

PARTIE VIII

Patrons de conception *(Design Patterns)*

Bruno Bachelet

Loïc Yon

- Introduction
 - Motivations
 - Réutilisation au niveau conceptuel
- Description
 - Référencement des patrons
 - Les patrons du GoF
 - Classification des patrons
- Patrons de création
 - Processus de création d'objets
- Patrons de structure
 - Structure d'objets complexes
- Patrons de fonctionnement
 - Organisation des algorithmes

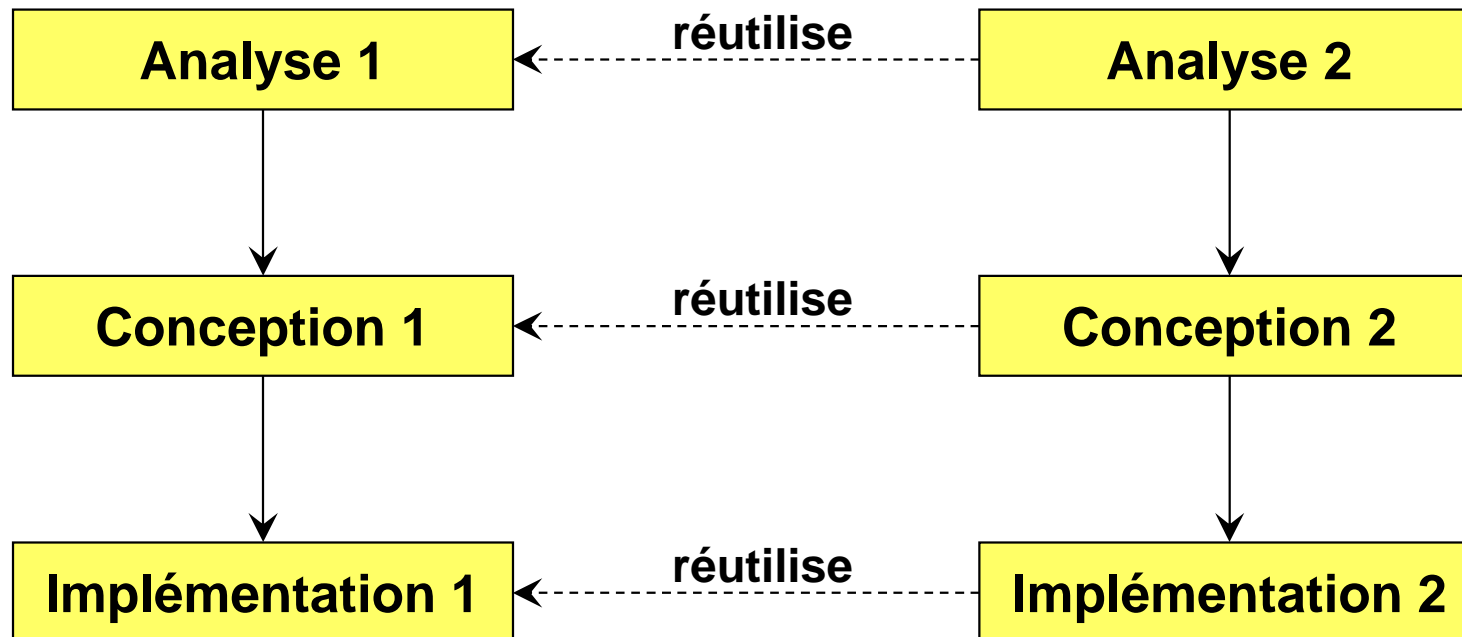
- Concevoir un système objet est difficile
- Beaucoup d'aspects à considérer
 - Décomposition du système
 - Factorisation du code
 - Relations entre les composants
 - Héritage, association, agrégation / composition, délégation
- Prévoir et intégrer dès la conception
 - Réutilisation du code
 - Evolutions / extensions possibles

⇒ Introduire de la réutilisabilité

- Bénéficier des bonnes pratiques de l'industrie
 - Minimiser les risques dans la phase de développement
 - Se référer à l'existant
 - Reprendre des solutions éprouvées

