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 qui ont un champ d’application général. Il y a donc tout intérêt à connaître ces patrons par cœur. D’autres ont été créés plus tard, mais 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. Dans ce cours, je n’en retiens que 19 qu’il me semble indispensables de connaître. J’ai écarté les patrons Interpréteur, Memento, Prototype et Poids Mouche qui me semble moins capitaux.

		Rôle		
		Création	**Structure**	**Comportement**
Portée	**Classe**	Fabrique simple (p. 15)	Adaptateur (p.3)	Patron de méthode (p. 29)
	Objet	Fabrique abstraite (p. 17) Prototype (p. Error: Reference source not found) Singleton (p. 35)	Adaptateur (p. 3)33 Pont (p. 31) Composite (p. 9) Décorateur (p. 11) Façade (p. 19) Procuration (p. 33)	Chaîne de responsabilités (p. 5) Commande (p. 7) Itérateur (p. 21) Médiateur (p. 23) Observateur (p. 27) État (p. 13) Stratégie (p. 37) Visiteur (p. 39)

1.1.1. Rôle

- **Patrons de création**
  - Ils décrivent 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**
  - Ils décrivent comment organiser les classes d’un programme dans une structure plus large en séparant l’interface de son implémentation.
- **Patrons de comportement**

- Ils décrivent le comportement d'interaction entre les objets et le partage des responsabilités pour composer des services.

1.1.2. Portée

- **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 par composition et modifiées à l'exécution.

1.2. Avertissement

Dans les modélisations UML de ce chapitre, les classes abstraites sont notées avec un stéréotype `<<abstract>>` alors que la norme UML impose soit la notation en italique du nom de la classe soit l'utilisation de la propriété `{abstract}` sous le nom de la classe à droite. Personnellement, je trouve l'italique peu visible et impossible à écrire sur un tableau lors du cours, et que la notation avec la propriété vient compliquer la syntaxe de base.
Puisque UML permet la définition de stéréotype personnel, j'ai ajouté le stéréotype `<<abstract>>` très visible et calqué sur le stéréotype `<<interface>>` dont il est proche.
Dans les trois représentations UML possibles d'une classe abstraite résumées dans la figure ci-dessous, la troisième notation nous semble la plus efficace.

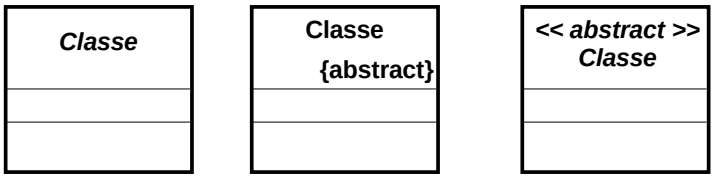


Figure 1: Trois notations UML de la notion de classe abstraite.

1.3. Adaptateur (*Adapter*)

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

1.3.1. Problématique

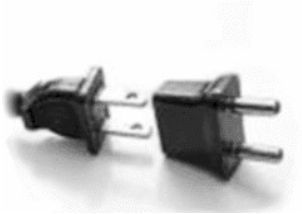
- On a besoin d'utiliser une classe existante, mais l'adaptation par dérivation ne peut pas être faite, car on ne dispose pas du code pour modifier la relation d'héritage de la classe.
- On souhaite créer une classe réutilisable qui collabore avec des classes encore inconnues qui n'auront pas forcément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

1.3.2. 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.

1.3.3. Métaphore : Prise électrique

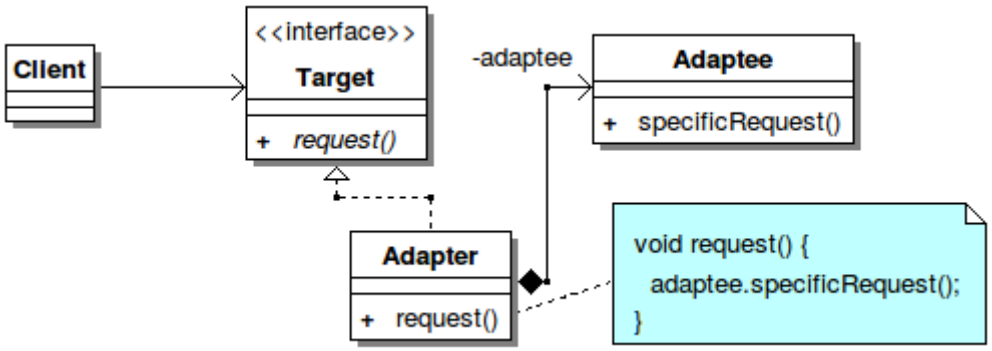
Parce que les deux pays n'utilisent pas le même standard de prise électrique, il est nécessaire d'utiliser un adaptateur pour brancher un équipement au standard anglais sur une prise de courant au standard français.



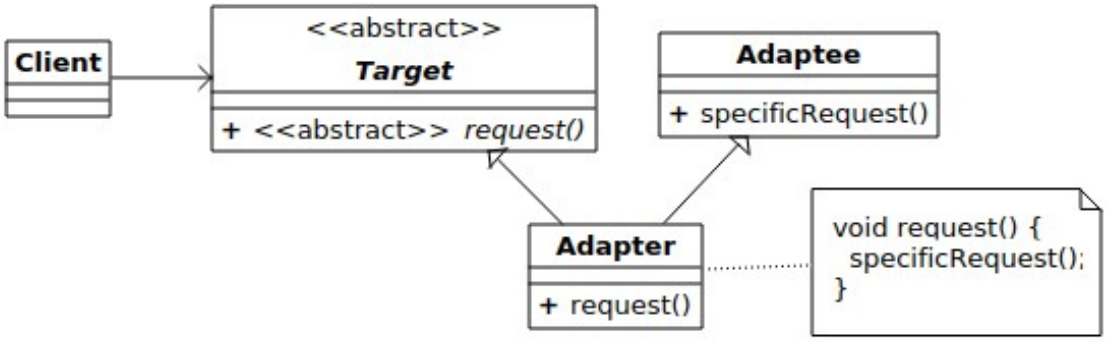
1.3.4. Structure

Il existe deux variantes selon que l'implémentation utilise l'héritage multiple ou non.

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



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

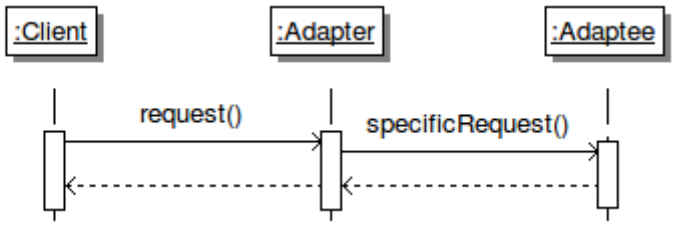


1.3.5. Constituants

- **Client** : il utilise le service `request()` de l'interface `Target`.
- **Target** : la classe qui 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`.

1.3.6. Collaboration

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.



1.3.7. 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 `Adaptee` 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(); }
}
```

1.4. 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.

1.4.1. Problématique

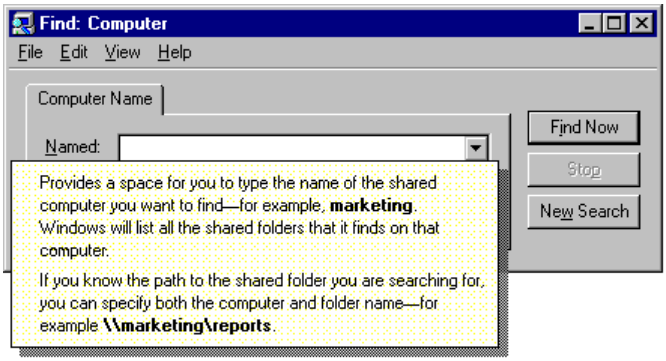
- Plus d'un objet peut gérer une requête, mais le destinataire effectif n'est pas connu a priori.
- On veut émettre une requête sans avoir à préciser le receveur explicitement.
- L'ensemble des objets qui peuvent gérer une requête ne peut être spécifié que dynamiquement.

1.4.2. Solution

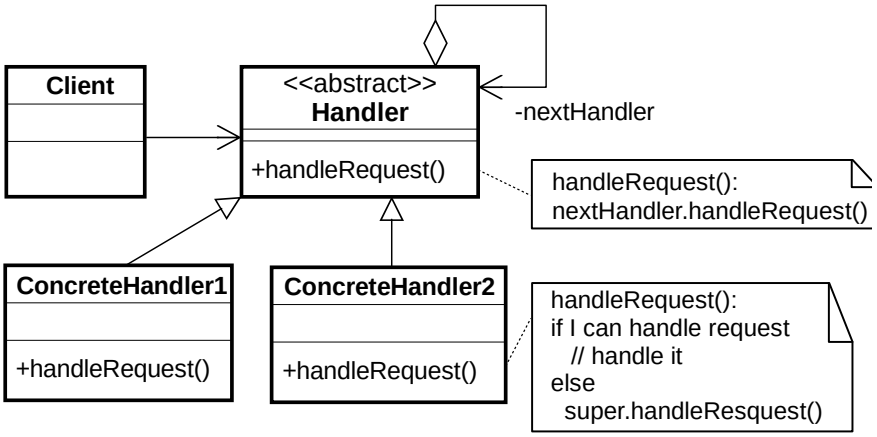
- Chaîner les objets récepteurs de façon à faire passer la requête le long de la chaîne jusqu'à ce qu'un objet soit en mesure d'y répondre.

1.4.3. Exemple : Gestion de l'aide contextuelle

L'aide interactive dans une interface graphique est un exemple de chaîne de responsabilité. L'utilisateur peut obtenir de l'aide sur n'importe quel objet de l'interface (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, sinon il demande cette aide à l'objet qui l'englobe.



1.4.4. Structure



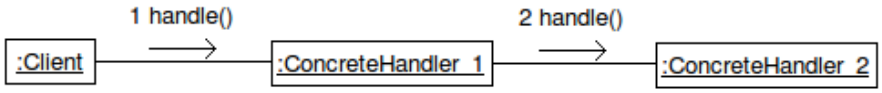
1.4.5. Constituants

- **Client** : l'émetteur de la requête vers 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.

1.4.6. 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.



1.4.7. 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 de répondre à la requête.
⊕ 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.

1.4.8. 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, sinon il revoie 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();
      }
   }
}
```

