



Design Patterns

- D'où viennent les design patterns ?
- Pourquoi utiliser des patterns ?
- Comment devient-on un pattern ?
- Les patterns vue de loin
- Les patterns vue d'un peu plus près

*Design patterns (catalogue de modèles de conception réutilisables)
Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides: gang of four*



D'où viennent les design patterns ?

- Plus de 25 ans d'expérience de la POO
- Des projets de très grande taille
- Construction progressive d'une véritable expertise dans la réalisation d'architecture de programmes orientés objets
- Les design patterns sont nés de la capitalisation de cette expérience



Pourquoi utiliser des patterns ?

- Les design patterns ou patrons de conception sont des solutions standardisées à des problèmes de conception classiques
- Proposés par des experts de la POO
- Utiles pour les applications de grande taille
- Il existe 23 patterns qui font l'unanimité répertoriés dans un ouvrage de référence (cf. première diapo)



Pourquoi utiliser des patterns ?

- Solutions indépendantes des langages
- Solutions abstraites, de haut niveau
- Souvent orientés vers le bon découpage en package (modularité, flexibilité, réutilisabilité)
- Exprimé sous forme d'architecture reliant quelques classes très abstraites
- Représentent beaucoup sur des interfaces



Comment devient-on un pattern ?

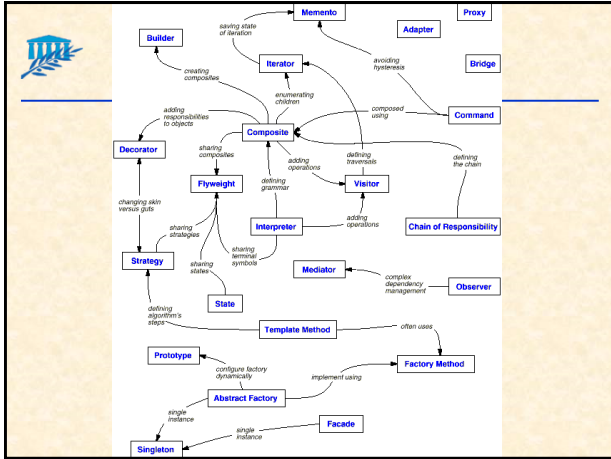
- Pour devenir un pattern, un modèle de conception doit vérifier les contraintes suivantes :
 - Être présent dans au moins trois gros projets très largement utilisés
 - Se trouver dans des programmes écrits dans des langages différents (C++, Eiffel, Smalltalk, Java, C#...)
 - Être minimal (non décomposable)
 - Être totalement documenté
- Seuls 23 en 10 ans !



Les patterns vue de loin

- 3 catégories de patterns : création (5), structurel (7) et comportemental (11)

Création	Structurel	Comportemental
Abstract Factory	Adapter	Interpreter
Factory Method	Composite	Template Method
Builder	Proxy	Chain of Responsibility
Prototype	Bridge	Command
Singleton	Decorator	Iterator
	Facade	Mediator
	Flyweight	Memento
		Observer
		State
		Strategy
		Visitor



CREATION Factory Method

- Définit une interface pour créer un objet mais laisse les sous-classes décider quelle classe instancier. L'usine permet à une classe de déléguer une instantiation à ses sous-classes

CREATION Factory Method

- Dans un Framework générique, il peut être nécessaire de créer des instances des objets manipulés
- Ces instances sont des objets de type spécifiques
- Donc leur type n'est pas connu par le concepteur du Framework
 - `MonInstanceGénérique = new MonTypeSpecifique();`
 - `MaFactory.creeMonInstanceGénérique();`
- Méthode écrite par l'utilisateur, qui encapsulera la création de l'objet spécifique

CREATION Factory Method

- Classe à laquelle est déléguée la création des objets spécifiques
- Création d'un objet en fonction du contexte appelant
- Définie dans le Framework comme une interface implémentée par l'utilisateur

```

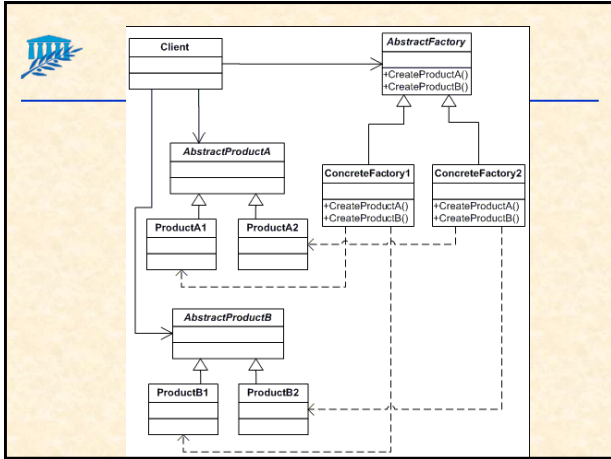
interface MaFactoryGénérique
{
    MonInstanceGénérique creeMonInstanceGénérique();
}
class MaFactoryConcrète implements MaFactoryGénérique
{
    Mon InstanceGénérique creeMonInstanceGénérique()
    {
        return new MonInstanceConcrète();
    }
}
  
