

Chapitre 3 : Catalogue de patrons de conception

« Les patrons de conception vous aident à apprendre des succès des autres plutôt que de vos propres échecs. » **Mark Johnson**

1. Les patrons de conception de la bande des quatre (Gang Of Four)

Le concept de patron de conception est apparu dans le livre “Design Patterns: Elements of Reusable Object-Oriented Software” publié en 1994 par quatre auteurs Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides que l’on nomme la « bande des quatre ». À leur apparition, on croyait que ce serait les premiers patrons d’une longue liste, mais il s’est avéré que c’étaient pratiquement les seuls, au moins généraux. Il y a donc tout intérêt à connaître par cœur ces patrons. D’autres sont apparus, ils sont plus spécialisés. Quelques uns sont présentés en fin de chapitre.

1.1. Organisation des patrons de conception

La bande des quatre présente 23 patrons de conception. Toutefois, je considère que le patron « interpréteur » n’a plus d’intérêt et est avantageusement remplacé par l’utilisation de bibliothèque d’analyse de langage. Je l’ai donc écarté de cette liste. Ne reste que 22 patrons de conception.

		Rôle		
		Création	Structure	Comportement
Domaine	Classe	Fabrique simple	Adaptateur	Patron de méthode
	Objet	Fabrique abstraite Monteur Prototype Singleton	Adaptateur Pont Composite Décorateur Façade Poids mouche Procuration	Chaîne de responsabilités Commande Itérateur Médiateur Memento Observateur État Stratégie Visiteur

1.1.1. Rôle

- **Patrons de création**
 - Décrit la manière d’abstraire le processus d’instanciation des classes pour rendre l’application indépendante du mode de création des instances qui la composent.
- **Patrons de structure**
 - Décrit comment organiser les classes d’un programme dans une structure plus large en séparant l’interface de l’implémentation.
- **Patrons de comportement**
 - Décrit le comportement d’interaction entre les objets et le partage des responsabilités

pour composer des services.

1.1.2. Domaine

- **Portée de classes**
 - Les relations entre les classes et leurs sous-classes sont établies statiquement (par héritage).
- **Portée d'objets**
 - Les relations entre les classes sont établies dynamiquement et modifiées à l'exécution (par composition).

1.2. Adaptateur (Adapter)

Faire correspondre un objet existant qu'on ne contrôle pas à une interface donnée.

1.2.1. Problématique

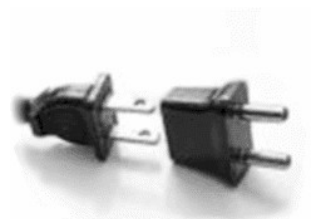
- On a besoin d'utiliser une classe existante, mais l'adaptation par dérivation ne peut pas être faite car on ne dispose pas du code pour modifier l'héritage de la classe.
- On souhaite créer une classe réutilisable qui collabore avec des classes encore inconnues qui seront développées par une autre équipe, c'est-à-dire avec des classes qui n'auront pas forcément une interface compatible.

1.2.2. Solution

- Créer une nouvelle classe **Adapter** qui fournit l'interface souhaitée et fait le lien entre la classe à utiliser et le programme l'utilisant.

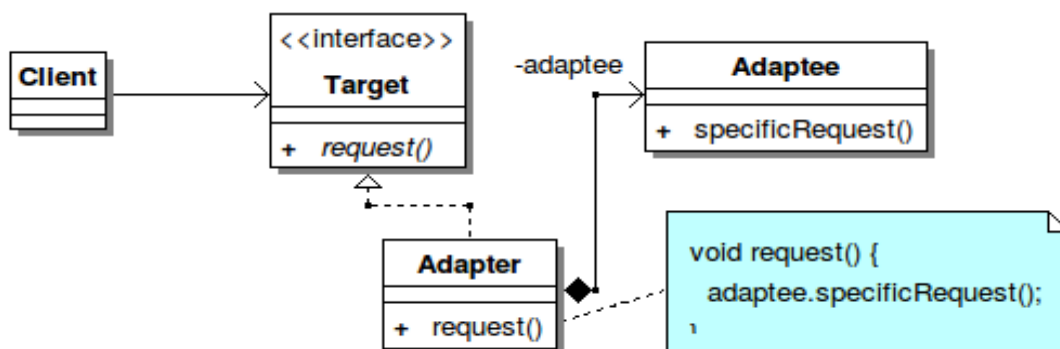
1.2.3. Métaphore : Prise électrique

Parce que les deux pays n'utilisent pas le même standard de prise électrique, il est nécessaire d'utiliser un adaptateur pour brancher un équipement au standard anglais sur une prise de courant au standard français.

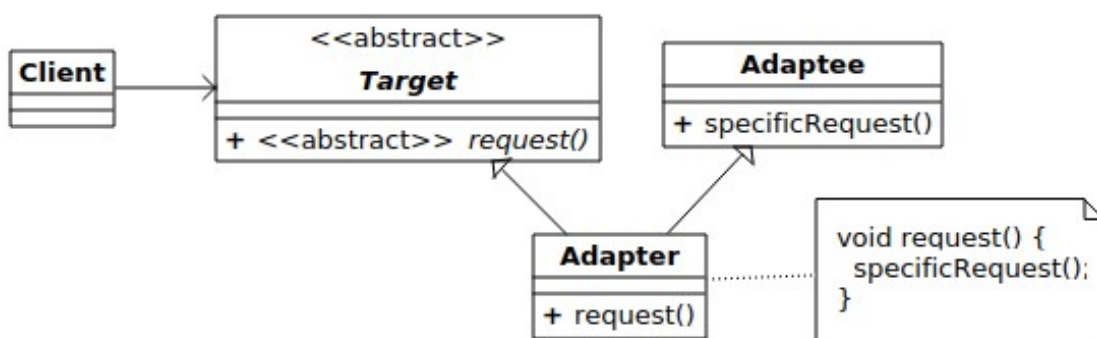


1.2.4. Structure

1. Adaptateur d'objet par composition d'objets :



2. Adaptateur de classe par héritage multiple (quand c'est possible) :

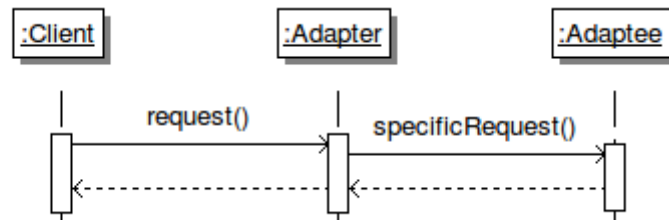


1.2.5. Constituants

- **Client** : utilise le service `request()` de l'interface `Target`.
- **Target** : présente une interface attendue pour un ensemble de services.
- **Adaptee** : la classe qui possède les services recherchés mais avec la mauvaise interface.
- **Adapter** : la classe qui adapte l'interface de `Adaptee` à l'interface définie par `Target`.

1.2.6. Collaborations

Les clients appellent les opérations d'une instance de `Adapter`. En réponse, l'instance de `Adapter` appelle des opérations de `Adaptee` pour exécuter la requête.



1.2.7. Considération d'implémentation

- En C++, l'adaptateur est préférentiellement de type adaptateur de classe. L'héritage vers la classe `Adaptee` doit être de type **privé** de manière à ne pas publier les services de l'adapté à travers l'adaptateur.

```
class Target {
    public : virtual void request() =0;
}
class Adapter: public Target, private Adaptee {
    public : void request() { Adaptee::specificRequest(); }
}
```

- En Java, l'adaptateur est de type adaptateur d'objet. `Target` est préférentiellement une interface et `Adapter` une classe qui implémente l'interface :

```
public interface Target {
    public void request();
}
public final class Adapter implements Target {
    private Adaptee _adaptee = new Adaptee();
    @Override public void request() { _adaptee.specificRequest(); }
}
```

1.3. Chaîne de responsabilité (Chain of Responsibility)

Éviter le couplage implicite entre un destinataire et un émetteur d'une requête en donnant à plus d'un objet la possibilité de gérer la requête.

1.3.1. Problématique

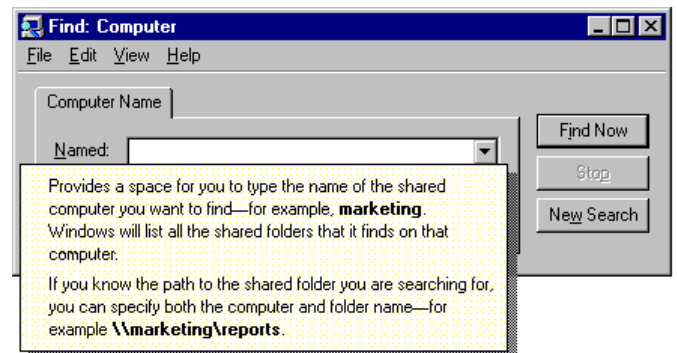
- Plus d'un objet peut gérer une requête, mais le destinataire effectif n'est pas connu a priori.
- On veut émettre une requête sans avoir à préciser le receveur explicitement.
- L'ensemble des objets qui peuvent gérer une requête ne peut être spécifié que dynamiquement.

1.3.2. Solution

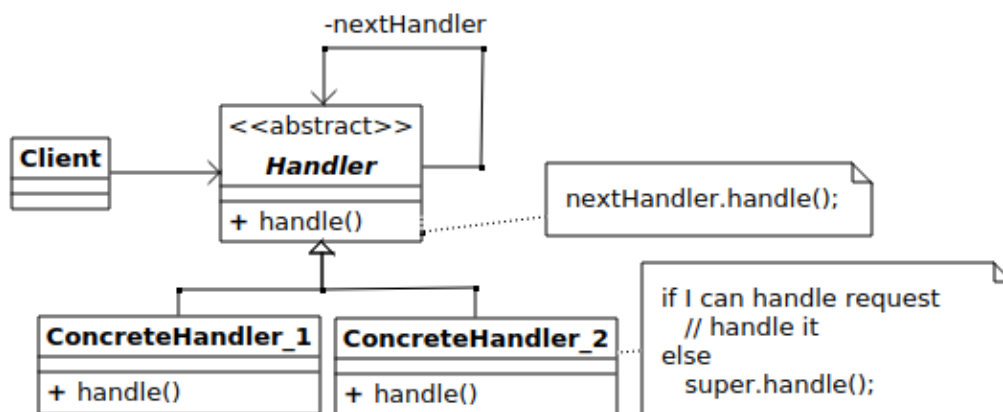
- Chaîner les objets récepteurs de façon à faire passer la requête le long de la chaîne jusqu'à ce qu'un objet soit en mesure d'y répondre.

1.3.3. Exemple : Gestion de l'aide contextuelle

La gestion de l'aide interactive dans une interface graphique est un exemple de chaîne de responsabilité. Dans la plupart des interfaces, l'utilisateur peut obtenir de l'aide sur n'importe quel objet (un bouton, une fenêtre, ...) en positionnant le curseur de la souris sur l'objet puis en pressant la touche F1. Si l'objet possède une aide associée, elle est affichée dans une fenêtre, sinon l'objet demande cette aide à l'objet qui l'englobe.



1.3.4. Structure

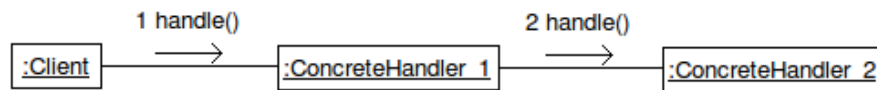


1.3.5. Constituants

- **Client** : l'émetteur de la requête à un objet `ConcreteHandler` de la chaîne.
- **Handler** : une classe abstraite qui définit une interface pour gérer la requête et détient le lien `nextHandler` vers le successeur dans la chaîne.
- **ConcreteHandler** : une classe qui gère la requête dont elle a la charge. Elle traite la requête si elle le peut, sinon elle la transmet à son successeur s'il existe.

1.3.6. Collaboration

Lorsqu'un client émet une requête, celle-ci se propage tout au long de la chaîne jusqu'à ce qu'elle trouve un `ConcreteHandler` pour prendre la responsabilité de la traiter.



1.3.7. Conséquences

- ⊕ Réduction du couplage : chaque objet ne connaît que son successeur dans la chaîne. Cela évite d'avoir à gérer un ensemble de références susceptibles d'apporter de l'aide.
- ⊕ Souplesse accrue dans l'attribution des responsabilités aux objets.
- ⊖ La réponse n'est pas garantie : la requête peut arriver au bout de la chaîne sans jamais avoir été traitée.

1.3.8. Considération d'implémentation

- L'interface `Handler` est représentée par une classe abstraite qui définit la méthode de propagation :

```
public abstract class Handler {
    private Handler _nextHandler;
    public void handle() {
        if (_nextHandler != null) {
            _nextHandler.handle();
        } else {
            /* Code pour indiquer qu'aucun objet n'a pu répondre à la requête */
        }
    }
}
```

- Chaque `ConcreteHandler` rend le service demandé s'il le peut (si `hasService()` est vrai) sinon renvoie la requête vers son successeur en utilisant la méthode de propagation associée à la classe abstraite :

```
public final class ConcreteHandler extends Handler {
    @Override public void handle() {
        if (hasService()) {
            service();
        } else {
            super.handle();
        }
    }
}
```

1.4. Commande (Command, Action)

Promouvoir l'invocation d'une méthode d'un objet comme un objet à part entière.

1.4.1. Problématique

- Besoin de garder un historique des commandes effectuées pour par exemple :
 - Assurer un service de réversion (undo / redo)
 - Réaliser un « film » des actions effectuées.
- Nécessité de traiter des requêtes sur des objets sans rien savoir de l'opération invoquée ou du receveur de la requête.

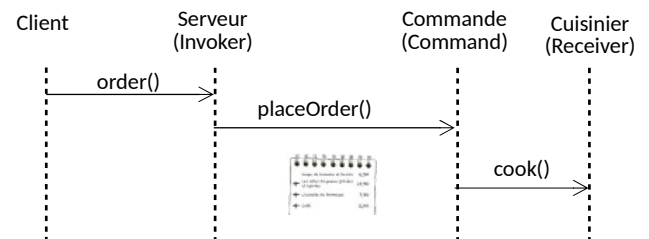
1.4.2. Solution

- Encapsuler l'invocation d'un service dans un objet à part entière.

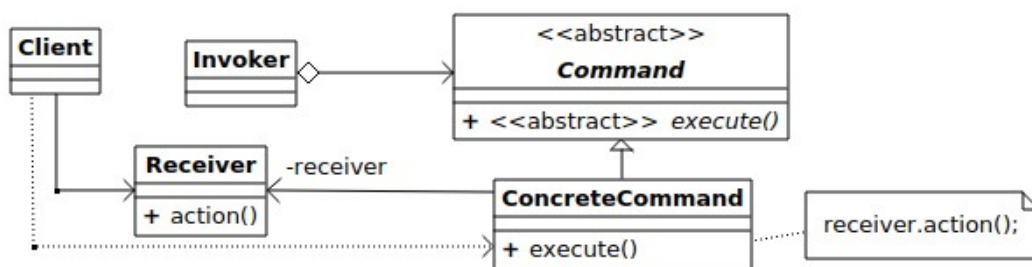
1.4.3. Métaphore : Commande dans un restaurant

La commande dans un restaurant est un exemple du patron Commande. Le serveur prend une commande auprès d'un client et l'enregistre sur le carnet de commande.

La commande est ensuite envoyée en cuisine dans la file courante de commandes. Les commandes sont enregistrées pour être traitées ultérieurement par les cuisiniers.



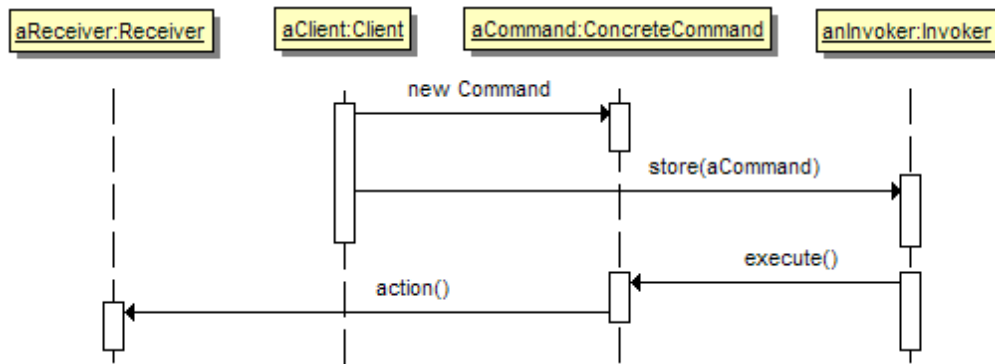
1.4.4. Structure



1.4.5. Constituants

- **Command** : une classe abstraite qui encapsule une opération pour l'exécuter en différé.
- **ConcreteCommand** (eg. Commande de menu client) : définit une astreinte entre un objet récepteur et une action, et concrétise execute() pour l'invocation des opérations du récepteur.
- **Client** (eg. Client) : crée un objet ConcreteCommand et positionne son récepteur (Receiver).
- **Invoker** (eg. Serveur) : demande à la Command d'exécuter sa requête.
- **Receiver** (eg. Cuisinier) : l'objet (ou un des objets) sur lequel s'applique la commande.