1.5. Commande (*Command*, *Action*)

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

1.5.1. Problématique

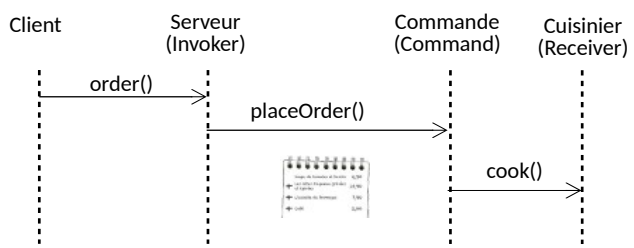
- Besoin de garder un historique des commandes effectuées pour :
  - assurer un service de réversion (undo / redo) ;
  - réaliser un « film » des actions effectuées.
- Nécessité de traiter des requêtes sur des objets sans rien savoir de l'opération invoquée ou du receveur de la requête.

1.5.2. Solution

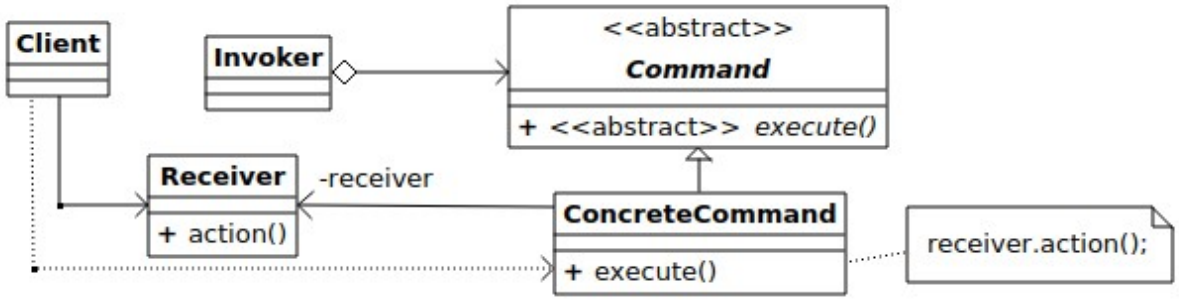
- Encapsuler l'invocation d'un service dans un objet à part entière (abstraction hypostatique).

1.5.3. 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.



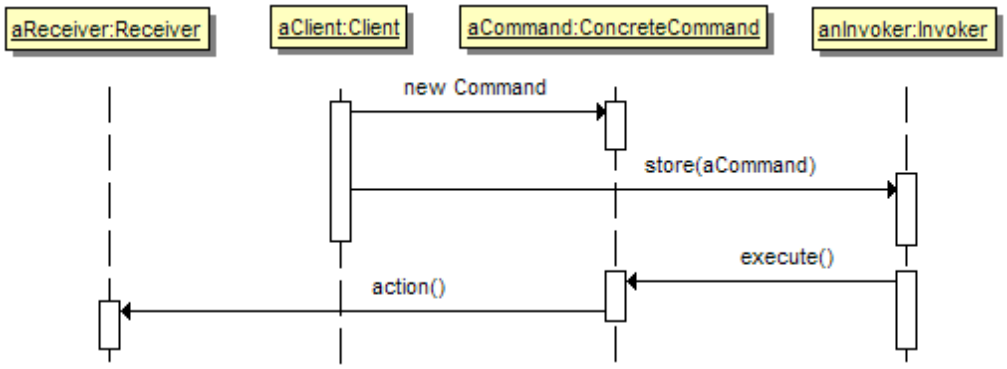
1.5.4. Structure



1.5.5. Constituants

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

1.5.6. Collaboration



1.5.7. 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.

1.5.8. 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 AjouteElement(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 un `Element`, mais nous ajoutons aussi l'objet `this` qui est nécessaire à l'exécution.

1.5.9. Exemple dans l'API Java

Les classes `AbstractAction` et `Undoable` et `UndoManager` en Java permettent d'implémenter le mécanise de réversion. La classe `Undoable` joue le rôle de la commande.

1.6. 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.

1.6.1. Problématique

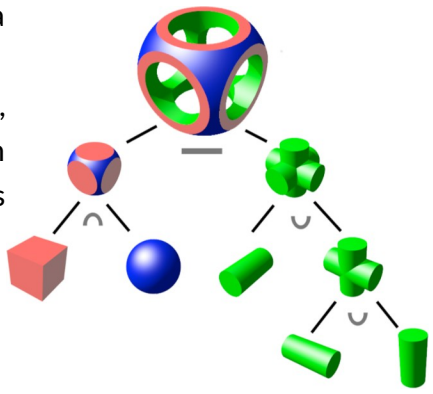
- Nécessité de représenter des hiérarchies de l'individu à l'ensemble.
- Ne pas de se soucier de la différence entre combinaisons d'objets et constituants.

1.6.2. Solution

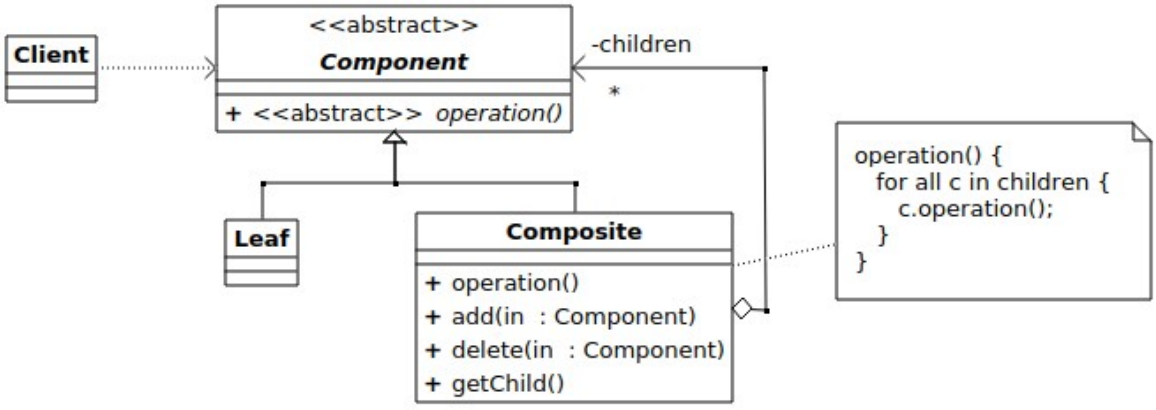
- Définir une classe abstraite qui spécifie le comportement commun à tous les composés et les composants.
- La relation de composition lie un composé à tout objet du type de la classe abstraite.

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

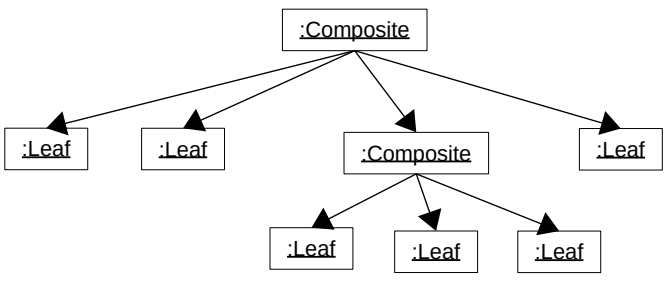
Une application de CAO se propose de modéliser les objets du monde à partir de formes primitives 3D : sphères, cubes et cylindres.
Un langage ensembliste avec les trois opérateurs « intersection », « union » et « moins » permet de traiter des formes 3D complexes. Un objet peut être une forme primitive ou un composite de plusieurs primitives.



1.6.4. Structure



Un composite est en fait un arbre abstrait :



1.6.5. Constituants

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

1.6.6. 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 ou après ce transfert.

1.6.7. Conséquences

⊕ Il définit des hiérarchies de classes qui peuvent être composées de façon récursive.
⊕ Il simplifie l'accès à la structure pour le client (unification des requêtes).
⊕ Il rend facile la création / l'ajout de nouveaux composants.

1.6.8. 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 déclassement (« 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 préférable au déclassement qui implique de savoir si on traite une feuille ou un composite.

1.7. Décorateur (*Decorator, Wrapper*)

Attacher dynamiquement des responsabilités supplémentaires à un objet.

1.7.1. 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.