```

CREATION Factory Method

- Exemple

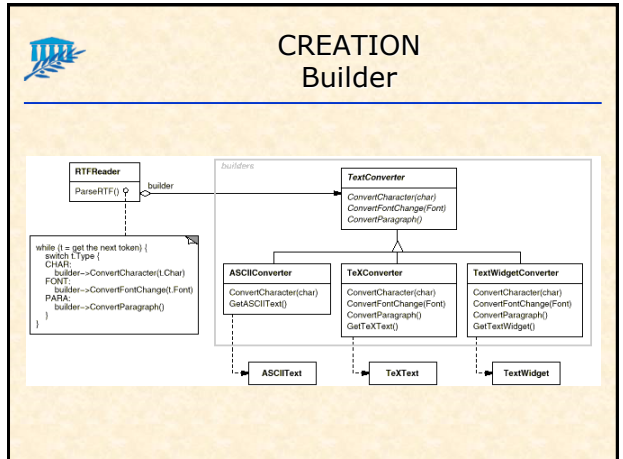
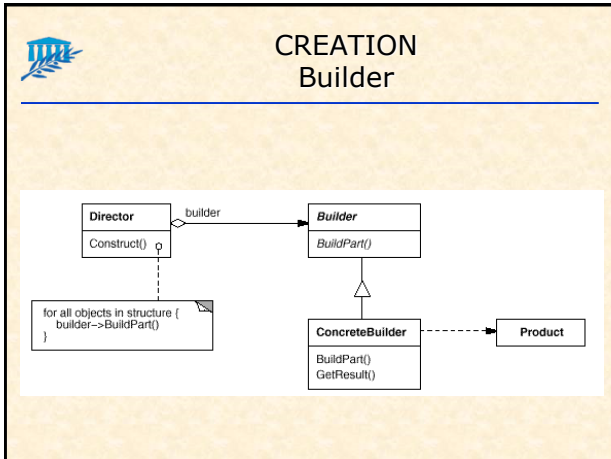
CREATION Abstract Factory

- C'est une généralisation de Factory pour la création de groupes d'objets liés
- Création d'une Factory en fonction du contexte appelant puis des objets en fonction du contexte



CREATION Builder

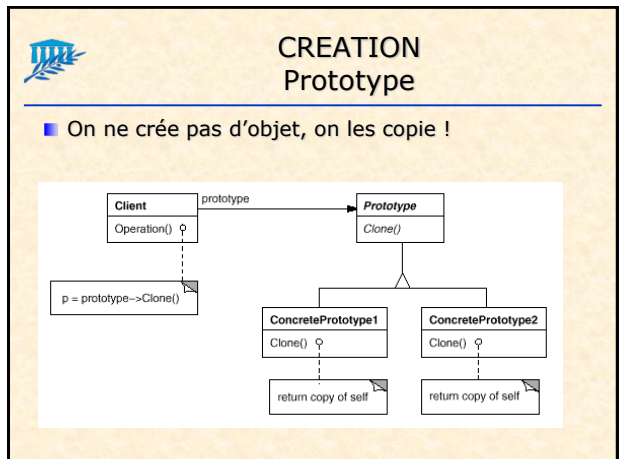
- Sépare la construction d'un objet complexe de son implémentation de façon à ce que plusieurs représentations différentes peuvent utiliser le même processus de construction (différents formats de fichier par exemple)
- Factory Method ou Abstract Factory ne le permettent pas de façon élégante (ajout de paramètres aux fabriques)