1.4.6. Collaborations



1.4.7. Conséquences

- ⊕ Supprime tout couplage entre l'objet qui invoque une opération et celui qui la réalise.
- ⊕ Les commandes sont des objets à part entière.
- ⊕ On peut assembler plusieurs commandes dans une commande Composite. Elles seront exécutées ensemble.

1.4.8. Considération d'implémentation

- Exemple d'une fonction **callback** associée à un bouton en JavaFX qui ajoute un objet vectoriel sur un canvas. L'action est enregistrée dans l'historique des commandes pour permettre une réversion :

```
Button button = new Button("Sélectionner..");
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override public void handle( ActionEvent event ) {
        Element element = event.getSource();
        Commande commande = new AjoutElement(element, this);
        _undoManager.addEdit(commande);
        commande.execute();
    }
});
```

- Remarque : on ajoute dans la commande tous les éléments pour exécuter la commande en différé. Ici, le receveur est `element`, mais nous ajoutons aussi l'objet `this` qui est nécessaire à l'exécution.

1.4.9. API Java

- Les classes `AbstractAction` et `Undoable` et `UndoManager` en Java.

1.5. Composite (Composite)

Créer des objets composites avec une structure arborescente pour permettre au client de traiter de la même façon des objets individuels et des groupements de ces objets.

1.5.1. Problématique

- Nécessité de représenter des hiérarchies de l'individu à l'ensemble.
- Ne pas se soucier de la différence entre combinaisons d'objets et constituants.

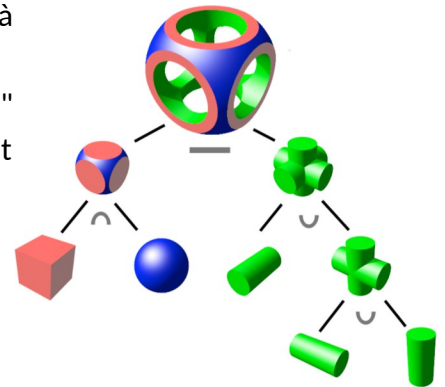
1.5.2. Solution

- Définir une classe abstraite qui spécifie le comportement commun à tous les composés et composants.
- La relation de composition lie un composé à tout objet du type de la classe abstraite.

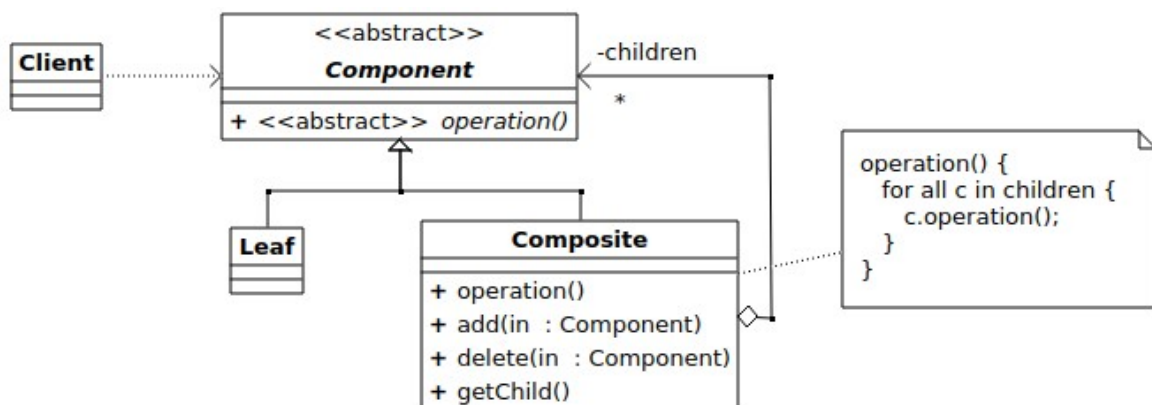
1.5.3. Exemple : Logiciel de conception assistée par ordinateur (CAO)

Une application de CAO se propose de modéliser les objets du monde à partir de formes primitives 3D : sphères, cubes et cylindres.

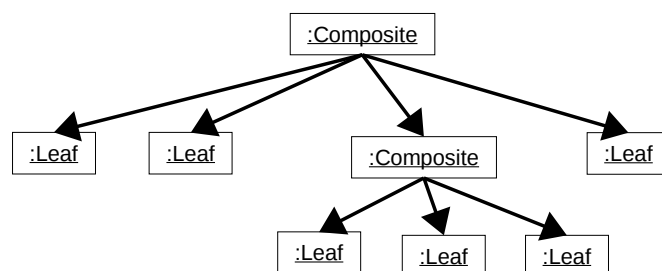
Un langage ensembliste avec les trois opérateurs "intersection", "union" et "moins" permet de traiter des formes 3D complexes. Un objet peut être une forme primitive ou composé de plusieurs primitives.



1.5.4. Structure



Cela permet de créer un arbre abstrait du type :



1.5.5. Constituants

- **Component** (eg. `Forme3D`) : déclare l'interface des objets entrants dans la composition, implante le comportement commun et déclare une interface pour accéder à ses composants enfants.
- **Leaf** (eg. `Sphère`) : représente les objets terminaux dans la composition et définit le comportement des objets primitifs.
- **Composite** (eg. opération union) : définit le comportement des objets composés et stocke les composants enfants sous la forme de listes.

1.5.6. Collaboration

Un client utilise l'interface de la classe `Component` pour manipuler les objets de la structure composite. Si l'objet manipulé est une feuille alors la requête est traitée directement. Si c'est un composite alors la requête est transférée à ses composants en effectuant éventuellement quelques opérations complémentaires avant / après ce transfert.

1.5.7. Conséquences

- ⊕ Définit des hiérarchies de classes qui peuvent être composées de façon récursive.
- ⊕ Simplifie l'accès à la structure pour le client (unification des requêtes).
- ⊕ Rend facile la création / l'ajout de nouveaux composants.

1.5.8. Considération d'implémentation

- L'un des buts est de dissimuler aux clients les classes `Feuille` et `Composite`. Pour atteindre ce but, la classe abstraite `Component` doit présenter les méthodes communes aux deux types de sous-classes.

```
Component c1 = new Leaf();  
Component c2 = new Composite().add(new Leaf()).add(new Leaf());
```

Mais que se passe-t-il quand les deux types de sous-classes définissent des méthodes spécifiques ? Nous avons alors deux choix. Soit nous restons fermes sur l'interface, mais l'utilisation des sous-classes nécessitent alors un déclasserment (« downcasting »), soit nous ajoutons toutes les méthodes spécifiques dans l'interface, mais dans ce cas, il faut définir un comportement par défaut pour les méthodes des classes non concernées. Cette dernière solution est ici préférable au déclasserment qui implique de savoir si on traite une feuille ou un composite.

1.6. Décorateur (Decorator, Wrapper)

Attacher dynamiquement des responsabilités supplémentaires à un objet.

1.6.1. Problématique

- L'objet que vous souhaitez utiliser possède la fonction de base dont vous avez besoin. Mais vous devrez lui ajouter des opérations supplémentaires individualisées parmi un ensemble qui s'exécuteront avant ou après la fonction de base.
- **Ce qui varie** : le comportement d'un service d'un objet en fonction de ses propriétés.

1.6.2. Solution

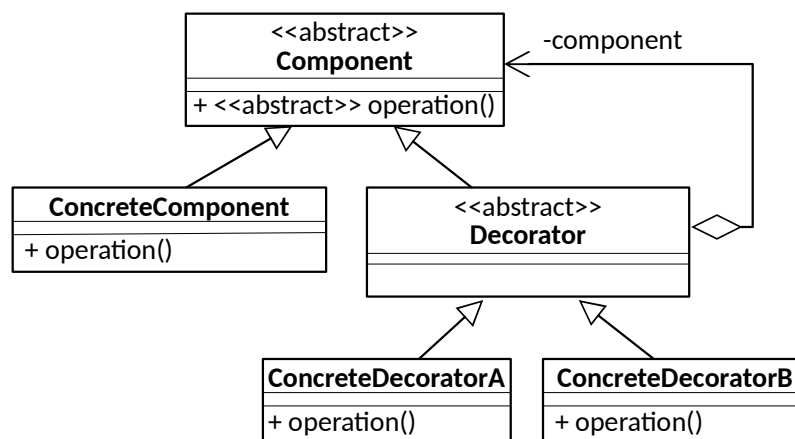
- Permettre l'extension de la fonctionnalité d'un objet sans avoir recours à des sous-classes, mais en utilisant une chaîne de fonctionnalités à partir de l'objet de base.

1.6.3. Exemple : Combattant d'un jeu de rôle

Lors de l'initialisation d'un jeu de rôle, chaque joueur doit construire son personnage en choisissant une liste de caractéristiques physiques, psychologiques et de pouvoirs. La classe `Combattant` possède la méthode `combattre()` dont le comportement dépend des caractéristiques de chaque personnage.



1.6.4. Structure



1.6.5. Constituants

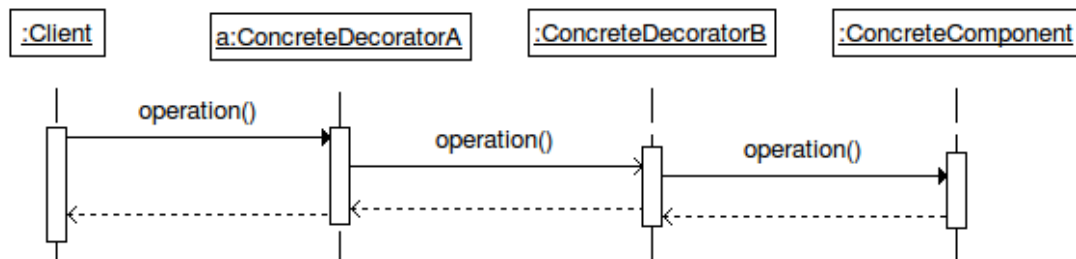
- **Component** : définit l'interface des objets qui peuvent recevoir dynamiquement des responsabilités supplémentaires pour la méthode `opération()`.
- **ConcreteComponent** : définit un objet auquel des responsabilités supplémentaires peuvent être adjointes. La méthode `opération()` fournit la responsabilité de base.
- **Decorator** : gère une référence à un objet composant et définit une interface conforme à celle du Composant.
- **ConcreteDecoratorA** : ajoute une responsabilité à un composant par la méthode `opération()`.

1.6.6. Collaborations

La création d'un objet décoré procède par emballage :

`ConcreteDecorator1( ConcreteDecorator2 (ConcreteComponent))` ;

Le service `operation()` est exécuté sous la forme d'un chaîne par l'intermédiaire du lien `component`. Chaque maillon ajoute sa part de responsabilité avant ou après la transmission de la requête à son successeur. En fin de chaîne, on retrouve le `concreteComponent` qui exécute l'opération de base.



1.6.7. Conséquences

- ⊕ Évite de surcharger les classes situées en haut de la hiérarchie avec des fonctionnalités.
- ⊕ Un décorateur et ses composants ne sont pas identiques. Un décorateur se comporte comme une enveloppe transparente. Un composant décoré n'est pas identique au composant lui-même.
- ⊖ **Conformité d'interface** : l'interface d'un objet Décorateur doit être conforme à l'interface du composant décoré. *On ne peut pas ajouter de méthodes supplémentaires aux décorateurs.*
- ⊖ Il devient impossible d'accéder à la classes de base `ConcreteComponent`.
- ⊖ Il est facile d'ajouter dynamiquement de nouvelles décorations à un composant, mais il est impossible d'en supprimer puisqu'elles sont enfouies dans les couches.

1.6.8. Considération d'implémentation

La création d'un objet décoré procède par emballage de l'objet de base dans les différentes couches de décorations souhaitées :

```
Component a = new ConcreteDecoratorA(new ConcreteDecoratorB(new  
ConcreteComponent())) ;
```

Au minimum, la création doit comporter l'objet de base s'il n'y a pas de décoration :

```
Component a = new ConcreteComponent() ;
```

1.6.9. Exemple dans l'API Java

- Les entrées-sorties en Java (`InputStream`, `FilterInputStream`, `OutputStream`, `FilterOutputStream`).

1.7. État (State)

Adapter le comportement d'un objet quand son état interne change.

1.7.1. Problématique

- Quand le comportement d'un objet dépend de son état, et que ce changement de comportement doit intervenir dynamiquement en fonction de cet état.
- On cherche une implémentation efficace d'un diagramme des cas d'utilisation.
- **Ce qui varie** : la nature des services rendus par un objet en fonction de son état.

1.7.2. Solution

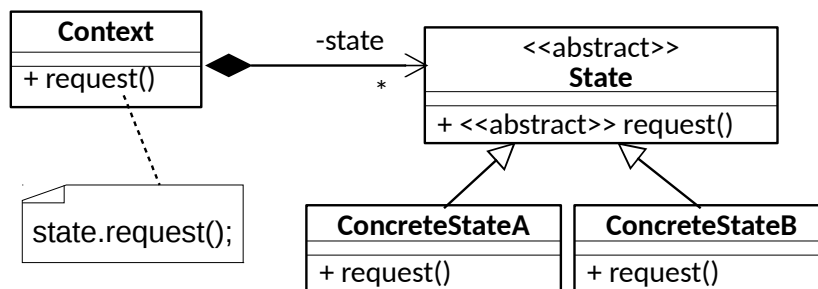
- Externaliser dans une hiérarchie annexe les méthodes qui sont dépendantes d'un état particulier d'un objet.
- Faire autant d'objets annexes que d'états.

1.7.3. Métaphore : La princesse et la grenouille

Dans le conte de fée de la grenouille et du prince, une créature se comporte différemment selon qu'elle se présente sous la forme d'une grenouille ou d'un prince. Au début du conte, la créature est une grenouille puis se transforme en prince charmant lorsque la princesse l'embrasse. La façon de s'exprimer change selon que la créature est une grenouille ou un prince.



1.7.4. Structure



1.7.5. Participants

- **Context** : définit l'interface qu'utilise les clients. Il gère une instance d'une sous-classe `ConcreteState` qui définit l'état en cours et la transition entre les différents états.
- **State** : définit une interface qui encapsule le comportement associé à un état particulier du contexte (ie. la liste des méthodes qui dépendent ensemble de l'état).
- **ConcreteState** : chaque classe implémente un comportement associé à l'état.

1.7.6. Collaborations

- Le `Context` délègue les requêtes spécifiques d'état à l'objet `ConcreteState` courant.
- La classe `Context` est l'interface primaire pour les utilisateurs. Les utilisateurs n'ont pas à traiter directement avec les objets `ConcreteState`.

- C'est soit à `Context`, soit aux sous-classes `ConcreteState` qu'il revient de décider de l'état qui succède à un autre état.

1.7.7. Conséquences

- ⊕ Isole les comportements spécifiques d'état et fait un partitionnement des différents comportements, état par état.
- ⊕ Rend les transitions d'état plus explicites.
- ⊕ Permet de simuler le changement de classe pour un objet.

1.7.8. Considérations d'Implémentation

- Les classes états sont une partie intrinsèque de la classe contexte et elles doivent avoir un accès privilégié aux données de la classe `Contexte`. C'est pourquoi elles doivent être préférentiellement des classes internes en Java et en C++.
- Une solution alternative basée sur des tables : les entrées correspondent aux transitions d'états. Alors que le modèle État se fonde sur le comportement propre d'un état, le modèle des tables se focalise sur les transitions.

1.8. Fabrique (Factory Method)

Définir une interface de création d'objets en laissant à ses sous-classes le soin de choisir la classe à instancier.

Localiser en un seul endroit la création d'instances d'une classe.

1.8.1. Problématique

- Un framework doit standardiser le modèle architectural pour une gamme d'applications tout en permettant aux applications individuelles de définir leurs propres objets de domaine.