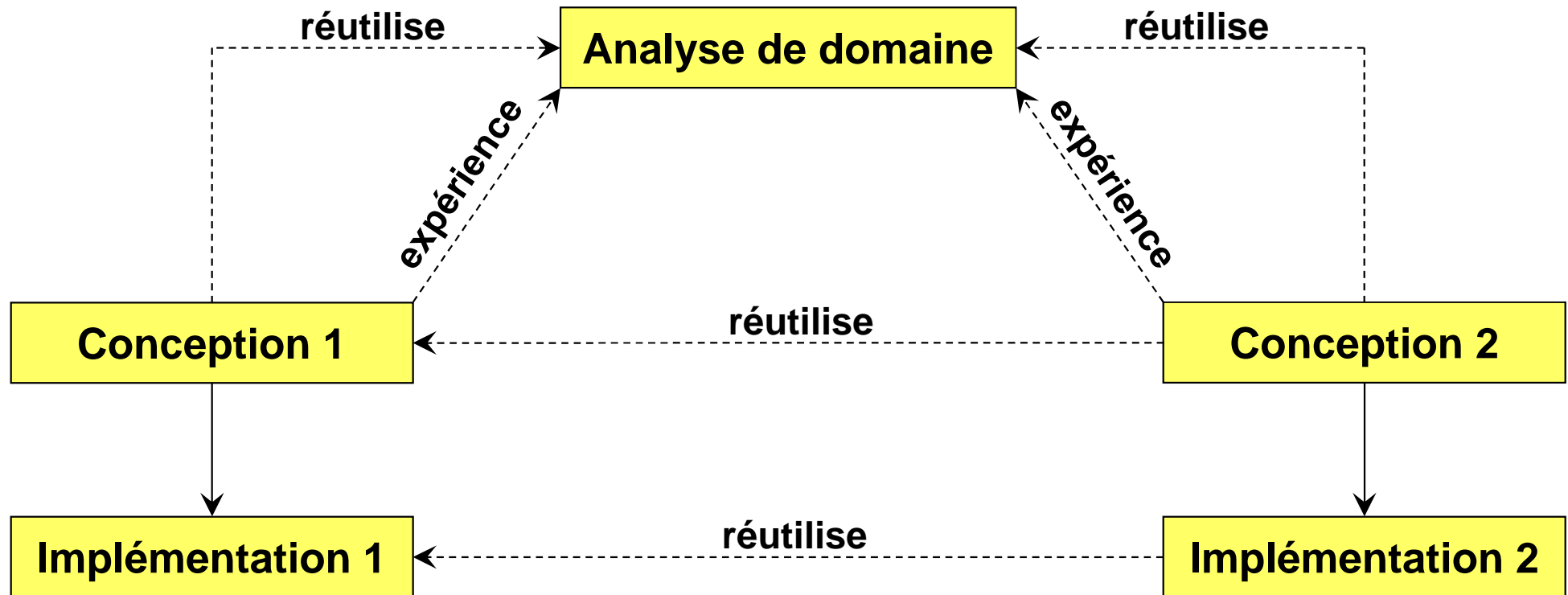
- Permettre une réutilisation
 - Au niveau implémentation
 - Mêmes structures de données / algorithmes
 - ⇒ Bibliothèques logicielles
 - Au niveau conception
 - Mêmes organisations des composants
 - ⇒ Patrons de conception (ou «*design patterns*»)

