Contenu du cours

- **Introduction**
  - principes de conception
  - introduction aux design patterns
    - ce qu'ils sont et ce qu'ils ne sont pas
    - bénéfices et inconvénients
    - leur classification des patterns du GOF
- **Les Design Patterns du GOF**
  - patterns de création
  - patterns de structure
  - patterns de comportement

Objectifs

- **Avoir à l'esprit les principes et pratiques de la COO**
- **Comprendre la classification et la documentation des DP**
- **Avoir une idée de l'intention des patterns du GoF**
- **Avoir une idée de solutions d'implémentation de quelques patterns clés**
- **Comprendre et connaître le vocabulaire pour mieux communiquer dans une équipe de conception**

- **A partir du problème, imaginer une solution, la corriger en utilisant le pattern adéquat et comprendre en quoi il s'agit d'une solution efficace**

Evaluation de l'UE

- **Nombre d'heures**
  - CM : 4h30
  - TP : 15h
    - utilisation d'un outil de modélisation (StarUML)
    - implémentation en Java de certains patterns
- **Contrôle final : 1h30 de devoir sur table en janvier – 50%**
- **Contrôle continu – 50%**
  - évaluation de ce qui est fait pendant les séances au CERI (plutôt que sur le rendu final…)

Bibliographie

- **Design Patterns. Elements of Reusable Object Oriented**
  - E. Gamma, R. Helm, R. Johnson et J. Vlissides. Addison Wesley : 1995.
- **Head First - Design Patterns**
  - E. Freeman, E. Robson, B. Bates, K. Sierra. O'Reilly : 2014.
- **Clean Code - A Handbook of Agile Software Craftmanship**
  - R.C. Martin. Prentice Hall : 2008.
- **Design Patterns Explained Simply**
  - A. Shvets.
- **https://sourcemaking.com/design_patterns**

Introduction

- Principes de conception
- Les patterns du GOF

Rappels Orientation Objet

- **Abstraction – Encapsulation - Modularité**
- **Classes concrètes, classes abstraites, interfaces**
- **Polymorphisme**
  - héritage
  - surcharge
  - sur définition
  - généricité
- **Relations entre classes**
  - association
  - agrégation – composition
  - généralisation/spécialisation - héritage

Principes de conception

- **Modularité**
  - principe de base de l'OO
  - simple à gérer
- **Cohésion**
  - mesure combien les fonctionnalités d'un même module se rapportent à un même domaine
- **Couplage**
  - niveau d'interactions entre les différents modules d'un système
  - favorise l'indépendance des modules entre eux
  - couplage faible = conception de bonne qualité
- **Réutilisabilité**
  - librairies, frameworks, …

Principes de conception

- **Les changements dans les besoins sont les principales causes de dégradation d'une conception : il faut donc des moyens de protéger la conception initiale**
- **Un module devrait être extensible mais pas modifiable**
  - de façon à changer ce que le module fait, sans changer le code source
  - techniques multiples : polymorphisme statique ou dynamique (templates)
- **On doit pouvoir substituer une sous-classe à sa classe de base**
  - l'utilisateur d'une classe de base doit pouvoir continuer à fonctionner correctement si on lui transmet une classe dérivée
- **Un client doit dépendre d'abstractions plutôt que de classes concrètes**
  - les modules contenant des implémentations ne devraient jamais être directement accédés, et eux-mêmes devraient dépendre d'abstractions
  - un lien de dépendance doit avoir pour cible une interface ou une classe abstraite
- **Une interface spécifique à un client est préférable à une d'intérêt général**
  - on construira une classe d'implémentation par héritage multiple d'interfaces

Penser "interface"

- On définit des interfaces plutôt que des implémentations
- Utiliser les classes abstraites/interfaces pour définir des ensembles de fonctionnalités communes à plusieurs classes
- Utiliser la substituabilité : déclarer les arguments comme des instances d'interface

Conséquence

- Moyen de renforcer la séparation logique entre la déclaration et l'implémentation
- Les clients ne dépendent pas directement des objets qu'ils utilisent (qui doivent respecter les spécifications d'interface)
- On évite les problèmes liés à l'héritage multiple

Ce que sont les DP

- Ils sont partout
  - on les retrouve dans de nombreux situations humaines dont le développement informatique : musique, architecture, langage, ergonomie, psychologie, linguistique, stylisme, décoration …
- Ils permettent une façon plus systématique de concevoir
- Ils synthétisent l'expérience des autres
- Ils sont de trop haut niveau pour être implémentés sous forme de librairies
- Ce sont juste des idées réutilisables, qui décrivent une possible partie de la solution
- Ils peuvent représenter une solution à un problème dans un contexte précis

Et ce qu'ils ne sont pas

- Ils ne sont pas des implémentations d'un problème
- Ils ne sont pas des composants : ils dépendent d'un environnement
- Ce ne sont pas des règles : ils doivent être adaptés au problème
- Ils ne sont pas une méthodologie : ils n'aident pas à décider, ils sont une solution possible
- Ce ne sont pas des solutions toutes faites à appliquer
- Il s'agit pas de réutilisation de code mais de réutilisation de stratégie

Patterns vs fonctionnalités de langage

- Les DP sont indépendants des langages (pas seulement OO)
  - mais certains patterns sont plus difficiles à implémenter dans certains langages et totalement inutiles dans d'autres
- Influence des langages sur les patterns
  - certains patterns utilisent des concepts spécifiques à certains langages
  - certains langages implémentent des patterns de bas niveau
  - les patterns complexes doivent être intensément transformés pour être implémentés dans un langage
- Influence des patterns sur les langages de programmation
  - les langages les plus récents mettent en œuvre de façon naturelle les idées sous-jacentes des patterns

Bénéfices

- **Permettre la réutilisation des conceptions réussies**
  - c'est la pratique des concepteurs expérimentés
  - en général des conceptions "élégantes" auxquelles ne pense pas un débutant
  - elles ont souvent été testées sur plusieurs systèmes
- **Réduire les dépendances des implémentations**
  - indépendantes de systèmes spécifiques ou de langage
- **Faciliter l'évolution du soft**
  - on ajoute de nouvelles fonctionnalités facilement sans détruire l'existant
  - on les combine pour des solutions plus complexes
- **Faciliter la communication en définissant un langage commun**
  - on améliore la documentation de la conception
  - on améliore la compréhension de la conception

Se focaliser sur des abstractions de haut niveau pour améliorer la qualité de la conception

Inconvénients

- **Ils impliquent une intense activité humaine de compréhension et pour identifier le pattern qui conviendra**
  - complexe par nature
  - nombreux : les diagrammes UML se ressemblent mais leurs objectifs sont différents
- **Ils sont validés par l'expérience et des discussions plutôt que des tests automatiques**
  - ne conduisent donc pas à de la réutilisation systématique de code
- **Ils sont définis à différents niveaux : certains patterns en utilisent d'autres**
- **Ils peuvent conduire à une surcharge : augmente la difficulté de compréhension d'une conception ou d'une implémentation en ajoutant des indirections ou augmentant la taille du code**

Historique des DP de software

- Issus du travail de l'architecte Christopher Alexander qui a approfondi les moyens d'améliorer le processus de conception de bâtiments et de zones urbaines
- 1987 - Cunningham et Beck utilisent ses idées pour développer un petit langage de patterns pour Smalltalk
- 1990 - The Gang of Four (Gamma, Helm, Johnson and Vlissides) : ils commencent à travailler à la définition d'un catalogue de DP
- 1991 - Bruce Anderson propose le premier atelier de travail à OOPSLA
- 1993 - Kent Beck et Grady Booch sponsorisent le premier meeting de ce qui sera connu sous le nom du Hillside Group
- 1994 – Première conférence Pattern Languages of Programs (PLoP)
- 1995 – Le Gang of Four (GoF) publie le livre des *Design Patterns*

Les patterns du Gang of Four

- Plusieurs niveaux d'abstraction
  - au niveau de l'application complète ou d'un sous-système
  - solution à un problème de conception dans un contexte particulier
  - simple outil réutilisable comme une liste chaînée, une hashtable, …
- Les DP du GoF se situent au milieu de ces niveaux d'abstraction
- Les DP du GoF sont “des descriptions d'objets et de classes qui communiquent et qui sont personnalisés pour résoudre un problème général de conception dans un contexte particulier.”

Classification selon GoF

- **Objectif – ce que fait un pattern**
  - Patterns de création
  - ils concernent le processus de création des objets
  - Patterns de structure
  - ils concernent la composition des classes et des objets
  - Patterns de comportement
  - ils concernent l'interaction des classes et des objets, en précisant comment ceux-ci interagissent et se distribuent les responsabilités
- **Portée – ce à quoi un pattern s'applique**
  - Patterns de classes
  - se concentrent sur la relation entre les classes et leurs sous-classes
  - impliquent la réutilisation par héritage
  - Patterns d'objets
  - se concentrent sur la relation entre objets
  - impliquent la réutilisation par composition

Catalogue

		OBJECTIFS		
		CREATION	STRUCTURE	COMPORTEMENT
PORTEE	CLASSE	Factory Method	Adapter (class)	Interpreter
				Template Method
	OBJET	Abstract Factory	Adapter (object)	Command
		Builder	Bridge	Iterator
		Prototype	Composite	Mediator
		Singleton	Decorator	Memento
			Facade	Observer
			Flyweight	State
			Proxy	Strategy
				Visitor
				Chain Of Responsibility

Description d'un pattern

- **Nom et classification du pattern :** transmet l'essence du pattern (Création, Structure, Comportement)
- **Problématique :** résumé court du type de problème que le pattern est supposé résoudre
- **Aussi connu comme :** autre nom possible du pattern
- **Motivation / Exemple :** un scénario qui illustre le problème de conception
- **Applicabilité :** dans quelles situations le pattern peut s'appliquer
- **Structure :** représentation graphique des classes en UML
- **Participants :** les classes et leurs responsabilités
- **Collaborations :** les collaborations entre les participants
- **Conséquences :** en quoi le pattern réalise ses objectifs
- **Implémentation :** pièges, indices, techniques dont il faut avoir conscience
- **Exemple de code :** code d'implémentation en C++ , Smalltalk, Java
- **Utilisations connues :** exemples du pattern dans des systèmes réels
- **Patterns associés**

