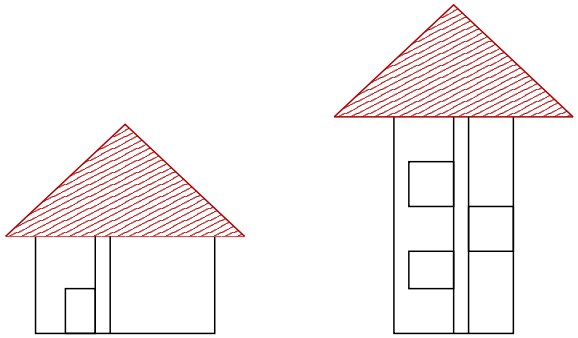
Introduction aux Design Patterns

C'est un architecte C. Alexander [a] qui a développé l'idée d'identifier des objets apportant :

"Une solution précise à problème donné, dans un context précis"



- Définir un pattern, c'est fournir le *pourquoi* et le *comment* de chaque solution.

[a] "Timeless way of building", 1979.

Spécificité des “Design Patterns”

E. Gamma and R. Helmand and R. Johnson and J. Vlissides, 1995.
Introduction du concept de patterns dans la modélisation objet.

- Décrire aussi les relations avec les autres objets du modèle et les conséquences liées à leur utilisation.
- Les patterns modélisent des objets qui ***n’existent pas*** en tant qu’***entités dans le système cognitif humain***.

Bibliographie

- *The Design Patterns Java Companion* James W. Cooper
  `www.patterndepot.com/put/8/JavaPatterns.htm`
- *Object-Oriented Software Development Using Java* Xiaoping Jia, Ph.D
  `se.cs.depaul.edu/Java/chap10.html`

Une parenthèse - Model View Controller

- G. Krasner et S. Pope (1988) - A cookbook for using the modelview controller user interface paradigm in SmallTalk-80 - J. of OOP
  - Modèle des données : l'application
  - Représentation du modèle, ex : affichage à l'écran
  - Contrôle : définition du protocole *abonnement/souscription*(subscribe/notify). Chaque fois que les données changent le modèle les notifie aux vues.
- MVC découple vues et modèles pour accroître la flexibilité et la réutilisation. Un modèle peut bénéficier de plusieurs vues .
- MVC permet également de modifier les réponses d'un objet sans modifier sa vue en encapsulant cette

Qu'est ce qu'un Design Pattern ?

- Les Design Patterns (DP) sont des architectures de classes permettant d'apporter une solution à des problèmes fréquemment rencontrés lors des **phases d'analyse et de conception** d'applications.
- Ces solutions sont facilement adaptables (donc réutilisables), elles sont utilisables sans aucun risque dans la grande majorité des langages de programmation orientés objet.

Intérêt des Design Patterns

- **Avantages** : les Design Patterns ont une architecture facilement compréhensible et identifiables pour un programmeur (améliore la communication).
- **Objectifs** : *ne pas réinventer la roue*
- **Remarque** : Les Design Patterns ne sont donc pas vraiment une solution miracle pour les problèmes, mais ce sont plutôt des méthodes de résolutions. C'est comme une formule mathématique, c'est la solution mais encore faut-il l'appliquer au bon moment avec les bonnes variables.

Design Patterns : le catalogue

Ce travail a été initié dans la thèse de doctorat d'E. Gamma et finalisé par *"The Gang of Four"* dans un ouvrage qui fait référence.

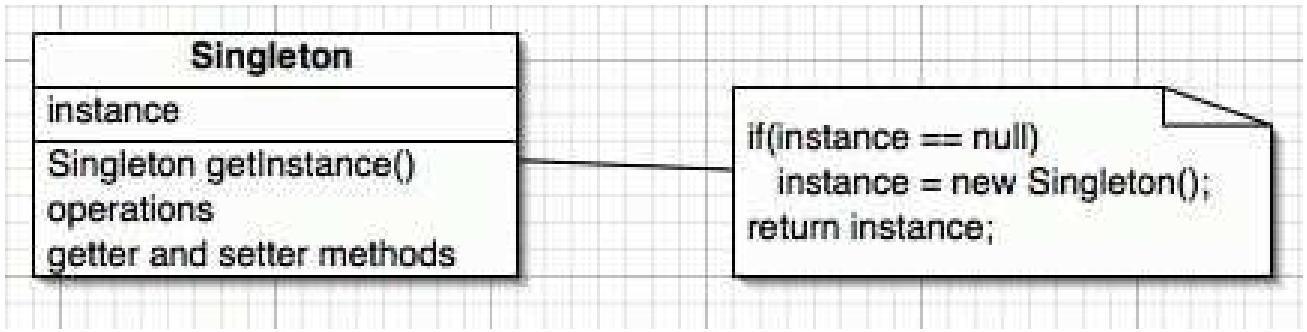
- 23 design patterns classifiés en 3 groupes d'après leur **rôle**

