Réseaux et systèmes répartis C3

Java 2 : Java et la conception orientée objets

CAMOS - CNAM

Plan du jour

- Principes avancés de la conception de systèmes orientés objet
- Les Design Patterns
  - avec un peu de Java
- Les Anti-Patterns

Les concepts de base de l'objet

- Nous avons déjà vu le principe Apple PIE
  - **A**bstraction
  - **P**olymorphisme
  - héritage (**I**nheritance)
  - **E**ncapsulation
- Ce principe permet de comprendre la théorie des langages orientés objet mais n'aide pas beaucoup pour la conception de systèmes et/ou programmes

Le besoin d'autre chose

- La conception orientée objet peut être facilitée par la compréhension :
  - de principes avancés de la théorie objet, qui influent sur la qualité de la conception
  - de principes de conception abstraits qui ont été observés dans les bons designs et constituent ainsi de bonnes pratiques
- Nous allons donc étudier
  - ces principes avancés
  - les Design Patterns

Phénomènes de dégénérescence d'une application

- La dégénérescence d'une application est causée par :
  - Rigidité : chaque évolution impacte de nombreuses parties de l'application
    - développement et modifications à coûts élevés
  - Fragilité : la modification d'une partie de l'application provoque des erreurs dans une autre partie
    - modifications risquées, coût de maintenance élevé
  - Immobilité : difficulté d'extraire une partie du code pour le réutiliser dans une autre application
    - coûts de développement élevés puisque l'on repart à chaque fois de zéro

Besoin : gestion des dépendances

- Une mauvaise gestion des dépendances fait apparaître les syndromes énoncés précédemment
- Un contrôle strict des dépendances permet d'aboutir à
  - robustesse : les changements n'introduisent pas de régression
  - extensibilité : ajout facile de nouvelles fonctionnalités
  - réutilisabilité : certaines parties d'une application peuvent être réutilisées dans une autre

Gestion des évolutions et des dépendances entre classes

- Principe d'ouverture / fermeture
  - OCP : Open-Closed Principle
- Principe de substitution de Liskov
  - LSP : Liskov Substitution Principle
- Principe d'inversion des dépendances
  - DIP : Dependency Inversion Principle
- Principe de séparation des interfaces
  - ISP : Interface Segregation Principle

Principe d'ouverture / fermeture (OCP)

- **"Tout module (package, classe, méthode) doit être ouvert aux extensions mais fermé aux modifications"**
- ouvert aux extensions pour être étendu et proposer des comportements non prévus au départ
- fermé aux modifications pour que les extensions introduites ne modifie pas le code du module d'origine
- Besoin d'abstraction et de délégation abstraite
  - interfaces, classes abstraites ou templates en fonction des langages de programmation objet

Principe de substitution de Liskov (LSP)

- "Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir"
- Ce principe repose sur la légitimité des abstractions : pour prétendre à l'héritage, une sous-classe doit être conçue de sorte que ses instances puissent se substituer à des instances de la classe de base partout où cette classe de base est utilisée

Principe de substitution de Liskov (LSP) suite

- Le principe n'est pas toujours simple à suivre dans les gros systèmes :
  - modification de la classe mère (avec impact sur les toutes les classes filles)
  - recours au downcasting (avec rupture de l'OCP)
- Notion de commonalité (avec variables *positives*) pour la classe de base et de variabilité (avec des variables *négatives*) (Coplien)
- Le même principe peut s'appliquer à la spécialisation de templates ou à la surcharge de fonctions

Principe d'inversion des dépendances (DIP)

- "Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstraction."
- "Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions"
- Le respect de ce principe permet de réutiliser des modules "métier" car ils ne dépendent pas de modules techniques type IHM

Principe d'inversion des dépendances (DIP) suite

- En respectant ce principe, les modules de bas niveau (techniques) doivent se conformer à des interfaces définies et utilisées par des modules de haut niveau
  - et non plus l'inverse où la logique métier est définie en utilisant des briques de base techniques
- Les interfaces introduites dans les modules métier permettent de mettre en œuvre une délégation abstraite
- Un même module métier devient ainsi utilisable dans plusieurs contextes techniques

Principe de séparation des interfaces (ISP)

- "Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas"
- La définition de classes de service trop riches peut se traduire par l'implémentation de clients :
  - n'utilisant pas tous le service disponible
  - pouvant être impactés par des modifications de sous-services qu'ils n'utilisaient pas
- La classe de service doit être scindée

Organisation de l'application en modules

- Principe d'équivalence livraison / réutilisation
  - REP : Reuse / Release Equivalence Principle
- Principe de réutilisation commune
  - CRP : Common Reuse Principle
- Principe de fermeture commune
  - CCP : Common Closure Principle

Principe d'équivalence livraison / réutilisation (RREP)

- "La granularité en terme de réutilisation est le package. Seuls des packages livrés sont susceptibles d'être réutilisés"
- Le code réutilisé reste la propriété de son auteur qui reste en charge des corrections et de l'évolution (anti eXtreme Programming)
- Le code est réutilisé tel quel et celui qui réutilise le code se contente de passer par des interfaces minimales sans avoir à comprendre le code (comme pour un ADT)