Patterns de création

- **Abstract Factory**
  - permet la création de familles d'objets ayant un lien ou interdépendants
- **Builder**
  - déplace la logique de construction d'un objet en dehors de la classe à instancier, pour en permettre une construction partielle ou le simplifier
  - sépare la construction d'un objet de sa représentation
- **Factory Method**
  - permet de créer un objet à partir d'un type abstrait, tout en laissant la possibilité de choisir la classe à instancier
- **Prototype**
  - permet d'obtenir une instance complètement initialisée à copier ou cloner
- **Singleton**
  - classe dont seule une instance peut exister (ou un nombre limité)

Patterns de structure

- **Adapter :** faire correspondre les interfaces de classes a priori différentes
- **Bridge :** séparation de l'interface d'un objet de son implémentation
- **Composite :** une structure d'arbre entre objets simples et composés
- **Decorator :** ajout de responsabilités à des objets dynamiquement
- **Façade :** une classe unique qui représente un sous-système complet
- **Flyweight :** utilisation d'une seule instance partageant des informations afin d'optimiser la consommation de mémoire
- **Proxy :** un objet qui en représente un autre

Patterns de comportement

- **Chain of responsibility :** propager une requête entre objets chainés
- **Command :** encapsule une requête en tant qu'objet
- **Interpreter :** définir les éléments d'un langage dans une application
- **Iterator :** pour accéder séquentiellement aux éléments d'une collection
- **Mediator :** définit une communication simplifiée entre classes
- **Memento :** mémoriser et restaurer l'état d'un objet
- **Observer :** moyen de notifier des changements à plusieurs objets
- **State :** on modifie le comportement d'un objet sans changer son état
- **Strategy :** encapsule un algorithme dans une classe
- **Template method :** déporter des parties d'algo dans des sous-classes
- **Visitor :** définit une nouvelle opération d'une classe sans en changer

Patrons de création

Patterns de création

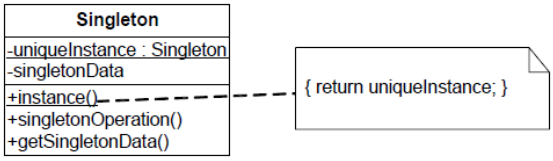
- **Singleton**
  - on s'assure qu'une classe ne peut avoir qu'une seule instance, et on en fournit un point d'accès
- **Factory Method**
  - on définit une interface qui permet de créer un objet, mais on laisse les sous-classes décider de quelle classe effectivement instancier
- **Abstract Factory**
  - procure une interface de création de familles d'objets interdépendants sans spécifier leurs classes concrètes
- **Builder**
  - on sépare la construction d'un objet complexe de sa représentation de façon à ce que le même processus de construction puisse créer plusieurs réalisations possibles de cet objet
- **Prototype**
  - on spécifie les catégories d'objets à créer à partir d'une instance prototype, et on crée des objets à partir de ce prototype

Singleton - Contexte

- **Garantit qu'une seule instance d'une classe peut être créée**
- **Fournit un accès global à cette instance**
- **Applicabilité**
  - lorsqu'une seule instance n'est possible et qu'on a besoin d'y accéder via un point connu
  - lorsque cette unique instance doit pouvoir être prolongée aux sous-classes, et que les clients doivent pouvoir utiliser une prolongation d'instance sans modification de code
- **Exemples**
  - gestionnaire d'impression unique pour une imprimante partagée
  - plusieurs threads nécessitant d'émettre des messages d'erreur dans une zone de messages partagée unique

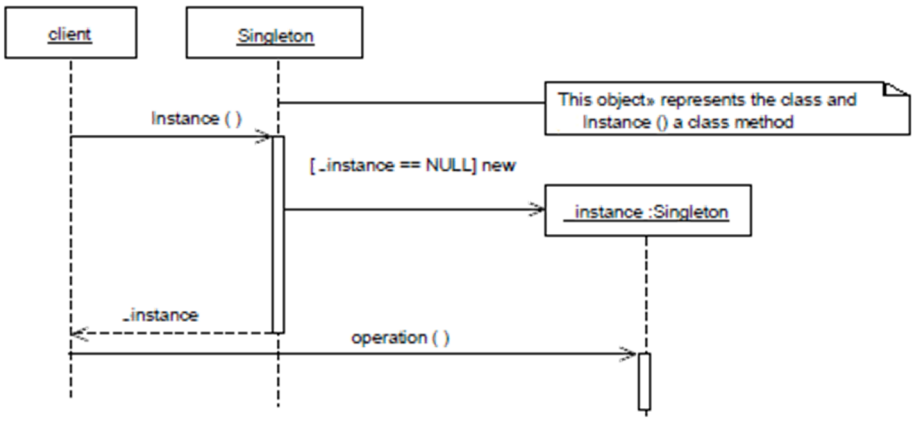
Singleton - Structure

- **On définit une opération de classe nommée "instance" qui fournit aux clients l'accès à cette unique instance**
- **Cette opération est chargée de la création de l'instance**



Singleton – Collaboration

- Un client ne peut utiliser cette instance singleton que via la méthode de classe "instance" de la classe



Singleton - Conséquences

- **Limite l'encombrement des packages**
  - pas de variable globale stockant cette unique instance
- **Permet un accès contrôlé à l'objet singleton**
- **Permet une extension par héritage**
- **Permet les modifications futures qui proposeraient plus d'une instance**
  - on peut contrôler le nombre d'instances
- **Mêmes points négatifs qu'une instance globale si mal utilisée**
  - problèmes en cas d'accès concurrents (applications multi-threads)
  - implémentation moins efficace qu'une variable globale

Singleton - Implémentation

- Une instance et une méthode statiques ; un constructeur privé

```
class Singleton {
     private static Singleton instance;
     public static Singleton getInstance(){
          if (instance == null)
               instance = new Singleton();
          return instance;
     }
     protected Singleton(){
          // initialisation des variables d'instance
     }
}
```


Singleton - Héritage

- Les instances avec lesquelles travaillent les clients doivent correspondent aux sous-classes
- L'instance est unique pour chaque sous-classe
  - on peut envisager une approche par registre de singletons
  - on y trouve une correspondance entre des noms (string) et les singletons
  - lorsque la méthode instance() a besoin d'un singleton, elle interroge le registre à partir du nom
  - on peut utiliser les templates
  - un template par type d'instanciation

Singleton – Implémentation par registre

- On utilise une interface commune pour toutes les classes Singleton

```
class Singleton {
   …
   public static void register(String name, Singleton s);
   protected static Singleton lookup(String name);
   private static Map<Name,Singleton> registry;
};
```


Singleton – Implémentation par template (C++)

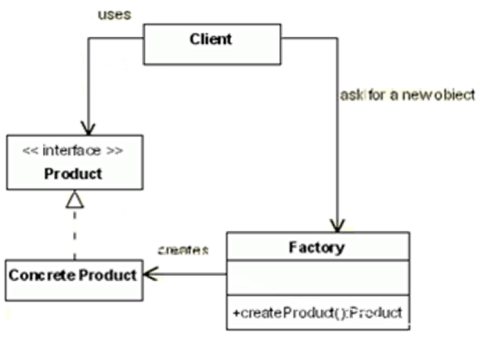
```
template <class T>
class Singleton
{
        public:
            static T* get_instance() {
                  if(!_instance) _instance = new T;
                  return _instance;
            }
        protected:
        private:
            static T* _instance;
    };
template <class T> T* Singleton<T>::_instance = NULL;
class A : public Singleton<A>{…};
```

Fabriques - Factories

- **Elles sont basées sur le principe de différer la création**
- **Ordre ascendant d'indirection :**
  - Simple factory
  - Factory Method,
  - Abstract factory

Simple Factory

- **Pas proposé par le GoF**
- **On crée des objets sans laisser le client gérer l'instanciation**
- **Le client utilisera l'objet créé à travers une interface commune**
- **Exemple :**
  connexion à une BD
- **Implémentation :**
  instruction if ou switch

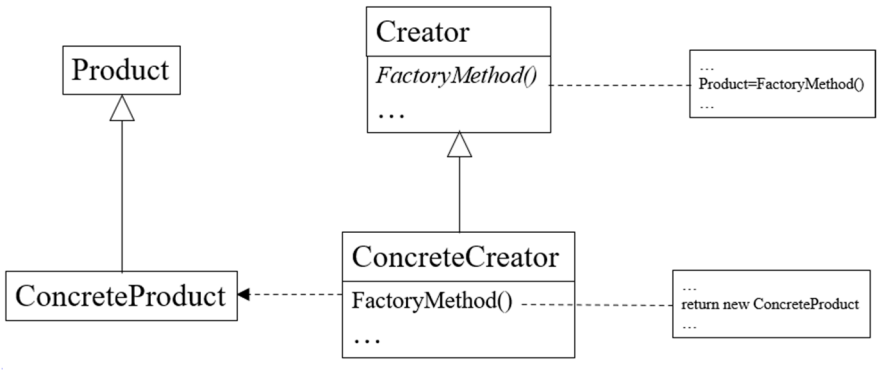


Factory Method – Contexte

- **Connu aussi comme Virtual Constructor**
- **A utiliser quand une classe ne peut pas anticiper quelle catégorie d'objets qu'elle devra créer**
  - il n'y a pas de classe unique qui décide de quelle sous-classe instancier
- **Ou quand une classe utilise ses sous-classes pour spécifier les objets à créer**
- **Exemples**
  - framework pour des applications de bureautique
  - connexion à des bases de données
  - collections et iterators

Factory Method – Structure

- **Chaque sous-classe redéfinit une opération abstraite qui retourne un objet de la sous-classe appropriée**



Factory Method - Participants

- **Product**
  - définit une interface d'objets crées par la fabrique
- **ConcreteProduct**
  - implémente l'interface Product
- **Creator (Fabrique)**
  - déclare la méthode qui retourne un objet de type Product
  - la méthode peut attendre des arguments permettant de déduire le type de classe à instancier
- **ConcreteCreator**
  - classe (spécifique à l'application) qui implémente l'interface Fabrique en implémentant la méthode de création d'objets concrets

Factory Method - Conséquences

- **Permet de supprimer un lien dans le code vers une classe spécifique à une application**
  - le client ne connaît que l'interface Product
- **Permet de travailler avec toute implémentation de ConcreteProduct**