Création	**Structuration**	**Def. de Comportement**
AbstractFactory	Adapter, Bridge	Interpreter, Command
Factory	Composite,	Chain of Responsability
Builder	Facade, Proxy	Iterator, Mediator,Template
Prototype	Flyweight	Memento, Observer
Singleton	Decorator	State, Strategy, Visitor

Création d'objets

- Exemple: `Singleton`
  - Un singleton est une classe qui ne produit qu'une seule instance, et cette dernière est accessible de partout.
  - Pour cela, il faut que la construction d'une nouvelle instance grâce à l'opérateur `new` soit interdite.
  - La solution est de rendre le constructeur de la classe privée.
  - Mais maintenant comment récupérer une instance de la classe ?

Schéma UML


Singleton
instance
Singleton getInstance()
operations
getter and setter methods
if(instance == null)
instance = new Singleton();
return instance;

Code Java

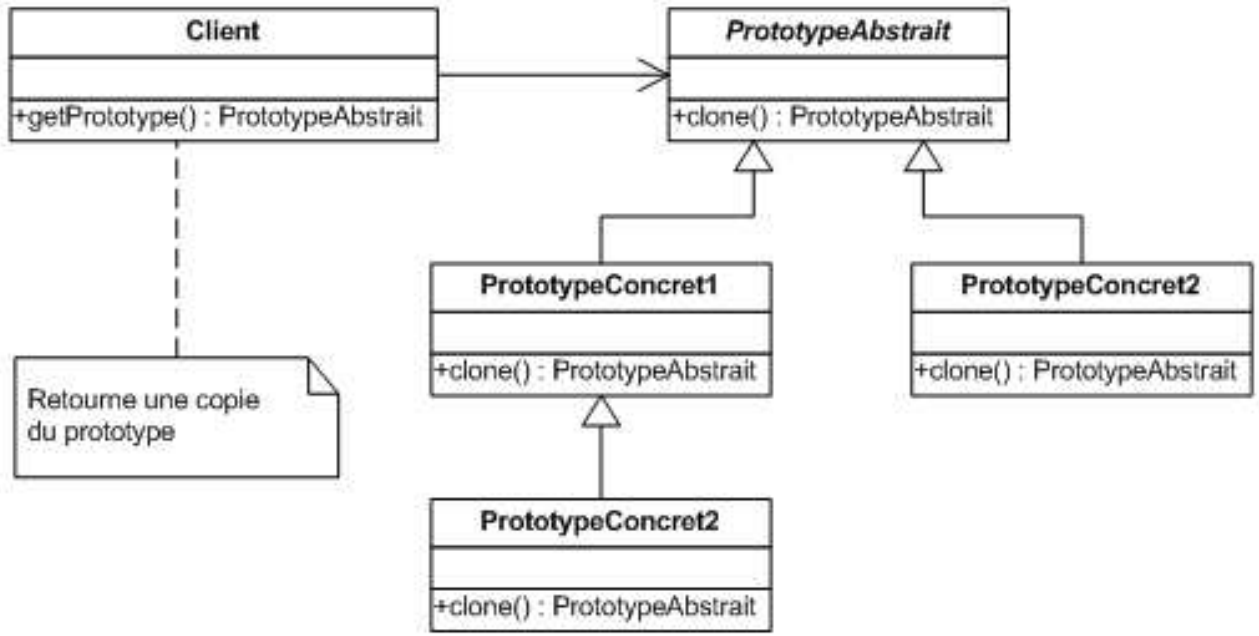
```
public class ClassicSingleton {
 private static ClassicSingleton instance = null;

 protected ClassicSingleton() {
    // Exists only to defeat instantiation.
 }
 public static ClassicSingleton getInstance() {
    if(instance == null) {
       instance = new ClassicSingleton();
    }
    return instance;
 }
}
```

Création d'objets

- Exemple: `Prototype`
  - Le prototype est le pattern qui va vous permettre de créer des copies exactes d'objets sans même connaître leurs types.
  - C'est donc une extension du constructeur par copie pour pouvoir travailler avec du polymorphisme.
  - Utilisation : génération automatique de classes et clonage d'objets.