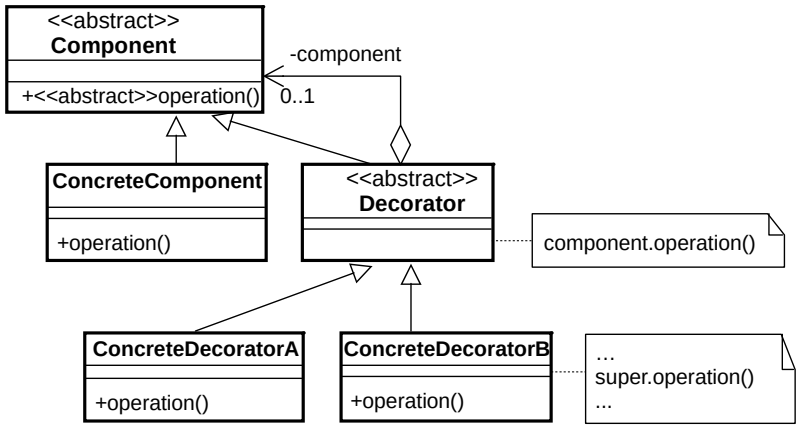
1.7.2. Solution

- Permettre l'extension de la fonctionnalité sans avoir recours à des sous-classes spécialisées, mais en chaînant dynamiquement les extensions de la fonctionnalité à partir d'un objet de base.

1.7.3. 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.

1.7.4. Structure

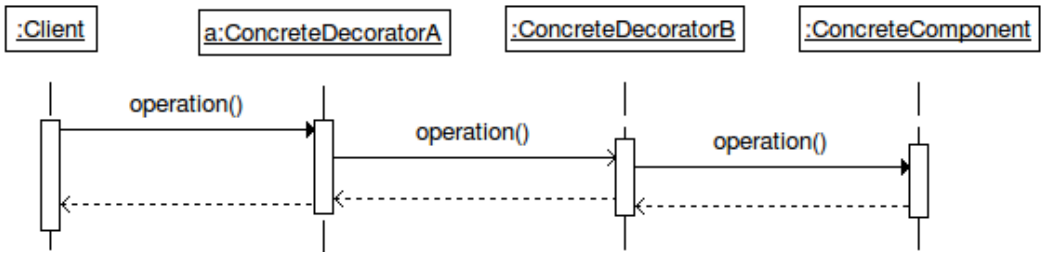


1.7.5. Constituants

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

1.7.6. Collaboration

Le service `operation()` est exécuté sous la forme d'une 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.



1.7.7. Conséquences

⊕ 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.

⊕ 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.

⊖ **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 est impossible d'accéder directement à la classe de base `ConcreteComponent` puisqu'elle est enfouie.

1.7.8. Considération d'implémentation

La création d'un objet décoré procède par emballement 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();
```

La méthode `operation`[1] doit faire appel à la méthode de la super classe pour continuer l'exécution de l'opération avec les autres couches du composant :

```
operation() { … super.operation(); … }
super.operation() { component.operation(); }
```

1.7.9. Exemple dans l'API Java

Les entrées-sorties en Java (`InputStream`, `FilterInputStream`, `OutputStream`, `FilterOutputStream`) sont construites sur le patron Décorateur.

1 Attention, telle que décrite ici, cette implémentation correspond à l'anti-patron « call super » (p. 47). Il faut trouver une implémentation concrète plus efficace, en s'appuyant par exemple sur le patron « patron de méthode » (p. 29).

1.8. État (*State*)

Adapter le comportement d'un objet aux changements de son état interne.

1.8.1. Problématique

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

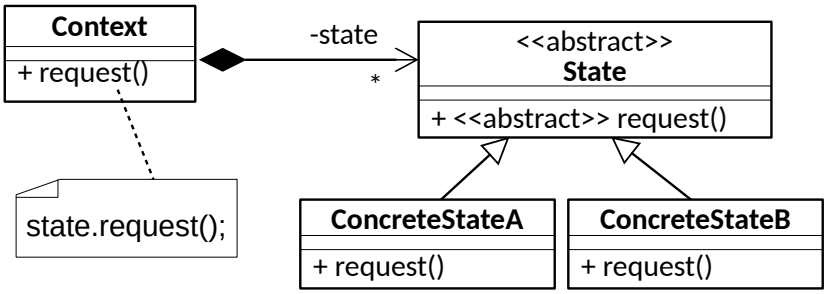
1.8.2. Solution

- 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.

1.8.3. 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. La façon de s'exprimer ou de se déplacer change selon que la créature est une grenouille ou un prince. Au début du conte, la créature est une grenouille puis se transforme en prince charmant lorsque la princesse l'embrasse.

1.8.4. Structure



1.8.5. Participants

- **Context** : une classe qui définit l'interface qu'utilisent 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** : une classe abstraite qui 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 un état.

1.8.6. Collaboration

- Le `Context` délègue les requêtes spécifiques de l'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` (cachés derrière la composition).
- C'est soit à `Context` soit aux sous-classes `ConcreteState` qu'il revient de décider de l'état qui succède à un autre état.

1.8.7. Conséquences

⊕ Ce patron isole les comportements spécifiques d'état et fait un partitionnement des différents comportements, état par état.
⊕ Il rend les transitions d'état plus explicites.
⊕ Il permet de simuler le changement de classe pour un objet.

1.8.8. 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 sont préférentiellement des classes internes en Java ou amies en C++.
- Une solution alternative est 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.

1.9. 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.

1.9.1. Problématique

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

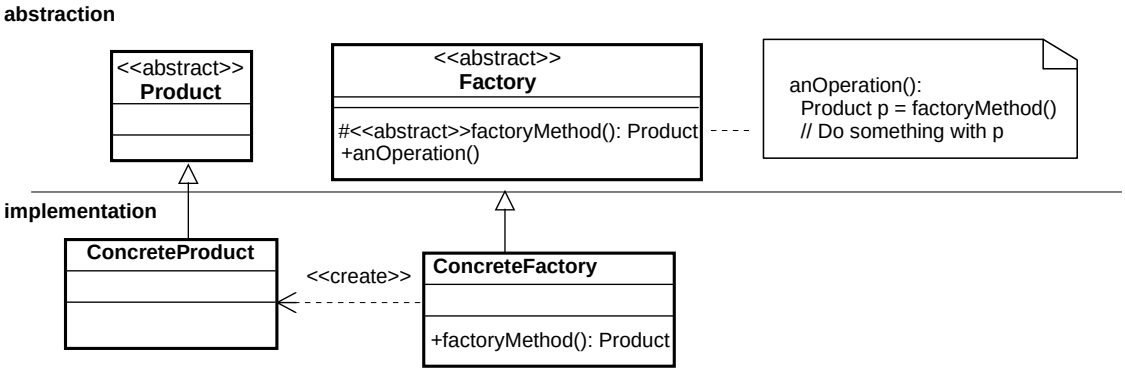
1.9.2. Solution

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

1.9.3. 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 la création des variétés de pizza effectives dans des classes dérivées. Chaque franchise est incarnée par une classe dérivée qui redéfinie la méthode de création des différentes pizzas de sa carte.

1.9.4. Structure



1.9.5. Participants

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

1.9.6. Collaboration

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

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

2- La classe `Factory` possède une méthode `factoryMethod()` statique avec un paramètre d'identification de l'objet à créer. 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(); }
  }
}
```

1.9.8. 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 abstraction et le niveau implémentation.

1.9.9. Exemple d'application

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

1.10. 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.

1.10.1. Problématique

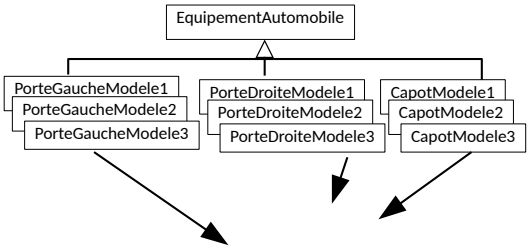
- On souhaite faire cohabiter des familles d'objets. Le choix de la famille est déterminé dynamiquement.
- On ne souhaite pas rendre accessible l'implémentation concrète d'une famille d'objets, le client n'aura accès qu'aux interfaces.

1.10.2. Solution

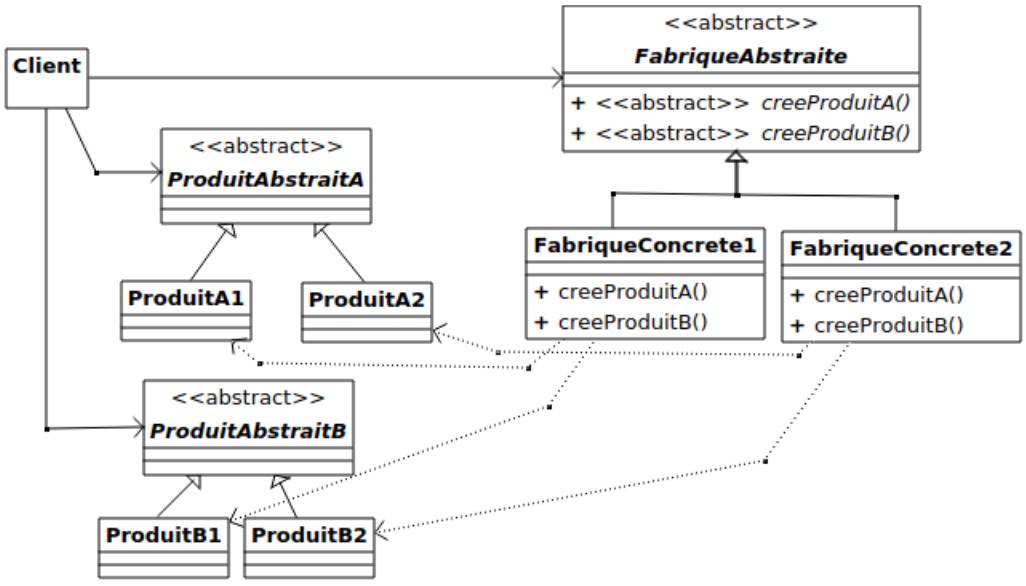
- Coordonner la création dans une hiérarchie de classes qui contiennent chacune les méthodes de création des objets d'une famille.