Factory Method – Implémentation

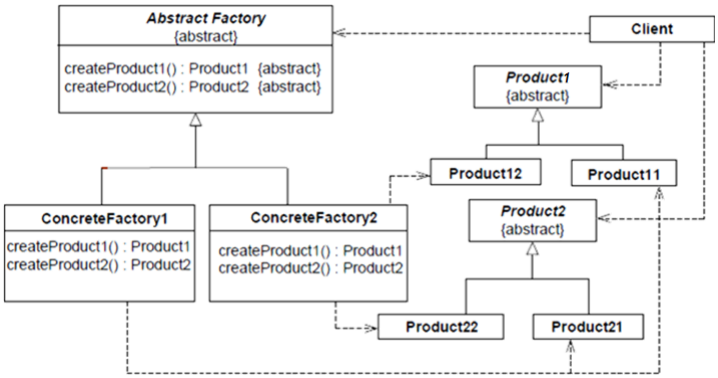
- **La classe Creator est une interface/classe abstraite**
  - les sous-classes en définissent une implémentation, pas d'implémentation par défaut. Cela résout le problème des classes imprévues.
- **La classe ConcreteCreator est une classe concrète**
- **Des méthodes fabriques paramétrées**
  - ce type d'implémentation permet à la méthode fabrique de créer plusieurs types d'objets. La méthode prend alors un paramètre identifiant le type d'objet à créer, tous les objets partageant alors la même interface
- **Utilisation de templates pour éviter l'héritage**
  - un des problèmes liées aux méthodes Fabrique réside dans le fait qu'il faut nécessairement hériter pour créer les objets appropriés. Un moyen de contourner ce pb est d'utiliser un template de classe Creator qu'on instanciera avec pour paramètre le type effectif de l'objet à créer

Abstract Factory – Contexte

- **Propose le moyen de créer des familles d'objets (c'est-à-dire des éléments apparentés ou interdépendants) sans mentionner aucune classe concrète**
- **A utiliser quand un système**
  - doit être indépendant des éléments avec lesquels il va réellement travailler
  - peut être configuré avec une ou plusieurs familles d'éléments
- **Exemples**
  - un ensemble de classes (API) interagissant avec des bases de données (Connexion, Requête, Résultat, …)
  - interface graphique, système d'exploitation, …

Abstract Factory – Structure

- On définit un niveau d'indirection qui masque la création des familles d'objets sans jamais adresser les classes concrètes
- La fabrique est responsable de la création de l'ensemble des services/objets d'une famille, les clients ne les créent jamais directement



Abstract Factory – Participants

- **AbstractFactory :**
  - interface déclarant les opérations de création d'éléments abstraits AbstractProduct
- **ConcreteFactory :**
  - implémente les opérations de création d'éléments abstraits AbstractProduct
- **AbstractProduct :**
  - interface commune aux éléments/services/objets (Product)
- **ConcreteProduct :**
  - définit les objets qui seront créés pas les ConcreteFactory associées et implémente l'interface AbstractProduct
- **Client :**
  - n'utilise que les interfaces AbstractFactory et AbstractProduct

Abstract Factory – Conséquences

- **AbstractFactory isole le processus de création**
- **Cela permet de changer de famille d'objets facilement**
- **Cela garantit que le client utilise des ensembles d'objets cohérents, de la même famille**
- **Inconvénient**
  - difficile d'ajouter une nouvelle catégorie d'objets dans la famille

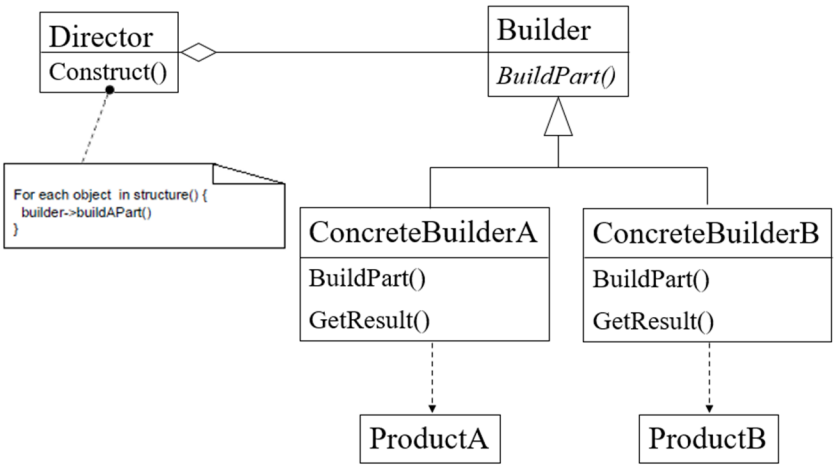
Abstract Factory – Implémentation

- **Habituellement une seule instance de ConcreteFactory par application, donc implémentée comme un Singleton**
- **Habituellement une FactoryMethod par Product**
- **Pour définir des fabriques extensibles, on définit uniquement une seule opération de création de Product paramétrée**
  - le paramètre identifie le type d'élément à créer
  - les objets créés (retournés) sont d'une même interface abstraite et doivent être convertis (downcast) dans leur type exact
  - solution très flexible mais pas toujours très sécurisée

Builder – Contexte

- **On sépare la construction d'un objet complexe de sa représentation afin qu'un même processus de construction puisse générer des représentations différentes**
  - on simplifie la construction de ces objets complexes en ne mentionnant que leur type et leur contenu
  - plusieurs Builder pour construire les différents représentations de ces objets complexes
  - chaque Builder concret construit une représentation différente
- **Exemple**
  - conversion entre fichiers à des formats différents (Save As)
  - construction d'objets à partir des messages de structures différentes

Builder – Structure

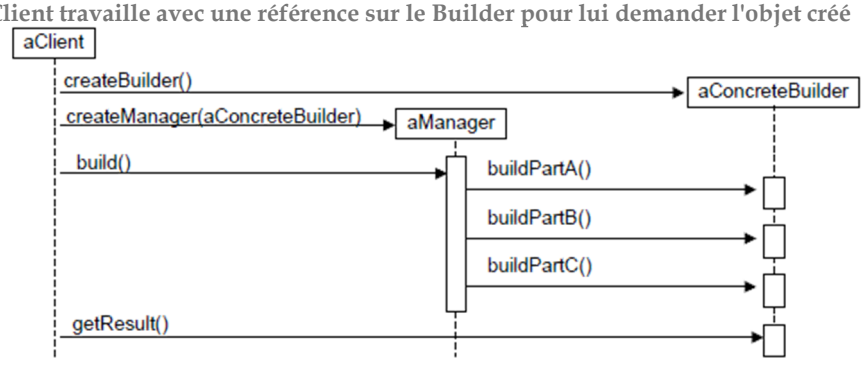


Builder – Participants

- L'objet Builder précise l'interface abstraite de création des différents parties de l'objet
- L'objet ConcreteBuilder crée et assemble les parties qui constituent l'objet complexe
- L'objet Director est chargé du processus de construction de l'objet complexe mais délègue la création effective et l'assemblage à l'interface Builder
- Le Product représente l'objet complexe créé par les objets ConcreteBuilder. Il est constitué des différentes parties créées séparément par les objets ConcreteBuilder

Builder – Collaboration

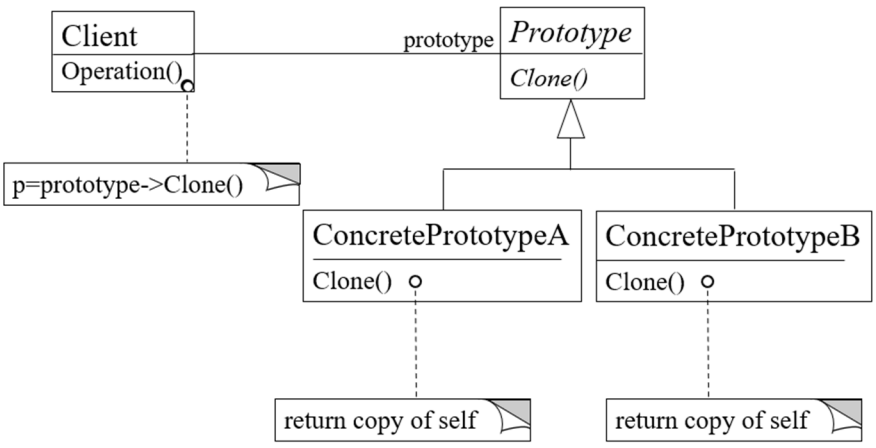
- Un client crée une instance de la classe Director et lui transmet le Builder approprié
- Le Director invoque les méthodes du Builder pour créer et initialiser les parties spécifiques du Product
- Le Builder reçoit du Director le contenu de l'objet à créer et l'ajoute au Product
- Le Client travaille avec une référence sur le Builder pour lui demander l'objet créé



Prototype – Contexte

- **Permet de spécifier comment créer des objets à partir d'une instance prototype, on crée de nouveaux objets en copiant ce prototype**
- **A utiliser quand**
  - le système ne dépend pas de comment les éléments sont créés
  - les classes à instancier sont précisées à l'exécution
  - on veut éviter la création d'une hiérarchie de fabriques
  - il est plus pratique, moins coûteux, de copier une instance plutôt que d'en créer
- **Exemples**
  - outils graphiques (dessin, musique, conception UML, …)
  - jeux avec labyrinthes

Prototype – Structure



Patrons structuraux

Patterns de Structure

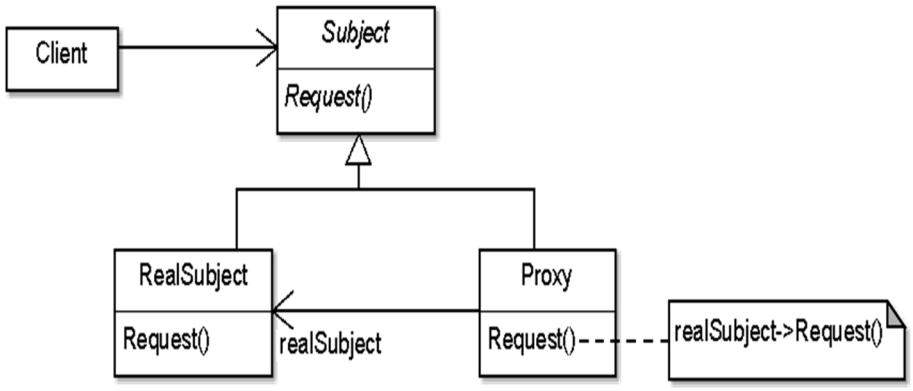
- **Proxy**
  - propose un substitut ou un remplaçant d'un objet afin d'en contrôler l'accès
- **Adapter**
  - pour convertir l'interface d'une classe en une autre que le client connait
- **Decorator**
  - ajoute dynamiquement de responsabilités supplémentaires à un objet
- **Composite**
  - organise des objets en structure hiérarchique (arbre) afin de représenter leurs relations composant-agrégat
- **Bridge**
  - découple une abstraction de son implémentation de façon à faire varier les 2 indépendamment
- **Facade**
  - fournit une interface unifiée à l'ensemble des interfaces d'un sous-système. Définit une interface de plus haut niveau rendant l'usage du sous-système plus simple
- **Flyweight**
  - optimise la consommation mémoire en utilisant le partage d'information lors de l'allocation d'un nombre important d'objets

