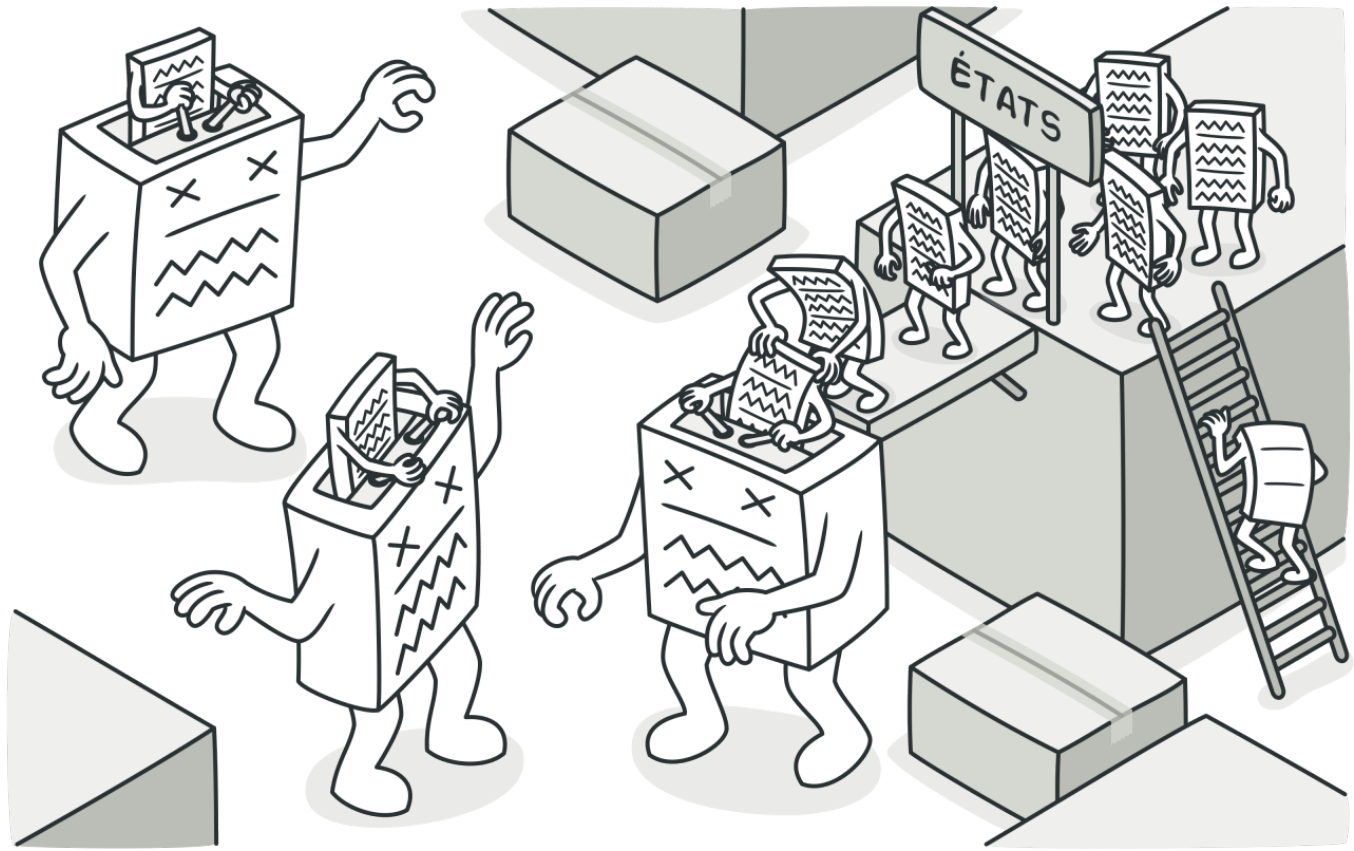
Inconvénients

- ☞ 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 choisir parmi les différentes stratégies.
- ☞ Peut être remplacé par des fonctions anonymes (lambdas).