1.8.2. Solution

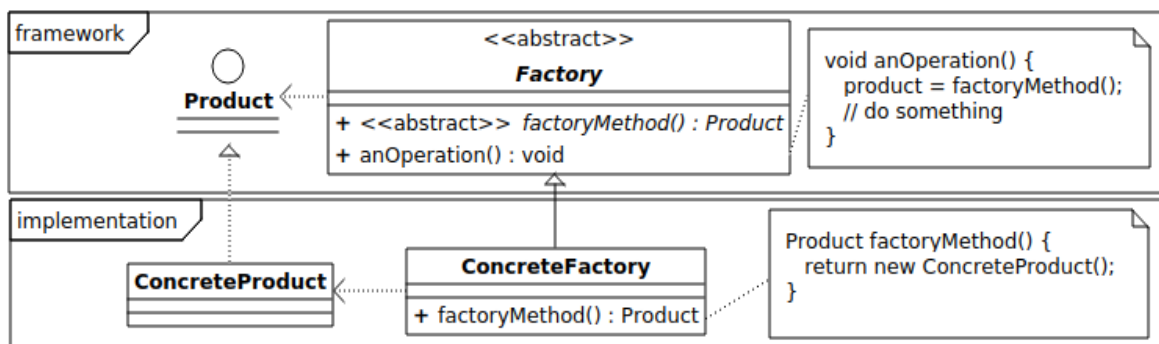
- Encapsuler la création d'instances d'objet dans une classe abstraite dédiée. C'est une conséquence directe du principe de « programmer pour une interface, pas une implémentation ».

1.8.3. Métaphore : Chaîne de pizzeria

Une chaîne de pizzeria propose une carte personnalisée de pizzas dans chacune de ses franchises. Toutefois, c'est toujours la même procédure pour préparer une pizza, seuls les ingrédients changent. On factorise donc la méthode de préparation des pizzas et on spécialise leur création effective dans des classes dérivées. Chaque franchise est incarnée par une classe dérivée qui redéfinit la méthode de création des différentes pizzas de sa carte.



1.8.4. Structure



1.8.5. Participants

- Product** : définit l'interface des objets créés par la fabrique.
- ConcreteProduct** : implémente l'interface `Product` avec un produit concret (eg. une pizza fromage).
- Factory** : déclare l'interface de la fabrique : celle-ci renvoie un type de produit.
- ConcreteFactory** : surcharge la fabrique pour renvoyer une instance d'un produit concret.

1.8.6. Collaboration

- La fabrique confie à ses sous-classes la création d'un produit concret adapté.

1.8.7. Considérations d'implémentation

- Il existe deux variantes principales :

1- La classe `ConcreteFactory` est une classe dérivée qui réifie la méthode de création d'instances.

```
public abstract class Factory {  
    protected abstract Product factoryMethod();  
    public void method( ) {  
        Product concreteProduct = factoryMethod();  
        ...  
    }  
}  
  
public final class ConcreteFactory extends Factory {  
    protected Product factoryMethod() {  
        Product product = new ConcreteProduct();  
        ...  
        return product;  
    }  
}
```

2- La classe `Factory` avec une méthode `factoryMethod()` statique. Le but est de localiser la création des instances à partir d'une seule méthode accessible de n'importe où :

```
public final class Factory {  
    public static Product factoryMethod( ProductId id ) {  
        if (id == ProductId.PRODUCT1) { return new Product1(); }  
        if (Id == ProductId.PRODUCT2) { return new Product2(); }  
    }  
}
```

1.8.8. Conséquences

- ⊕ Respect du principe de responsabilité unique par séparation des responsabilités sur la hiérarchie.
- ⊕ Localisation de la création de différents objets apparentés dans une seule méthode.
- ⊕ Bonne séparation entre le niveau framework et le niveau implémentation.

1.8.9. Exemple d'application

- Ce patron se retrouve de façon inhérente dans les Frameworks. Un Framework est construit avec une classe abstraite de fabrique d'objet, et l'implémentation exige des instances concrètes de cette classe pour créer les objets de l'application.

1.9. Fabrique abstraite (Abstract Factory, Kit)

Fournir une interface pour la création de familles d'objets apparentés, sans qu'il soit nécessaire de spécifier leur classe concrète.

1.9.1. Problématique

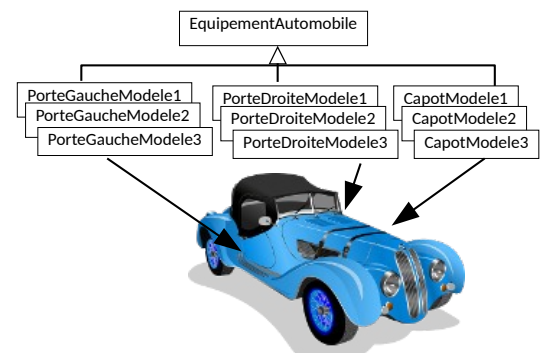
- On souhaite faire cohabiter des familles d'objets. Le choix de la famille est déterminé dynamiquement.
- On ne souhaite pas rendre accessible l'implémentation concrète d'une famille d'objets, le client n'aura accès qu'aux interfaces.

1.9.2. Solution

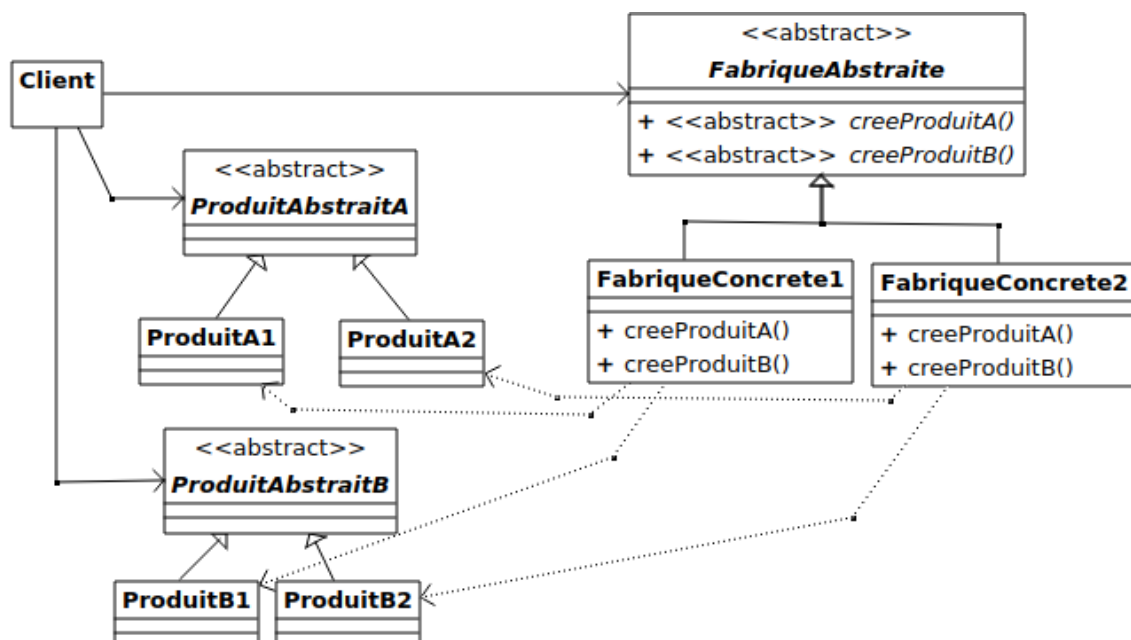
- Coordonner la création dans une classe qui contient les méthodes de création des objets d'une même famille.

1.9.3. Métaphore : Chaîne de montage de voiture

L'assemblage d'une voiture nécessite toujours les mêmes éléments (p. ex. les portes, le capot) mais le modèle de ces éléments changent selon le modèle de voiture. Pour le montage d'une voiture d'un modèle particulier, il est nécessaire de récupérer tous les éléments appartenant à ce même modèle. Il est donc important de localiser la récupération des éléments d'un même modèle de voiture à un seul endroit : une fabrique abstraite d'éléments. Chaque modèle possède sa propre fabrique concrète qui est dérivée de la fabrique abstraite.



1.9.4. Structure



1.9.5. Constituants

- **Fabrique abstraite** : déclare une interface contenant les opérations de création d'objets `Produit`.
- **Fabrique concrète** : implémente les opérations de création des différents produits de la famille.
- **Produit abstrait** : déclare une interface pour un type d'objet `Produit`.
- **Produit concret** : définit un objet `Produit` qui doit être créé par la fabrication concrète correspondante.
- **Client** : n'utilise que les interfaces déclarées par les classes abstraites (`Fabrique` et `Produit`)

1.9.6. Collaborations

Une instance unique de la classe `FabriqueConcrete` est créée à l'exécution et permet au client de créer à la demande les différents produits concrets de cette famille.

1.9.7. Conséquences

- ⊕ Sépare les règles de sélection des objets de leur logique d'utilisation.
- ⊕ Les classes concrètes sont isolées. Il y a encapsulation de la prise en charge et de la création des objets produits.
- ⊕ La substitution des familles de produits est facilitée.
- ⊖ Gérer de nouveaux types de produit est laborieux, parce qu'il faut changer l'interface de la fabrique abstraite et donc de ses sous-classes.

1.9.8. Considérations d'Implémentation

- Les fabriques peuvent être des singletons : une seule interface concrète par classe de produits.
- `FabriqueAbstraite` ne fait que déclarer une interface pour la création de produits. C'est aux sous-classes `FabriqueConcrete` de les créer effectivement.

1.9.9. Exemple dans l'API Java

- La classe `java.awt.Toolkit` est la classe abstraite de toutes les implémentations concrètes du toolkit du système de fenêtrage abstrait AWT. Les sous-classes de la classe `Toolkit` sont utilisées pour lier les différents composants graphiques à des implémentations particulières du toolkit natif de la machine hôte.

Les implémentations concrètes sont par exemple `Metal`, `Windows` ou `Motif`. Pour chaque toolkit, la fabrique présente les méthodes pour créer les éléments graphiques :

- `createMenu()`
- `createScrollbar()`
- `createImage()`, ...

1.10. Façade (Façade)

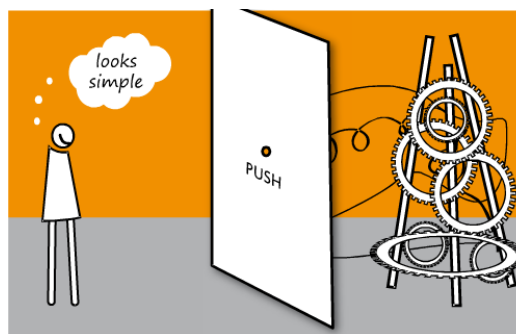
Fournir une interface unifiée pour un ensemble d'interfaces d'un sous-système.

1.10.1. Problématique

- On veut disposer d'une interface simple pour masquer un sous-système complexe.
- Il n'est pas nécessaire d'utiliser toutes les fonctionnalités du sous-système d'origine.
- Il faut découpler le sous-système du client des autres sous-systèmes.

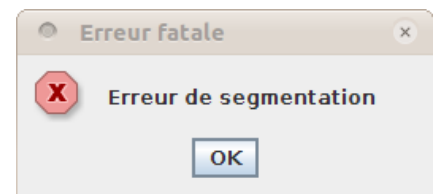
1.10.2. Solution

- Une classe qui fournit des services qui sont rendus en combinant plusieurs services de plusieurs classes.

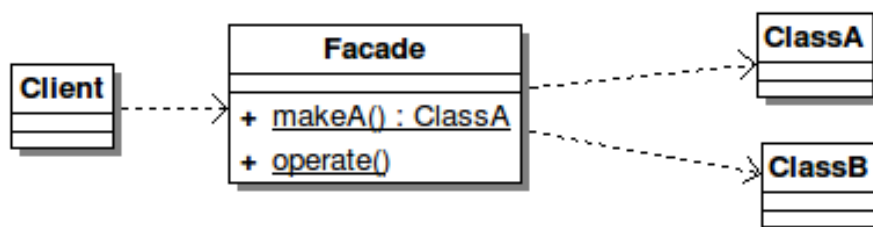


1.10.3. Exemple : création de fenêtre d'alerte en Java Swing

La création des fenêtres d'erreur en Swing est un exemple de façade. La création d'une telle fenêtre nécessite de faire appel à plusieurs classes : `JLabel`, `JPanel`, `JButton`. La classe `JOptionPane` permet alors de simplifier la création de la fenêtre d'alerte en une seule fonction statique `JOptionPane.showMessageDialog()`.



1.10.4. Structure



1.10.5. Constituants

- **Facade** : une classe qui regroupe toutes les méthodes permettant de manipuler simplement le sous-système. Elle connaît les classes du sous-système et délègue le traitement des requêtes du client aux objets appropriés du sous-système.
- **Les classes du sous-système, ClassA, ClassB** : implémentent les fonctionnalités du sous-système et gèrent les travaux assignés par l'objet façade. Ces classes ne connaissent pas la façade.

1.10.6. Collaborations

- Le client du sous-système communique avec la façade et seulement avec la façade.
- La façade répercute la demande de services auprès des objets appropriés du sous-système.

1.10.7. Conséquences

- ⊕ Masque au client la multiplicité des communications et des liens de dépendance entre les constituants du sous-système.
- ⊕ Couplage faible entre client et sous-système.
- ⊕ Très utile pour la communication entre paquets développés par différentes équipes indépendantes au sein d'un même projet.
- ⊕ N'empêche pas les applications d'utiliser directement des classes du sous-système.

1.10.8. Considérations d'Implémentation

- Les façades sont généralement implémentées par des **classes utilitaires**, c'est-à-dire qu'elles ne possèdent que des méthodes statiques et le constructeur est désactivé (mis en mode privé).

```
public final class Facade {  
    private Facade() { }  
    public static ClassA makeA() { ... }  
    public static void operator() { ... }  
}  
ClassA a = Facade.makeA();
```

- Il est toutefois possible de faire une classe abstraite qui se dérive en sous-classes pour chaque variante de dérivation de la façade.

1.11. Itérateur (*Iterator*, *Cursor*)

Séparer le mécanisme de parcours d'un agrégat de sa représentation.

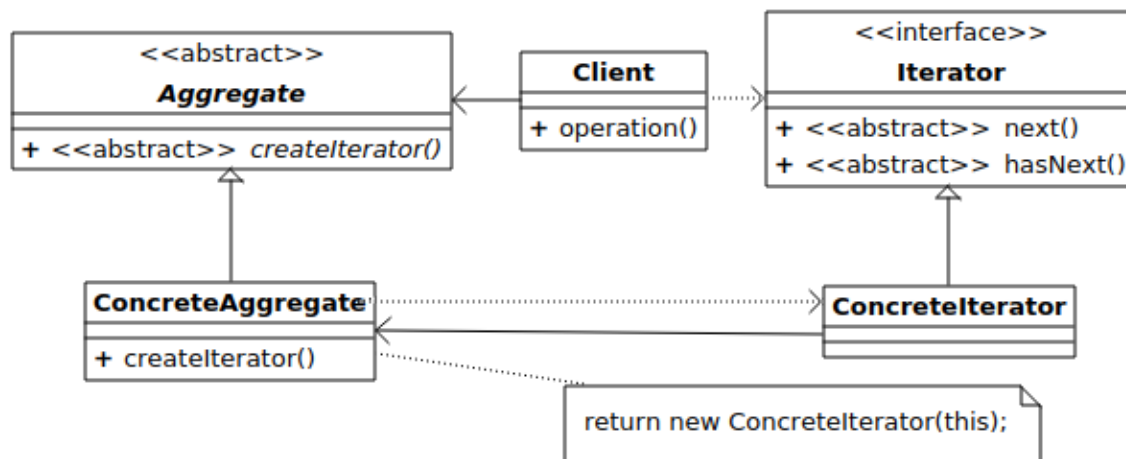
1.11.1. Problématique

- Accéder au contenu d'un objet agrégat sans révéler la structure interne de l'agrégat.
- Gérer simultanément plusieurs parcours dans d'un même agrégat (file, pile, ordonné, aléatoire).
- Offrir une interface uniforme pour le parcours de diverses structures d'agrégats (pour permettre ainsi une itération polymorphe).

1.11.2. Solution

- Déléguer le parcours d'un agrégat dans un objet annexe qui sera généré à la demande par l'agrégat.

1.11.3. Structure



1.11.4. Constituants

- **Iterator** : définit une interface pour accéder aux éléments et parcourir la liste.
- **Concrete Iterator** : assure le suivi de l'élément courant lors de la traversée d'un agrégat.
- **Aggregate** : définit une interface pour la création d'un objet itérateur.
- **ConcreteAggregate** : Implante l'interface de création de l'itérateur afin de retourner l'instance adéquate d'un itérateur concret.

1.11.5. Conséquences

- ⊕ Aucun risque pour la structure interne de l'agrégat.
- ⊕ Permet pour des accès variés, au gré des projets.
- ⊕ Permet plusieurs parcours simultanés sur un même agrégat.

1.11.6. Considérations d'Implémentation

Allocation des itérateurs

- Les itérateurs doivent être alloués dynamiquement par une Fabrique avec la méthode

`createIterator()`.