Proxy - Contexte

- Fournit un substitut (ou pseudo-objet) à un objet afin d'en contrôler l'accès ou la création (si complexe)
- On utilise un niveau d'indirection supplémentaire pour permettre un accès local/distribué, contrôlé ou intelligent
- On ajoute une enveloppe ou une délégation pour protéger le composant effectif
- Exemple
  - accès à des objets distants (CORBA)
  - protection, synchronisation, comptage, proxys virtuels …

Proxy – Structure

- Le Proxy transfère des requêtes au RealSubject si nécessaire



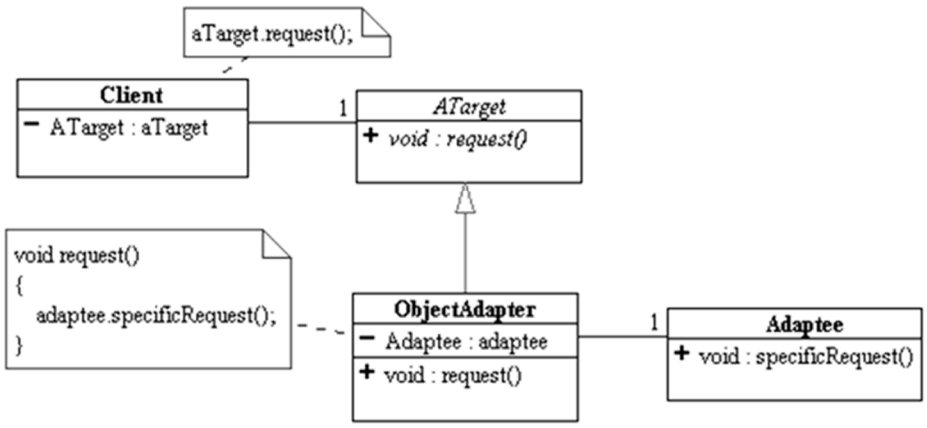
Proxy – Participants

- **Subject**
  - interface implémentée par RealSubject et qui représente ses services
- **Proxy**
  - contient une référence lui permettant d'accéder au RealSubject
  - implémente la même interface que celle implémentée par RealSubject de façon à qu'il s'y substitue
  - contrôle l'accès à RealSubject et peut éventuellement être responsable de sa création ou destruction
  - peut avoir d'autres responsabilités suivant le type de proxy
- **RealSubject**
  - l'objet réel qui est représenté par le proxy

Adaptor - Contexte

- **Convertit une interface entre une autre à laquelle l'application cliente s'attend**
- **Permet à des classes qui a priori ne le pourraient pas en raison d'interfaces incompatibles d'interagir**
- **On enveloppe une classe existante dans une nouvelle interface**
- **Permet aussi d'adapter un ancien composant à un nouveau système ou de masquer des opérations possibles à un client**
- **Connu aussi sous le nom de Wrapper**
- **Exemples**
  - stack, queue et priority_queue dans la STL C++

Adaptor – Structure



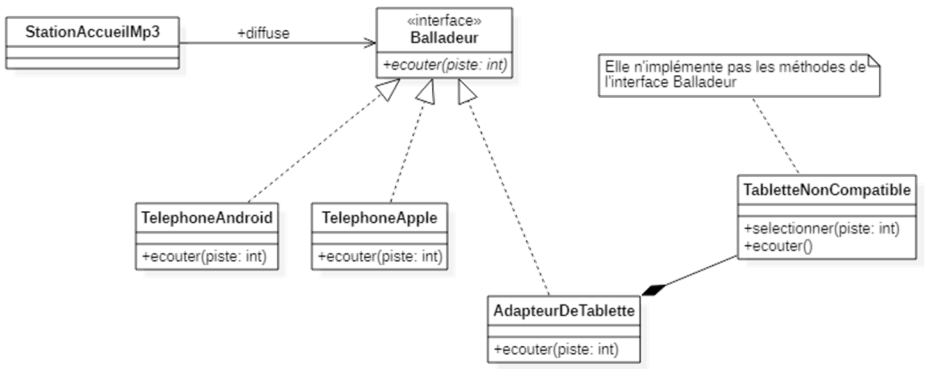
Adaptor - Participants

- **Target**
  - définit l'interface que le Client sait utiliser
- **Adapter**
  - adapte/transforme l'interface Adaptee à sa forme spécifiée par l'interface Target
- **Adaptee**
  - c'est l'interface existante qu'on souhaite adapter
- **Client**
  - a connaissance d'objets correspondant à l'interface Target

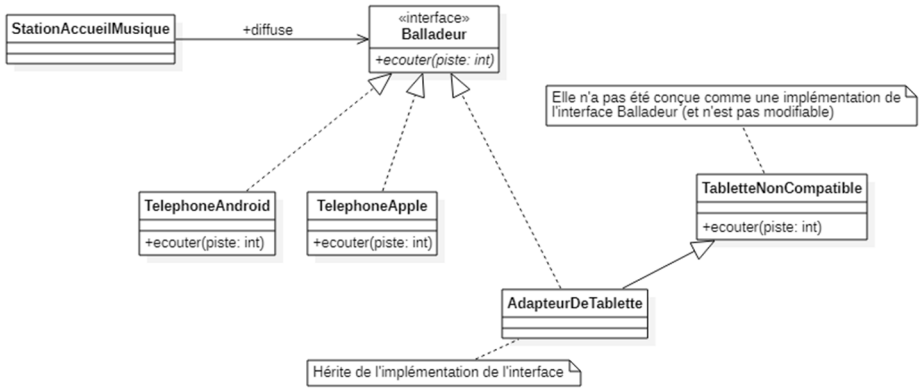
Adaptateur d'objet ou de classe

- **Adaptateur d'objet**
  - s'appuie sur la relation de composition
  - permet au client et à l'objet adapté d'être complètement découplés, seul l'adaptateur les connaît tous les deux
- **Adaptateur de classe**
  - s'appuie sur un héritage multiple
  - il peut y avoir des conflits de noms si des méthodes de même signature existent à la fois dans la cible et l'adapté
  - utile lorsque le découplage total entre le client et l'adapté n'est pas nécessaire

Adapteur d'objet



Adapteur de classe

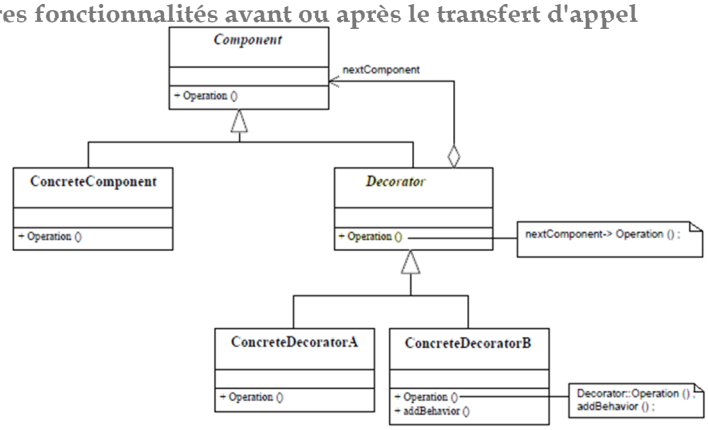


Decorator – Contexte

- **Etend dynamiquement les fonctionnalités d'un objet de façon transparente pour les clients**
  - on utilise une instance d'une autre classe qui invoquera les méthodes de l'objet initial (décoré)
  - c'est une alternative à l'héritage
  - à utiliser lorsqu'on souhaite ajouter des responsabilités à des objets individuels et pas à la classe entière
- **Connu aussi comme Wrapper**
- **Exemple**
  - flots (streams) : text, buffered, network, …
  - fenêtres avec ajout d'ascenseurs, de bordures, ...

Decorator – Structure

- Ce pattern possède une interface identique à celle d'un objet qu'il contient et qui est une réalisation concrète d'une classe abstraite
- Il retransmet tout appel de méthode à l'objet qu'il contient, mais en ajoutant ses propres fonctionnalités avant ou après le transfert d'appel



Decorator - Participants

- **Component**
  - définit l'interface d'objets auxquels on peut ajouter dynamiquement des responsabilités
- **ConcreteComponent**
  - définit un objet auquel des responsabilités sont rattachées
- **Decorator**
  - contient une référence vers un objet Component et définit une interface conforme à celle de Component
- **ConcreteDecorator**
  - ajoute des responsabilités au composant

Decorator – Conséquences

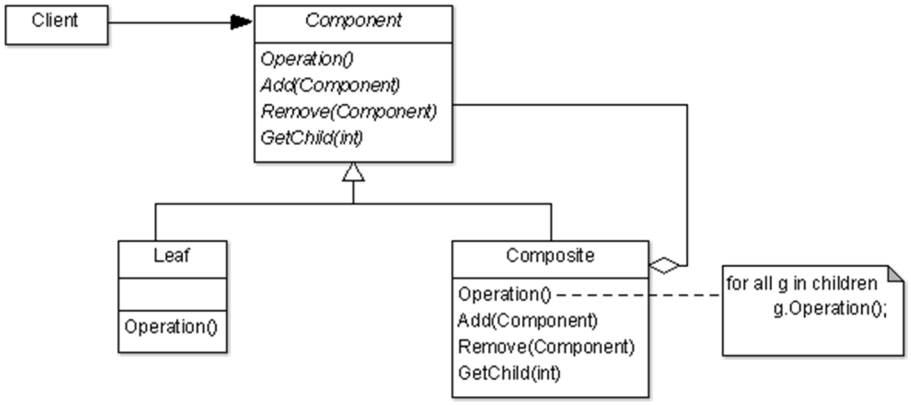
- **Permet plus de flexibilité : on peut changer ce que fait un Decorator dynamiquement, par opposition à un changement statique implémenté par une sous-classe**
  - il est aussi possible de supprimer les fonctionnalités ajoutées dynamiquement
- **On ne peut pas distinguer le Decorator de l'objet qu'il contient (instance concrète) ou des autres Decorators**