Principe de réutilisation commune (CRP)

- "Réutiliser une classe d'un package, c'est réutiliser le package entier"
- Si plusieurs classes doivent être réutilisées ensemble, il est plus simple de les intégrer dans un même package (qui devient une librairie)
- Cependant, cette technique force à dépendre d'une libraire : il faut faire attention à ne pas transposer la "non séparation des interfaces" dans le monde des packages

Principe de fermeture commune (CCP)

- "Les classes impactées par les même changements doivent être placées dans un même package"
- Le principe ouverture / fermeture ne peut pas être appliqué totalement : il restera toujours des endroits impactés par des modifications
- Le CCP propose de regrouper dans un même package l'ensemble des classes impactées par un changement (ce qui manque parfois de logique)

Gestion de la stabilité de l'application

- Principe des dépendances acycliques
  - ADP : Acyclic Dependencies Principle
- Principe de relation dépendance / stabilité
  - SDP : Stable Dependencies Principle
- Principe de stabilité des abstractions
  - SAP : Stable Abstractions Principle

Principe des dépendances acycliques (ADP)

- "Les dépendance entre package doivent former un graphe acyclique"
- Il est possible de trouver une boucle dans les dépendances entre package
- Ceci pourrait avoir un effet désastreux lors du changement d'un package qui impacterait les packages qui en dépendent (besoins de modifications) et ainsi de suite jusqu'au package qui a été changé en premier lieu
- Pour éviter les problèmes, il faut faire de l'inversion de dépendances (DIP)

Principe de relation dépendance / stabilité (SDP)

- "Un package doit dépendre uniquement de package plus stables que lui"
- La stabilité est une notion probabiliste !
- Mais : plus un module dépend d'autres modules et plus il est susceptible d'être impacté par une modification d'un de ces modules
- Stabilité : un module ne dépend d'aucun autre module
  - Un exemple en Java ?

Principe de stabilité des abstractions (SAP)

- "Les packages les plus stables doivent être les plus abstraits. Les packages instables doivent être concrets. Le degré d'abstraction d'un package doit correspondre à son degré de stabilité"
- Selon OCP, les interfaces stabilisent certaines parties de l'application et génèrent des packages abstraits et
- Selon le DIP, les packages les plus stables contiennent la logique fonctionnelle
- La somme de ceux deux principes entraîne les SAP

Et après les principes avancés…

- Ces principes sont bons !
  - Certains outils aident même à vérifier leur application en plus du calcul de métriques sur le code (dont je vous fait l'économie !)
  - Il est bon de connaître ces principes pour avoir une idée générale de comment il faut faire
- Il existe également de bonnes et de mauvaises manières de créer du code
  - Les "bonnes manières de faire" sont regroupées dans les Design Patterns

Design patterns

- Les Design Patterns représentent une avancée très profitable dans les phases de conceptions de systèmes orientées objet
- “Design Patterns” se traduit le mieux par “modèles de conception réutilisables”
- Ils partent du principe que les mécanismes standards sont au cœur de l’architecture

Design patterns (2)

- Les Design Patterns viennent du monde de l’architecture (bâtiments)
- Ils ont été mis en avant en 1977 par Christopher Alexander
- Ils ont ensuite été repris en informatique
  - Design Patterns – Elements of Reusable Object Oriented Software, 1994
  - Gamma, Helm, Johnson et Vlissides : le GoF, Gang of Four

Design patterns (3)

- Les Design Patterns « décrivent des problèmes qui arrivent encore et encore dans un environnement et décrivent alors une solution qui peut être utilisée un grand nombre de fois sans que cette solution ne soit jamais appliquée exactement de la même manière » (traduit de Christopher Alexander)

Design patterns (4)

- Les Design Patterns sont constitués par :
  - Un nom
  - Une intention et une motivation
  - Des indications d'utilisation
  - Une structure
  - Des constituants
  - Des collaborations
  - Des conséquences

Design patterns (5)

- L'intention
  - Que fait le pattern, quelle est sa raison d'être et son but, quel problème tente-t-il de résoudre ?
- Motivation
  - Exemple d'un scénario qui permet de comprendre le pattern. La motivation est plus compréhensible que le modèle abstrait qui accompagne le Design Pattern

Design patterns (6)

- Indications d'utilisation
  - Quels cas justifient l'utilisation du pattern ? Quelles situations peu satisfaisantes peuvent tirer avantage à l'utiliser ?
- Structure
  - Représentation graphique du pattern ; bien souvent en UML lorsqu'il s'agit d'informatique même si les premiers patterns étaient décrits avec une notation OMT

Design patterns (7)

- Constituants
  - Description des différents objets ou classes qui interviennent dans le pattern
- Collaborations
  - Interaction entre les constituants, responsabilités des constituants

Design patterns (8)

- Conséquences
  - Quelles sont les conséquences de l'application du pattern ? Comment les objectifs sont-ils atteints ? Quels sont les compromis nécessaires et leurs impacts ?