Patrons de conception
Plan

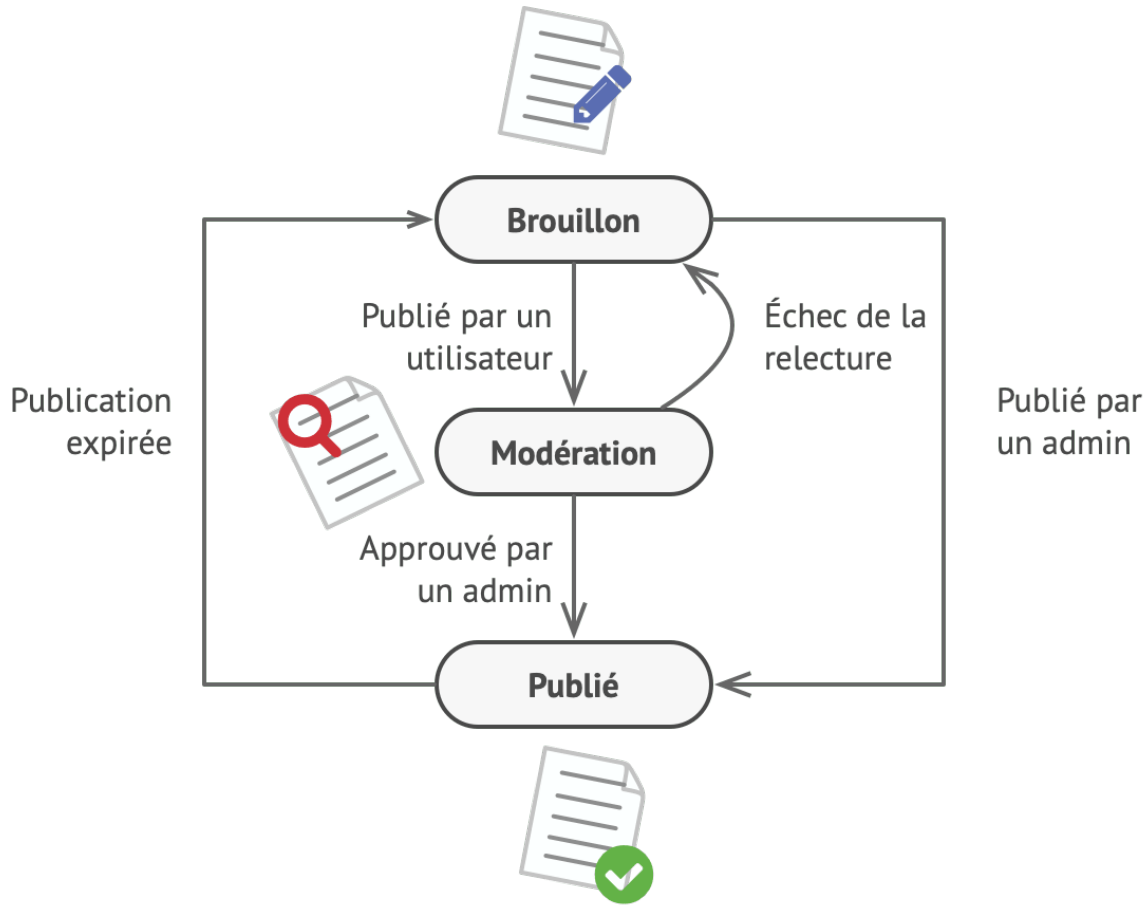
Intention

État (State) 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

- Imaginons une classe Document. Un document peut être dans l'un des trois états suivants : Brouillon (draft), Modération et Publié.
- Le comportement de la méthode publier du document dépend de son état :
  - Brouillon : passe le document en modération.
  - Modération : rend le document public si l'utilisateur est administrateur.
  - Publié : on ne fait rien du tout.