Composite - Contexte

- **On compose des objets en une structure arborescente pour représenter une hiérarchie de type composant-agrégat**
- **Ce pattern permet aux clients de traiter uniformément les objets individuels et les compositions d'objets**
- **A utiliser quand**
  - on a besoin de composants contenant d'autres composants, qui à leur tour peuvent être des Composites
  - on souhaite que les clients ignorent la différence entre les objets composés et les objets individuels
- **Exemples**
  - répertoire et systèmes de fichiers
  - représentation graphique schématique
  - Swing, DOM

Composite – Structure

- **Un client interagit avec les objets d'une structure composite par le biais de la classe Component. Si le destinataire de la requête est une feuille, dans ce cas elle la traite directement. Si le composant est un agrégat, alors la requête est transférée à ses composants internes**



Composite – Participants

- **Component**
  - déclare l'interface des objets de la composition
  - peut implémenter un comportement par défaut d'une interface commune aux différentes classes
  - permet d'associer ou de gérer les composants internes
- **Leaf**
  - représente les objets feuille d'une composition
  - définit le comportement basique des objets d'une composition
- **Composite**
  - définit le comportement des composants agrégats
  - contient les composants internes
  - implémente des opérations associées aux composants internes
- **Client**
  - utilise les objets d'une composition à travers l'interface Component

Composite – Conséquences

- **Minimise la complexité d'un objet composite**
  - les interactions du client sont simplifiées grâce à une interface commune
- **Facilite l'ajout de nouveaux composants**
  - le code du client ne change pas
- **Inconvénient**
  - on ne peut pas restreindre les composants contenus dans un composite, il faut utiliser des vérifications en exécution

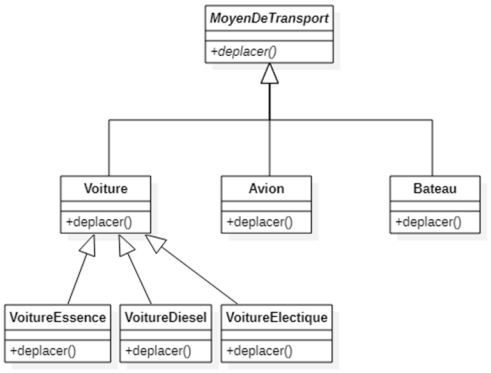
Decorators, Adapters, Composites

- **Les Adapters peuvent aussi être vus comme des décorateurs d'une classe existante**
  - cependant, leur fonction est de changer une interface d'une ou plusieurs classes en une plus adaptée à un client particulier
  - les décorateurs ajoutent des responsabilités supplémentaires ("amélioration") à des instances précises plutôt qu'à toutes
- **Composite et Decorator ont des diagrammes similaires, qui reflètent le fait qu'ils s'appuient tous les deux sur une composition récursive d'un nombre indéterminé d'objets**
  - un Decorator peut être considéré comme un Composite d'un seul composant, mais son intention n'est pas de faire de l'agrégation
  - un Composite ne vise pas "l'embellissement" mais la représentation d'information
  - ces intentions sont différentes mais complémentaires. Par conséquent, ces deux patterns sont souvent utilisés ensemble
- **Decorator et Proxy ont des objectifs différents mais des structures similaires**
  - les deux décrivent comment ajouter un niveau d'indirection à un autre objet, et leurs implémentations contiennent une référence sur l'objet auquel sont transférés les requêtes
  - Proxy propose la même interface

Bridge - Contexte

- **Découple une abstraction de son implémentation de façon à les faire changer indépendamment**
  - à utiliser quand
  - une abstraction peut avoir plusieurs implémentations
  - il existe une hiérarchie d'abstractions et une hiérarchie correspondante d'implémentations
- **Ce pattern implémente les abstractions et les implémentations comme des classes indépendantes qu'on pourra combiner dynamiquement**
- **L'utilisation de l'héritage n'est pas approprié en raison de l'explosion combinatoire**
- **Exemples**
  - GUI :
  - des fenêtres de différentes boites à outils (Motif, KDE, Gnome, MSWindows, …)
  - différentes catégories de fenêtres (IconWindow, PopUp, Dialog, …)
  - drivers de périphériques, adaptateurs réseau, …

Bridge - Motivation

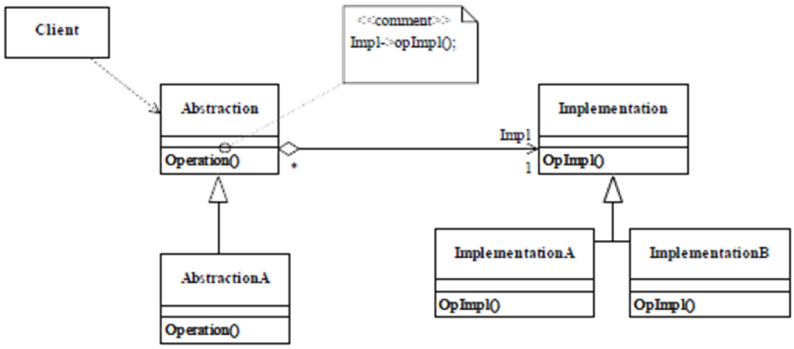


Des implémentations différentes possibles suivant le moyen de propulsion essence, diesel, électrique, hydrogène, …

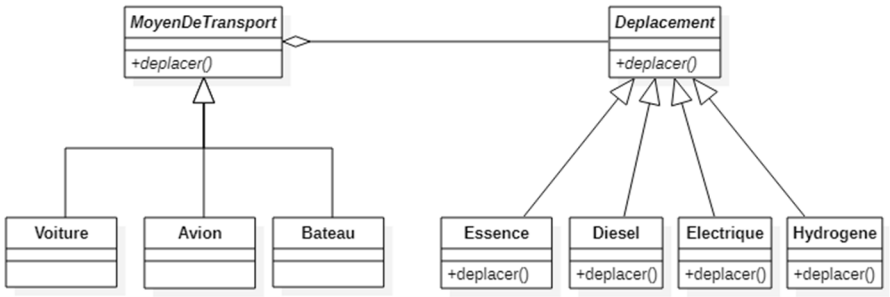
Pourrait se traduire par une hiérarchie, qui peut devenir trop importante

Bridge - Structure

- Le Client ne veut pas s'encombrer de détails dépendant de l'implémentation
- Chaque abstraction sera affinée avec des détails spécifiques



Bridge – Mise en œuvre

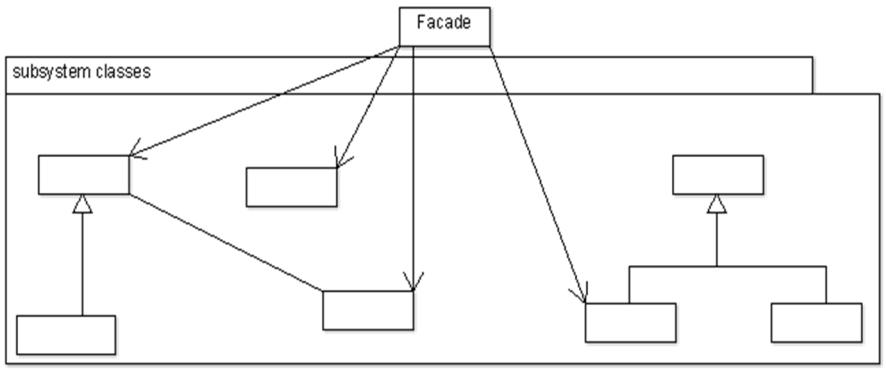


Facade - Contexte

- **Propose une interface unifiée pour un ensemble d'interfaces d'un sous-système**
- **Ce pattern définit une interface de plus haut niveau rendant ainsi le sous-système plus simple à utiliser**
- **A utiliser quand**
  - on souhaite proposer une interface simple à un système complexe
  - il existe de nombreuses dépendances entre les clients et les classes d'implémentation
  - on souhaite organiser les sous-systèmes en couche
- **Exemple**
  - outils de compilation : scanner, parser, linker, générateur de code, …
  - la plupart des clients n'ont pas besoin de connaître les détails, mais certains pourraient en avoir besoin
  - portails internet

Facade – Structure

- **Les clients communiquent avec le sous-système en émettant des requêtes à Façade, qui les transfère aux objets appropriés**



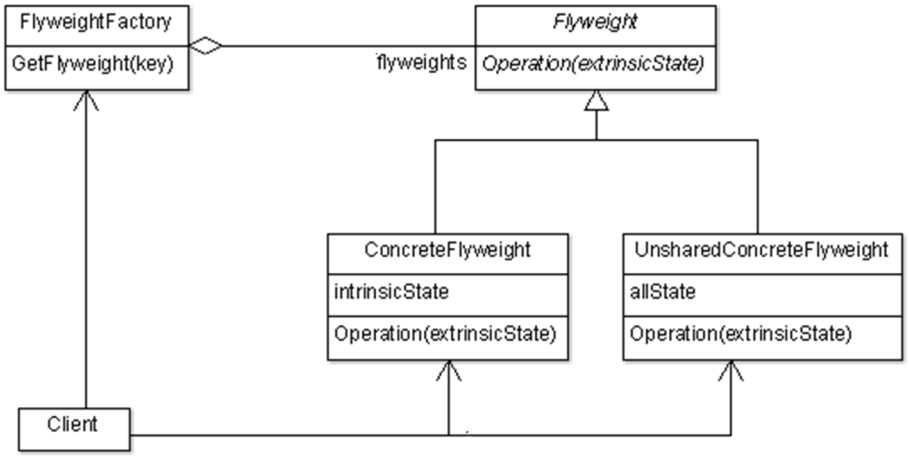
Facade – Conséquences

- **Les clients de Facade n'ont pas à savoir combien de classes il y a derrière la façade**
  - système plus simple à utiliser parce qu'il réduit le nombre d'objets que le client peut adresser
- **Cela clarifie les accès au système**
- **Cela réduit le couplage entre le système et ses clients**
  - les composants du sous-système peuvent varier sans affecter les clients
- **N'empêche pas une application d'utiliser le sous-système si besoin**