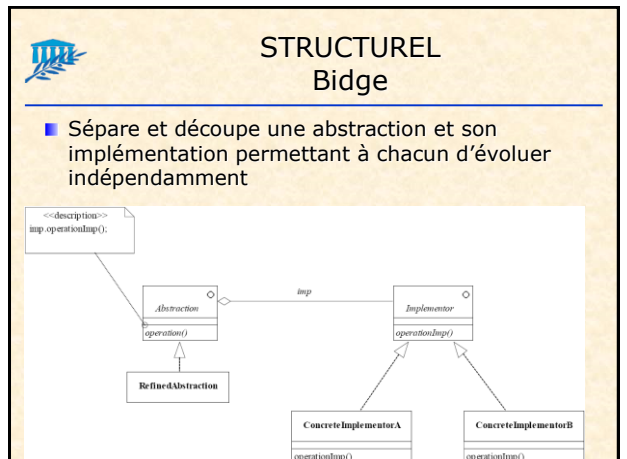
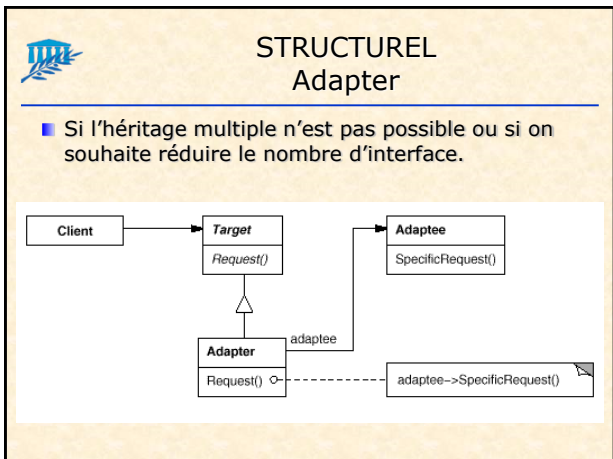
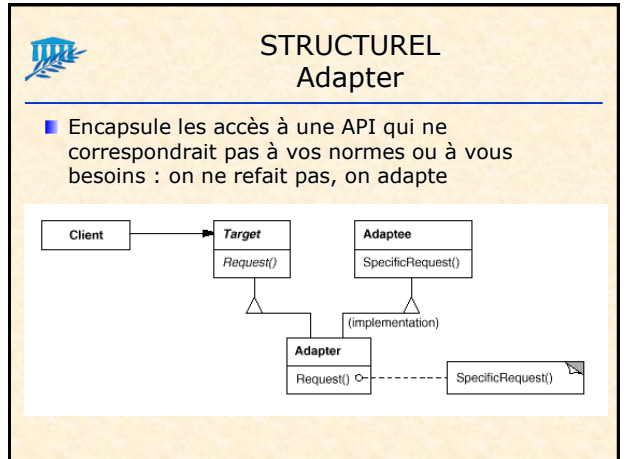
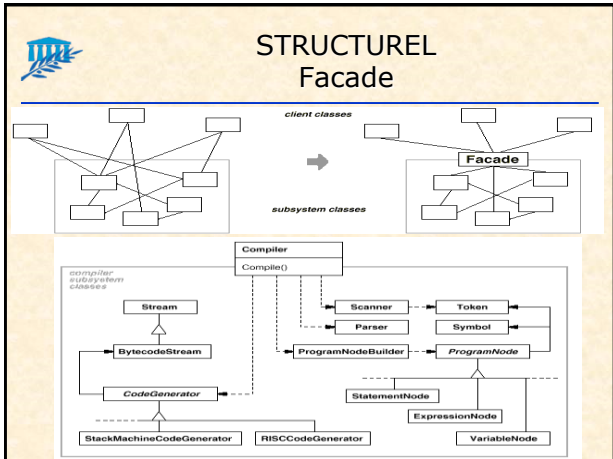
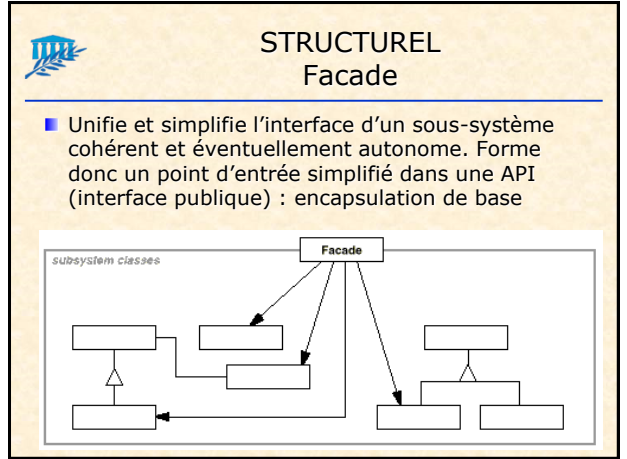
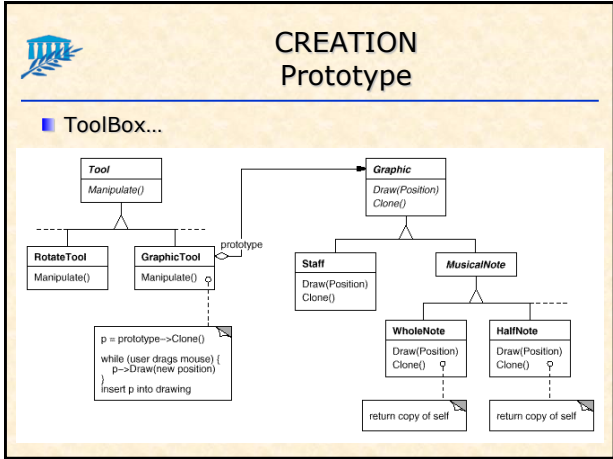