1.10.3. Métaphore : Chaîne de montage de voiture

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



1.10.4. Structure



1.10.5. Constituants

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

1.10.6. Collaboration

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.

1.10.7. 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 ses sous-classes.

1.10.8. 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.

1.10.9. 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()`…

1.11. Façade (*Facade*)

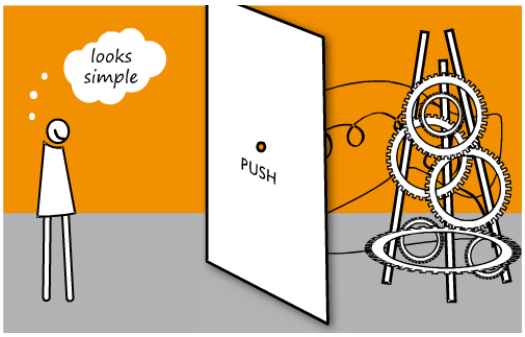
Fournir une interface unifiée pour un ensemble d'interfaces d'un sous-système.

1.11.1. 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.

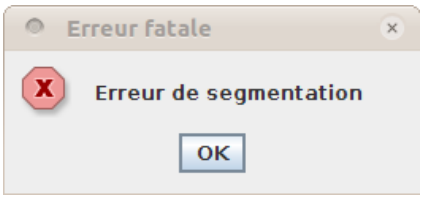
1.11.2. Solution

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

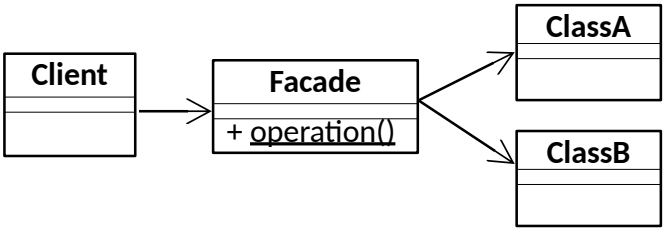


1.11.3. Exemple : Fenêtre d'alerte en Java Swing

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()`.



1.11.4. Structure



1.11.5. 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.
- **ClassA, ClassB** : des classes qui implémentent les fonctionnalités du sous-système et gèrent les

travaux assignés par l'objet façade. Ces classes ne connaissent pas la façade.

1.11.6. Collaboration

- 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.

1.11.7. 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.