Réutilisation: niveaux d'abstraction



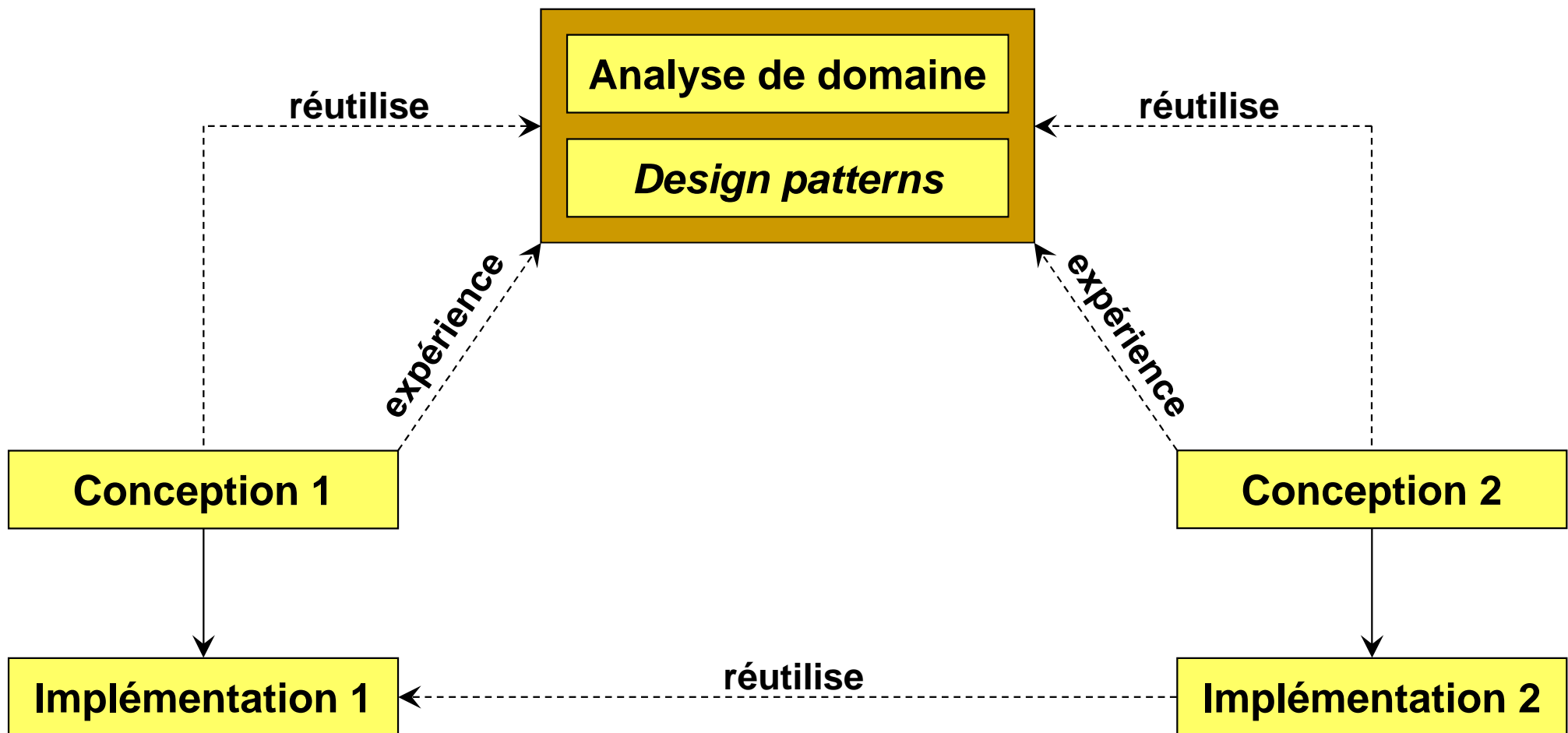
- A chaque nouvelle expérience, on peut réutiliser
- Sans outil particulier: réutilisation niveau par niveau

Réutilisation: analyse de domaine



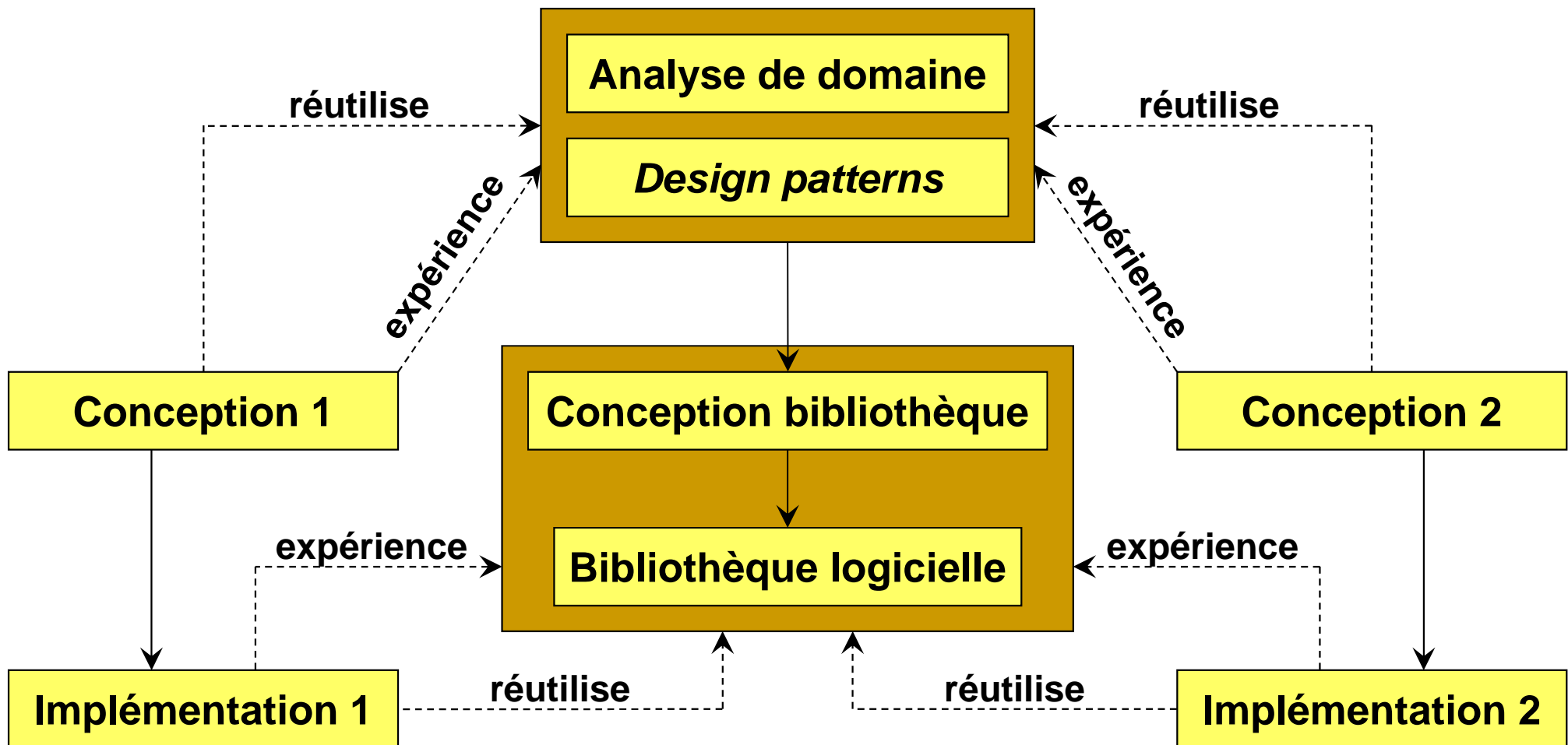
- Profiter de plusieurs expériences du même domaine