- La responsabilité de la suppression incombe au client. Pour cela, le patron Procuration fournit un remède : on utilise une procuration allouée dans la pile et le destructeur élimine l'itérateur effectif.
- Les itérateurs doivent avoir un accès privilégié à l'agrégat. Pour ne pas rompre l'encapsulation :
 - En C++, l'itérateur est une classe amie (friend) de son agrégat.
 - En Java, l'itérateur sera une classe interne de son agrégat.

Itérateur actif / itérateur passif

- Si le client contrôle les itérations (par exemple à partir d'une méthode `next()`), on parle d'itérateur actif. Si c'est l'itérateur qui exerce ce contrôle, on parle d'itérateur passif (eg, les streams en java).

Variante : Curseur

- L'itérateur n'est pas le seul endroit où peut se définir l'algorithme de parcours. L'agrégat lui-même peut le faire et n'utiliser l'itérateur que pour le seul stockage de l'état des itérations. Ce type d'itérateur s'appelle un curseur. Toutefois, il ne peut y avoir qu'un seul parcours simultané.

Opérations additionnelles

- Les agrégats ordonnés peuvent avoir une opération `previous()`. Dans une collection indexée, on peut rencontrer `goto()` qui utilise des critères spécifiques.

Itérateur pour composite

- La structure étant récursive, un itérateur externe doit conserver un cheminement d'accès dans le composite. Un itérateur interne peut simplement conserver sa position interne, et en s'appelant récursivement, il conserve implicitement le chemin dans les piles d'appels.

Variante : Itérateur nul

- C'est un itérateur dégénéré utilisé pour les conditions aux limites : l'opération `hasNext()` retourne toujours faux... Il est par exemple, défini dans les structures arborescentes où les feuilles n'ont pas à parcourir leurs enfants.

1.11.7. API Java

- L'interface des itérables : `Iterable` (qui définit la méthode `iterator()`).
- L'interface des itérateurs d'agrégat : `Iterator`.

1.12. Médiateur (*Mediator*)

Définir un objet qui encapsule les modalités d'interaction entre divers objets.

1.12.1. Problématique

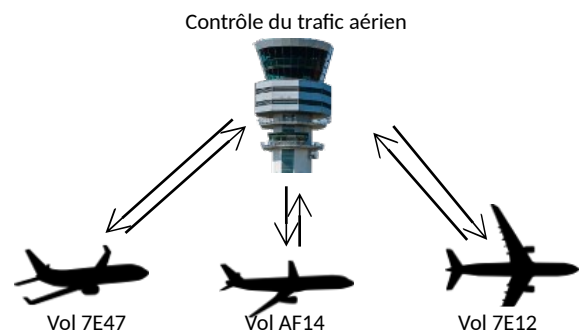
- Quand les objets d'un ensemble dialoguent de façon bien définie, mais très complexe. Le résultat des interdépendances est non-structuré et difficile à appréhender.
- Quand la réutilisation d'un objet est très difficile, du fait qu'il fait référence à beaucoup d'autres objets et communique avec eux.
- Quand le contrôle d'exécution d'un ensemble d'objet doit être centralisé.

1.12.2. Solution

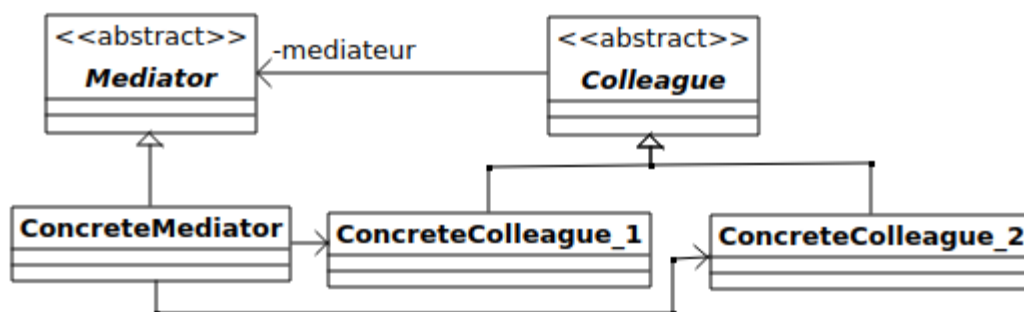
- Créer un objet qui organise la communication entre un ensemble d'objets.

1.12.3. Métaphore : Tour de contrôle d'un aéroport

La tour de contrôle d'un aéroport est un exemple de médiateur. Les pilotes des avions en approche d'atterrissage ou au décollage communiquent avec la tour de contrôle au lieu de communiquer directement entre eux pour organiser le trafic. La décision de qui peut décoller et qui peut atterrir est prise par la tour de contrôle. Il est important de noter que la tour ne contrôle pas le vol de chaque avion mais uniquement l'organisation entre les avions.



1.12.4. Structure



1.12.5. Constituants

- **Médiateur** : définit une interface pour communiquer avec les collègues.
- **ConcreteMediator** : réalise le comportement coopératif en coordonnant les objets collègue. Il connaît et gère ses collègues.
- **Colleague** : chaque classe Collègue connaît son objet Médiateur. Chaque Collègue s'adresse à son médiateur et n'a pas besoin de connaître les collègues concernés.

1.12.6. Collaborations

Les Collègues émettent et reçoivent des requêtes de l'objet Médiateur.

Le Médiateur implémente le comportement coopératif en assurant le routage entre les collègues pertinents.

1.12.7. Conséquences

- ⊕ Pas de limitation du nombre de sous-classes.
- ⊕ Réduction du couplage entre collègues : le médiateur favorise un couplage faible.
- ⊕ Simplification des protocoles objets : remplacement de relation plusieurs à plusieurs par des relations 1 à plusieurs, plus facile à appréhender, à maintenir, et à faire évoluer.
- ⊕ Formalisation de la coopération des objets : en encapsulant le comportement coopératif des objets, on ne mélange pas comportement coopératif et comportement individuel.
- ⊖ Centralisation du contrôle : le Médiateur peut devenir très complexe à faire évoluer et à maintenir.

1.12.8. Considérations d'Implémentation

- Omission de la classe abstraite Médiateur. Il n'est pas nécessaire de définir une classe abstraite Médiateur quand les collègues travaillent avec un seul Médiateur.

1.13. Memento (Memento)

Sans violer l'encapsulation, acquérir et délivrer à l'extérieur une information sur l'état interne d'un objet, afin que celui-ci puisse être rétabli plus tard dans cet état.

1.13.1. Problématique

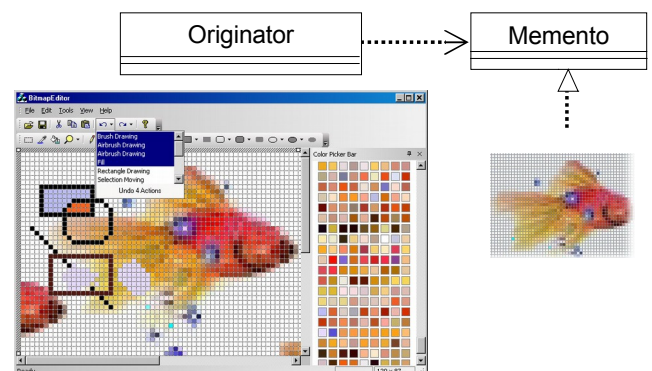
- Quand un instantané de tout ou d'une partie d'un état doit être sauvegardé et restauré.
- Quand l'utilisation d'une interface directe pour atteindre l'état conduirait à révéler des détails de l'implémentation et rompre l'encapsulation.

1.13.2. Solution

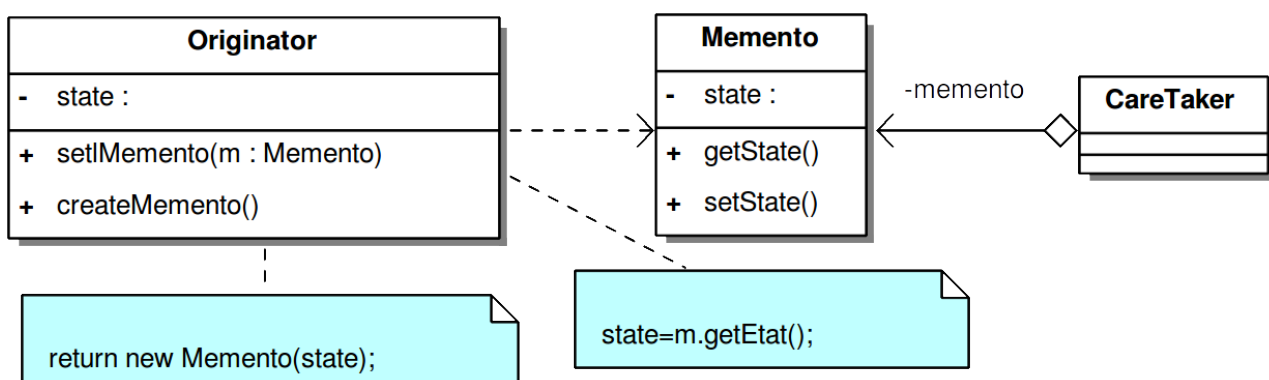
- Délocaliser le stockage de l'état interne d'un objet vers un autre objet (memento).
- Le memento propose une interface large pour l'objet créateur et une interface étroite pour les objets extérieurs afin de ne pas rompre l'encapsulation.

1.13.3. Exemple : Logiciel de dessin bitmap

Les logiciels de dessin bitmap stockent différents instantanés du dessin à chaque modification pour réaliser le service réversion (Undo/Redo).



1.13.4. Structure



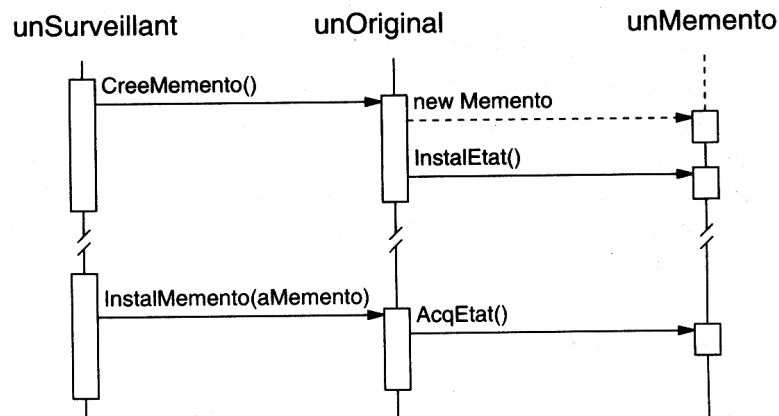
1.13.5. Constituants

- **Memento** : mémorise l'état interne de l'objet *Originator*. C'est l'objet *Originator* qui décide des informations à stocker. Le *Memento* protège des incursions d'autres que *Originator*. Il dispose d'une interface "large" visible par *Originator*. Il dispose d'une interface "étroite", visible du *Caretaker*.

- **Originator** : crée un objet `Memento` contenant un instantané de son état interne. Il utilise `Memento` pour restaurer son état interne.
- **Caretaker** : est responsable de la sauvegarde du `Memento` mais n'agit jamais sur le `Memento`, ni ne l'examine.

1.13.6. Collaborations

Un surveillant (`Caretaker`) réclame un memento (`Memento`) à un objet original (`Originator`), le conserve et le renvoie en cas de besoin. Les mementos sont passifs. Seul l'auteur qui a créé le memento affectera et récupérera son état.



1.13.7. Conséquences

- ⊕ Préservation des frontières d'encapsulation.
- ⊕ Simplification de l'objet `Originator` qui est directement déchargé de la gestion mémoire de ses différents états.
- ⊖ L'utilisation de memento peut être coûteuse si de grands volumes d'information doivent être sauvegardés (envisager un stockage incrémental).
- ⊖ Coûts cachés de la surveillance des mementos. Un surveillant est responsable de la suppression des mementos qu'il surveille, mais il n'a pas connaissance du nombre d'états stockés dans le memento.

1.13.8. Considérations d'Implémentation

- En C++, `Memento` sera une classe amie de `Originator` pour accéder à l'interface large. Seule l'interface étroite pourra être publique.
- En java, ce sera souvent une classe interne.
- Stockages incrémentaux des états : Si les mementos sont créés et renvoyés selon une séquence ordonnée, seules les différences entre 2 états peuvent être stockées.

1.14. Monteur (Builder)

Créer des objets complexes dont les différentes parties doivent être créées suivant un ordre ou un algorithme spécifique.

1.14.1. Problématique

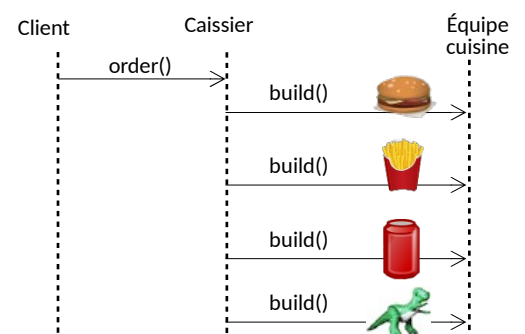
- La création d'un objet s'effectue par parties qui peuvent être variables.
- La création d'un objet nécessite beaucoup d'arguments dont certains sont optionnels ou ont plusieurs variations en termes de type. On veut éviter l'anti-pattern « Constructeur télescopique » (voir page 56).
- **Ce qui varie** : les parties d'un composant générique.

1.14.2. Solution

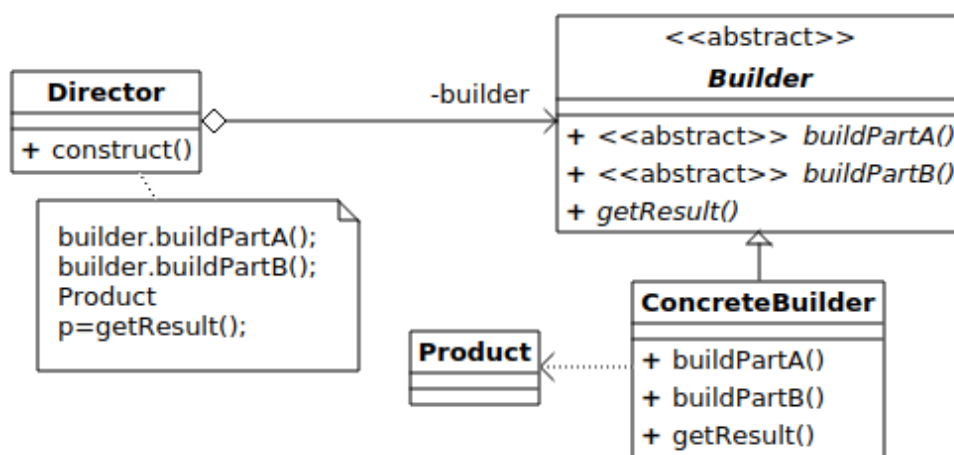
- Encapsuler la construction d'un objet par la liste de ses parties et laisser la responsabilité de l'algorithme de création du produit à une classe externe.

1.14.3. Métaphore : Chaîne de restauration rapide

La composition d'un menu enfant dans un fast-food utilise un monteur. Le restaurant propose des menus tout fait dans lesquels la composition est fixée. Chaque menu est construit à la demande avec un sandwich, un accompagnement, une boisson et/ou un jouet déterminés.



1.14.4. Structure



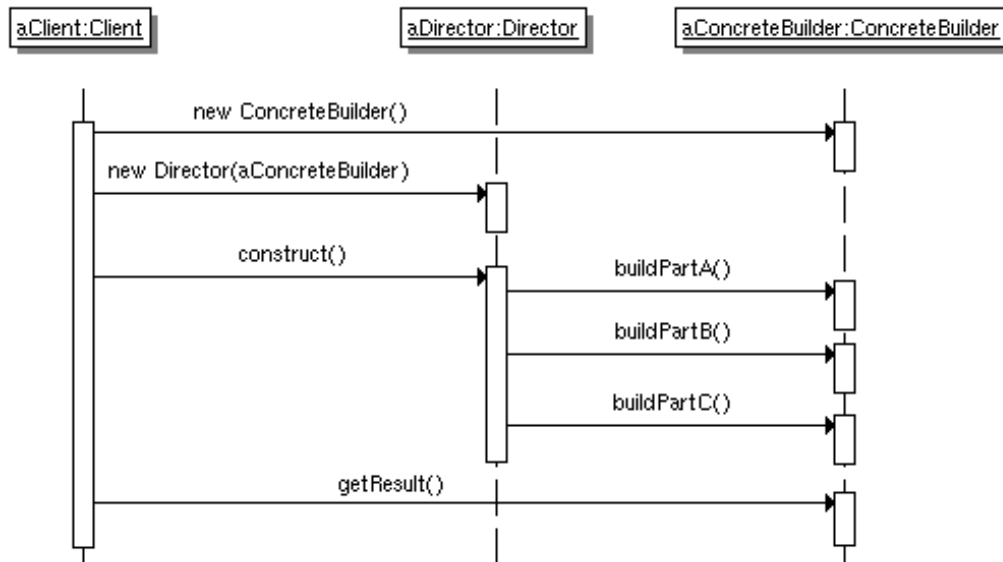
1.14.5. Constituants

- **Builder** (eg. Équipe cuisine) : précise une interface abstraite pour la création de parties d'un objet Produit.
- **ConcreteBuilder** : construit et assemble des parties du produit par implémentation de l'interface *Builder*. Il fournit aussi une interface pour la récupération du résultat.
- **Director** (eg. Cassier) : construit un objet en utilisant l'interface de Monteur.