Schéma UML


Client
+getPrototype() : PrototypeAbstrait
PrototypeAbstrait
+clone() : PrototypeAbstrait
Retourne une copie
du prototype
PrototypeConcret1
+clone() : PrototypeAbstrait
PrototypeConcret2
+clone() : PrototypeAbstrait
PrototypeConcret2
+clone() : PrototypeAbstrait

Autres paterns de création

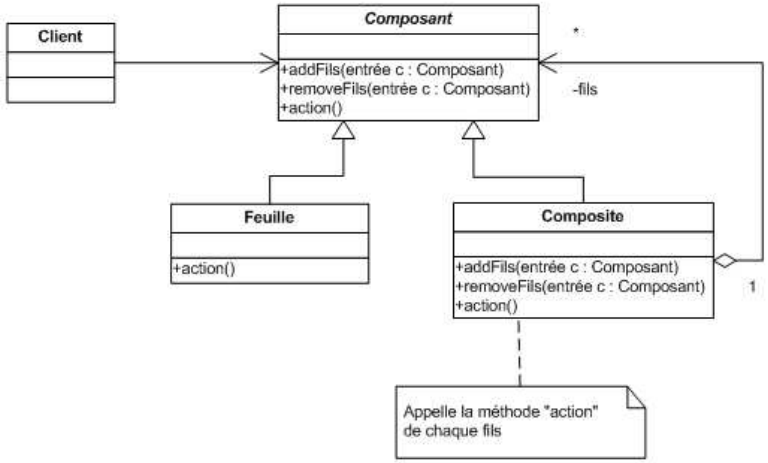
- **Abstract Factory** : création de familles d'objets apparentés ou interdépendants (parfois équivalent à Prototype)
- **Builder** : lorsque la création d'un objet est complexe et indépendante des parties qui compose l'objet - exemple : Convertisseur de texte.
- **Factory** : permet de déléguer l'instanciation des objets aux sous-classes - exemple : manipulateur de figure

Design Pattern de Structure

- Les Design Patterns de structure sont les patterns qui concernent des problèmes d'assemblage d'objets et de structure de code.
- Plutôt que de composer des interfaces et des implémntations, ces Design Patterns décrivent les moyens de composer des objets pour réaliser de nouvelles fonctionalités

Composite

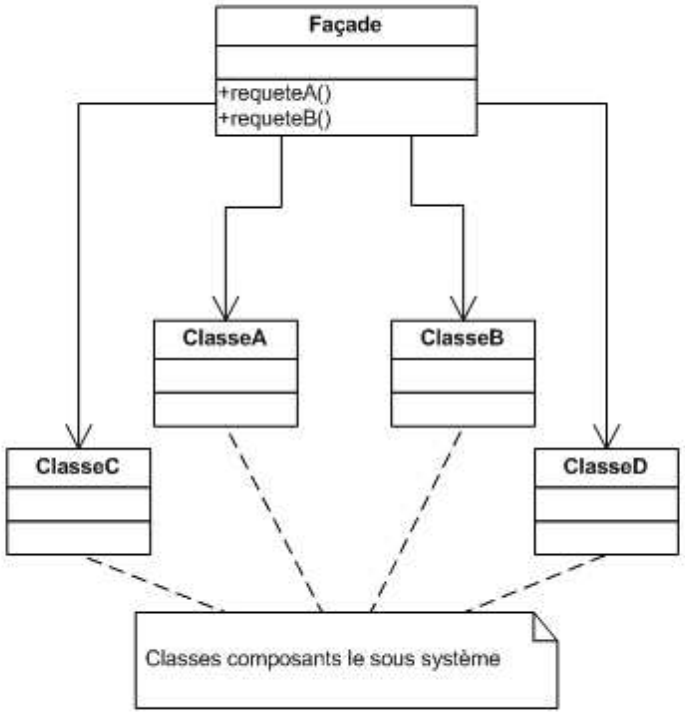
- **Objectif** : représentation des structures d'arbres, ou composition récursives.
- **Avantages** : pouvoir traiter de la même manière un objet ou une collection d'objets.