Réutilisation: patrons de conception



- Problèmes de conception récurrents \Rightarrow Patrons de conception
- Solutions génériques réutilisables au niveau conception

Réutilisation: cadriciel (*framework*)



- Réutilisation au niveau implémentation \Rightarrow Bibliothèques logicielles
- Cadriciel = composants réutilisables (conception + implémentation)

Patrons de conception (ou *design patterns*)

■ Définition

- ❑ Un *design pattern* traite un problème de conception récurrent
- ❑ Il apporte une solution générale, indépendante du contexte

■ En clair

- ❑ Description de l'organisation de classes et d'instances en interaction pour résoudre un problème de conception

■ Solution générique de conception

- ❑ Doit être «élégante» et réutilisable
- ❑ Doit être testée et validée dans l'industrie logicielle
- ❑ Doit viser un gain en terme de génie logiciel
- ❑ Doit être indépendante du contexte

Référencement des *design patterns* (1/2)

- Tentatives de référencement des patrons de conception
- Livre fondateur
 - ❑ «Design Patterns: Elements of Reusable Object-Oriented Software»
 - ❑ Gamma, Helm, Johnson, Vlissides
 - Surnommé le «GoF» (*Gang of Four*)
 - ❑ Addison-Wesley, 1994
 - ❑ 23 patrons de conception
- Lecture conseillée
 - ❑ «Design patterns tête la première», O'Reilly
 - ❑ «Pour mieux développer avec C++», Dunod
 - ❑ «Modern C++ Design», Addison-Wesley



Référencement des *design patterns* (2/2)

- Les patrons présentés ici sont issus du *GoF*
- Mais ce ne sont pas les seuls !
 - Patron MVC (Modèle-Vue-Contrôleur)
 - Il n'est pas dans la liste du *GoF*
 - Patrons GRASP
 - Proposés par Craig Larman
 - Plus conceptuels
- Communauté active
 - De nouveaux patterns proposés régulièrement
 - Démocratique: adoptés si utilisés et généraux
 - Exemples (cf. Wikipedia)
 - *Reversible command (undo)*
 - *Lazy initialization*
 - Patrons de concurrence

Patrons de conception du *GoF* (1/3)

- Quatre éléments principaux définissent un patron
- Objectif
 - ❑ Description de son utilité
- Problème / Motivation
 - ❑ Quand appliquer le patron de conception
 - ❑ Relations problématiques entre les classes
- Solution proposée
 - ❑ Éléments impliqués
 - ❑ Leurs relations
 - ❑ Schémas conceptuels (e.g. diagrammes UML)
- Conséquences
 - ❑ Compromis éventuels
 - ❑ Qualité de la solution