1.11.8. 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 (visibilité privée).

```
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.

1.12. Itérateur (*Iterator, Cursor*)

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

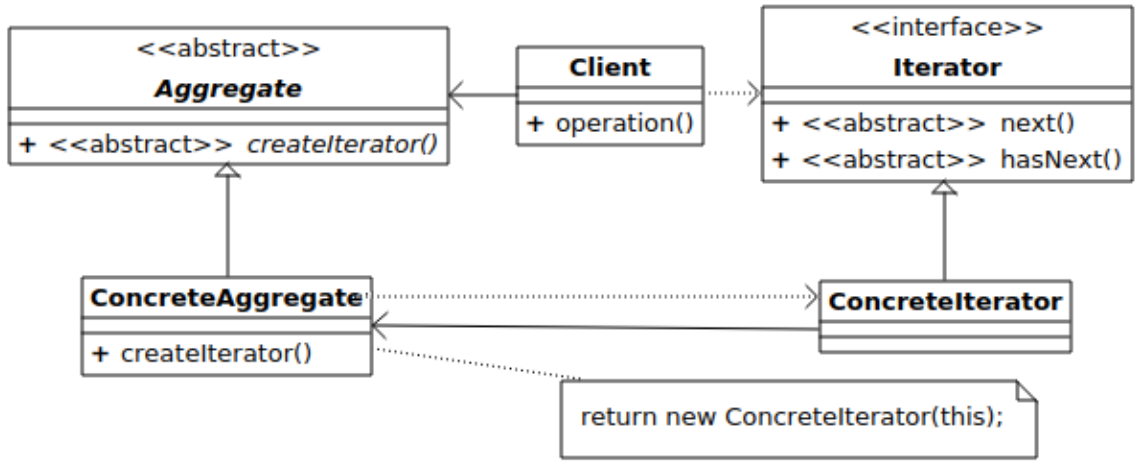
1.12.1. Problématique

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

1.12.2. Solution

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

1.12.3. Structure



1.12.4. Constituants

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

1.12.5. Conséquences

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

1.12.6. Considérations d'implémentation

Implémentation en Java

La classe `Aggregate` doit implémenter l'interface Iterable qui définit la méthode `iterator()`. Elle permet en plus de bénéficier directement de la méthode `foreach()` sans rien ajouter.

Allocation des itérateurs

- Les itérateurs doivent être alloués dynamiquement par une Fabrique avec la méthode `createIterator()`.
- En C++, 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()` pour des parcours aléatoires.

Variante : Itérateur nul

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

1.12.7. Exemple dans l'API Java

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

1.13. Médiateur (*Mediator*)

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

1.13.1. Problématique

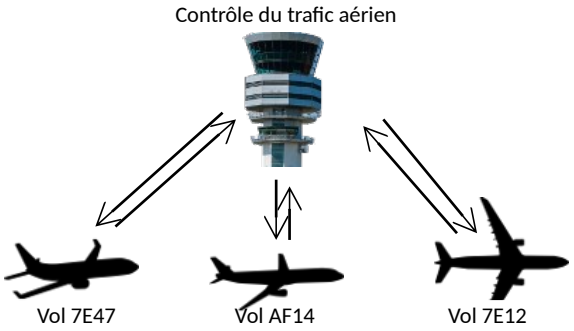
- Quand les objets d'un ensemble dialoguent de façon bien définie, mais très complexe. Le résultat des interdépendances est non-structuré et difficile à appréhender.
- Quand la réutilisation d'un objet est très difficile, du fait qu'il fait référence à beaucoup d'autres objets et communique avec eux.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

1.13.2. Solution

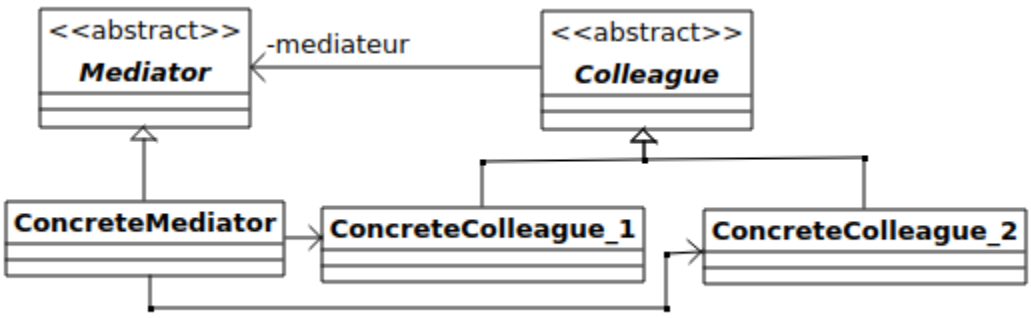
- Créer un objet qui organise la communication entre un ensemble d'objets.

1.13.3. 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 de 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.



1.13.4. Structure



1.13.5. Constituants

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

1.13.6. Collaboration

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.

1.13.7. 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 relations 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.

1.13.8. Considérations d'implémentation

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

1.14. Monteur (*Builder*)

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

1.14.1. Problématique

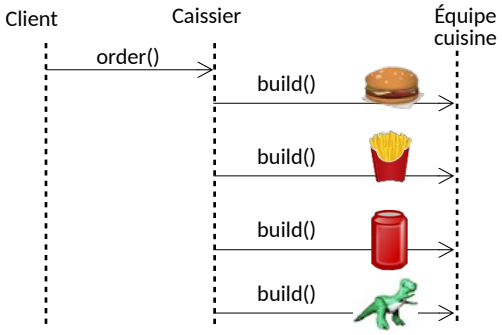
- La création d'un objet s'effectue par parties qui peuvent être variables.
- La création d'un objet nécessite beaucoup d'arguments dont certains sont optionnels. On veut éviter l'anti-patron « Constructeur télescopique » (voir page 49).
- **Ce qui varie** : les parties d'un composant générique.

1.14.2. Solution

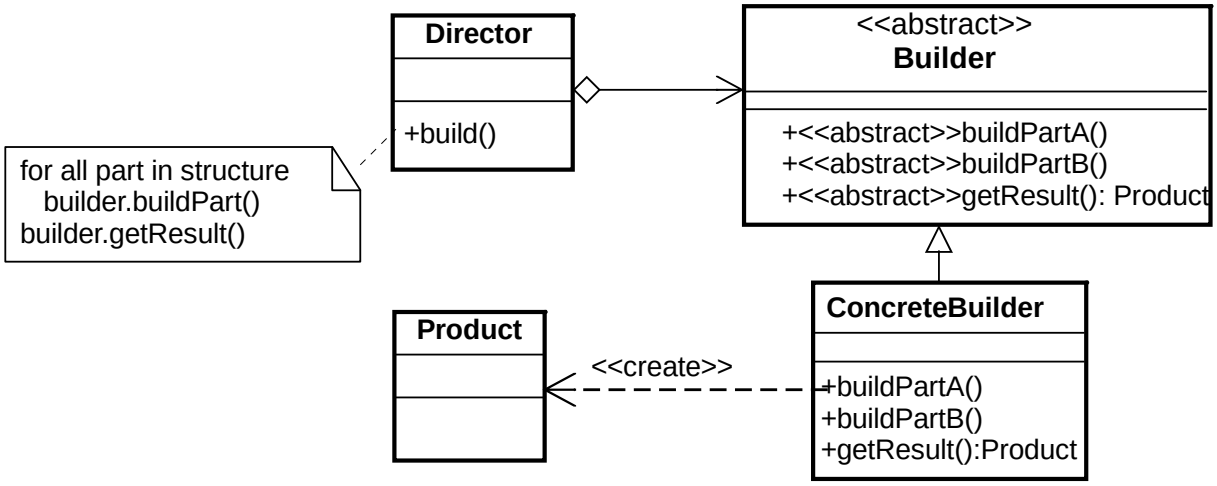
- Encapsuler la construction d'un objet par la liste de ses parties et laisser la responsabilité de l'algorithme de création du produit à une classe externe.

1.14.3. Métaphore : Chaîne de restauration rapide

La composition d'un menu enfant dans un fast-food utilise un monteur. Le restaurant propose plusieurs menus tout faits 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



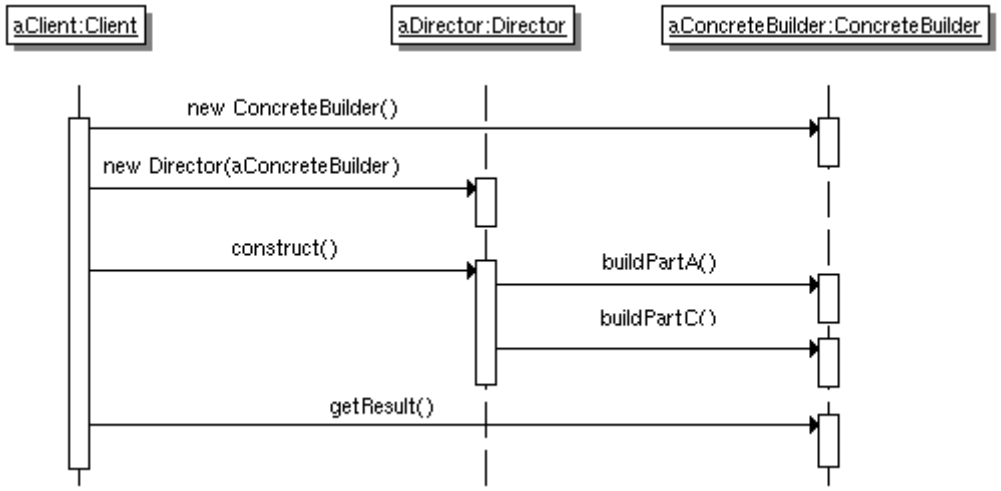
1.14.5. Constituants

- **Builder** (eg. Équipe cuisine) : une classe qui définit l'interface abstraite pour la création de parties d'un objet Produit.
- **ConcreteBuilder** : une classe qui construit et assemble des parties du produit par implémentation de l'interface `Builder`. Elle fournit aussi une interface pour la récupération du résultat.

- **Director** (eg. Caissier) : une classe qui construit un objet en utilisant l'interface de Monteur.
- **Product** (eg. Menu) : la classe qui représente l'objet complexe en cours de construction.

1.14.6. Collaboration

- Le client crée l'objet `Director` en lui passant une instance de `ConcreteBuilder`.
- 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 ConcerteBuilder().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

La classe `StringBuilder` est un exemple de monteur. Une chaîne de caractères est construite à partir appels successifs de la méthode `append()`.

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

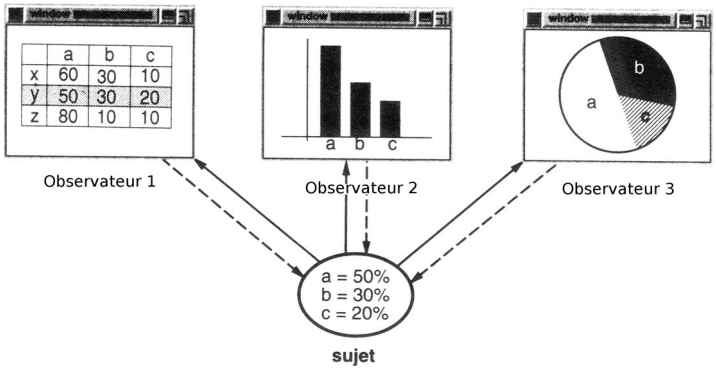
- Informer une liste variable d'objets qu'un événement a eu lieu.
- **Ce qui varie** : la liste des objets intéressés par des événements de modification d'un objet de référence.

1.15.2. Solution

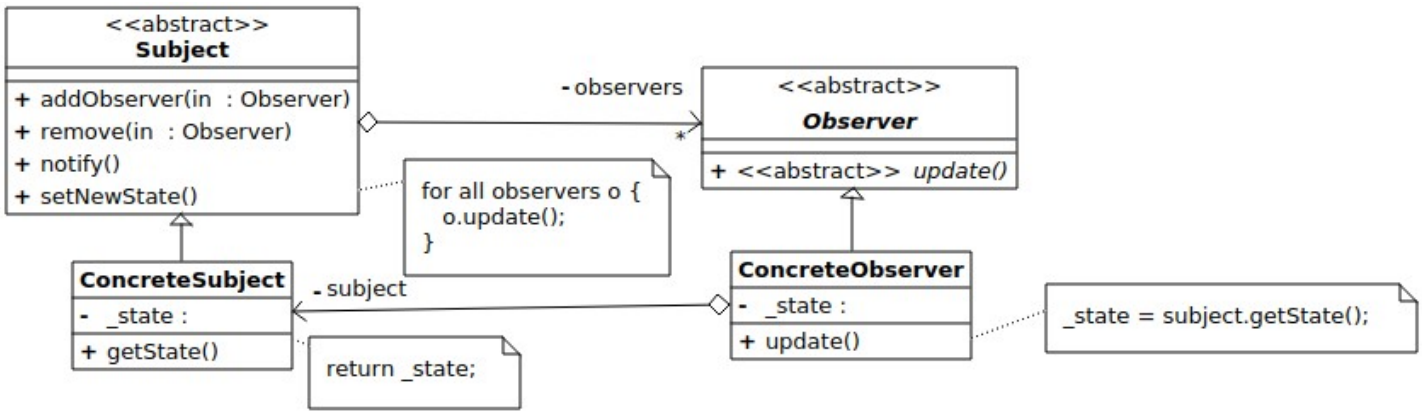
- Le modèle de base est constitué d'un sujet (détenteur de données) et d'observateurs. Tous les observateurs reçoivent une notification chaque fois que le sujet subit une modification.
- Les observateurs doivent s'**enregistrer** dynamiquement auprès du client.

1.15.3. Exemple : Tableur

Un tableur visualise des données sous plusieurs formes : un tableau contenant les chiffres bruts et des représentations graphiques type camembert ou histogramme que l'utilisateur choisit d'ajouter dynamiquement. Dès que qu'une donnée est changée sur le calque, toutes les représentations graphiques sont immédiatement mises à jour.



1.15.4. Structure



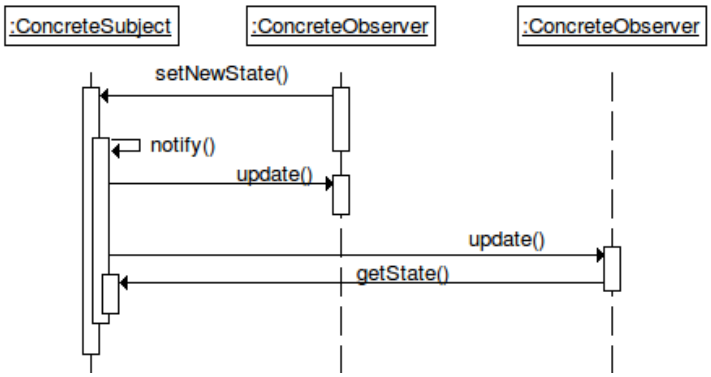
1.15.5. Composants

- **Subject** : il connaît les objets observateurs, car ces derniers s'enregistrent auprès de lui. Il doit les avertir dès qu'un événement a eu lieu.
- **ConcretSubject** (eg. Tableur) : il mémorise les états qui intéressent les objets observateurs

concrets et envoie une notification à ses observateurs par la méthode `notify()`.

- **Observer** : 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()`.
- **ConcreteObserver (**eg. Histogramme) **:** Il fait l'implémentation de l'interface de mise à jour de l'observateur pour conserver la cohérence de son état avec le sujet. Il est responsable de son inscription auprès du sujet en utilisant la méthode `addObserver()` du sujet.

1.15.6. Collaboration



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 pour le processus de notification :

- **push** : le sujet envoie aux Observateurs des informations détaillées sur les modifications. Dans ce cas, le sujet risque d'envoyer aux observateurs plus d'information qu'ils n'ont besoin chacun.

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

- **pull** : le sujet n'envoie aucune 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 );
```