Problème

- ☞ 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.

```
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     // ...
```

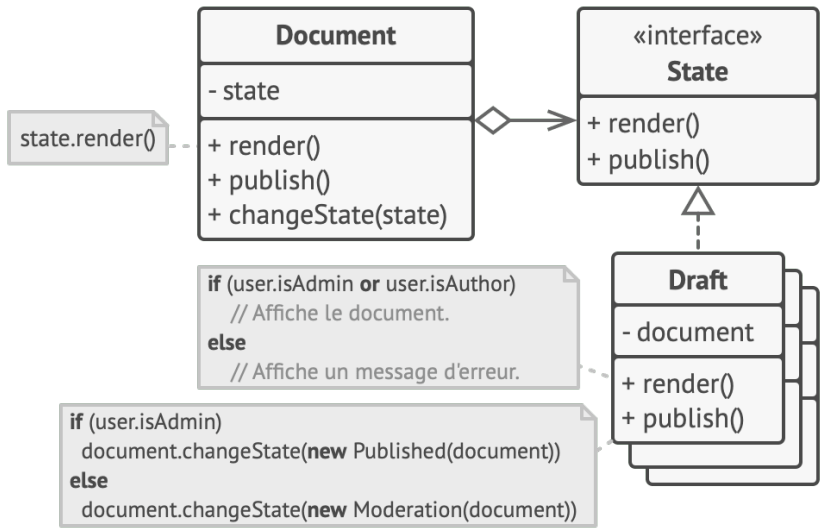
Problème

- 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 d'énormes 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 transition 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

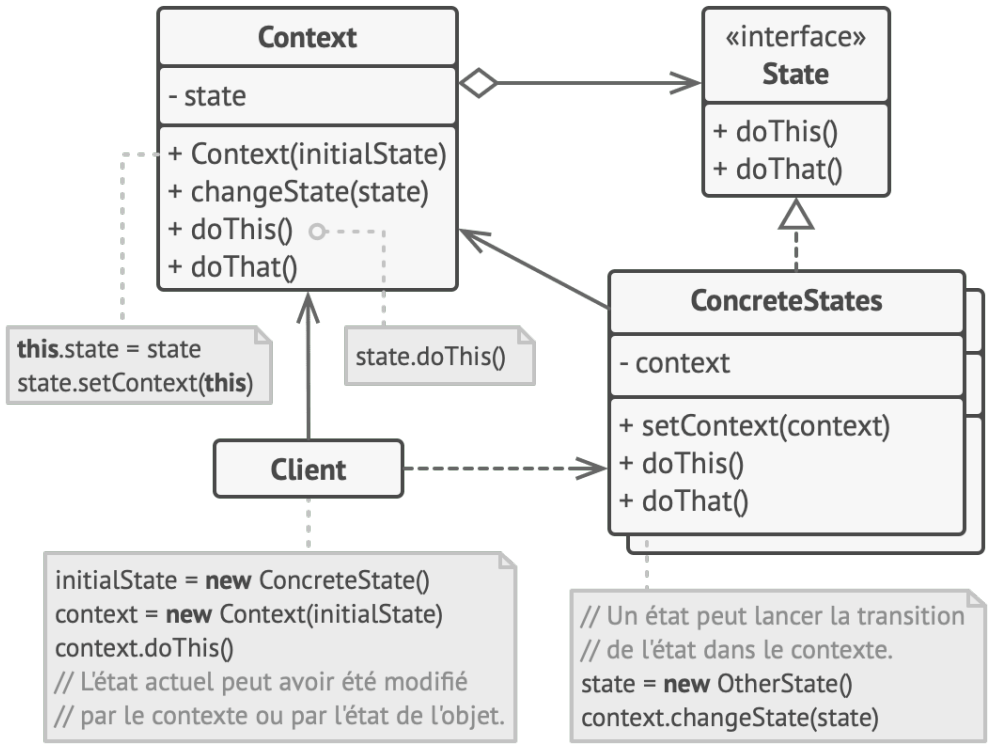
- Le patron de conception état propose de créer des classes pour tous les états possibles d'un objet et d'y mettre les comportements liés aux états.
- Plutôt que d'implémenter tous les comportements, l'objet original (le **contexte**) stocke une référence vers un objet **état**. Il **délègue** tout ce 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.
- Nécessite que :
  - toutes les classes suivent la même interface
  - le contexte utilise cette dernière pour manipuler ces objets.
- **Différence avec Stratégie :** 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.