- **Product** (eg. Menu) : représente l'objet complexe en cours de construction.

1.14.6. Collaborations

- Le Client crée l'objet Directeur et le compose avec l'objet Monteur choisi.
- Le Directeur utilise le Monteur chaque fois qu'une partie du produit doit être construite.
- Le Monteur gère les requêtes du Directeur et le Produit en parties.
- Le Client récupère le Produit auprès du Monteur.



1.14.7. Variante

L'objet Directeur n'est pas toujours présent et c'est même la variante la plus utilisée. L'implémentation classique d'un monteur interactif dans lequel les parties ne sont pas figées dans un Directeur, mais choisies par le client, utilise une suite de méthodes chaînées, suivie d'un `build()` final pour construire et récupérer l'objet créé :

```
Product p = new ConcreteBuilder().buildPartA().buildPartC().build();
```

Il faut donc faire des méthodes qui retournent une référence sur le monteur :

```
public Builder buildPartA() {
    ...
    return this;
}
```

Cette variante est la réponse à l'anti-patron « Constructeur Télescopique ».

1.14.8. Exemple dans l'API Java

- `StringBuilder`

1.15. Observateur (Observer)

Définir une dépendance "1 à plusieurs" entre un objet et ses multiples observateurs de manière à ce que les observateurs soient automatiquement informés d'un changement d'état de l'objet.

1.15.1. Problématique

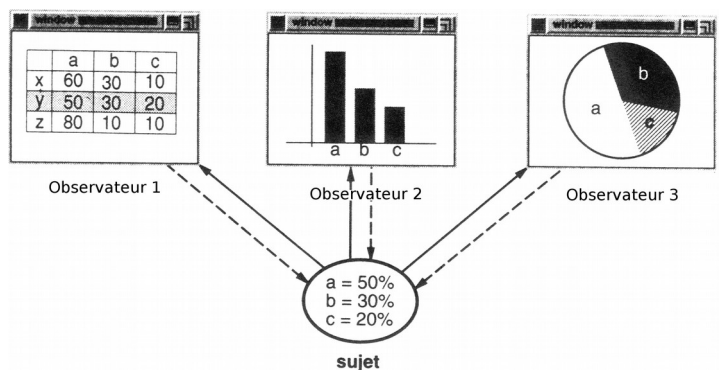
- Informer une liste variable d'objets qu'un événement a eu lieu.
- **Ce qui varie** : la liste des objets dépendant des modifications d'un objet de référence.

1.15.2. Solution

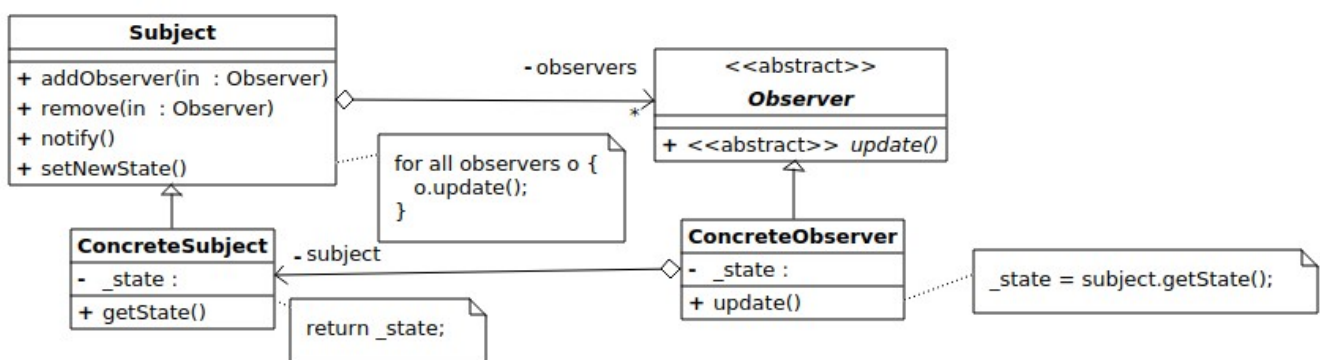
- Le modèle de base est constitué d'un sujet (détenteur de données) et d'observateurs. Tous les observateurs reçoivent une notification chaque fois que le sujet subit une modification.
- Les observateurs doivent s'**enregistrer** dynamiquement auprès du client.

1.15.3. Exemple : Tableur

Un tableur permet de visualiser des données sous plusieurs formes : un calque contenant les chiffres et des représentations graphiques (eg. camembert, histogramme) que l'utilisateur choisit d'ajouter. Dès que l'on change une donnée dans le calque, le calque et les représentations graphiques présentes sont immédiatement mises à jour.



1.15.4. Structure

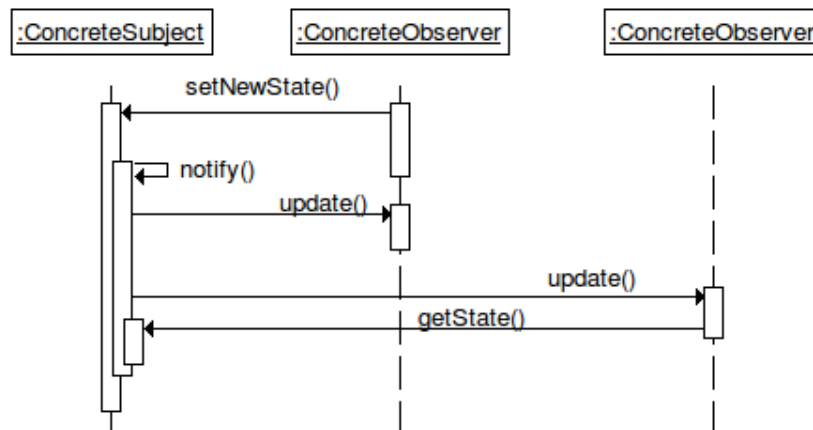


1.15.5. Composants

- **Subject** (eg. Tableur) : il connaît les objets observateurs car ces derniers s'enregistrent auprès de lui. Il doit les avertir dès qu'un événement a eu lieu.
- **Observer** (eg. Histogramme) : Il est responsable de son inscription auprès du sujet en utilisant la méthode `addObserver()` du sujet. Il définit une interface de mise à jour pour les objets qui doivent être notifiés lors d'un changement d'état par la méthode `update()`.

- **ConcretSubject** : il mémorise les états qui intéressent les objets observateurs concrets et envoie une notification à ses observateurs lorsqu'il change d'état par la méthode `notify()`.
- **ConcreteObserver** : Il fait l'implémentation de l'interface de mise à jour de l'observateur pour conserver la cohérence de son état avec le sujet.

1.15.6. Collaborations



1.15.7. Conséquences

- ⊕ Forte réduction du couplage entre l'observé et les observateurs.
- ⊕ Il est possible d'ajouter et de supprimer à chaud des observateurs.

1.15.8. Considérations d'Implémentation

Il existe deux modèles du processus de notification :

- **push** : le sujet envoie aux Observateurs des informations détaillées sur les modifications. Dans ce cas, le sujet risque de donner aux observateurs des informations sur des événements superflus si certains observateurs ne sont intéressés que par un sous-ensemble d'événements seulement.

```
public void update( Data data );
```

- **pull** : le sujet n'envoie aucune l'information et ce sont les observateurs qui réclament les informations complémentaires par des méthodes de type `getState()` du sujet en passant une référence sur l'observé dans les paramètres de la méthode `update()`. Dans ce cas, il faut avoir recours à des accesseurs ce qui augmente le couplage entre les observateurs et l'observé.

```
public void update( Subject s );
```

1.15.9. Exemple dans l'API Java

- La classe `Observable` et l'interface `Observer`. À noter que la méthode `update()` définie dans l'interface `Observer` permet les deux modèles pull et push.
 - `public void update(Observable o, Object data);`
- Les `listeners` : tous les objets graphiques (bouton, boîte de dialogue, champs de texte) de la bibliothèque JavaFX fonctionnent sur le modèle d'observateur.

1.16. Patron de méthode (Template Method)

Définir le squelette de l'algorithme d'une opération en déléguant le traitement de certaines étapes variables à ses sous-classes.

1.16.1. Problématique

- On souhaite construire plusieurs variations d'un même algorithme ayant une partie commune.
- On souhaite des comportements par défaut (opérations socles) et des comportements qui peuvent être surchargés dans les classes concrètes.
- **Ce qui varie** : les parties d'un service (algorithme).

1.16.2. Solution

- Un patron de méthode définit les différentes étapes d'un algorithme dont certaines sont abstraites (car spécialisées), lesquelles seront surchargées de façon concrète dans des sous-classes spécialisées. Chaque classe concrète donne alors une implémentation complète de l'algorithme.

1.16.3. Métaphore : Jeu de société

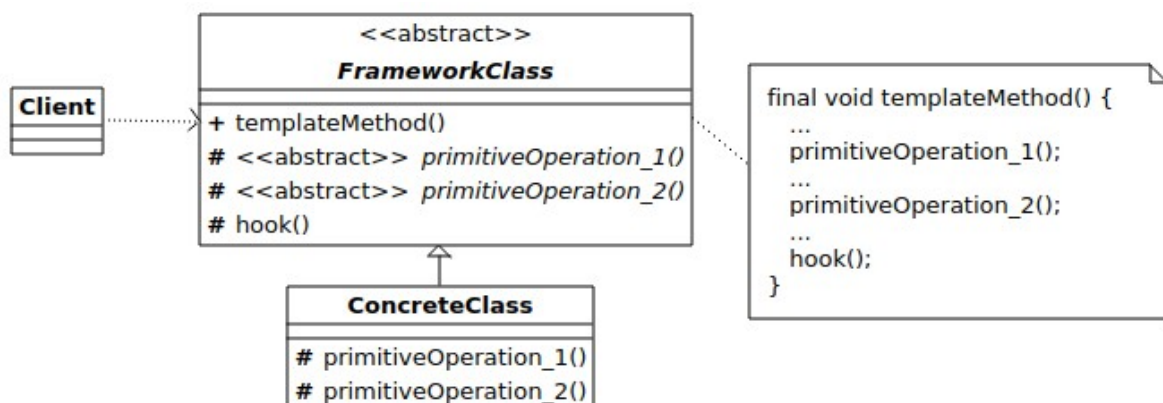
Les étapes d'un jeu de société (eg. Monopoly, les petits chevaux) ont toujours les mêmes :

- D'abord initialiser le jeu.
- Puis, tant que la partie n'est pas terminée faire jouer les joueurs à tour de rôle.
- À la fin, proclamer le vainqueur.

Toutefois, chacun des jeux possède sa propre façon de faire l'initialisation du jeu, de déterminer quand la partie est terminée et de décider du vainqueur.



1.16.4. Structure



1.16.5. Constituants

- **FrameworkClass** : fournit le squelette d'un algorithme utilisant certaines opérations abstraites

spécialisées dans des sous-classes concrètes (patron de méthode) et définit aussi des opérations primitives abstraites, rendues concrètes par des sous-classes destinées à implémenter diverses étapes d'un algorithme.

- **ConcreteClass** : implante les opérations concrètes pour effectuer les étapes d'un algorithme.
- **Client** : c'est le client qui choisit l'implémentation qu'il souhaite parmi les classes concrètes.

1.16.6. Participants et collaborateurs

La classe concrète s'en remet à la classe abstraite pour implanter les parties invariantes de l'algorithme.

1.16.7. Considérations d'Implémentation

- Les opérations appelées par le patron de méthode sont **abstraites** et en mode "**protected**".
- La méthode `templateMethod()` qui définit l'algorithme doit être protégée contre la surcharge (`final` en Java).
- Il est possible d'ajouter une méthode `hook() {}` concrète mais vide dans la classe de base, qui peut être redéfinie par des classes dérivées pour ajouter une fonctionnalité optionnelle.

```
public abstract class FrameworkClass {
    final void templateMethod() {
        ...
        primitiveOperation();
        ...
        hook() { }
        ..
    }
    protected abstract void primitiveOperation();
}

public class ConcreteClass extends FrameworkClass {
    @Override
    protected void primitiveOperation() { .. }
    @Override
    protected void hook() { .. }
}
```

1.17. Poids mouche (Flyweight)

Assurer en mode partagé le support d'un grand nombre d'objets à fine granularité qu'il serait trop coûteux en mémoire de tous les instancier.

1.17.1. Problématique

Les conditions suivantes doivent être réunies :

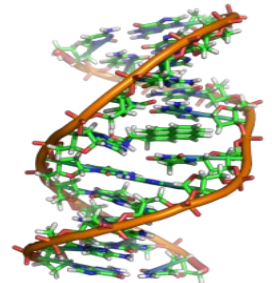
- l'application utilise un grand nombre d'objets ;
- les coûts de stockage sont élevés du fait d'une réelle quantité d'objets ;
- les propriétés des objets peuvent être considérés comme extrinsèques (ie, ne dépendent pas de l'instance).

1.17.2. Solution

- Créer une fabrique d'objets qui contrôle l'unicité de chaque type d'objet. L'état intrinsèque (invariant) du type d'objet est stocké. L'état extrinsèque (variant) dépend du contexte du poids mouche est fourni comme argument des méthodes.

1.17.3. Métaphore : Une chaîne d'ADN

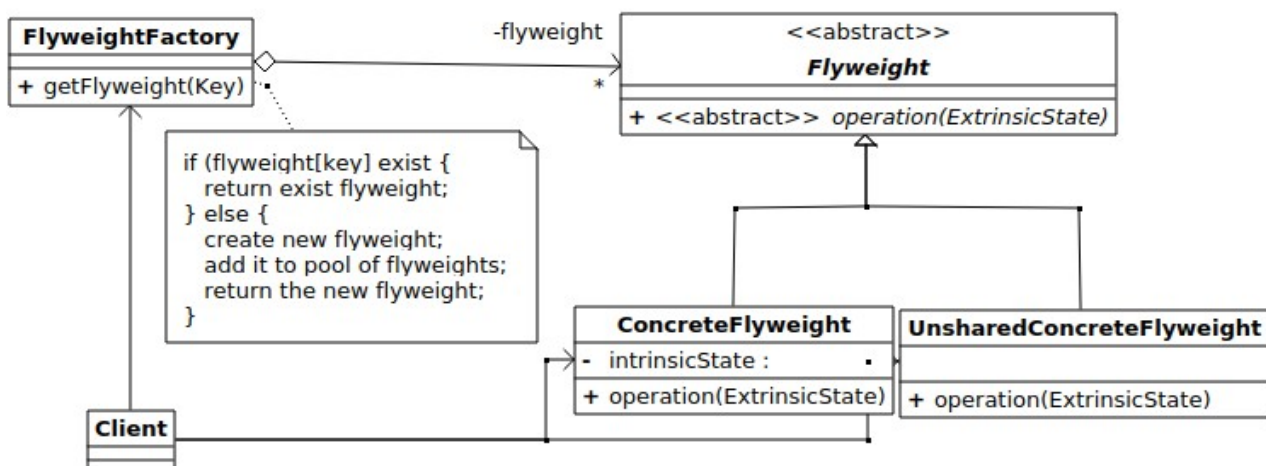
Une chaîne d'ADN est composée de plusieurs millions de nucléotides. Mais puisque les nucléotides ne sont que de quatre types, Adenine, Guanine, Cytosine et Thymine, il est possible de représenter une chaîne d'ADN avec seulement quatre instances de nucléotide obtenus à la demande auprès d'une usine de fabrication de nucléotides.



1.17.4. Exemple

Un exemple classique est celui des caractères manipulés dans un traitement de texte. Un texte aussi long qu'il soit peut-être finalement implémenté avec une liste fixe de caractères. Chaque caractère correspond à une instance ayant une police de caractères, une taille de caractères, et d'autres données de formatage qui sont des propriétés extrinsèques.