- Implémentation en Java : Java fournit 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);
```

1.15.9. Exemple dans l'API Java

Les `listeners en Java et` les objets graphiques (bouton, boîte de dialogue) JavaFX.

1.16. Patron de méthode (*Template Method*)

Définir 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. Les étapes abstraites seront surchargées de façon concrète dans des sous-classes spécialisées. Chaque classe concrète donne alors une implémentation complète de l'algorithme.

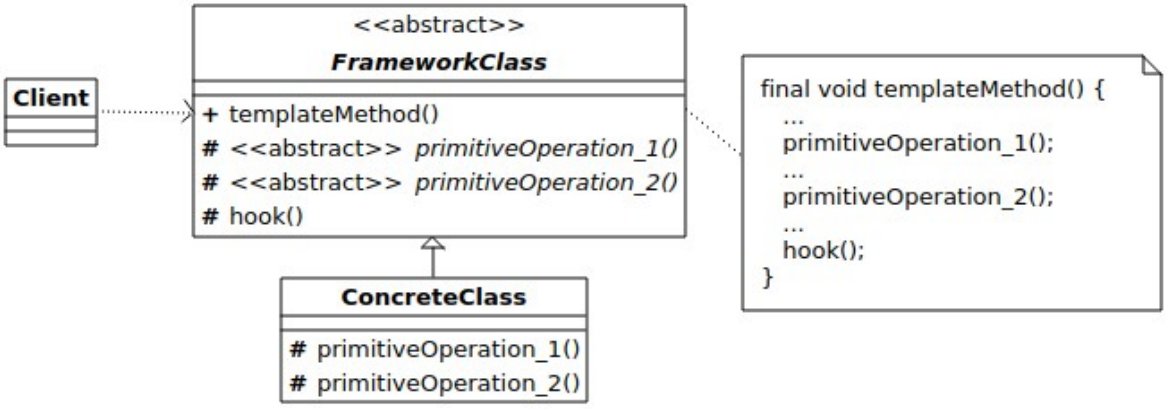
1.16.3. Métaphore : Jeu de société

Les étapes d'un jeu de société (eg. Monopoly, les petits chevaux) sont 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, chaque jeu 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 : une classe abstraite qui fournit l'algorithme utilisant certaines opérations abstraites spécialisées dans des sous-classes concrètes (patron de méthode). Elle définit aussi des opérations primitives abstraites, rendues concrètes par des sous-classes destinées à implémenter diverses étapes d'un algorithme.
- **ConcreteClass** : une classe qui 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();
  protected void hook(){ }
}

public class ConcreteClass extends FrameworkClass {
  @Override
  protected void primitiveOperation() { /* code spécifique obligatoire */ }
  @Override
  protected void hook() { /* code spécifique si nécessaire */ }
}
```

1.17. Pont (*Bridge*)

Découpler une abstraction de son implémentation afin que chacune puisse être modifiée indépendamment de l'autre.

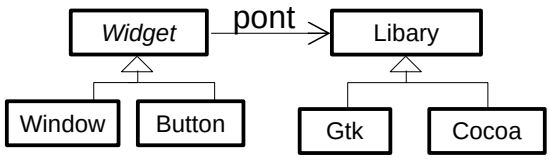
1.17.1. Problématique

- 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 au nombre d'abstractions multiplié par le nombre d'implémentations pour couvrir toutes les variations possibles.
- **Ce qui varie** : les abstractions et les implémentations peuvent évoluer indépendamment l'une de l'autre.

1.17.2. Solution

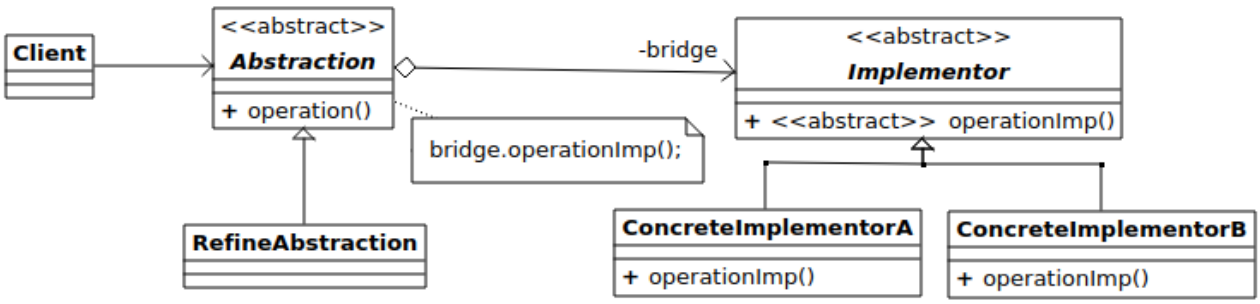
- Le patron Pont traite ces problèmes en séparant l'abstraction et son implémentation et en les reliant par une association : le pont. Il y a deux hiérarchies de classes distinctes.

1.17.3. Exemple : bibliothèque d'interface graphique



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

1.17.4. Structure



1.17.5. Constituants

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

1.17.6. Collaboration

- 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.

1.17.7. Conséquences

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

1.18. Procuration (*Proxy*, *Surrogate*)

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

1.18.1. Problématique

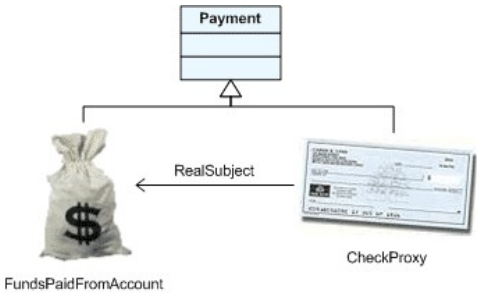
- On souhaite contrôler l'accès à un objet dans le but de différer sa création ou son initialisation avant son utilisation effective.
- On a besoin de références à un objet qui soient plus sophistiquées qu'un simple pointeur.

1.18.2. 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.

1.18.3. Métaphore : Chèque bancaire

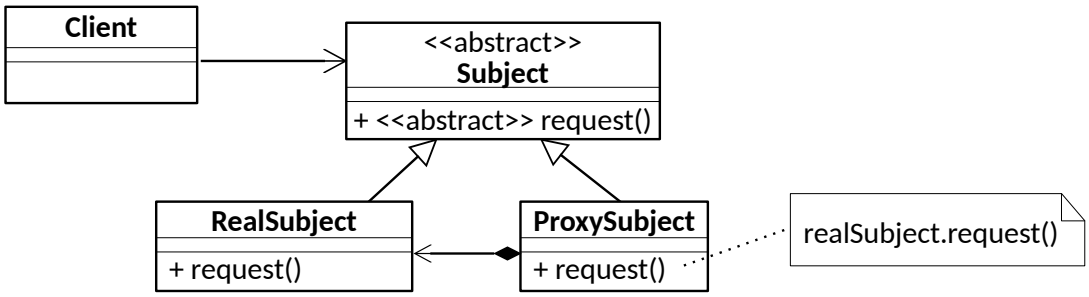
Un chèque est un subrogé pour une somme d'argent d'un compte bancaire. Il peut être utilisé à la place du numéraire pour faire des achats, et il peut à tout moment donner accès au numéraire.



1.18.4. 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 (eg, Le type `Image` de Java).
3. Une procuration de protection contrôle l'accès à l'original : droits d'accès.
4. Une procuration vérifie, avant d'y accéder, qu'un objet réel est verrouillé pour être le seul à le modifier.
5. Une référence intelligente réalise des opérations supplémentaires lors de l'accès à un objet. L'exemple typique est celui des **smart pointers** en C++ :
   - Le proxy décompte des références faites à un objet réel ;
   - Le proxy détruit l'objet réel quand le décompte est égal à 0.

1.18.5. Structure

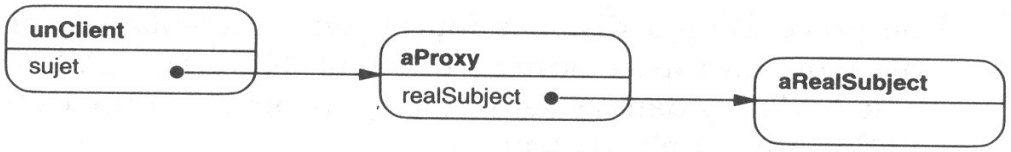


1.18.6. Constituants

- **Subject** : une classe abstraite qui définit une interface commune pour `RealSubject` et `Proxy`, de sorte que la classe `Proxy` puisse être utilisée à la place de `RealSubject`.
- **ProxySubject** (eg. CheckProxy) : une classe qui 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. Il présente la même interface que l'objet ce qui le rend transparent pour le client.
- **RealSubject** (eg. FundsPaidFromAccount) **:** une classe qui définit l'objet réel représenté par la procuration.

1.18.7. Collaboration

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



1.18.8. Considérations d'implémentation

- Il est préférable que la procuration et l'objet réel partage 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.

1.18.9. Exemple dans l'API Java

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

1.19. Singleton (*Singleton*)

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

1.19.1. 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 (eg, incohérences, écrasement de données).