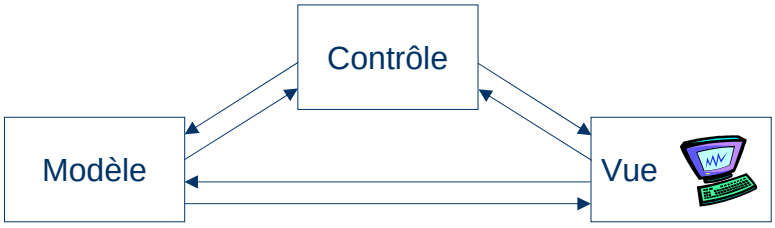
Design patterns (9)

- Le plus ancien des patterns informatiques a en fait été créé pour faciliter la définition des interfaces graphiques dans un langage orienté objet en 1978
  - Smalltalk (Xerox PARC, première implémentation des environnements fenêtrés)
  - Besoin d'un pattern simple pour identifier les responsabilités d'affichage, de gestion des interactions et de coordination

Design patterns (10)

- MVC, Model-View-Controller
  - Le modèle est l'objet de l'application (abstrait)
  - La vue est la présentation à l'écran
  - Le contrôleur gère l'interaction de l'utilisateur
- Ceci permet de découpler les différentes catégories d'objets intervenant dans un dialogue homme-machine
- Facilite l'usage de plusieurs vues et de plusieurs interactions sur un même objet

Design patterns (11)



*** Stocke l'état de l'application**
*** Fonctionnel de l'application**
*** Conserve la liste des vues et des contrôles**
*** Notifie les changements de données au contrôleur**

*** Transforme les actions de l'utilisateur en évènements**
*** Traduit les évènements en requêtes auprès du modèle**

*** Crée les contrôleurs**
*** Met à jour l'écran en fonction de l'état du modèle**
*** Met à jour l'écran en fonction des actions de l'utilisateur**

Design patterns (12)

- Le MVC, inventé pour Smalltalk chez Xerox, a été repris plus ou moins clairement
  - Par les stations graphiques Unix puis par
  - Macintosh puis par
  - Microsoft Windows (Apple ayant même fait un procès en paternité à Microsoft) puis par
  - …
- Le MVC est peu ou prout sur tous les écrans mais attention, le MFC Document/Frame/View n'est pas un MVC

Design patterns (13)

- Le MVC est parfois critiqué car il est parfois un peu lourd
- Il en existe bien sûr des variantes telle que PAC, Presentation-Abstraction-Control
  - Remplacer
    - Presentation par View
    - Abstraction par Model

Design patterns (14)

- Les Design Patterns du GoF sont moins spécialisés que le MVC
- Un Design Pattern décrit un problème arrivant régulièrement, décrit ensuite le cœur de la solution à ce problème, de manière à pouvoir utiliser le pattern dans de nombreux cas
- Un Design Pattern représente une connaissance générale de design

Design patterns (15)

- Répartition des 23 Design Patterns du GoF

		Domaine		
		Création	**Structure**	**Comportement**
Applicabilité	**Classes**	Fabrication	Adaptateur	Interpréteur Patron de méthode
	Objets	Fabrique abstraite Monteur Prototype Singleton	Adaptateur Pont Composite Décorateur Façade Poids Mouche Procuration	Chaîne de responsabilité Commande Itérateur Médiateur Mémento Observateur État Stratégie Visiteur

Design patterns (16)

- Les patterns de création
  - Donnent une abstraction au processus d'instanciation
  - Rendent un système indépendant de la manière dont les objets sont créés, composés et représentés

Design patterns (17)

- Les patterns de structure :
  - Réduisent le couplage dans un système, simplifiant sa maintenance
  - Introduisent des classes abstraites favorisant les extensions futures
  - Encapsulent les structures complexes

Design patterns (18)