Flyweight - Contexte

- **On utilise le partage pour rendre plus efficace la gestion d'un grand nombre d'objets de faible granularité**
- **Le pattern Flyweight s'utilise lorsqu'on a un nombre très important d'objets dont une partie de leur état interne est commune tandis qu'une autre partie est individualisée**
  - utilisé pour optimiser lorsque la partie la plus importante de certains objets peut être externalisée (extérieure à l'objet)
- **Concept clé : état extrinsèque et intrinsèque**
  - l'information intrinsèque est stockée dans le Flyweight
- **Exemple**
  - application de traitement de texte
  - graphiques

Flyweight - Structure



Patrons comportementaux

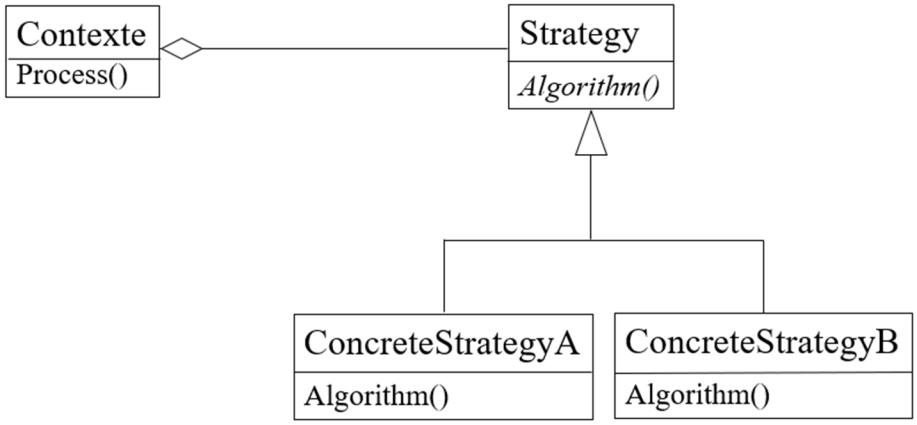
Patterns de comportement

- **Strategy :** définit une famille d'algorithmes en les encapsulant et les rendant interchangeable. Les algorithmes pourront alors être adaptés suivant les besoins des clients
- **Observer :** définit une dépendance entre objets de type one-to-many de façon à ce que lorsque l'état d'un objet change, ses dépendants en sont automatiquement notifiés et mis à jour
- **Chain Of Responsibility :** permet d'éviter le couplage entre l'émetteur d'une requête et l'objet qui doit la traiter en permettant à plus d'un objet d'avoir la possibilité de la traiter
- **Mediator :** définit un objet encapsulant les interactions au sein d'un ensemble d'objets
- **Command :** encapsule une requête en tant qu'objet, afin de paramétrer les clients avec différents types de requêtes
- **Visitor :** représente une opération devant être appliquée aux éléments d'une structure. On peut ainsi définir de nouvelles opérations sans changer les classes des objets sur lesquels les invoquer
- **Template Method :** définit le squelette d'un algorithme dans une opération, en laissant des sous-classes en gérer des parties élémentaires
- **Memento :** capture et externalise l'état interne d'un objet (sans violer le principe d'encapsulation), de façon à pouvoir y revenir plus tard.
- **Iterator :** fournit le moyen d'accéder séquentiellement aux éléments d'un objet agrégat sans en publier la structure sous-jacente
- **State :** permet à un objet de changer de comportement lorsque ses états internes changent, comme si l'objet changeait de classe
- **Interpréter :** permet de représenter la grammaire et l'interpréteur d'un langage donné

Strategy - Contexte

- **Définit une famille d'algorithmes, en encapsulant chacun d'eux dans des classes, sous-classes d'une classe de base commune, les rendant ainsi interchangeables**
- **L'algorithme pourra varier selon les clients et aussi dans le temps**
- **Un algorithme encapsulé est appelé Strategy**
- **Exemple**
  - sélection d'un algorithme de tri
  - découpage d'un texte en lignes suivant des algorithmes différents
- **A utiliser quand**
  - on a besoin de plusieurs variantes d'un algorithme
  - une classe peut avoir plusieurs comportements

Strategy – Structure



Strategy – Participants

- **Strategy**
  - déclare l'interface commune de tous les algorithmes possibles
- **ConcreteStrategy**
  - implémente un algorithme de l'interface Strategy
- **Contexte**
  - est configuré avec un objet ConcreteStrategy
  - gère une référence à un objet Strategy

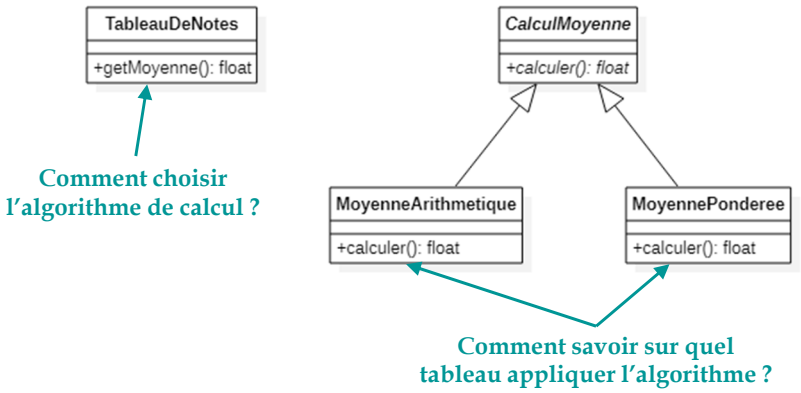
Strategy - Conséquences

- **Le code du client est simplifié**
  - déchargé de tout choix de l'algorithme ou d'implémentation de comportements alternatifs
  - ce qui élimine les instructions if ou switch
- **La Strategy peut changer dynamiquement**
- **L'héritage va aider à factoriser des fonctionnalités communes aux algorithmes**

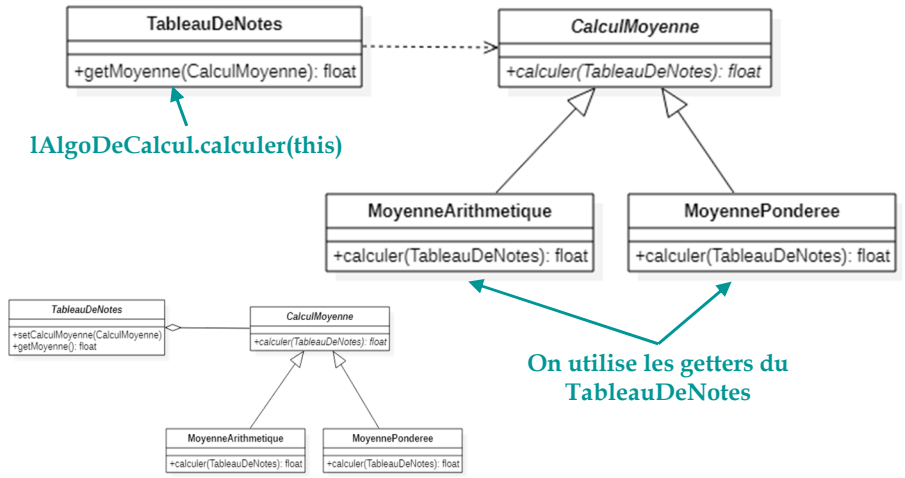
Strategy – Implémentation

- **Les classes ConcreteStrategy héritent/implémentent de Strategy**
- **Tout objet de la classe Contexte est configuré avec le ConcreteStrategy dont a besoin le client (par exemple à la construction)**
- **L'objet Contexte appelle l'interface Strategy pour appeler l'algorithme défini dans le ConcreteStrategy**

Strategy - Exemple



Strategy – Implémentation



monTableauDeNotes.getMoyenne(new MoyenneArithmetique());

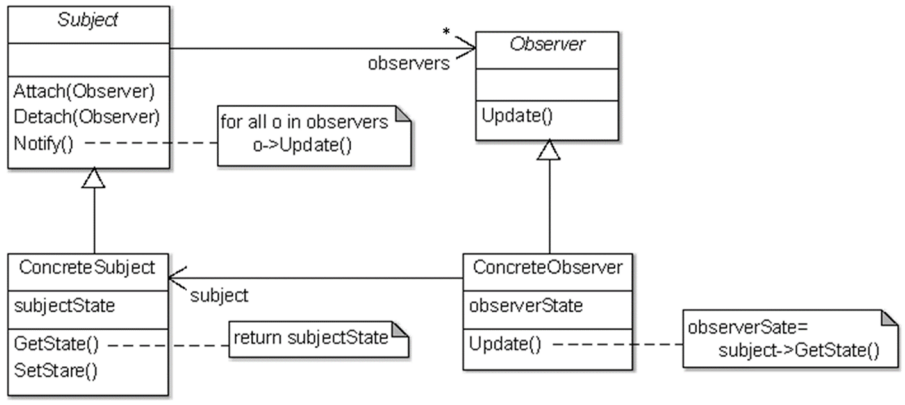
Strategy vs Bridge

- **Les diagrammes de classes sont similaires, mais ces deux patterns n'ont pas le même objectif**
  - le pattern Strategy est un pattern de comportement, c'est un pattern dynamique à utiliser lorsqu'on souhaite inter-changer des algorithmes
  - le pattern Bridge pattern est un pattern de structure (structural pattern), on l'utilise quand il peut y avoir plusieurs hiérarchies possibles
- **Le couplage entre le contexte et les stratégies (lors de la construction) est plus fort que le couplage entre l'abstraction et l'implémentation dans le pattern Bridge (liaison dynamique)**

Observer - Contexte

- **Définit une dépendance un-plusieurs entre objets de façon à ce que lorsque l'état d'un objet change, ses dépendants en sont automatiquement notifiés et mis à jour**
- **Les objets qui dépendent d'un sujet doivent être informés lorsque celui-ci change**
  - ils ne doivent pas dépendre des détails d'implémentation du sujet
  - ils ne sont concernés que par comment il change et non pas par comment il est implémenté
- **Exemple**
  - gestionnaire d'événements
  - interface graphique
  - visualisation d'objets à travers plusieurs vues
  - ces vues doivent être mises à jour lorsque les valeurs de l'objet changent