1.19.2. Solution

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

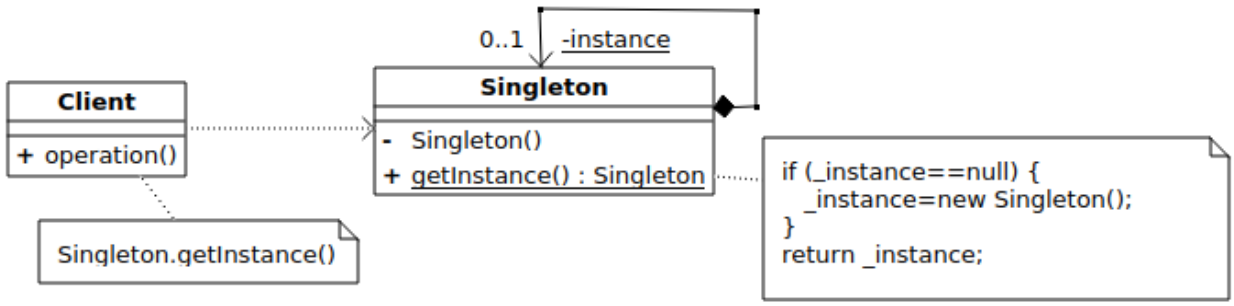
1.19.3. Métaphore : Équipe championne du monde de football

L'équipe championne de monde de football en cours est typiquement un Singleton. L'appellation « équipe championne du monde de football » est un point d'accès global qui identifie l'équipe de façon univoque.

1.19.4. Exemples

- 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 pour ces fenêtres.

1.19.5. Structure



1.19.6. Constituants

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

1.19.7. Collaboration

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

1.19.8. 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. Ceci fait que ce patron est aussi un *anti-patron* (cf. les anti-patrons STUPID). Son utilisation doit donc être justifiée comme incontournable.

1.19.9. 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 public `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 »). 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.

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

Remarque : La Programmation Orientée Aspect (POA) offre une solution plus élégante pour l'implémentation de ce patron. Elle permet de conserver un constructeur d'apparence normale.

1.20. Stratégie (Strategy)

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

1.20.1. Problématique

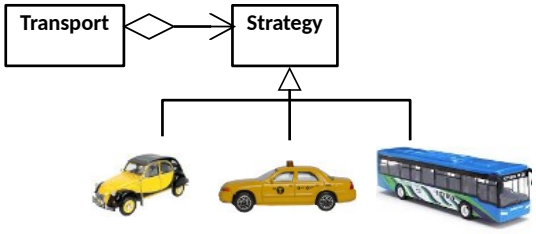
- On souhaite que la sélection d'un algorithme dépende du client à l'origine de la demande.
- **Ce qui varie** : l'algorithme implémentant un service.

1.20.2. Solution

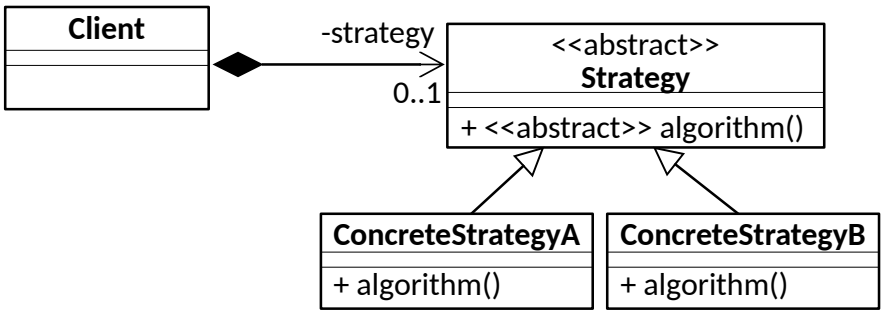
- Encapsuler les algorithmes dans une hiérarchie et lier la hiérarchie à l'objet appelant par composition.
- Séparer la sélection de l'algorithme de son implémentation.
- Le patron permet une sélection dynamique basée sur le contexte.

1.20.3. 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.



1.20.4. Structure



1.20.5. Constituants

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

1.20.6. Conséquences

⊕ Famille d'algorithmes apparentés : tous les algorithmes ont la même interface.

⊕ Alternative aux dérivations de Client en sous-classes : la stratégie n'est pas codée en dur. Elle peut être choisie dynamiquement.

⊕ Des implémentations différentes peuvent être proposées pour un même comportement.

1.20.7. Variante

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 quand elles évoluent toutes 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 et le calcul de la taxe de douane.

1.20.8. Considérations d'implémentation

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

1. Les informations sont passées sous forme de paramètres :

```
public void algorithm( par1, par2, par3 );
```

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 **Messager** pour le paramètre (ie, une classe qu'avec des attributs publics, voir section 2.2).

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

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

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

Dans ce cas, il faut un lien privilégié entre les deux classes, en faisant par exemple des Stratégies des classes internes. Cela peut aussi se réaliser en C++ avec des classes amies.

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

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

1.20.9. 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.

1.21. Visiteur (Visitor)

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

1.21.1. Problématique

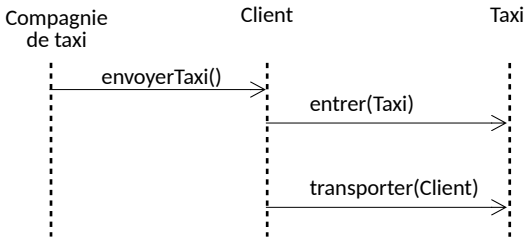
- Différentes opérations distinctes doivent être réalisées sur les nœuds d'une structure composite, mais on ne souhaite pas polluer les classes des nœuds avec ces opérations, ni tester le type de chaque nœud pour transtyper le pointeur avec le bon type avant de réaliser l'opération.
- Ce qui varie :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

1.21.2. Solution

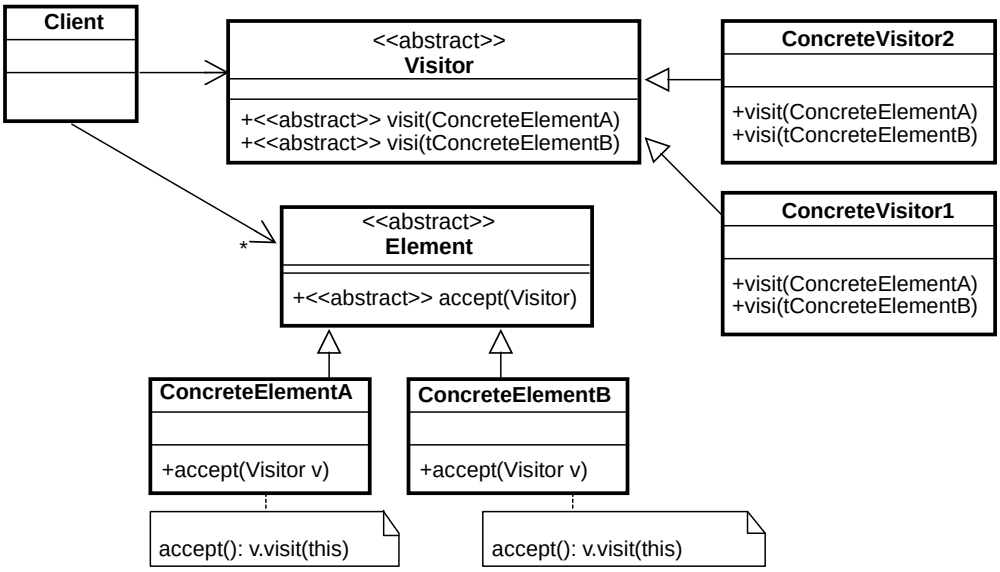
- Externaliser les opérations d'une structure d'objets dans une hiérarchie séparée et ajouter une méthode dans la structure d'objets pour accueillir des instances des opérations.

1.21.3. 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.



1.21.4. Structure



1.21.5. Constituants

- **Element** : une interface qui déclare une opération `accept(Visitor)` pour chaque classe ConcreteElement de la structure d'objets.
- **ConcreteElement** (eg. client) : une classe qui réalise le codage d'une opération `accept()` qui

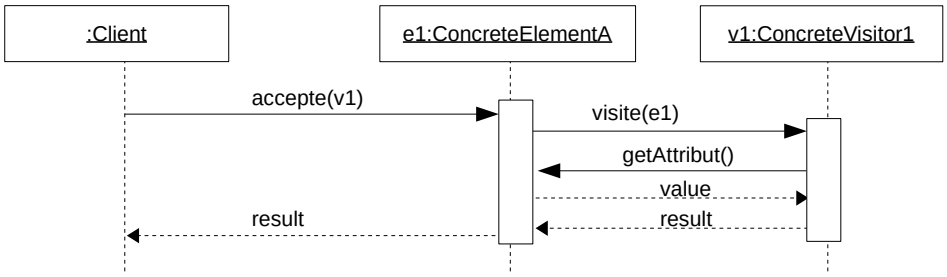
prend un paramètre de type `Visitor`.

- **Visitor** : une interface qui définit les opérations réalisables sur les éléments.
- **ConcreteVisitor** (eg. taxi) : une classe qui détient 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.

1.21.6. Fonctionnement

Une opération `operationA()` qui devraient être réparties dans chacun des objets de la hiérarchie `Element` est maintenant regroupée dans une seule classe sous la classe abstraite Visitor. L'accès à cette opération se fait en passant un pointeur ce `Visiteur` dans l'objet `Element`. La méthode `accept()` fait l'indirection.

1.21.7. Collaboration



1.21.8. Considérations d'implémentation

La séquence d'appel pour appliquer une opération implantée dans l'objet `ConcreteVisitor1` à une structure d'objet représentée par l'objet `ConcreteElementA` est :

```
ConcreteElement element = new ConcreteElementA();
element.accept(new ConcreteVisitor1());
```

1.21.9. Conséquences

⊕ Le patron Visiteur facilite l'addition de nouvelles opérations 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.