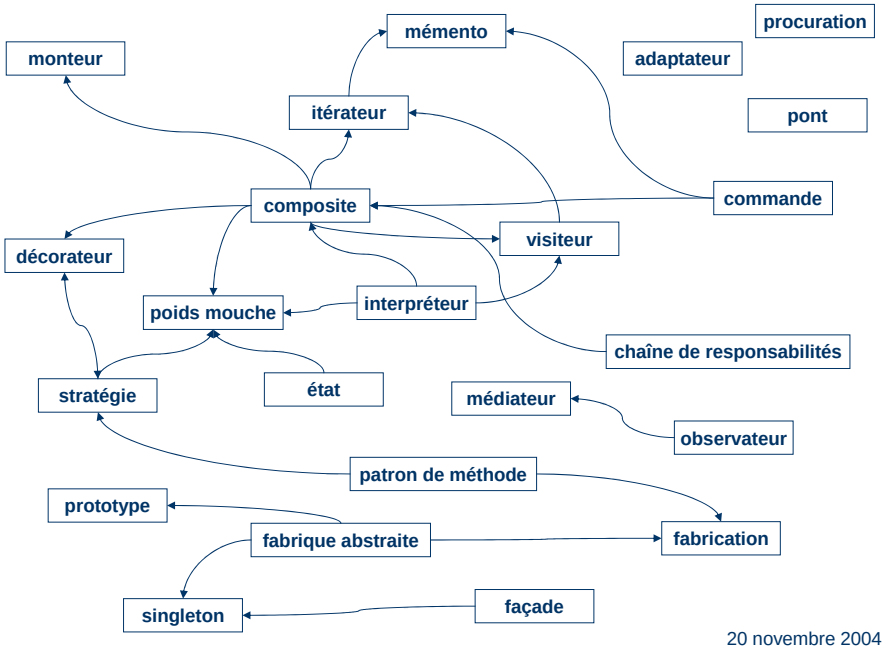
- Les patterns de comportement
  - Permettent de savoir qui fait quoi
    - Dans le domaine algorithmique
    - Au niveau des responsabilités des classes
  - Modélisent des flux de contrôle parfois très complexes et permettent ainsi d'éviter certains soucis en ce concentrant sur la nature de la communication

Design patterns (19)

- Classement top ten des patterns selon Gamma
  - État, stratégie
  - Composite
  - Fabrication
  - Patron de méthode
  - Adaptateur
  - Observateur
  - Médiateur
  - Interface / implémentation
  - Commande
  - Chaîne de responsabilité

Design patterns (20)

- Les 23 design patterns n'existent pas indépendamment des autres :
  - Certains en utilisent d'autres de manière explicite (certains vont de pair)
  - Certains en utilisent d'autres de manière fréquente sans que cela soit une obligation
- La page qui suit présente les relations entre design patterns



Design patterns (24)

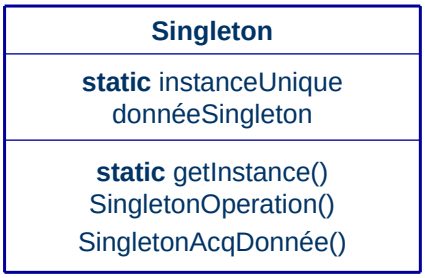
- Entendu de la bouche de Gamma dans une conférence (TOOLS Europe 1999) :
  - « J'ai croisé un développeur qui était triste de n'avoir réussi à utiliser que 22 design patterns dans une de ses applications. Il se désolait de ne pas avoir trouver une place pour le 23ème. Je crois qu'il n'a pas tout compris… »

Design Pattern : Singleton

- Le singleton est un pattern de création
- Le respect de ce pattern permet de disposer dans un système d'une classe qui possèdera une instance unique et qui pourra ainsi servir de référence à travers toute l'application pour stocker certaines informations

Design Pattern : Singleton (2)

Principe de base :

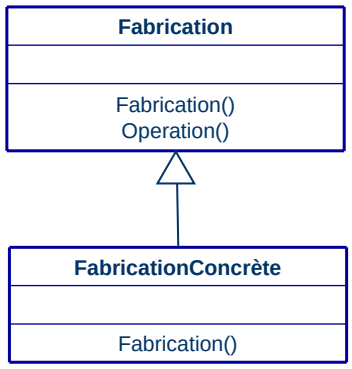


Design Pattern : Fabrication

- La fabrication est un pattern de création
  - Factory en anglais
- La fabrication permet de déléguer la fabrication d'instances d'objets à des sous-classes
  - Peut être utilisé pour fabriquer différent types de sous-classe en fonction de la valeur des paramètres placés dans ces sous-classes
- Il s'agit d'un constructeur virtuel
  - Le constructeur de la sous-classe peut faire appel au constructeur de la classe mère

Design Pattern : Fabrication (2)

Principe de base :

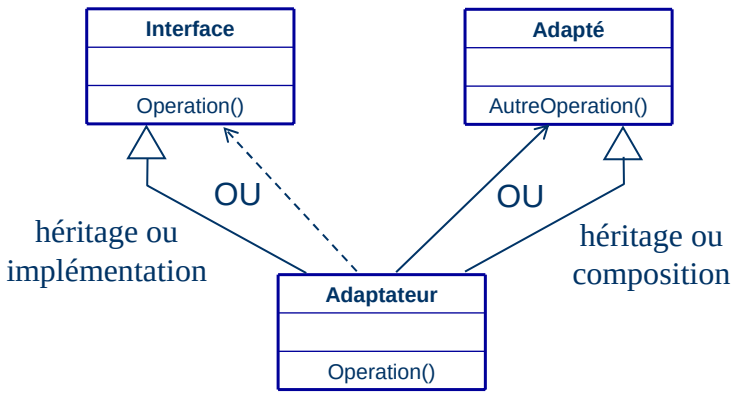


Design Pattern : Adaptateur

- L'adaptateur est un pattern de structure
  - Adapter en anglais
- L'adaptateur permet de fournir l'interface désirée vis-à-vis d'un objet qui ne la possédait pas. Ce dernier objet devient alors utilisable dans un autre contexte sans pour autant avoir été lui-même modifié
- Rappelez-vous, il y a des adaptateurs dans la Standard Template Library !
- ISP...