1.17.5. Structure



1.17.6. Constituants

- **Flyweight** : déclare une interface à travers laquelle les poids mouches peuvent recevoir les états extrinsèques et agir sur eux.
- **ConcreteFlyweight** : implémente l'interface d'un poids mouche. Ce sont toutes les classes partageables.
- **UnsharedConcreteFlyweight** : toutes les sous-classes de poids mouche ne sont pas nécessairement partagées. On garde chaque occurrence des poids mouches non partageables dans des classes.
- **FlyweightFactory** : crée et gère les objets poids mouche.
- **Client** : gère une référence aux poids mouches, calcule et mémorise l'état extrinsèque des poids mouches.

1.17.7. Collaborations

- L'état intrinsèque est stocké dans l'objet `PoidsMoucheConcret`. L'état extrinsèque est calculé ou stocké par le client puis communiqué au poids mouche lors de l'appel de ses services.
- Les clients obtiennent des objets poids mouche par l'intermédiaire d'une fabrique. Celle-ci garantit qu'une seule instance est retournée par type lorsqu'il s'agit d'un objet avec état extrinsèque. Par contre, elle retourne une instance par appel lorsqu'il s'agit d'un objet à état intrinsèque.

1.17.8. Conséquences

- ⊕ Réduire le nombre d'instances stockées en mémoire.
- ⊕ Centraliser l'état de beaucoup d'objets virtuels à un seul endroit.
- ⊖ L'utilisation des poids mouches peut introduire des coûts à l'exécution résultant du transfert et du calcul/stockage des états extrinsèques. De tels coûts sont compensés par l'économie de place mémoire réalisée.

1.17.9. Considérations d'Implémentation

- Il est nécessaire d'avoir des moyens de comptage et de récupération mémoire pour les langages sans machine virtuelle (par exemple les *smart Pointers* en C++).

1.17.10. Modèles apparentés

- Fabrique et Singleton : l'unicité est garantie par la fabrique de poids mouche.

1.18. Pont (Bridge)

Découpler une abstraction de son implémentation afin que les 2 éléments puissent être modifiés indépendamment l'un de l'autre.

1.18.1. Problématique

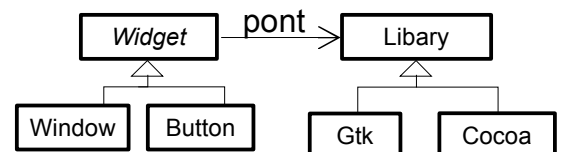
- Quand une abstraction peut avoir toute une série d'implémentations possibles, le moyen usuel pour les fédérer est d'utiliser l'héritage : une classe abstraite définit l'interface de l'abstraction et les sous-classes représentent toutes les combinaisons possibles entre les variantes de l'abstraction et les variantes de l'implémentation.
 - Cette approche n'est pas suffisamment souple : l'héritage lie l'implémentation de façon permanente, ce qui rend difficile la modification, l'extension, la réutilisation des abstractions et des implémentations indépendamment l'une de l'autre.
 - Le couplage conduit à une explosion du nombre de classes. Il faut créer un nombre de classes égal à (abstractions × implémentations) pour couvrir toutes les variations possibles.
- **Ce qui varie** : les abstractions et les implémentations peuvent évoluer indépendamment l'une de l'autre.

1.18.2. Solution

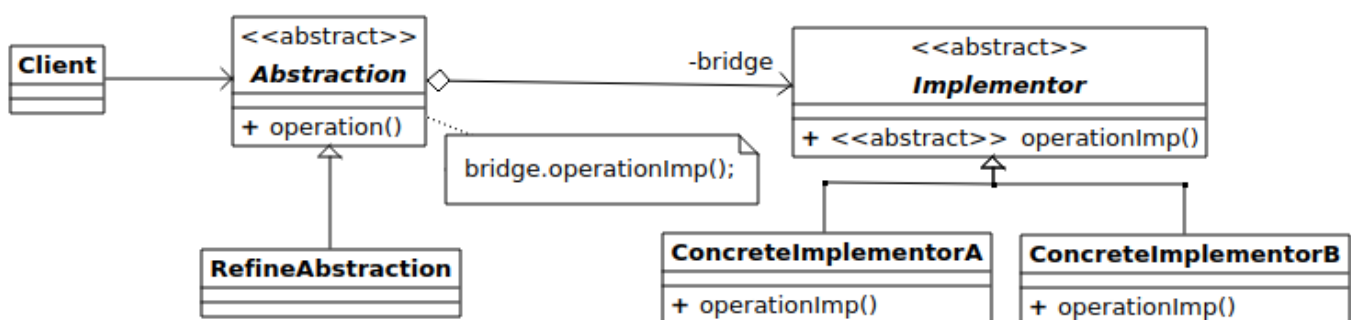
- Le patron Pont traite ces problèmes en séparant l'abstraction et son implémentation et en les reliant par une association : le pont. Il y a deux hiérarchies de classes distinctes.

1.18.3. Exemple : bibliothèque d'interface graphique

Pour qu'une application soit portable d'une plateforme à une autre, il ne faut pas que les éléments graphiques (ascenseurs, fenêtres, boutons) soient codés en dur pour une bibliothèque donnée. Les éléments graphiques possèdent leur propre hiérarchie et leur implémentation avec une bibliothèque graphique donnée est faite dynamiquement en liant un élément à une bibliothèque. Cela évite d'avoir à créer autant de classes que de combinaisons : élément graphique × bibliothèque (eg. WindowGtk et WindowCocoa, ButtonGtk, ButtonCocoa).



1.18.4. Structure



1.18.5. Constituants

- **Abstraction** (eg. Interrupteur) : définit l'interface de l'abstraction ; gère une référence à un objet de type Implementor.
- **RefineAbstraction** (eg. Bouton presseur) : enrichit l'interface définie par l'abstraction pour constituer une classe concrète.
- **Implementor** (eg. Appareil) : définit l'interface des classes d'implémentation. En général, elle définit des opérations primitives.
- **ConcreteImplementor** (eg. Télévision) : réalise concrètement l'implémentation de l'interface.
- **bridge** : lien d'association qui fait le pont entre l'abstraction et l'implémentation.

1.18.6. Collaborations

- À la création de l'abstraction, il faut lier l'instance à une instance de l'implémentation.
- L'abstraction transmet les requêtes du client à l'objet implémentation qui lui est lié.

1.18.7. Conséquences

- ⊕ Le découplage des implémentations à partir des objets les utilisant accroît les possibilités d'extension.
- ⊕ Les clients ignorent les problèmes d'implémentation.

1.19. Procuration (Proxy, Surrogate)

Fournir un remplaçant d'un objet pour en contrôler l'accès.

1.19.1. Problématique

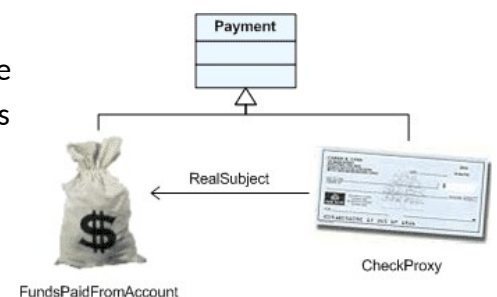
- Ce qui justifie de contrôler l'accès à un objet, c'est le souci de différer sa création ou son initialisation avant son utilisation effective.
- L'utilisation de procuration est indiquée quand on a besoin de références à un objet qui soient plus sophistiquées qu'un simple pointeur.

1.19.2. Solution

- Créer un objet subrogé qui instancie l'objet réel la première fois que le client en fait la requête, mémoriser l'identité de l'objet réel et rediriger les requêtes vers cet objet.

1.19.3. Métaphore : Chèque bancaire

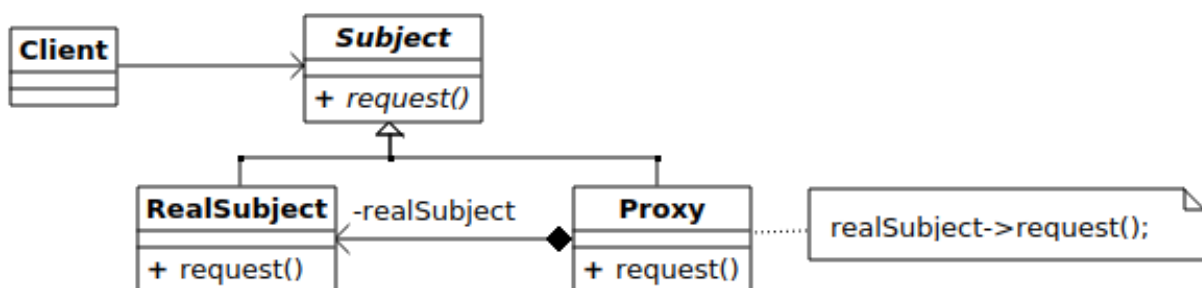
Un chèque est un subrogé pour une somme d'argent d'un compte bancaire. Il peut être utilisé à la place du numéraire pour faire des achats, et il peut à tout moment donner accès à du numéraire.



1.19.4. Exemples d'application

1. Une procuration à distance fournit un représentant local (*ambassadeur*) dans un espace différent (principe de base des objets itinérants type RMI).
2. Une procuration virtuelle crée des objets lourds à la demande : type Image de Java.
3. Une procuration de protection contrôle l'accès à l'original : droits d'accès.
4. Une référence intelligente réalise des opérations supplémentaires lors de l'accès à un objet : **smart pointers en C++**
 - décompte des références faites à un objet réel ;
 - destruction de l'objet réel quand le décompte est égal à 0.
5. Une procuration vérifie, avant d'y accéder, qu'un objet réel est verrouillé pour être le seul à le modifier.

1.19.5. Structure

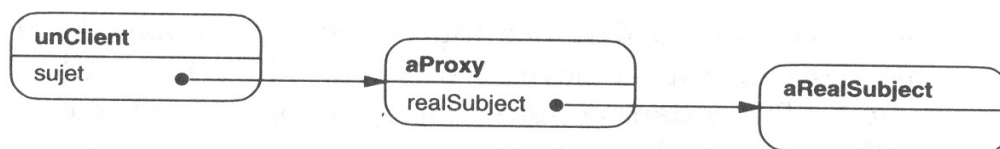


1.19.6. Constituants

- **Subject** : définit une interface commune pour `RealSubject` et `Proxy`, de sorte que la classe `Proxy` puisse être utilisée à la place de `RealSubject`.
- **Proxy** (eg. `CheckProxy`) : gère une référence qui lui permet d'accéder à un objet réel. Il procure une interface identique à celle du sujet, ce qui permet d'agir à sa place. Il contrôle l'accès au sujet réel, ce qui lui permet de gérer la création ou la suppression.
- **RealSubject** (eg. `FundsPaidFromAccount`) : définit l'objet réel représenté par la procuration.

1.19.7. Collaborations

Une procuration retransmet les requêtes au sujet réel selon les règles de son type.



1.19.8. Considérations d'Implémentation

- Il est préférable que la procuration et l'objet réel partagent la même interface, dans le sens où la procuration est « mise pour » l'objet réel (*respect du principe de substitution totale de Liskov*).
- Une procuration utilisée pour contrôler des droits d'accès peut limiter l'exécution de certaines opérations.

1.19.9. API Java

- Dynamic proxy : `java.lang.reflect.Proxy`.
- Remote Method Invocation (RMI) : `java.rmi.*`;

1.20. Prototype (Prototype)

Utiliser une instance prototypique pour créer de nouveaux objets par copie de ce prototype.

1.20.1. Problématique

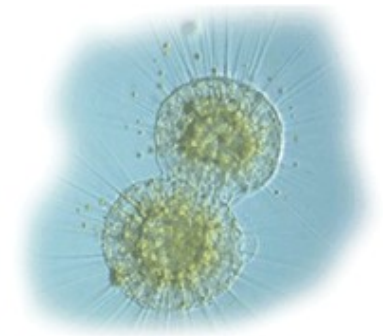
- La copie d'un objet est difficile à réaliser parce qu'il y a beaucoup d'informations à recopier ou la structure de l'objet est trop complexe ou une partie des informations est protégée.

1.20.2. Solution

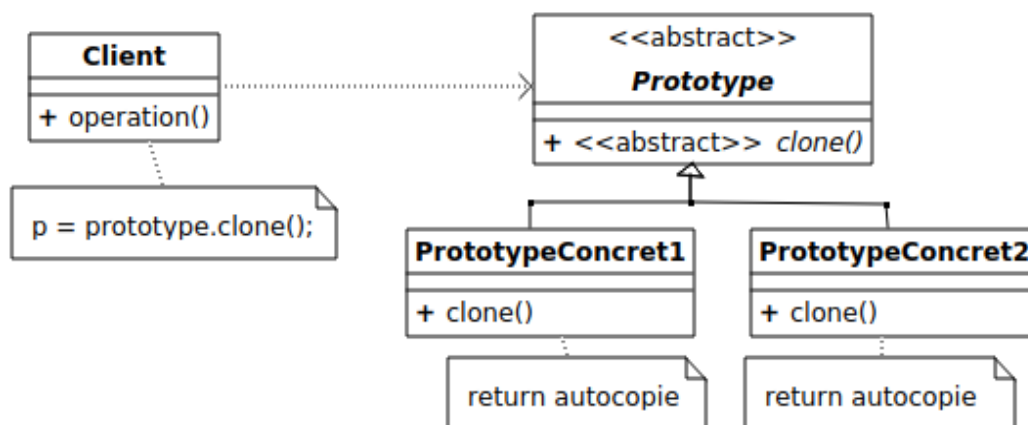
- Donner aux objets la possibilité de créer des copies d'eux-mêmes.

1.20.3. Métaphore : cellules biologiques

La création d'une nouvelle cellule biologique s'opère par clonage. Les deux cellules résultantes sont totalement identiques (aux mutations près...).



1.20.4. Structure



1.20.5. Constituants

- Prototype** : déclare une interface pour se cloner lui-même.
- PrototypeConcret** : implante une opération capable pour une instance de se cloner elle-même.
- Client** : crée un nouvel objet en demandant à un objet de se cloner.

1.20.6. Collaboration

Un client demande à une instance de se cloner puis manipule une copie de cette instance.

1.20.7. Conséquences

- ⊕ Le prototype cache au client la complexité de création de produits concrets ayant beaucoup de caractéristiques particulières (celui-ci a moins de noms à mémoriser).
- ⊕ Le prototype permet de créer des objets dont on ne connaît pas la classe.
- ⊖ La principale contrainte du modèle prototype est l'implantation de l'opération `clone()`, ce qui peut être

difficile quand les classes préexistent.

⊖ Le clonage peut devenir difficile pour les objets qui ne peuvent pas être copiés ou qui ont des références circulaires.

1.20.8. Considérations d'Implémentation

En Java, la méthode `clone()` est définie dans la classe de base `Object`. Toutefois, cette méthode possède la visibilité `protected` pour éviter la copie par défaut dans les classes dérivées. Cela signifie que l'on ne peut pas appeler `clone()` par une référence sur l'objet. Ainsi, les lignes de code ci-dessous produisent une erreur à la compilation :

```
Integer x = new Integer(1);  
Integer y = x.clone();
```

Cependant, une classe dérivée peut faire appel à la méthode `clone()` de la classe de base. Cela produit une copie bit à bit de l'objet de la classe dérivée.

Ainsi pour fournir la méthode `clone()` dans la liste de ses services, une classe doit :

1. implémenter l'interface `java.lang.Cloneable` (une « tagging interface » = interface vide),
2. surcharger la méthode `clone()`,
3. la rendre publique,
4. faire appel à la méthode `super.clone()` à l'intérieur de la méthode `clone()` pour produire une copie bit à bit de l'objet,
5. ajouter du code pour faire la copie particulière des éléments de la classe si nécessaire.

▪ Remarque : la méthode `clone()` définie dans une classe fonctionne aussi pour ses classes dérivées sans avoir à la redéfinir. Ainsi, si `B` est une classe dérivée de `A` qui définit la méthode publique `clone()`, alors les lignes suivantes suffisent à créer un clone de l'objet `b1` :

```
B b1 = new B(); B b2 = b1.clone();
```

Cela produit une copie de `b1`, selon le code de `A.clone()`, même si `clone()` n'a pas été redéfinie dans `B`. C'est la magie de Java !

- Une alternative au prototype est l'utilisation de la **serialization** java, mais elle est plus chronophage.

1.21. Singleton (Singleton)

Garantir qu'une classe n'a qu'une instance et fournir un point d'accès global à cette classe.

1.21.1. Problématique

- Divers clients doivent se référer à une même chose et on veut être sûr qu'il n'existe qu'une seule instance.
- Si on ne garantit pas l'unicité, il peut y avoir de graves problèmes de fonctionnement (incohérences, écrasement de données).

1.21.2. Solution

- S'assurer qu'il n'existe qu'une seule instance en contrôlant le constructeur.