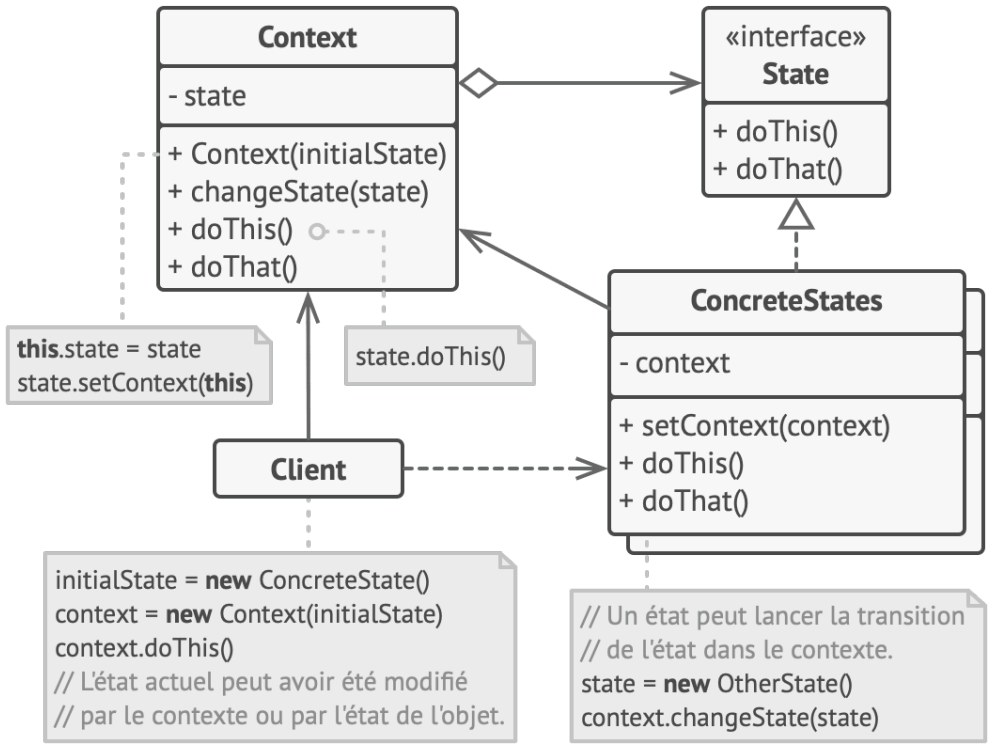
Structure

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.