Design Pattern : Adaptateur (2)

Principe de base :



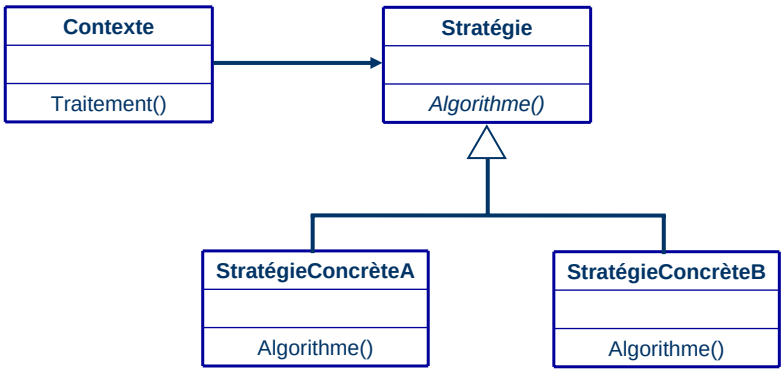
Design Pattern : Composite

- Un des Design Patterns les plus utilisés est le composite
- Il s'agit d'une sous version de l'interprète
- Un composite permet normalement de représenter une structure mais avec l'orienté objet et l'encapsulation, les structures sont capables de réaliser des traitements
  - Les Design Patterns prêtent parfois à confusion

Design Pattern : Stratégie

- Définit une famille d'algorithmes, les encapsule et les rend interchangeables
- S'utilise lorsque plusieurs classes apparentées ne diffèrent que par leur comportement
- Par exemple, un tri peut être réalisé de plusieurs manières pour aboutir au même résultat, plus ou moins vite...
  - L'utilisation du DP stratégie permet d'implanter différents algorithmes de manière extensible

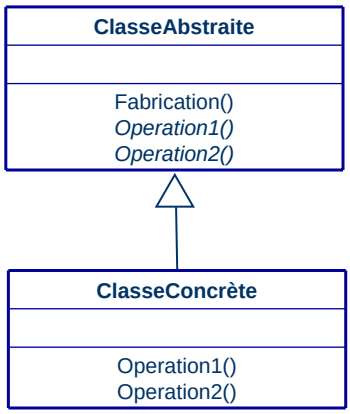
Design Pattern : Stratégie (2)



Design Pattern : Patron de méthode

- Définition d'un squelette d'un algorithme dans une méthode, en déléguant l'implémentation de ses parties d'algorithmes aux sous-classes
  - Cas restrictif de la stratégie : on expose certaines parties de l'algorithme
- Exemple : pour le tri, l'algorithme est implémenté mais il faut fournir le comparateur entre deux objets

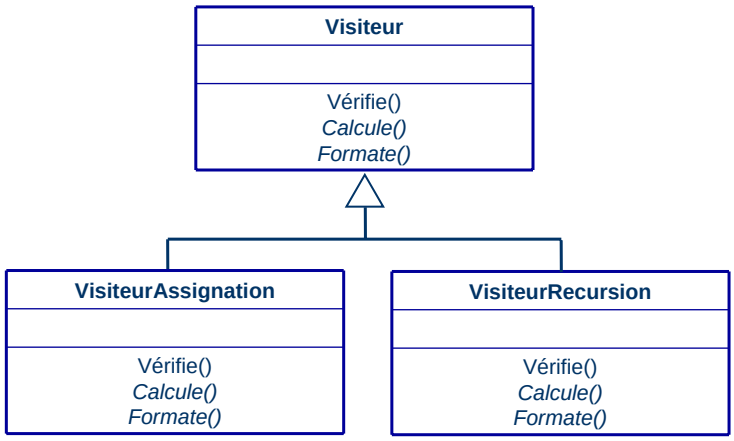
Design Pattern : Patron de méthode (2)



Design Pattern : Visiteur

- Le visiteur implémente une opération applicable à une structure d'objets
- Il permet de définir des opérations sans qu'il soit nécessaire de modifier la classe des éléments
- Le visiteur permet d'une certaine manière de se libérer du principe de l'encapsulation des attributs et des méthodes (données et traitements associés)

Design Pattern : Visiteur (2)