1.21.3. Métaphore : Équipe championne du monde de football

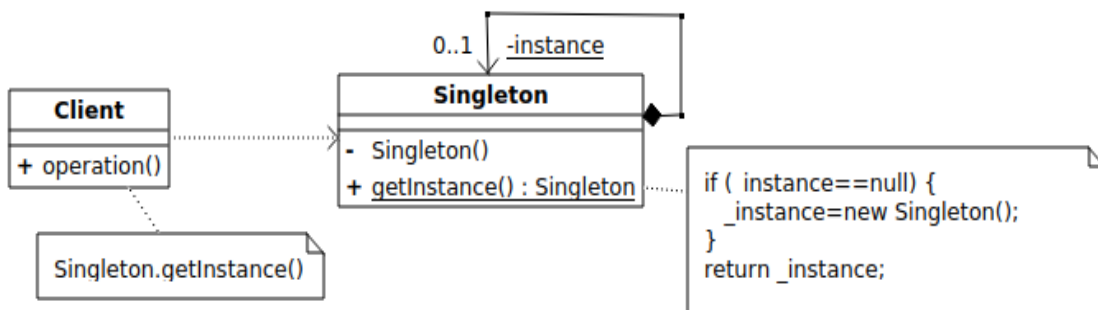
L'équipe championne de monde de football est typiquement un Singleton. Il n'existe à un instant donné qu'une seule équipe championne du monde de football.



1.21.4. Exemple

- Il peut y avoir plusieurs imprimantes sur un système, mais un seul serveur d'impression.
- Plusieurs fenêtres sont présentes sur un écran, mais un seul gestionnaire de ces fenêtres.

1.21.5. Structure



1.21.6. Constituants

- **Singleton** : construit sa propre instance unique et définit une opération `getInstance()` qui donne l'accès à son unique instance.

1.21.7. Collaborations

Les clients accèdent à l'instance uniquement par l'intermédiaire de l'opération `getInstance()` de la classe Singleton.

1.21.8. Conséquences

- ⊕ Les clients n'ont pas à prendre en charge la création initiale du Singleton, puisque lui-même s'en charge.
- ⊖ La construction d'un singleton n'est pas naturelle (pas d'appel à l'opérateur `new` pour créer un objet).
- ⊖ S'apparente à une variable globale avec tous ses inconvénients (cf. principes STUPID).

1.21.9. Considérations d'Implémentation

Trois implémentations sont possibles selon le degré de contrôle souhaité :

1- L'instance peut être créée au chargement de la classe. Il y a donc toujours exactement une instance :

```
public final class Singleton {  
    private static Singleton _instance = new Singleton();  
    private Singleton() { }  
    public static Singleton getInstance() {  
        return _instance;  
    }  
}
```

- Déclarer la classe `final` pour éviter toute surcharge qui pourrait contourner l'effet du Singleton.
- Ajouter un membre statique publique `getInstance()` qui renvoie la valeur de l'instance.
- Ajouter un constructeur affecté du statut **privé** pour forcer la création par `getInstance()`.

2- L'instance est créée seulement à la première demande (« lazy initialization »)

Dans ce cas, il faut, en plus de l'implémentation précédente, se prémunir contre le risque de création simultanée par de différents threads, d'où la synchronisation de la méthode.

```
public static synchronized Singleton getInstance() {  
    if (_instance == null) {  
        _instance = new Singleton();  
    }  
    return _instance;  
}
```

- Il est aussi nécessaire de définir la donnée membre `_instance` en volatile.

```
private static volatile Singleton _instance;
```

3- Le « singleton du pauvre ». Le singleton est un attribut statique de la classe cliente. L'unicité de l'instance est alors créée lors du chargement de la classe cliente. Dans ce cas, c'est le client qui est le garant de l'unicité de l'instance.

```
public final class Client {  
    private static Singleton _singleton = new Singleton();  
}
```

1.21.10. Attention

Ce patron est aussi un anti-patron (cf. les anti-patterns STUPID). Son utilisation doit donc être justifiée comme incontournable.

1.22. Stratégie (Strategy)

Définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables.

1.22.1. Problématique

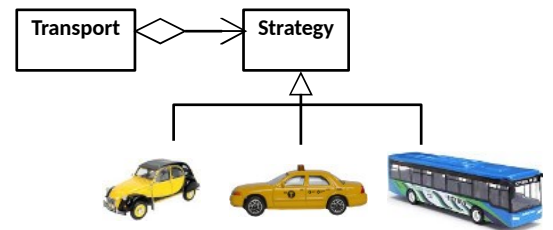
- On souhaite que la sélection d'un algorithme dépende du client à l'origine de la demande.
- **Ce qui varie** : l'algorithme implémentant un service.

1.22.2. Solution

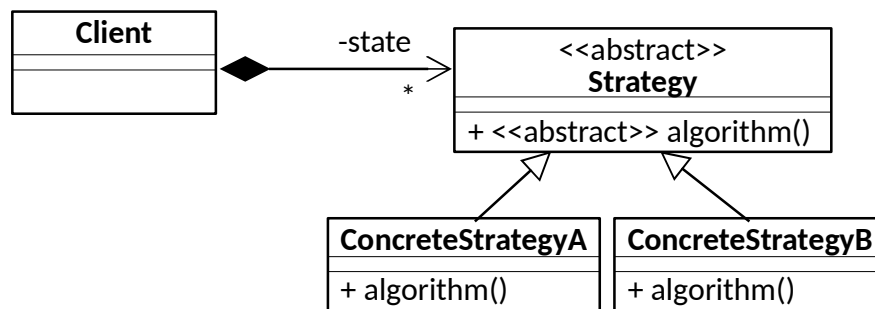
- Encapsuler les algorithmes dans une hiérarchie et lier la hiérarchie à l'objet appelant par composition.
- Séparer la sélection de l'algorithme de son implémentation.
- Le patron permet une sélection dynamique basée sur le contexte.

1.22.3. Métaphore : Mode de transport pour se rendre à l'aéroport

Un voyageur dispose de plusieurs modes de transport pour se rendre à un aéroport. Il choisit la stratégie qui lui semble la plus adaptée en fonction de critères du moment tels que le prix, le confort, le temps de parcours ou l'encombrement.



1.22.4. Structure



1.22.5. Constituants

- **Strategy** : déclare une interface commune à tous les algorithmes représentés. Le client utilise cette interface pour appeler l'algorithme défini par une stratégie concrète.
- **ConcreteStrategy** : implémente un algorithme en utilisant l'interface `Strategy`.
- **Client** : choisit une référence à un objet `Strategy` concret.

1.22.6. Conséquences

- ⊕ Famille d'algorithmes apparentés : tous les algorithmes ont la même interface.
- ⊕ Alternative aux dérivations en sous-classes de Client : la stratégie n'est pas codée en dur. Elle peut être choisie dynamiquement.
- ⊕ Des implémentations différentes peuvent être proposées pour un même comportement.

1.22.7. Variante

Le patron de conception **Politique** : dans son acceptation la plus stricte, le patron Stratégie ne concerne qu'une seule méthode. Il est possible de généraliser le concept en encapsulant plusieurs méthodes pour une même implémentation mais qui évoluent en même temps (par exemple, le calcul du prix d'exportation d'un produit se distingue par les politiques appliquées par chaque pays destinataire pour le calcul des frais de port, le calcul de la taxe d'importation ou la taxe de douane).

1.22.8. Considérations d'Implémentation

- Les interfaces de Stratégie et de Client doivent assurer à une stratégie concrète l'accès aux données d'un client dont elle a besoin. Il y a deux possibilités d'implémentation :

1. Les informations peuvent être passées sous forme de paramètres :

```
public void algorithm( Parameter par );
```

2. Le client peut fournir sa référence en paramètre, et la stratégie lui réclame les données par l'intermédiaire d'accesseurs :

```
public void algorithm( Client client );
```

Dans ce dernier cas, il faut un lien privilégié entre les deux classes. On fera alors des classes internes.

- L'interface stratégie est partagée par toutes les sous-classes, ce qui peut provoquer l'apparition de paramètres qui ne seront pas tous utilisés par toutes les stratégies. Une solution consiste à utiliser un **Messenger** (section 2.2) qui n'utilise pas d'accesseur.

```
public void algorithm( Parameter parameter ) {  
    ... parameter.get("arg1") ... parameter.get("arg3") ...  
}
```

- En C++, la stratégie peut être un argument d'un template si l'on souhaite faire une liaison statique entre le Client et la Stratégie (liaison à la compilation) :

```
template <typename T >  
class Client {  
    ... T _strategy;  
    _strategy.algorithm();  
}  
class Strategy1: public Strategy { ... }  
Client<Strategy1> c;
```

1.22.9. Modèles apparentés

- Patron de méthode : la différence réside dans le fait que le « Patron de Méthode » ne varie que sur une partie de l'algorithme alors que « Stratégie » redéfinit entièrement l'algorithme.

1.22.10. Exemple dans l'API Java

- L'interface `Comparable` donne la possibilité de personnaliser des fonctions de comparaison entre deux objets. Elle est utilisée entre autres pour trier des collections d'objets.

1.23. Visiteur (Visitor)

Ajouter de nouvelles opérations sans modifier les classes des éléments sur lesquels elles opèrent.

1.23.1. Problématique

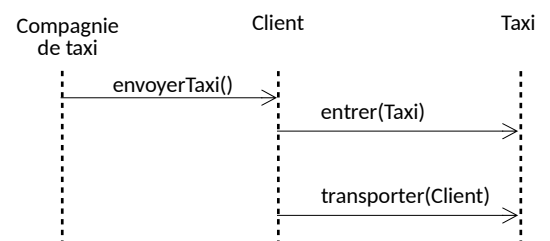
- Différentes opérations distinctes doivent être réalisées sur les nœuds d'une structure composite, mais on ne souhaite pas polluer les classes des nœuds avec ces opérations, ni tester le type de chaque nœud pour transtyper le pointeur avec le bon type avant de réaliser l'opération.
- **Ce qui varie** : la liste des services rendus par une structure d'objet.

1.23.2. Solution

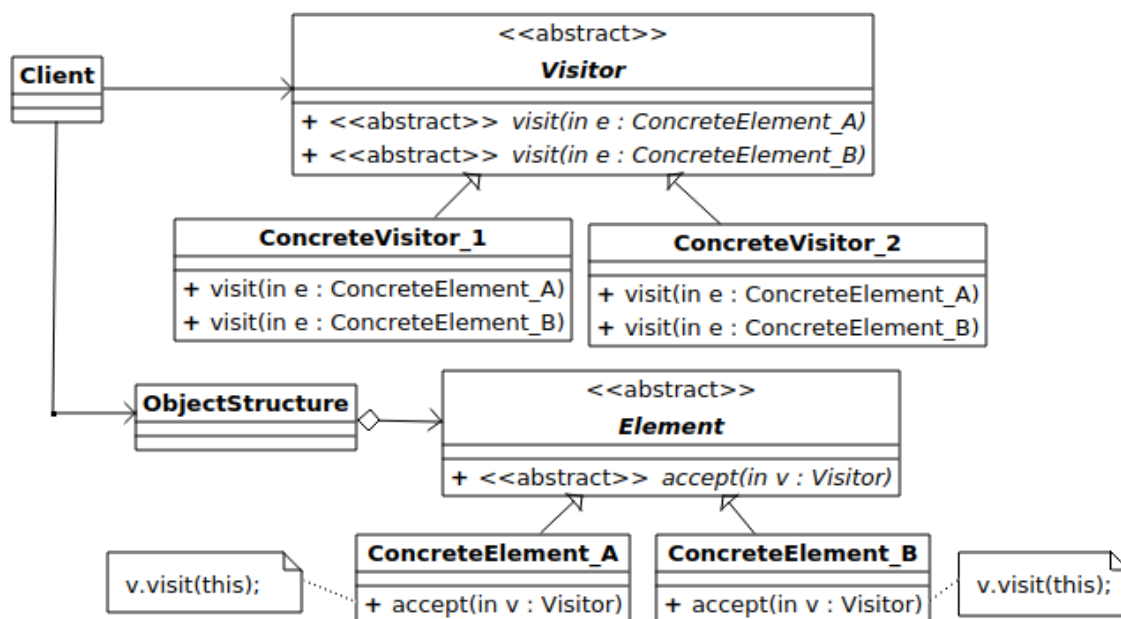
- Externaliser les opérations d'une structure d'objets dans une hiérarchie séparée et ajouter une méthode dans la structure d'objet pour accueillir des instances des opérations.

1.23.3. Métaphore : Compagnie de taxi

Ce patron peut être observé dans les opérations d'une compagnie de taxi. Quand une personne appelle une compagnie de taxi (accepter un Visiteur), la compagnie envoie un véhicule au client. Quand il entre dans le taxi, le client (Visiteur) ne gère plus son transport, c'est le taxi qui le fait.



1.23.4. Structure



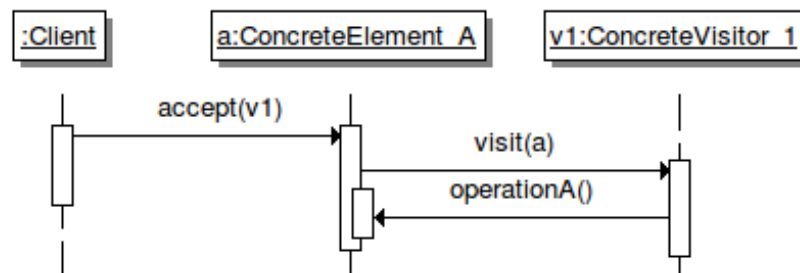
1.23.5. Constituants

- **Visitor** (eg. Compagnie de taxi) : une interface qui déclare une opération `accept()` pour chaque classe **ConcreteElement** de la structure d'objets. Le nom de l'opération et sa signature identifient la classe émettrice de la requête `accept()` chez le visiteur. Le visiteur peut ensuite accéder à

l'élément directement, au travers de son interface particulière.

- **ConcreteVisitor** (eg. taxi) : le code de chaque opération déclarée de la classe Visitor. Chaque opération code un fragment de l'algorithme défini pour les classes d'objets qui lui correspondent dans la structure. Le visiteur concret fournit le contexte pour l'algorithme et mémorise son état.
- **Element** : une interface qui définit une opération `accept()` ayant pour argument un visiteur.
- **ConcreteElement** (eg. client) : réalise le codage d'une opération `accept()` qui prend pour argument un Visitor.
- **ObjetStructure** (eg. liste de clients) : fournit une interface de haut niveau permettant au visiteur de visiter ses éléments ; peut être un Composite ou un conteneur tel qu'une liste ou un ensemble.

1.23.6. Collaborations



1.23.7. Considérations d'Implémentation

La séquence d'appel pour appliquer une opération implantée dans l'objet `ConcreteVisitor_1` à une structure d'objet représentée par l'objet `ConcreteElement_A` est :

```
ConcreteElement element = new ConcreteElement_A();
element.accept(new ConcreteVisitor_1());
```

1.23.8. Conséquences

- ⊕ Le patron Visiteur facilite l'addition de nouvelles opération, par simple ajout d'un nouveau visiteur.
- ⊕ Le Visiteur rassemble les opérations du même type : les comportements coopératifs sont implantés dans un Visiteur concret.
- ⊖ **Ce patron est absolument contre-indiqué si la structure visitée est amenée à évoluer** : le coût de modification des visiteurs devient prohibitif.
- ⊖ **Ce patron va à l'encontre de la modélisation objet** en diminuant la cohésion et en augmentant le couplage. Il rompt ainsi avec l'encapsulation : le modèle impose de rendre publique des opérations qui accèdent à une partie de l'état interne d'un élément.

2. Autres patrons de conception (hors bande des quatre)

2.1. Collecteur de paramètres (Collecting Parameter)

Glaner des données lors de l'exécution successive de plusieurs méthodes, à la manière d'une abeille qui collecte du pollen.

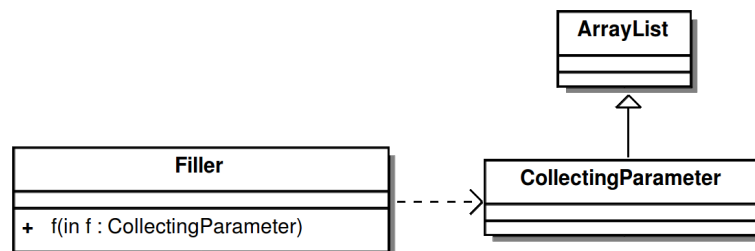
2.1.1. Problématique

- Construire une structure pour collecter les valeurs de différents attributs lors de l'exécution de méthodes.

2.1.2. Solution

- Construire une classe qui est une extension d'une collection.
- L'idée est que le collecteur soit utilisé et modifié par les méthodes dont il est le paramètre.