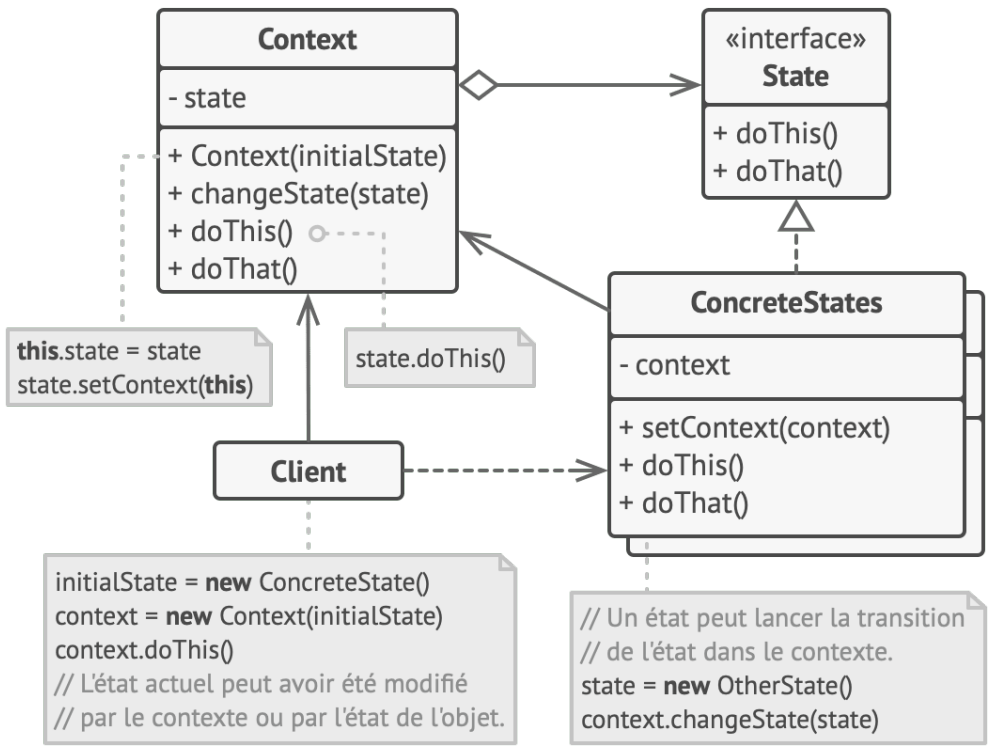
Structure

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.



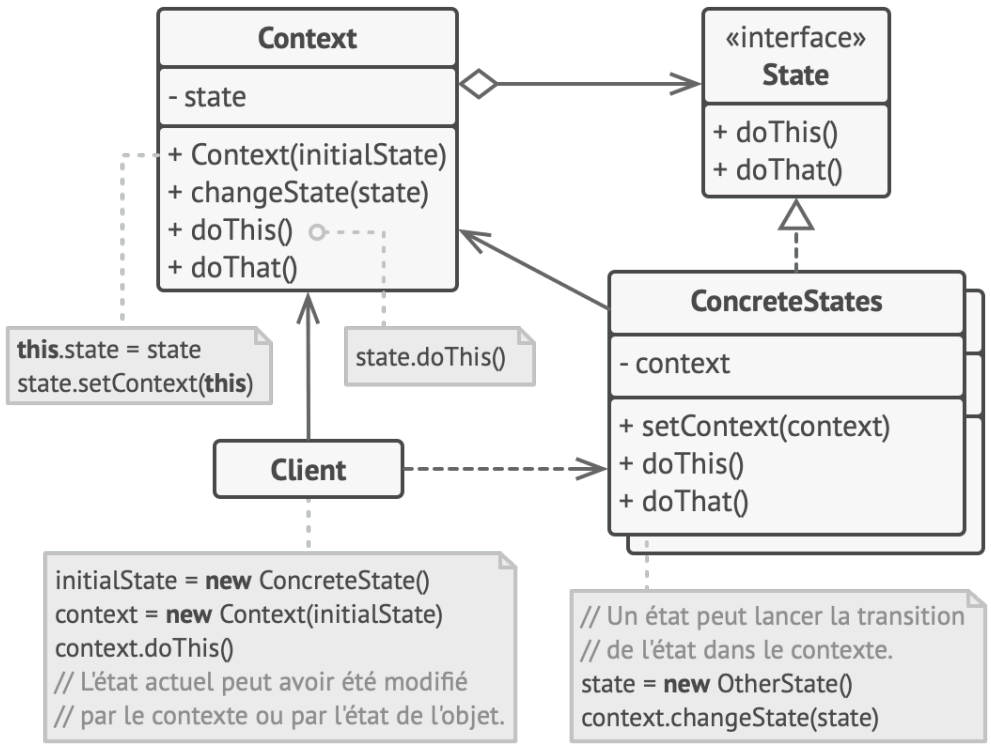
Structure

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.