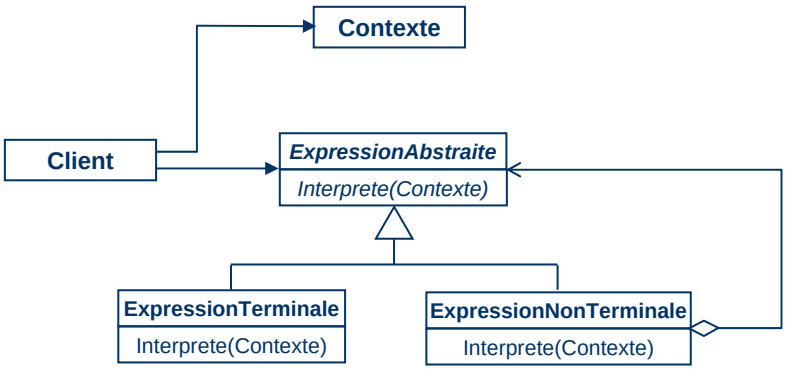


Design Pattern : Interpréteur

- L'interpréteur est un DP de comportement
  - Interpreter en anglais
- Il permet de traiter des structures et d'en obtenir une représentation objet (ou plutôt hiérarchique)
- En plus d'une structure, il permet d'implémenter des traitements sur les éléments de cette structure

Design Pattern : Interpréteur (2)

Principe de base :

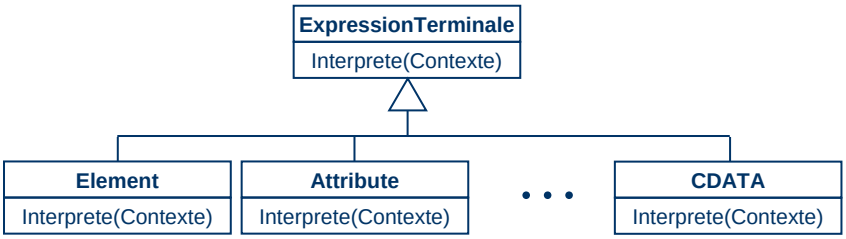


Design Pattern : Interpréteur (3)

- Comme pour tout DP, il s'agit de la vue la plus abstraite de la solution
  - Cette solution repose sur un traitement récursif d'une arborescence
- La fonction Interprete() permet d'ajouter des traitements tant au niveau du corps de l'arbre qu'au niveau des feuilles
- La classe ExpressionTerminale est généralement la racine d'une hiérarchie
  - ExpressioNonTerminale le peut aussi

Design Pattern : Interpréteur (4)

Exemple de hiérarchie avec « ExpressionTerminale » :



Cette hiérarchie, très incomplète, est à la base du DOM, Document Object Model, qui permet de traiter les documents XML

Les principes avancés dans les design patterns

- OCP et strategy : le code initial est ouvert/fermé à l'ajout de nouveaux algorithmes en ayant recours à une classe d'interface commune à tous les algorithmes. Cela permet d'ajouter de nouveaux algorithmes sans changer le code client
- OCP et abstract factory : le code client est ouvert/fermé à la création de nouveaux objets

Les principes avancés dans les design patterns (2)

- OCP et template method : on ferme la structure générale d'une méthode en ouvrant certaines sous-parties à de nouvelles implémentations. Cela permet de changer les sous-parties sans modifier la structure de base
- OCP et visitor : la structure parcourue est ouverte/fermée à l'ajout de nouveau algorithmes de traitement

Les principes avancés dans les design patterns (3)

Et enfin :

- adapter : principe de séparation des interfaces pour représenter des services

Patterns d'architecture

- Il existe aussi des patterns pour représenter les différents types d'architecture informatique qu'il est possible de rencontrer
- Ces patterns tentent de donner une définition des différents constituants de divers architectures qu'il est possible d'observer dans le domaine informatique
- Une liste en anglais :

Patterns d'architecture (2)

- Distributed
- Event-driven
- Frame-based
- Batch
- Pipes and filters
- Repository-centric
- Blackboard
- Interpreter
- Rule-based
  - Layered
  - MVC
  - IR-centric
  - Subsumption
  - Disposable

Ce que les DP ne sont pas

- Un pattern n'est pas une solution à un problème dans un contexte
  - Un pattern est une solution abstraite, le contexte peut donc être très simplifié et il faut donc adapter le pattern
- Les Design Pattern ne sont pas que du jargon ou des règles de programmation
  - On les apprécient d'autant plus qu'on les connaît

Ce que les DP ne sont pas (2)

- Tous les pattern ne sont pas égaux
  - Certains seront très largement applicables tandis que d'autres, bien qu'abstraits, ont peu de champs d'application
- Les patterns n'ont pas besoin de méthodologies et d'outils
  - Les outils et méthodologies sont de bonnes choses mais on peut faire sans pour utiliser les patterns !