CREATION Singleton

- S'assure de l'unicité d'une instance de classe et évidemment le moyen d'accéder à cette instance

Singleton static Instance() ◊ SingletonOperation() GetSingletonData() static uniqueInstance singletonData	return uniqueInstance
---	-----------------------

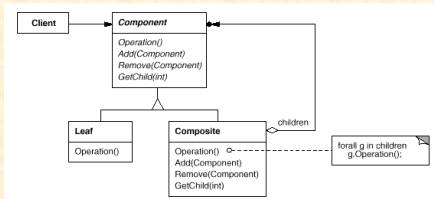




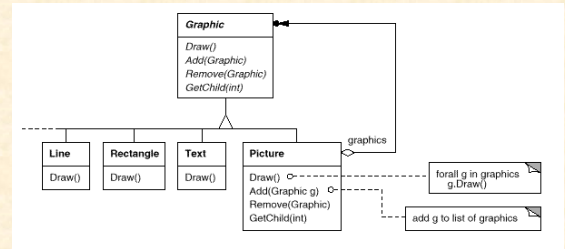


STRUCTUREL Composite

- Compose les objets dans une structure arborescente qui représente une hiérarchie «partie de». Le pattern Composite permet au client de traiter de la même manière un objet ou un ensemble d'objets.

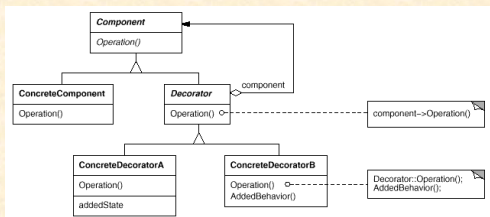


STRUCTUREL Composite

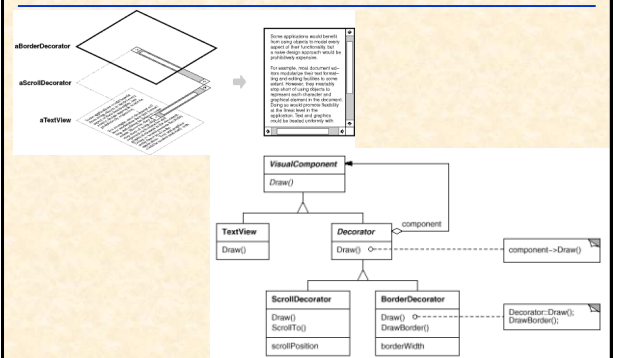


STRUCTUREL Decorator

- Attache une nouvelle responsabilité à un objet dynamiquement. Le Decorator fournit une alternative souple à l'héritage pour étendre les fonctionnalités (parfois plus efficace)

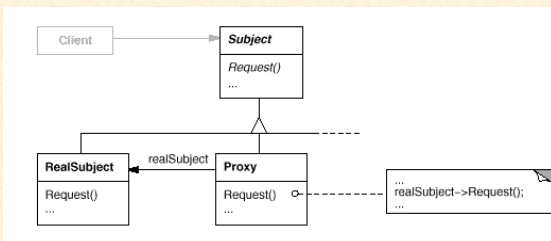


STRUCTUREL Decorator

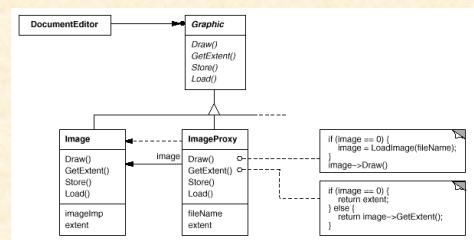


STRUCTUREL Proxy

- Sorte de suppléant d'un objet afin de vérifier son accès et parfois utiliser comme « représentant » léger de l'objet