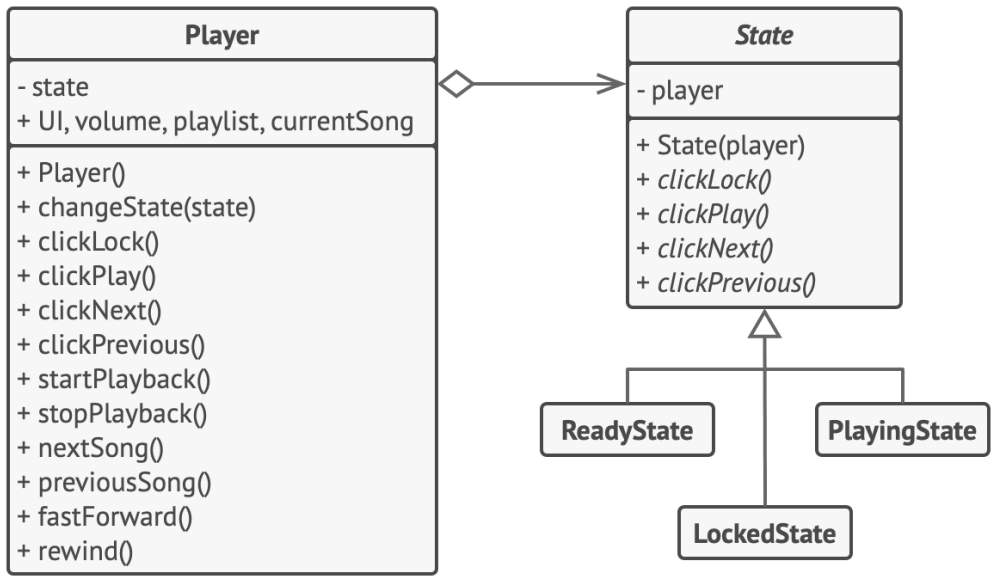


Structure

Le contexte et les états concrets peuvent modifier le prochain état du contexte et lancer une transition en remplaçant l'état lié au contexte.



Exemple : lecteur audio



Exemple : lecteur audio

```
class AudioPlayer is
  field state: State
  field UI, volume, playlist, currentSong

  constructor AudioPlayer() is
    this.state = new ReadyState(this)

    UI = new UserInterface()
    UI.lockButton.onClick(this.clickLock)
    UI.playButton.onClick(this.clickPlay)
    UI.nextButton.onClick(this.clickNext)
    UI.prevButton.onClick(this.clickPrevious)

  method chgSt(state: State) is
    this.state = state

  // Les méthodes de l'UI délèguent l'exécution à l'état.
  method clickLock() is
    state.clickLock()
  method clickPlay() is
    state.clickPlay()
  method clickNext() is
    state.clickNext()
  method clickPrevious() is
    state.clickPrevious()

  method startPlayback() is ...
  method stopPlayback() is ...
  method nextSong() is ...
  method previousSong() is ...
  method fastForward(time) is ...
  method rewind(time) is ...
```

Exemple : lecteur audio

```
abstract class State is
  protected field player: AudioPlayer

  constructor State(player) is
    this.player = player

  abstract method clickLock()
  abstract method clickPlay()
  abstract method clickNext()
  abstract method clickPrevious()
```

```
class Playing extends State is
  method clickLock() is
    player.chgSt(new Locked(player))

  method clickPlay() is
    player.stopPlayback()
    player.chgSt(new Ready(player))

  method clickNext() is
    if (event.doubleclick)
      player.nextSong()
    else
      player.fastForward(5)

  method clickPrevious() is
    if (event.doubleclick)
      player.previous()
    else
      player.rewind(5)
```

```
class Ready extends State is
  method clickLock() is
    player.chgSt(new Locked(player))

  method clickPlay() is
    player.startPlayback()
    player.chgSt(new Playing(player))

  method clickNext() is
    player.nextSong()

  method clickPrevious() is
    player.previousSong()
```

```
class Locked extends State is

  method clickLock() is
    if (player.playing)
      player.chgSt(new Playing(player))
    else
      player.chgSt(new Ready(player))

  method clickPlay() is
    // Verrouillé, ne rien faire.

  method clickNext() is
    // Verrouillé, ne rien faire.

  method clickPrevious() is
    // Verrouillé, ne rien faire.
```

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.
2) Déclarez l'interface état.
3) Pour chaque état, créez une classe qui dérive de l'interface état.
   En effectuant cette manipulation, vous pourriez tomber sur des éléments privés du le contexte. Il faudra alors ajouter une manière publique d'y accéder.
4) Dans votre classe contexte, ajoutez un attribut « état » et un setter public.
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.

Avantages et inconvénients

Avantages

- **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.

Inconvénient

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.

Patrons de conception
Plan

1. Patrons structurels
   1.1. Composite
   1.2. Procuration
   1.3. Autres patrons structurels

2. Patrons de création
   2.1. Fabrique
   2.2. Singleton

3. Patrons comportementaux
   3.1. Patron de méthode
   3.2. Stratégie
   3.3. État