Ce que les DP ne sont pas (3)

- Les patterns ne garantissent pas la réutilisabilité, la productivité, etc.
  - Les patterns ne garantissent rien, ils permettent juste de simplifier un peu le travail. Ils ne sont que ce que vous en faites
- Les patterns ne génèrent pas l'architecture
  - Les patterns ne sont que des éléments d'une architecture plus ou moins bien faite

Ce que les DP ne sont pas (4)

- Les patterns ne sont pas spécifiques au design ou au codage (orienté objet)
  - Certains patterns s'appliquent à de plus hauts niveaux
- Les patterns ne sont pas une charge
  - Ils permettent d'avoir un langage commun permettant de simplifier les échanges de connaissances et d'idées

Ce que les DP ne sont pas (5)

- Les patterns ne sont pas élitistes
  - Les connaître ne permet pas de faire partie d'une communauté de gourous
- Les patterns ne sont pas des balles en argent (silver bullet) résolvant tous les problèmes
  - Et l'usage du terme dans un discours commercial ne garantit rien non plus…

Des Patterns pour tout

- Il existe, finalement, des patterns pour tout
  - Certains s'intéressent au design des applications
    - les applications de facturation se ressemblent toutes pour la bonne et simple raison que tous les comptables ont étudiés les mêmes livres !
  - D'autres sont spécialisés dans les processus
- Pendant que les Design Patterns étaient à la mode, chacun voulait faire **son** design pattern : on en trouve à foison…

Des Patterns pour tout (2)

- Il existe même des Design Pattern de ce qu'il ne faut pas faire, en informatique mais même de manière plus générale
- Il ne s'agit pas de Design Patterns mais d'**AntiPatterns**
  - Finalement, il est possible de les utiliser lorsque l'on fait de la refonte (réécriture) d'applications
  - Il suffit de les trouver et de les supprimer !

AntiPatterns

- Les Design Patterns synthétisent les bonnes manières de faire, les AntiPatterns, les mauvaises…
- Il existe des AntiPatterns portant sur
  - Le développement de logiciel
    - La manière d'écrire le code a ses mauvaises habitudes
  - L'architecture logicielle
    - Un logiciel s'inscrit dans un ensemble, le système d'information, d'où il récupère des informations
  - La gestion de projet
    - Des tas de manière de mal faire

AntiPatterns (2)

Dans ce qui suit, cherchez ce que vous connaissez !

AntiPatterns : développement

- Blob
  - Lors d'un mauvais design orienté objet, un seul et même objet se retrouve avec une énorme quantité de responsabilités tandis que la grande majorité des autres objets ne font que stocker des données
  - Les objets sont vu comme des structures et l'un d'eux sera le programme
  - Facile à faire en C++ en reprenant un programme C !

AntiPatterns : développement (2)

- Obsolescence continue
  - La technologie informatique change si souvent que les développeurs sont tentés de passer à une nouvelle technologie, beaucoup plus sexy, avant de terminer un projet en cours
  - Avec le temps d'adaptation à la nouvelle technologie et les adaptations nécessaires de ce qui a déjà été réalisé, le développement ne finit jamais

AntiPatterns : développement (3)

- Flot de lave
  - Les parties d'un programme devenues inutiles, ou dont les raisons de développement ont été oubliées, sont conservées dans le code même si l'application évolue par ailleurs
  - Analogie avec la lave : les blocs de lave refroidis dans le flot de lave en fusion
  - « si c'est là et que ça marche, on le laisse là »

AntiPatterns : développement (4)

- Décomposition fonctionnelle
  - Cet antipattern est très pratiqué par les développeurs qui passent à l'orienté objet après avoir longtemps pratiqué les langages de programmation procéduraux
  - Les différentes classes d'une applications représentent autant de parties d'une décomposition fonctionnelle traditionnelle : une succession de tâches plutôt que des objets

AntiPatterns : développement (5)

- Ancre de marine
  - Une ancre de marine est un ensemble d'objets permettant de faire le lien avec une librairie logicielle ou un matériel périphérique qui ne sert à rien
  - L'ancre de marine justifie l'achat antérieur du bien inutile, qui a également tendance à être hors de prix

AntiPatterns : développement (6)

- Marteau en or
  - C'est une technologie ou un concept connu appliqué de manière obsessionnelle dans toutes les situations possibles
  - Le marteau en or indique un manque de connaissances tant dans le design ou la programmation

AntiPatterns : développement (7)

- Code spaghetti
  - Résultat d'une analyse et d'un design bâclé pour une quelconque raison, le code spaghetti se traduit bien souvent par un flot de programme très difficile à suivre
  - Mais après tout, on programmait déjà comme ça en BASIC et en Fortran…

AntiPatterns : développement (8)

- Promenade dans un champ de mine
  - Le renouvellement fréquent de technologies informatiques utilisées pour le développement peut se révéler être une promenade dans un champ de mine : l'utilisation de logiciels tout juste disponibles sur le marché oblige bien souvent à identifier des bugs dans ceux-ci
  - Les logiciels acquis auraient bien sûr dus en être exempts mais ils ne sont pas garantis contre les bugs !