Facade

- Consiste à donner une interface unifiée et haut niveau à un sous-système composé de plusieurs interfaces et objets aux interactions complexes.
- Typiquement, la façade peut répondre à des requêtes de l'utilisateur en appelant des méthodes de divers objet du sous-systême.
- L'utilisateur appelle alors des méthodes de la façade sans se soucier du travail effectué par le sous-systême.

Schéma UML


Façade
+requeteA()
+requeteB()
ClasseA
ClasseB
ClasseC
ClasseD
Classes composants le sous système

Les autres Patterns de structuration

- **Bridge** : découplage entre une abstraction et une implémentation. Exemple : création de fenêtre indépendemment de la plate-forme.
- **Adapter** : convertit l'interface d'une classe en une autre répondant aux attentes du client, soit par composition soit par héritage multiple.
- **Decorator** : ajoute dynamiquement des caractéristiques (ou des comportements) à un objet (une classe). Exemple : attributs d'un windown manager.

Les autres Patterns de structuration

- **Proxy** : intercepte les requêtes vers un objet et instancie celui-ci quand il est réellement necéssaire. Exemple : affichage d'image et création de bitmaps.
- **Flyweight** : objet partagé et sucesptible d'être utilisé par plusieurs contextes simultanément. Exemple les caractères d'un texte.

Design Patterns de Comportement

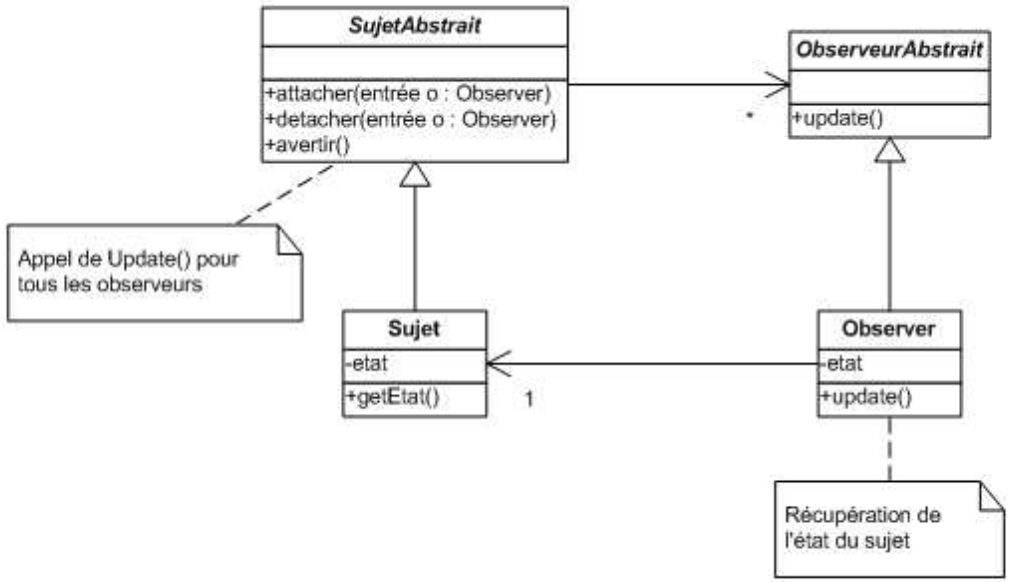
Deux groupes de patterns :

- Les patterns orientés Classes qui utilisent les notions d'héritage et de polymorphisme pour décrire des algorithmes et des flux de traitements.
  - **Template pattern** et **interpreter**
- Les patterns orientés Objets,décrivent les interactions dans un groupe d'objets pour, par exemple effectuer certains traitements ne pouvant pas être effectués par un objet seul. Ils utilisent la composition.

Observer

- Permet de définir une dépendance entre plusieurs objets, telle que, lorsqu'un des objets (le sujet) change d'état (modification de certains de ses attributs), les autres objets (observeurs) en soit avertis immédiatement.
- **Utilisation** : quand un objet doit faire de la notification à d'autres sans faire d'hypothèse sur la nature de ces autres.
- **Exemple** : des données qui sont visualisées dans un tableur et sous forme graphique : diagramme, histogramme... Quand les données sont modifiées toutes les représentations doivent en être informées, sans qu'on construise de dépendance entre ces classes.