2.1.3. Structure



2.1.4. Considérations d'Implémentation

- Une liste chaînée (`ArrayList`) définit un bon collecteur de paramètres, puisqu'elle est déjà faite pour ajouter dynamiquement des objets :

```
public final class CollectingParameter extends ArrayList<Object> {}

public final class Filler {
    public void f( CollectingParameter cp ) {
        cp.add(new Object(1));
    }
    public void g( CollectingParameter cp ) {
        cp.add(new Object(2));
    }
}

@Test
public void testCollectingParameter() {
    Filler filler = new Filler();
    CollectingParameter cp = new CollectingParameter();
    filler.f(cp);
    filler.g(cp);
    assertEquals("[ " + cp + " ]","[accumulating, items as we go]");
}
```

```
}
```

- Une alternative est l'utilisation d'un dictionnaire (`hashtable`) qui permet de nommer les valeurs collectées.

2.2. Messenger (Messenger)

Englober des données liées sémantiquement dans un objet pour les passer comme paramètres de méthodes au lieu de les passer séparément.

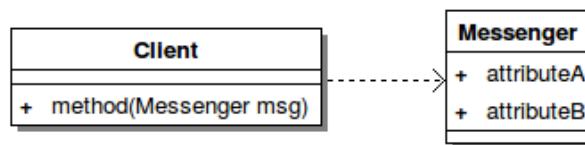
2.2.1. Problématique

- Trop d'arguments dans le prototype d'une méthode.
- Coupler des données liées sémantiquement.

2.2.2. Solutions

- Une classe sans aucune méthode. Puisque le but d'un messenger est seulement de porter des données, ces données sont déclarées **publiques** pour un accès facile.
- Un dictionnaire (`hashtable`). Le dictionnaire permet de nommer les paramètres dynamiquement. C'est la version utilisée par Python pour ses paramètres de fonction variables.

2.2.3. Structure



2.2.4. Constituants

- **Client** : possède une méthode qui utilise les données liées sémantiquement entre elles.
- **Messenger** : structure contenant les données sous forme publique.

2.2.5. Exemple

La liste des coordonnées d'une figure est regroupée dans le messenger `Point`. Sans le messenger, le code de `translate()` serait plus confus à lire.

```
final class Point {
    public int x, y, z;
    public Point( int x, int y, int z ) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    public Point( Point p ) {
        x = p.x;
        y = p.y;
    }
}
```

```

        z = p.z;
    }
    public String toString() {
        return "x: " + x + " y: " + y + " z: " + z;
    }
}

final class Point extends Hashtable<String, int> {

    public String toString() {
        return entrySet().stream()
            .map(x -> x.getKey() + ": " + x.getValue().stream()
                .map(Object::toString)
                .collect(Collectors.joining(", ")))
            .collect(Collectors.joining("; "));
    }
}

```

2.2.6. Remarque

Ce patron va à l'encontre du principe d'encapsulation en révélant la représentation interne de la classe `Messenger`. Mais, ces données sont la raison d'être de la classe, donc il n'est pas nécessaire de cacher la représentation. Toutefois, il est possible de limiter l'accès aux données en passant par des mutateurs et accesseurs pour des raisons de sécurité des valeurs.

En C++, le messenger est typiquement représenté par une *structure*.

2.3. Data Transfert Object (DTO)

Data Transfer Object est un conteneur qui est utilisé pour transporter des données entre des couches ou des étages d'une architecture. Ce n'est rien de plus qu'un messenger qui dispose en plus de capacité de sérialisation permettant de l'exploiter sur le réseau.

2.4. Data Access Object (DAO)

Data Access Object est essentiellement un objet ou une interface qui fournit un accès à une base de données ou toute forme de stockage persistant. Ce patron permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métier.

C'est essentiellement une classe `Messenger` avec un ensemble de mutateurs et d'accesseurs.

2.5. Pool d'objet (Object Pool)

Créer un nombre limité d'objets qui peuvent être partagés par d'autres objets.

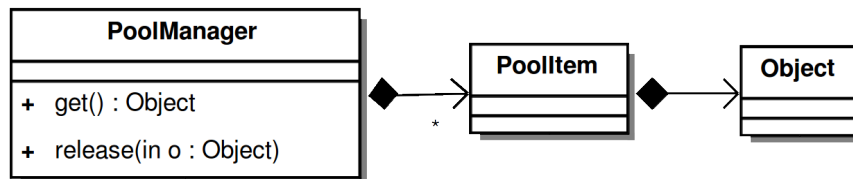
2.5.1. Problématique

- Le patron Singleton est restreint à une seule instance.

2.5.2. Solution

- Créer une classe avec des méthodes qui contrôlent l'ajout en nombre limité et la suppression des éléments.

2.5.3. Structure



2.5.4. Considérations d'Implémentation

```
public final class PoolManager {
    static class EmptyPoolException extends Exception {}

    private static class PoolItem {
        boolean inUse = false;
        Object item;
        PoolItem( Object item ) { this.item = item; }
    }

    private ArrayList<PoolItem> _items = new ArrayList<PoolItem>();

    public void add( Object item ) {
        _items.add(new PoolItem(item));
    }

    public Object get() throws EmptyPoolException {
        for (PoolItem pitem : _items) {
            if (! pitem.inUse) {
                pitem.inUse = true;
                return pitem.item;
            }
        }
        throw new EmptyPoolException();
    }

    public void release(Object item) {
        for (PoolItem pitem : _items) {
            if (item == pitem.item) {
                pitem.inUse = false;
                return;
            }
        }
        throw new RuntimeException(item + " not found");
    }
}
```

2.5.5. Exemple

Les bases de données commerciales restreignent souvent le nombre de connexions qui peuvent être utilisées en même temps par plusieurs clients.

Le pool d'objets est implémenté par une classe séparée.

```
public interface Connection {
    Object get();
    void set(Object x);
}

class ConnectionImplementation implements Connection {
    public Object get() { return null; }
    public void set(Object s) {}
}

class ConnectionPool {

    // Singleton du pauvre.
    private static PoolManager _pool = new PoolManager();

    public static void addConnections(int number) {
        for (int i = 0; i < number; i++)
            _pool.add(new ConnectionImplementation());
    }

    public static Connection getConnection() throws EmptyPoolException {
        return (Connection)_pool.get();
    }

    public static void releaseConnection(Connection c) {
        _pool.release(c);
    }
}

public final class ConnectionPoolDemo {
    static { ConnectionPool.addConnections(5); }

    public static void main( String[] args) {
        Connection c = null;
        try {
            c = ConnectionPool.getConnection();
        } catch (PoolManager.EmptyPoolException e) {
            throw new RuntimeException(e);
        }
        c.set(new Object());
        c.get();
        ConnectionPool.releaseConnection(c);
    }
}
```

2.6. Objet Nul (Null Object)

Définir un objet qui permet d'encapsuler l'absence d'un objet en fournissant un substitut qui implante le comportement de ne rien faire.

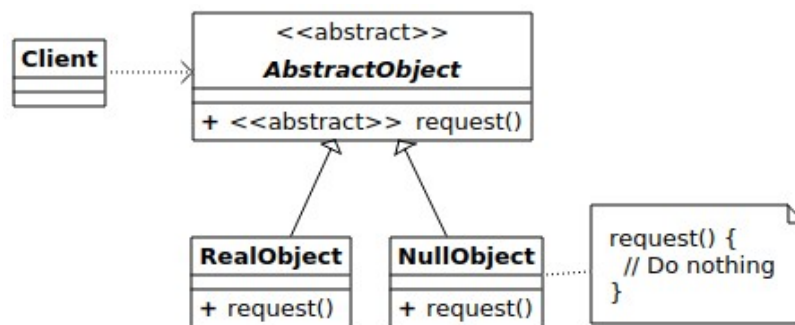
2.6.1. Problématique

- Si une référence d'objet peut être nulle, et que le résultat de la vérification est de ne rien faire ou de faire un comportement par défaut, comment l'absence de l'objet peut être traitée de façon transparente ?
- Éviter les problèmes induits par le test `(object==null)` source de nombreux bugs.

2.6.2. Solution

- La clé du patron « Objet Nul » est une classe abstraite qui définit l'interface pour tous les objets de ce type. L'Objet Null est implémenté comme une sous-classe de cette classe abstraite. Parce qu'il est conforme à l'interface abstraite de classe, il peut être utilisé n'importe où ce type d'objet est nécessaire. Cette classe est un gain par rapport à l'utilisation de la valeur spéciale « null » qui doit toujours être vérifiée dans n'importe quel objet qui utilise l'interface abstraite.

2.6.3. Structure



2.6.4. Constituants

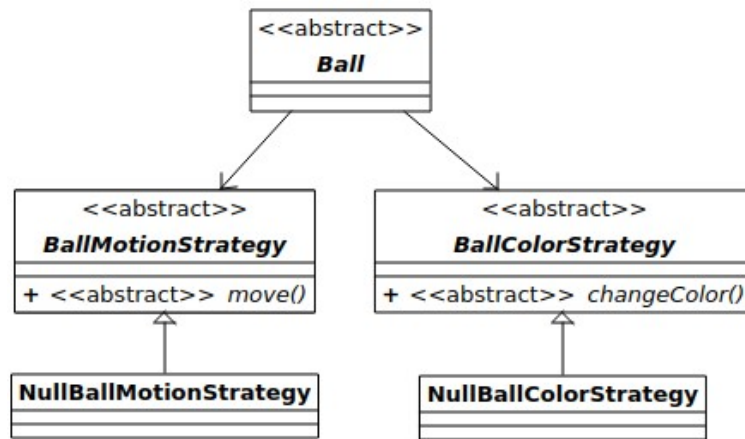
- **Client** : réclame le service `request`.
- **AbstractObject** : déclare l'interface pour les collaborateurs du Client.
- **RealObject** : Définit une sous-classe concrète qui fournit le comportement souhaité.
- **NullObject** : Définit le comportement correspondant à un objet nul.

2.6.5. Exemple

Considérons par exemple, un simple écran de veille qui affiche des balles qui se déplacent sur l'écran et qui ont des effets de couleur spéciaux. Ceci est facilement réalisé en créant une classe `Balle` et en utilisant un patron Stratégie pour proposer une variété de mouvements de la balle et un autre patron Stratégie pour varier la couleur de la balle.

La stratégie la plus simple est de ne pas bouger et de ne pas changer de couleur. Cependant, le modèle de stratégie exige d'avoir des objets qui implémentent les interfaces de Stratégie. C'est là que le patron d'objet Null devient utile. Il suffit de mettre en place une classe `NullBallMotionStrategy` qui ne bouge pas la balle et un `NullBallColorStrategy` qui ne change pas la couleur de la balle. Toutes les

méthodes de ces classes ne font "rien".



3. Anti-patrons

3.1. Call super

Obliger les utilisateurs d'une classe à redéfinir une méthode de la classe pour compléter son comportement afin de pouvoir l'utiliser.

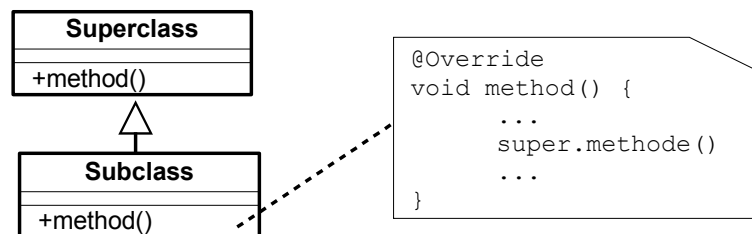
3.1.1. Problématique

La classe de base fournit un comportement de base pour une méthode mais qui n'est pas complet. Ce sont les utilisateurs de cette classe qui doivent compléter le comportement en appelant la méthode de base puis en la complétant avec du code spécifique.

C'est le fait d'obliger verbalement l'utilisateur à compléter le comportement de la méthode de base pour pouvoir l'utiliser qui est un anti-patron. Le compilateur n'a aucun moyen de vérifier que le contrat n'est pas respecté et donc l'erreur ne pourra se révéler qu'à l'exécution (dans le meilleur des cas).

3.1.2. Structure

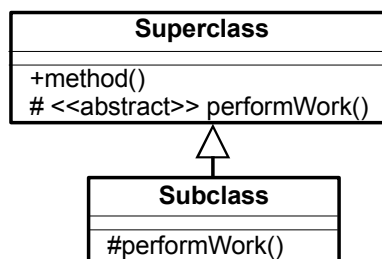
Le code de la méthode de la sous-classe fait appel au code de la méthode de base.



Ce faisant, rien n'oblige l'utilisateur de la classe à appeler la méthode de base. Or a priori, le code de la méthode redéfinie n'est pas complet sans l'appel à la méthode de la classe base. Cela provoque un défaut d'exécution qui n'est pas détectable à la compilation.

3.1.3. Solution

Une solution consiste à utiliser le patron de conception « patron de méthode ».



Cette fois, c'est le compilateur qui oblige l'utilisateur de la classe à définir la méthode `performWork()` pour compléter le code de la méthode de base puisqu'elle est abstraite.

3.1.4. Remarque

Java propose une autre solution avec l'annotation `@CallSuper` que l'on met sur la méthode de la classe

de base pour obliger le compilateur à vérifier bien que si la méthode de base est redéfinie, la méthode redéfinie appelle bien la méthode parent. Ainsi, cet anti-pattern est utilisé massivement dans les méthodes du cycle de vie d'Android (eg. `onCreate`, `onResume`). Mais, il est résolu par l'utilisation de l'annotation `@CallSuper`.

3.2. Objet Omniscient (God Object)

Concentrer trop de méthodes dans une seule classe.

3.2.1. Problématique

C'est l'erreur du débutant en programmation orientée objet qui a tendance à tout mettre dans une seule classe. Ceci est contraire au principe de responsabilité unique.

3.3. BaseBean

Hériter une fonctionnalité d'une classe utilitaire plutôt que de la déléguer.

3.3.1. Problématique

En programmation orientée objet, l'héritage doit être utilisé pour des relations de type « est-un » et pas « a-un ». Un BaseBean est une classe utilitaire à partir de laquelle des entités concrètes ont été obtenues par héritage, créant une fausse implication que les classes dérivées représentent des sous-types de la classe utilitaire dans le domaine métier. Par exemple, si une classe utilitaire appelée `DatabaseUtils` contient des méthodes pour l'établissement et la fermeture de connexions de base de données alors l'anti-pattern consiste à faire hériter une classe `Employé` pour profiter de ces méthodes. Dans le monde réel, un employé n'est pas une « base de données ».

3.3.2. Solution

Une bonne conception suggère que la fonctionnalité héritée devrait être fournie via une délégation. Une classe ne doit pas hériter d'une autre classe simplement parce que la classe parente contient une fonctionnalité nécessaire à la sous-classe. La classe héritée peut se comporter de manière incorrecte lorsqu'elle est utilisée en remplacement de la classe parent, violant ainsi le principe de substitution de Liskov. La délégation doit être utilisée pour obtenir la fonctionnalité ou les données qui lui sont nécessaires. En d'autres termes, cet anti-patron suggère de préférer la composition à l'héritage.

3.4. Couplage séquentiel

Une classe qui exige que ses méthodes soient appelées dans un ordre particulier sans que rien n'y oblige.

3.4.1. Solution

Une solution peut être trouvée avec le patron de conception « Patron de méthode ».

3.5. Problème Yo-yo

Une modélisation basée sur une cascade d'héritage.

3.5.1. Problématique

Le graphe d'héritage est difficile à comprendre en raison de sa fragmentation excessive. Le programmeur doit lire et comprendre un programme dont le graphe UML est si long et compliqué qu'il doit jongler entre plusieurs définitions de classes afin de suivre le flux de contrôle du programme.

3.6. Constructeur télescopique

Une classe a de multiples constructeurs, où chaque constructeur appelle un constructeur plus spécifique dans la hiérarchie, qui a plus de paramètres que lui, en lui fournissant les valeurs par défaut pour les paramètres supplémentaires.

3.6.1. Exemple :

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

Le problème c'est que ces constructeurs sont difficiles à lire. Ils ont beaucoup de paramètres - qui sait ce qu'est le sixième ? Si les paramètres adjacents ont le même type, vous pouvez accidentellement changer leurs valeurs sans erreur de compilation et tout semble aller bien. De plus, vous savez rarement quels paramètres facultatifs le client souhaite utiliser pour son scénario, ce qui se traduit souvent par des appels de constructeur où le constructeur possède de nombreux paramètres, mais beaucoup sont nuls. Ce modèle est difficile à maintenir et à partir d'un certain nombre de paramètres, les constructeurs sont incroyablement longs et trop nombreux.

3.6.2. Solution

Une solution est d'utiliser le patron Monteur (Section 1.14).