Patrons de conception du *GoF* (2/3)

- Classification selon deux critères
- Cible: qui est concerné ?
 - ❑ Les classes
 - Relations d'héritage
 - Aspect statique
 - ❑ Les instances
 - Relations de composition
 - Aspect dynamique
- Objectif: que veut-on faire ?
 - ❑ Création de composants \Rightarrow Patrons de création
 - ❑ Assemblage de composants \Rightarrow Patrons de structure
 - ❑ Comportement des composants \Rightarrow Patrons de comportement

Patrons de conception du *GoF* (3/3)

Critères		Objectif		
		Création	Structure	Comportement
Cible	Classe	<i>Factory Method</i>	<i>Adapter</i>	<i>Interpreter</i> <i>Template Method</i>
	Instance	<i>Abstract Factory</i> <i>Builder</i> <i>Prototype</i> <i>Singleton</i>	<i>Adapter</i> <i>Bridge</i> <i>Composite</i> <i>Decorator</i> <i>Facade</i> <i>Flyweight</i> <i>Proxy</i>	<i>Chain of Responsibility</i> <i>Command</i> <i>Iterator</i> <i>Mediator</i> <i>Memento</i> <i>Observer</i> <i>State</i> <i>Strategy</i> <i>Visitor</i>

Principes généraux des *design patterns*

- Favoriser une bonne conception

- ❑ Facile à appréhender
- ❑ Facile à faire évoluer
- ❑ Résistante aux changements

⇒ Quelques principes permettent de tendre à ces buts

- ❑ Responsabilité unique
- ❑ Connaissance minimale
- ❑ Ouvert/fermé
- ❑ Encapsuler ce qui varie
- ❑ Programmer envers une interface
- ❑ Favoriser la composition à l'héritage

- Faire une seule chose, et le faire bien
⇒ cohésion forte des classes et des modules
- Une classe devrait avoir une seule raison de changer
 - Facilite la compréhension
 - Limite le risque d'introduction de bugs
 - Particulièrement lors d'évolution
 - Facilite les tests
- Exemple: séparer le calcul de données de leur lecture/écriture dans un fichier

- Ne parler qu'à ses «connaissances» proches
- Loi de Déméter
 - Invocation de méthodes sur
 - Soi-même
 - Paramètres de méthode
 - Objets créés (variables locales)
 - Attributs
 - Éviter d'appeler les méthodes d'un objet retourné par une autre méthode
- Limite les couplages
- Facilite la compréhension

- Etre ouvert aux extensions...
 - Permettre l'ajout de fonctionnalités
 - Permettre la modification du comportement
- ...mais fermé aux modifications
 - Le code d'un module ne devrait pas devoir être modifié si les besoins changent
- Besoins changent régulièrement
⇒ nécessité de pouvoir évoluer
- Tout en évitant de casser du code existant

Encapsuler ce qui varie

- Séparer les aspects susceptibles de changer de ce qui ne changera pas
- Protège contre le changement
 - Stabilité du code face aux modifications
- Flexibilité pour les comportements sujets à variation
- Exemple
 - Comparateur dans un algorithme de tri

Programmer envers une interface

- Programmer envers une interface et pas une implémentation
- Module sans dépendance avec les détails d'implémentation
 - Modification de l'implémentation sans impact sur le module
 - Changement d'implémentation facilitée
- Exemple (Java)
 - Dépendre de `Collection` plutôt que de `ArrayList`

Favoriser la composition à l'héritage

- Héritage → statique
- Composition → dynamique
- Utilisation de la délégation
 - ❑ Couplage plus faible
 - ❑ Changement de fournisseur de services à l'exécution
- Évite souvent des héritages sans réelle relation «est un»
 - ❑ Principe de substitution de Liskov
- Exemple
 - ❑ **EnregistreurResultats** agrège **Writer**
 - ❑ Plutôt que de spécialiser **Writer**

Mécanisme de délégation (1/2)

- Principe
 - ❑ Rediriger un message vers un autre objet
 - ❑ Utilise la composition: délégant vers délégué
- Intervient dans de nombreux patrons du *GoF*
 - ❑ Peut être une alternative à l'héritage
- Plusieurs manières de rediriger
 - ❑ Contrôle du message (e.g. *Proxy*)
 - ❑ Changement de message (e.g. *Adapter*)
 - ❑ Changement de délégué (e.g. *Chain of Responsibility*)
 - ❑ Redéfinition du message (e.g. *Decorator*)

Mécanisme de délégation (2/2)