Observer - Structure



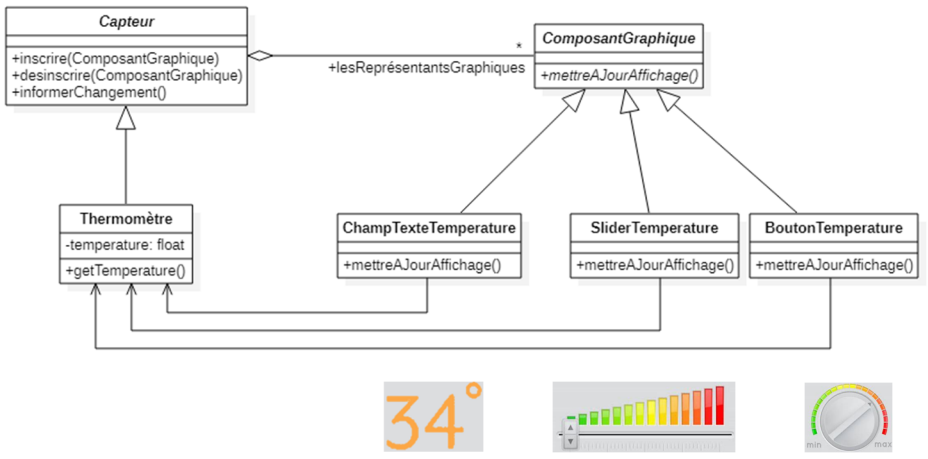
Observer - Participants

- **Subject**
  - connaît ses observateurs, quel qu'en soit leur nombre
  - propose une interface pour inscrire ou désinscrire les observateurs
- **Observer**
  - définit une interface de mise à jour pour les objets devant être notifiés d'un changement dans le sujet
- **ConcreteSubject**
  - gère l'état de l'objet et quand un changement se produit, en notifie les observateurs inscrits
- **ConcreteObserver**
  - gère une référence vers un ConcreteSubject
  - implémente l'interface de mise à jour Observer de façon à conserver un état de l'observateur cohérent avec celui du sujet

Observer – Collaborations

- Le client instancie l'objet ConcreteSubject
- Puis il instancie et enregistre les observateurs concrets en utilisant les méthodes définies dans l'interface Subject
- Chaque fois que l'état du sujet change, celui notifie tous les Observers enregistrés en utilisant les méthodes définies dans l'interface Observer
- Lorsqu'un nouvel Observer est ajouté à l'application, il est enregistré auprès de l'objet Subject

Observer – Exemple



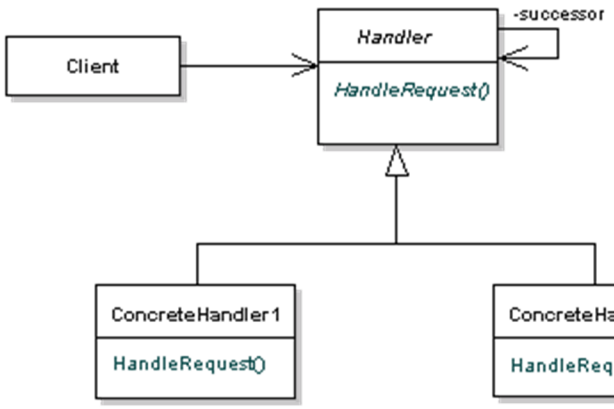
Observer – Conséquences

- **Avantages**
  - sujets et observateurs sont indépendants : réutilisabilité
  - couplage abstrait entre le Subject et l'Observer
  - le sujet n'en connaît que l'interface
  - permet une communication de type broadcast
  - à la différence d'une requête classique, le sujet ne spécifie aucun destinataire
  - les observateurs sont ajoutés ou supprimés dynamiquement sans aucun impact sur le sujet
- **Inconvénients**
  - mises à jour inattendues
  - les observateurs n'ont aucune connaissance de leurs pairs
  - les modifications et mises à jour en cascade peuvent être coûteuses

Chain of Responsibility - Contexte

- **Permet d'éviter le couplage entre l'émetteur d'une demande et son récepteur, en permettant à plus d'un objet de répondre à cette demande**
- **On chaîne les objets récepteurs et on fait passer la demande le long de cette chaîne jusqu'à ce qu'un objet parmi eux la traite**
- **A utiliser quand**
  - on ne sait pas quel objet pourra traiter la demande
  - l'accès à l'objet qui doit traiter la demande n'est pas autorisé
- **Exemples**
  - événements d'une GUI
  - filtres d'un processus, exceptions dans un système distribué

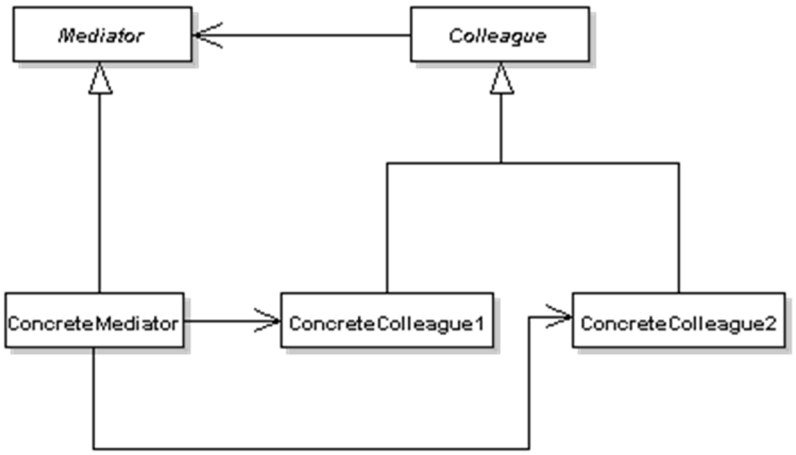
Chain of Responsibility - Structure



Mediator – Contexte

- **Définit un objet qui gère les interactions d'un ensemble d'objets**
- **Permet le couplage faible en évitant que des objets se réfèrent mutuellement, et en permettant de faire varier leurs interactions**
- **C'est un mécanisme facilitant l'interaction entre objets en faisant en sorte qu'aucun d'entre eux n'ait connaissance des autres**
- **A utiliser quand**
  - plusieurs objets communiquent suivant des règles bien définies mais complexes. Les dépendances qui en découlent ne sont pas structurées et difficiles à comprendre
  - la réutilisation d'un objet est difficile parce qu'il fait référence et communique avec de nombreux autres objets
  - un comportement distribué entre plusieurs classes peut être personnalisé en utilisant intensément l'héritage
- **Exemple**
  - dans une interface graphique, les formulaires contenant des contrôles
  - application de chat

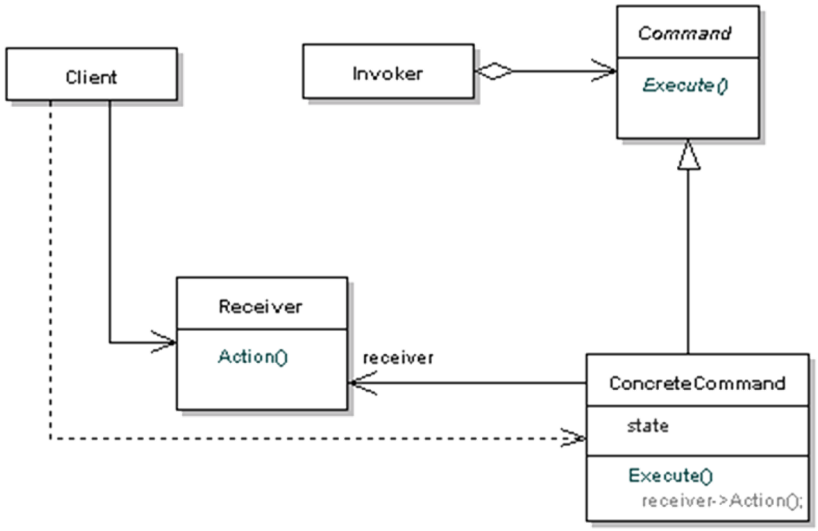
Mediator – Structure



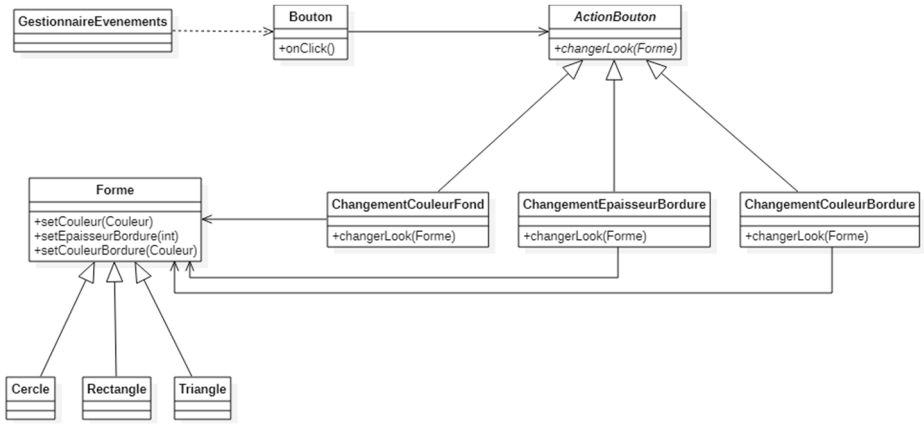
Command – Contexte

- **Encapsule une requête comme un objet, de façon à paramétrer les clients avec plusieurs requêtes**
- **A utiliser quand on a besoin**
  - de paramétrer des objets en fonction de l'action qu'ils doivent effectuer
  - d'ajouter dans une queue puis d'exécuter les requêtes à des moments différents
  - permet des actions annulables (la méthode Execute peut mémoriser l'état afin d'y revenir)
- **Exemple**
  - boite à outils d'interface utilisateur