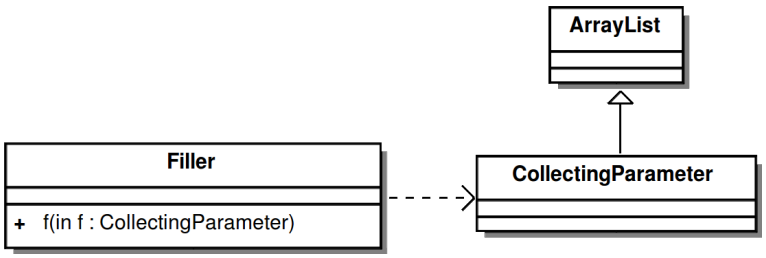
2.1.1. Problématique

- 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 est 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, Arrays.asList(new Object(0), new Object(1)));
}
```

- Une alternative est l'utilisation d'un dictionnaire, tel que `Hashtable` en Java, qui permet en plus de nommer les valeurs collectées.

2.2. Messager (*Messenger*)

Englober des données liées sémantiquement dans un objet pour le passer comme paramètre unique de méthodes au lieu de passer toutes les données séparément.

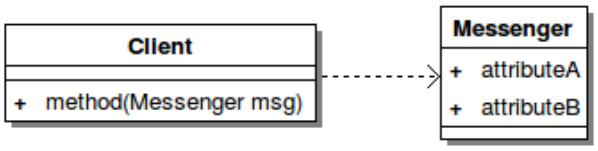
2.2.1. Problématique

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

2.2.2. Solutions

- Une classe avec des données publiques. Puisque le but d'un messager est seulement de porter des données, ces données sont déclarées **publiques** pour un accès facile.

2.2.3. Structure



2.2.4. Constituants

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

2.2.5. Considérations d'implémentation

Un dictionnaire (`Hashtable`) est une variante d'implémentation qui présente l'avantage de nommer les paramètres dynamiquement, à la manière des paramètres de fonction des langages Python ou Dart.

2.2.6. Exemple

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

```
final class Point { // un messager
  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) {
    this.x = p.x;
    this.y = p.y;
    this.z = p.z;
  }
  public String toString() {
    return "x: " + x + " y: " + y + " z: " + z;
  }
}
class Vector {
  public int vx, vy, vz;
  public Vector(int vx, int vy, int vz) {
    this.vx= vx;
    this.vy= vy;
    this.vz= vz;
  }
}
class Space {
  public static Point translate(Point p, Vector v) {
    p = new Point(p);
    p.x = p.x + vx;
    p.y = p.y + vy;
    p.z = p.z + vz;
    return p;
  }

  public static void main(String[] args) {
    Point p1 = new Point(1, 2, 3);
    Point p2 = Space.translate(p1, new Vector(1, 1, 1));
    String result = "p1: " + p1 + " p2: " + p2;
    System.out.println(result); // -> "p1: x: 1 y: 2 z: 3 p2: x: 2 y: 3 z: 4;
  }
}
```

2.2.7. Remarque

Ce patron va à l'encontre du principe d'encapsulation en révélant la représentation interne de la classe Messager. 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 messager est typiquement représenté par une *structure*.

2.2.8. Variante : Data Transfert Object (*DTO*)

Le patron Data Transfer Object définit 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 messager qui dispose en plus

de la capacité de sérialisation permettant de l'exploiter sur le réseau.

2.2.9. Variante : Data Access Object (*DAO*)

Le patron Data Access Object définit 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 Messager avec un ensemble de mutateurs et d'accesseurs.

2.3. Pool d'objet (*Object Pool*)

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

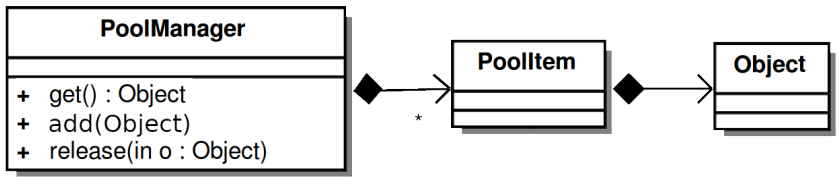
2.3.1. Problématique

- Le patron Singleton est restreint à une seule instance.

2.3.2. Solution

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

2.3.3. Structure



2.3.4. 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 List<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 IllegalStateException(item + " not found");
   }
}
```

2.3.5. Exemple

Les bases de données 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);
   }
}
```

2.3.6. Remarque

Ce patron est finalement une forme spécialisée du patron procuration.

2.4. 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.

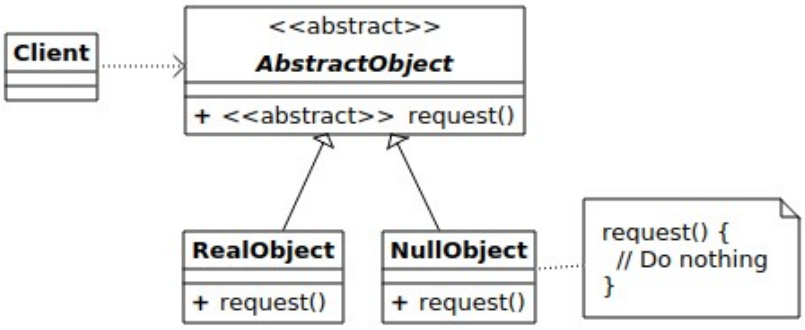
2.4.1. Problématique

- Comment l'absence d'un objet peut être traitée de façon transparente si la référence de l'objet peut être nulle ?
- Éviter les problèmes induits par le test « `if (object==null)` » source de nombreux bugs.

2.4.2. 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 Cette classe est un gain par rapport à l'utilisation de la valeur spéciale « null » qui doit toujours être vérifiée avant l'accès à n'importe quel objet qui utilise l'interface abstraite.

2.4.3. Structure

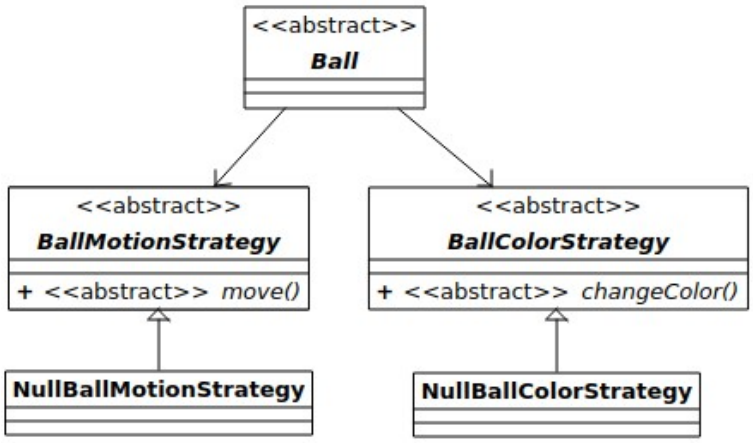


2.4.4. Constituants

- **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.

2.4.5. Exemple

Considérons le cas d'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 abstraite `Balle` et en utilisant un patron Stratégie qui propose une variété de mouvements de la balle et un autre patron Stratégie qui varie 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

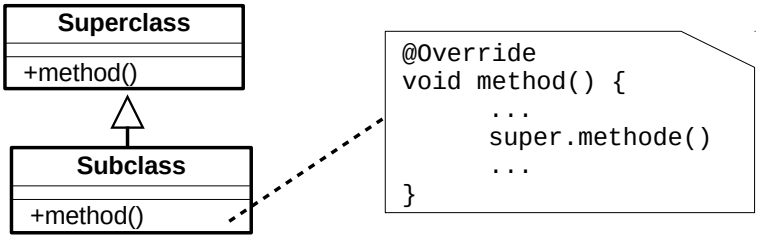
Obliger les utilisateurs d'une classe à redéfinir une méthode de la classe en compléter le comportement de base afin de pouvoir l'utiliser.

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 redéfinir la méthode en faisant appel au code de la méthode de base puis en ajoutant 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 est respecté et donc l'erreur ne pourra se révéler qu'à l'exécution (dans le meilleur des cas).

3.1.2. Structure

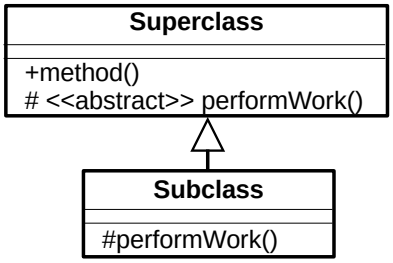
Le code de la méthode de la sous-classe fait appel au code de la méthode de base.



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 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 que si la méthode de base est redéfinie, la méthode redéfinie appelle bien la méthode de base.

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

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 », pas plus qu'en génie logiciel.

3.3.2. Solution

Selon la règle de Coad, l'héritage ne doit être utilisé que pour des relations de type « est-un » et pas « a-un ». 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 développeur 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

Définir plusieurs constructeurs dans une classe, chacun appelant un autre constructeur dans la hiérarchie qui a plus de paramètres que lui-même, fournissant des valeurs par défaut pour les paramètres supplémentaires.

3.6.1. Problématique

Le problème s'illustre sur l'exemple suivant de la classe Pizza :

```
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){
     ...
  }
}
```

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, une inversion des valeurs ne peut pas être détectée par le compilateur. De plus, il est difficile d'anticiper 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.

3.6.2. Solution

Une solution est d'utiliser le patron Monteur.