AntiPatterns : développement (9)

- Copier/coller
  - Réutilisation du code par usage abusif des Ctrl-C et Ctrl-V
  - Dans un design orienté objet, cela se traduit par une arborescence très pauvre de classes énormes car possédant un très grande nombre de fonctions et ne faisant que très peu appel à la réutilisation de code par héritage

AntiPatterns : architecture

- Système de canalisations (stovepipe)
  - Certains systèmes informatiques sont réalisés de manière indépendante les uns des autres
  - La mise en place d'une communication entre ces systèmes impose le développement de canaux de communication ad hoc et en grand nombre : pour N systèmes, $O(N^2)$ canaux
  - De plus, de tels systèmes sont très difficiles à faire évoluer

AntiPatterns : architecture (2)

- Dépendance par rapport à un vendeur
  - Arrive lorsque de nombreux logiciels sont acquis auprès d'un même fournisseur. Celui-ci peut alors forcer la main de ses clients pour acquérir de nouvelles versions à grande échelle car celles-ci résolvent certains bugs des anciennes et sont, bien sûr, partiellement incompatibles avec les anciennes
  - Malheureusement, les fonctionnalités annoncées ne sont pas toujours là…

AntiPatterns : architecture (3)

- Viande chaude
  - Dans certains projets informatiques, la seule justification de la présence de certains développeurs est le respect de la planification et du budget prévisionnel
  - Ils n'ont aucun travail effectif et finissent par distraire ou ralentir les autres développeurs par des réalisations périphériques inutiles

AntiPatterns : architecture (4)

- Design par comités et groupe de travail
  - Un mauvais symptôme des organismes de normalisation, et d'autres entreprises, où l'architecture est réalisée partiellement par de petits groupes de travail presque indépendants
  - Les résultats manquent le plus souvent de cohérence et de clarté

AntiPatterns : architecture (5)

- Réinvention de la roue
  - La réécriture d'un système ancien impose très généralement de réinventer la roue par manque de documentation
  - L'étude du code source peut aider à extraire les fonctionnalités et règles de gestion mais la roue reste cependant à réinventer

AntiPatterns : architecture (6)

- Couteau suisse
  - Mise en place, en divers endroits de l'architecture, d'interfaces permettant un très grand nombre d'opérations
  - L'architecte a tenté de fournir la totalité des manières possibles d'utiliser cette interface, créant une charge de développement et de maintenance significative bien qu'inutile

AntiPatterns : gestion de projets

- Il existe enfin des AntiPatterns pour la gestion de projets
- Ils étudient les cause de blocage d'une analyse, les problèmes de gestion de planning, les problèmes de violence intellectuelle dans les choix et quelques erreurs de management

Lectures…

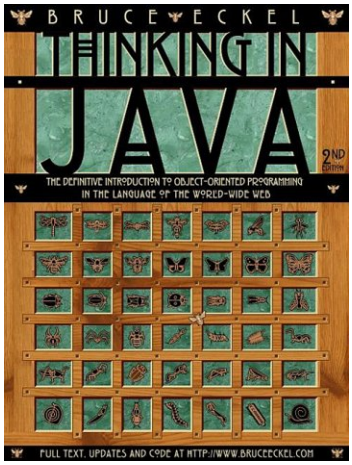
Design Patterns

Bruce Eckel

Prentice Hall, 2000

ISBN: 0130273635

gratuit sur
http://www.mindview.net/



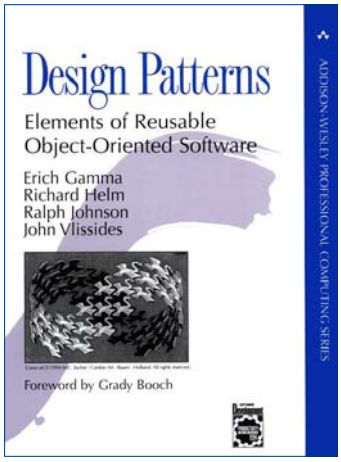
Lectures…

Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Addison-Wesley, 1994

ISBN 0-201-63361-2



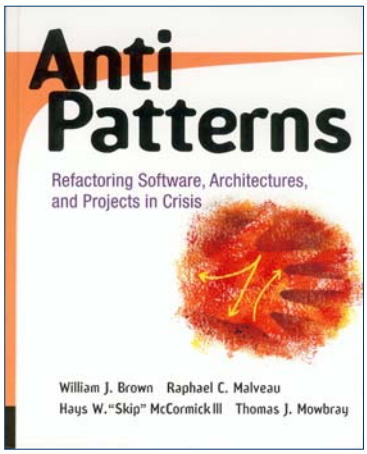
Lectures…

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis

William J. Brown, Raphael C. Malveau, William H. Brown, Hays W., III McCormick, Thomas J. Mowbray

Wiley Computer Publishing, 1998

ISBN 0-471-19713-0

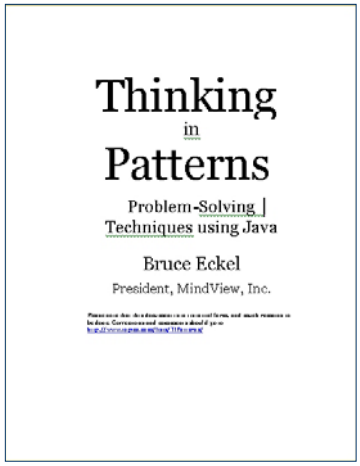


Lectures…

Thinking in Patterns with Java

Bruce Eckel

gratuit sur http://www.mindview.net/



Questions / Remarques