Command – Structure



Command - Exemple



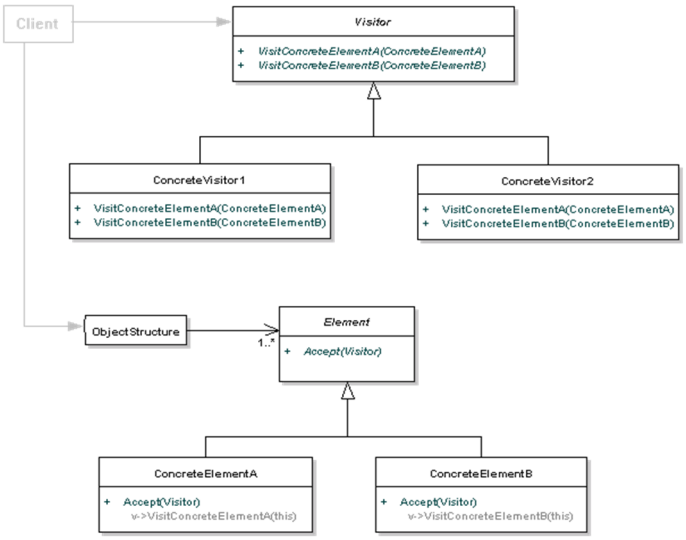
Command – Participants

- **Command**
  - déclare une interface d'exécution d'une opération
- **ConcreteCommand**
  - hérite de l'interface Command, en implémentant la méthode Execute, qui invoque les opérations correspondantes du Receiver
  - définit un lien entre le Receiver et l'action
- **Client**
  - crée un objet ConcreteCommand et initialise son Receiver
- **Invoker**
  - demande à Command d'exécuter l'action
- **Receiver**
  - sait comment réaliser les opérations

Visitor – Contexte

- **Représente une opération à effectuer sur des éléments d'un objet structuré**
  - on peut définir une nouvelle opération sans changer les classes des éléments sur lesquels elle agira
- **A utiliser quand**
  - des opérations similaires doivent être effectuées sur des objets de types différents, mais regroupés dans une même structure (une collection ou une structure plus complexe)
  - de nombreuses opérations, sans lien entre elles, doivent être effectuées
  - Visitor permet de créer une classe concrète séparée pour chaque type d'opération, et de séparer l'implémentation de l'opération de la structure des objets
  - la structure des objets n'est pas censée changer mais il est probable qu'on ait besoin de nouvelles opérations
  - puisque le pattern sépare le visiteur (qui représente les opérations, les algorithmes, les comportements) de la structure de l'objet, et tant que la structure ne change pas, il est facile d'ajouter de nouveaux Visiteurs
- **Exemple :**
  - implémentation de variantes d'analyses sur le même type de données

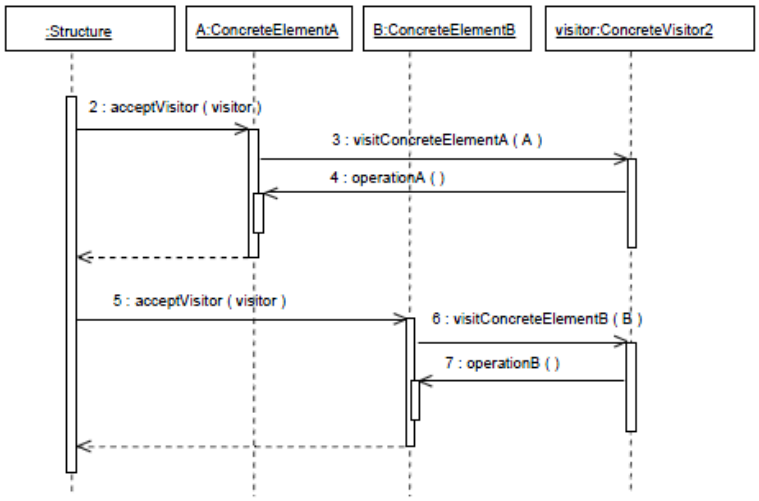
Visitor – Structure



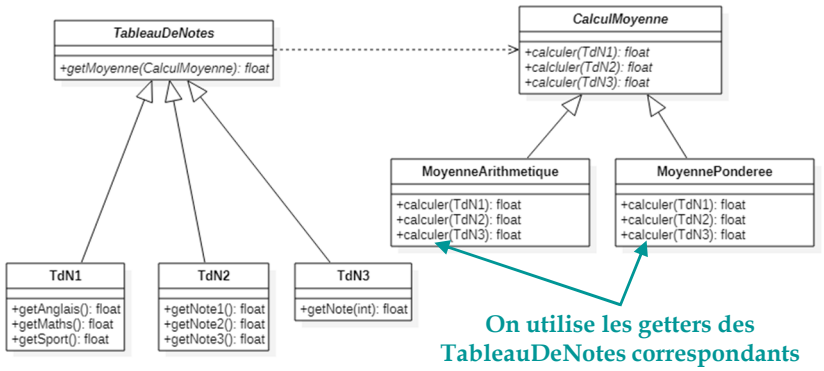
Visitor – Participants

- **Visitor**
  - c'est une interface ou une classe abstraite utilisée pour déclarer les opérations de visite pour tout type d'élément. Généralement le nom de l'opération est le même mais celles-ci diffèrent pas leur signature, le type du paramètre permettant d'identifier quelle méthode sera exécutée
- **ConcreteVisitor**
  - pour chaque type de visiteur, toutes les méthodes de visite doivent être implémentées. Chaque visiteur sera responsable des différentes opérations. Lorsqu'un nouveau visiteur est ajouté, on lui passe l'objet structuré
- **Element (Visitable)**
  - c'est une abstraction qui déclare l'opération accept. C'est le point d'entrée qui permet à un objet d'être visité par un objet visiteur. Chaque objet d'une collection implémentera cette abstraction de façon à permettre d'être visité
- **ConcreteElement**
  - ces classes implémentent l'interface/classe Element en définissant l'opération accept. L'objet visiteur sera passé à cet objet par cette opération accept
- **ObjectStructure**
  - cette classe contient tous les objets pouvant être visités. Elle propose un mécanisme de parcours de la structure. Ce n'est pas nécessairement une collection, cela peut être une structure complexe comme un objet composite

Visitor – Collaborations



Visitor – Exemple



On utilise les getters des TableauDeNotes correspondants

Visitor – Implémentation

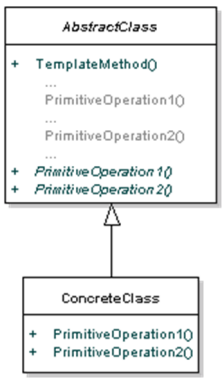
- **Double-dispatch**
  - l'opération à exécuter dépend du type de la requête et des types des deux éléments Visitor et Element
- **Qui est responsable du parcours de l'objet structuré ?**
  - l'objet structuré est parcouru dans Visitor, via un itérateur séparé
  - habituellement dans l'objet structuré
  - une collection itère sur ses éléments, en appelant accept sur chacun
  - un objet composite se parcourt en faisant en sorte que chaque opération accept parcourt les enfants et appelle accept récursivement

Template method – Contexte

- **Définit le squelette d'un algorithme dans une opération, tout en déferrant certains étapes à des sous-classes. Permet de redéfinir certaines étapes d'un algorithme sans en changer la structure générale**
- **A utiliser**
  - pour implémenter des parties variables d'un algorithme et en laissant des sous-classes implémenter les variantes
  - quand on factorise suite à l'identification d'un comportement commun à plusieurs classes. Une classe abstraite est créée qui contiendra le code commun (dans un Template Method)

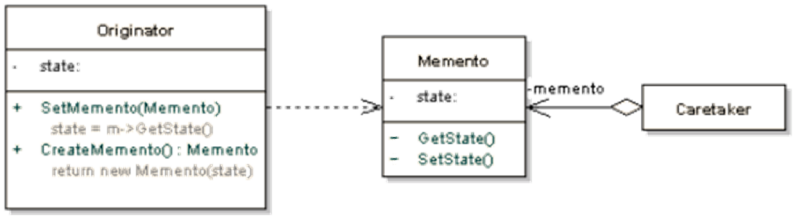
Template method – Structure

- **AbstractClass**
  - déclare les opérations primitives abstraites que des classes concrètes devront implémenter pour définir des étapes d'un algorithme
  - implémente une méthode templateMethod qui définit le squelette de l'algorithme. Cette méthode appelle les opérations primitives ainsi que d'autres opérations de la même ou d'autres classes
- **ConcreteClass**
  - implémente les opérations primitives qui effectueront des étapes spécifiques de l'algorithme
  - lorsqu'une la méthode template est appelée sur une classe concrète, le code exécuté est celui de la classe de base, avec invocation des méthodes définies dans cette classe dérivée



Memento – Contexte

- **Permet de capturer et d'externaliser l'état interne d'un objet, sans violer le principe d'encapsulation, afin d'y revenir plus tard**
- **Encapsule l'état courant d'un objet quelconque dans un objet mémento de façon à restaurer cet état si besoin, et sans jamais publier la représentation interne de cet objet**
- **Utile pour réaliser un instantané afin de revenir à un état original, comme on le fait avec un "Undo" ou "Rollback"**

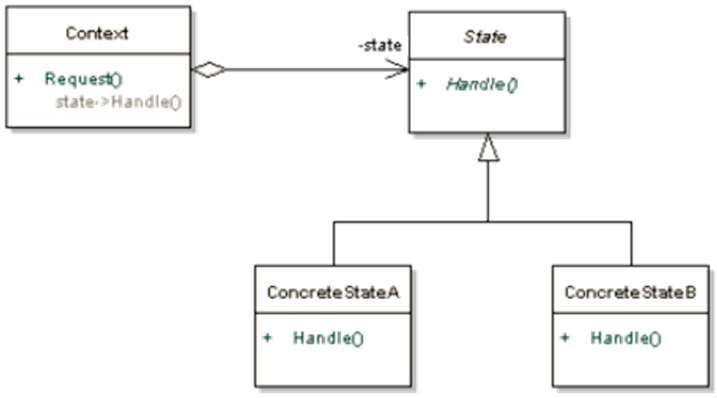


Memento – Participants

- **Memento**
  - stocke l'état interne de l'objet origine. Cet état correspond à un nombre quelconque de variables de l'objet
  - le Memento doit avoir deux interfaces
  - une pour le gardien (caretaker) : elle ne permet aucune opération d'accès à l'état interne stockée dans le mémento, respectant ainsi l'encapsulation
  - une pour le créateur (originator) : elle permet au créateur d'accéder à toute variable nécessaire pour le retour à l'état précédent
- **Originator**
  - créer un objet mémento enregistrant son état interne
  - utilise l'objet mémento pour revenir à son état précédent
- **Caretaker**
  - a la responsabilité de conserver le mémento
  - le mémento ne peut pas être utilisé par le caretaker

State

- **Permet à un objet de changer de comportement lorsque son état interne change, donnant l'illusion que l'objet change de classe**



Iterator

- Fournit le moyen d'accéder séquentiellement aux éléments d'un objet agrégat sans en connaître la représentation interne