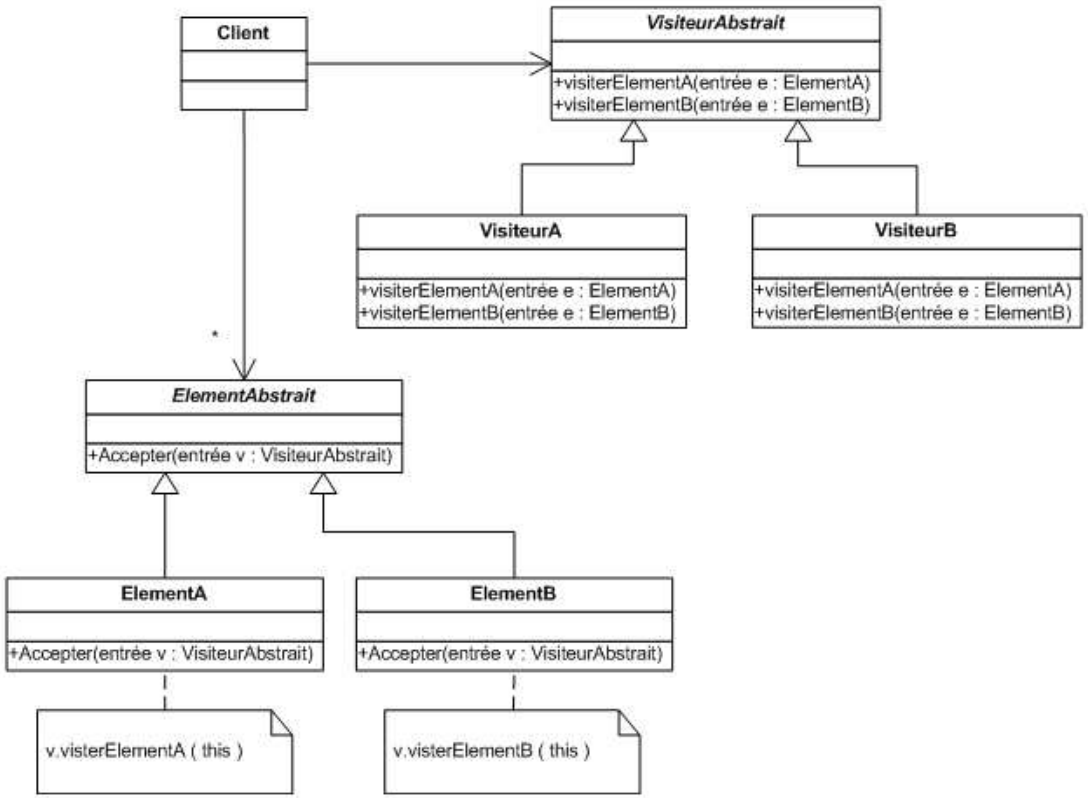
Schéma UML


SujetAbstrait
+attacher(entrée o : Observer)
+detacher(entrée o : Observer)
+avertir()
ObserveurAbstrait
+update()
*
Appel de Update() pour
tous les observeurs
Sujet
-etat
+getEtat()
1
Observer
-etat
+update()
Récupération de
l'état du sujet

Un exemple : Visitor

- **Objectif** : permet d'ajouter des traitements à une classe sans avoir à la modifier.
- **Utilisation** :
  - Les classes ElementA et ElementB dérivent de ElementAbstrait, on ajoute des fonctionnalités spécifi ques sans perdre l'avantage d'une interface commune bien défi nie et du polymorphisme.
  - Pas de modifi cation des classes ElementA et ElementB mais création d'une méthode virtuelle Accepter(Visiteur),
  - Le visiteur est l'objet qui va nous permettre d'effectuer les traitements.
  - Création d'une classe VisiteurAbstrait contenant deux méthodes visiterElementA(ElementA) et visiterElementB(ElementB) et on peut ainsi créer des classes Visiteur dérivées qui redéfi nissent ces méthodes.

Schéma UML


Client
VisiteurAbstrait
+visiterElementA(entrée e : ElementA)
+visiterElementB(entrée e : ElementB)
VisiteurA
+visiterElementA(entrée e : ElementA)
+visiterElementB(entrée e : ElementB)
VisiteurB
+visiterElementA(entrée e : ElementA)
+visiterElementB(entrée e : ElementB)
*
ElementAbstrait
+Accepter(entrée v : VisiteurAbstrait)
ElementA
+Accepter(entrée v : VisiteurAbstrait)
ElementB
+Accepter(entrée v : VisiteurAbstrait)
v.visterElementA (this)
v.visterElementB (this)