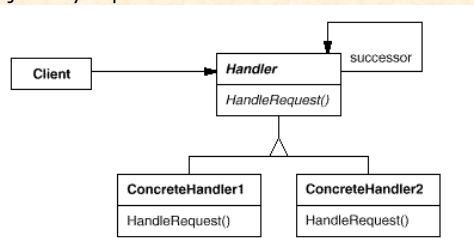
STRUCTUREL Proxy



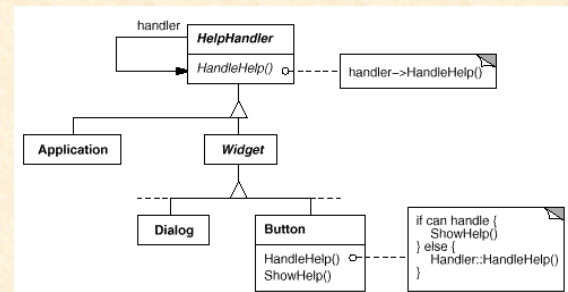


Comportemental Chain of responsibility

- Évite de coupler l'émetteur d'une requête et son receveur, en donnant la possibilité à plusieurs objets d'y répondre.

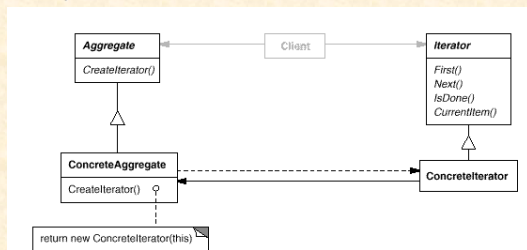


Comportemental Chain of responsibility



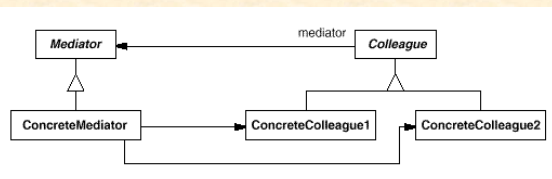
Comportemental Iterator

- Fournit un moyen d'accéder séquentiellement aux éléments d'une agrégation d'objets sans dévoiler sa représentation

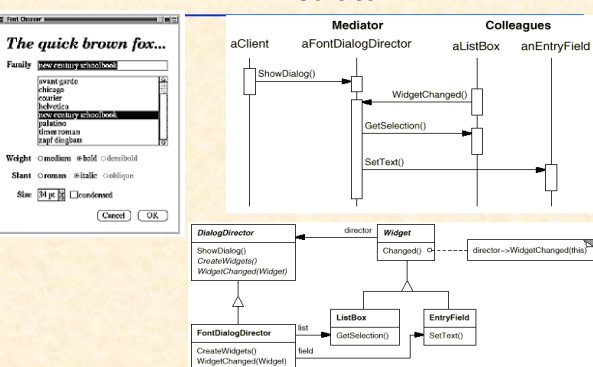


Comportemental Mediator

- Définit un objet qui encapsule comment un ensemble d'objet interagissent
- Évite le couplage fort entre objet

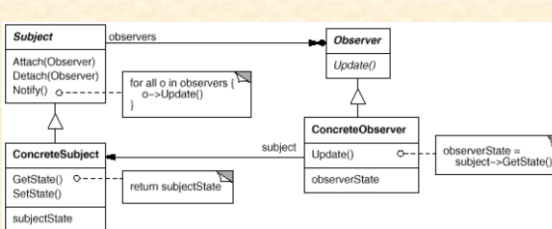


Comportemental Mediator



Comportemental Observer

- Définit une relation un-à-plusieurs telle que lorsqu'un objet change, tous les objets liés sont notifiés de cette modification et mis à jour



Comportemental Observer

- Plusieurs vues pour une même document

Comportemental Strategy

- Définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. S'applique sur les mêmes données

Comportemental Strategy

- Découpage en phrase par ligne, par paragraphe ou d'un tableau

Héritage vs Délégation

- Hiérarchie complexe

→ **Mauvais critère de classification**

Héritage vs Délégation

- Hiérarchie complexe

→ **critère structurant**

- Si on doit modifier la hiérarchie cela implique des réorganisations majeures du code → conception préalable importante

Héritage vs Délégation

- Avantages de l'héritage
 - partie commune en commun !
- Inconvénients de l'héritage
 - Couplage fort entre les classes mères et filles, une modification dans la classe mère modifie toutes les filles
 - Typage dynamique donc lenteur d'exécution
 - D'après the gang of four, il est préférable de faire de la délégation



Conclusion

- Ne pas utiliser le Design Patterns si on n'est pas sûr d'avoir clairement identifié le bon pattern
- Le métier de concepteur s'éloigne de plus en plus du code
- Outils liés uniquement à la conception donc pas de code
- Les AGL utilisent de plus en plus les Design Patterns