

Réutilisation dans les SI : patrons et frameworks

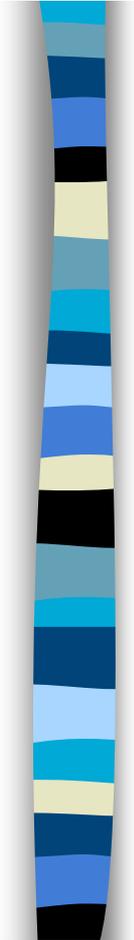
M1 MIAGE - SIMA - 2005-2006

Yannick Prié

UFR Informatique - Université Claude Bernard Lyon 1

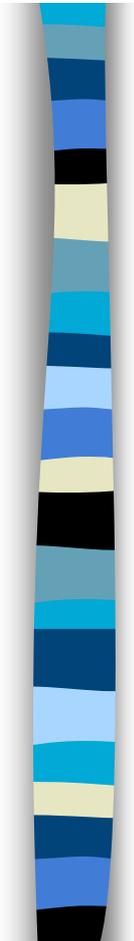
Introduction : réutilisation (5)

- Constante de la conception d'outils en général
 - Ex. : je dois fabriquer quelque chose pour s'asseoir. Que puis-je réutiliser ?
- En informatique
 - réutilisation de code
 - sous la forme de composants
 - à acheter / fabriquer
 - sous la forme de framework
 - à spécialiser
 - réutilisation de principes de conception
 - à connaître
- Dès que des principes se révèlent pertinents
 - abstraction / réutilisation



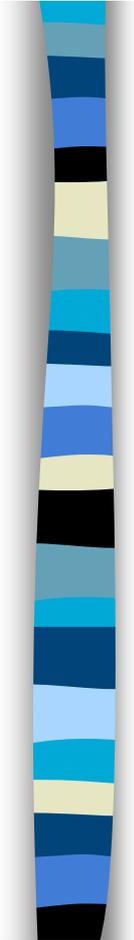
Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks



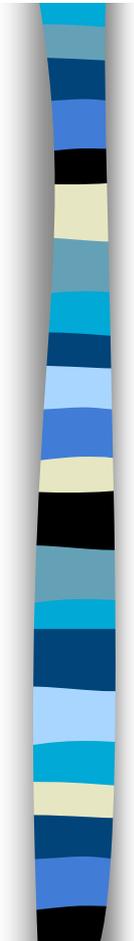
Généralités sur les patterns

- Pattern
 - solution classique à un problème de conception classique dans un contexte donné
- Pattern de conception
 - structure et comportement d'une société de classes
 - description nommée d'un problème et d'une solution
 - avec conseils d'application



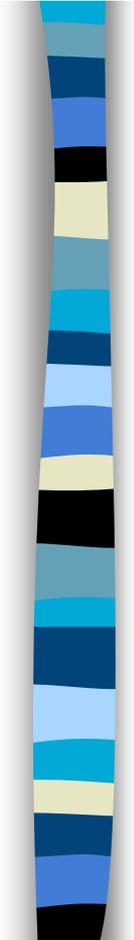
Description d'un pattern

- Nom du pattern
 - un ou deux mots
 - le nom d'un pattern permet d'en parler, il n'est pas significatif en soi
- Problème
 - quand appliquer le pattern ?
 - explication du problème et de son contexte
- Solution
 - élément de conception, relation, responsabilités, collaboration
 - description abstraite
- Discussion
 - conseils sur la façon de l'appliquer
 - présentation des avantages, inconvénients, conseils d'implémentation, variantes, ...



Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks



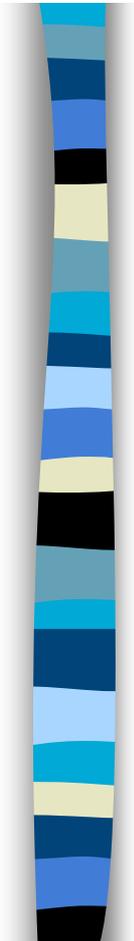
Conception pilotée par les responsabilités

■ Métaphore

- communauté d'objets responsables qui collaborent (*cf.* humains)
- penser l'organisation des composants (logiciels ou autres) en termes de responsabilités par rapport à des rôles, au sein de collaborations

■ Responsabilité

- abstraction de comportement (contrat, obligation par rapport à un rôle)
 - une responsabilité n'est pas une méthode
 - les méthodes s'acquittent des responsabilités



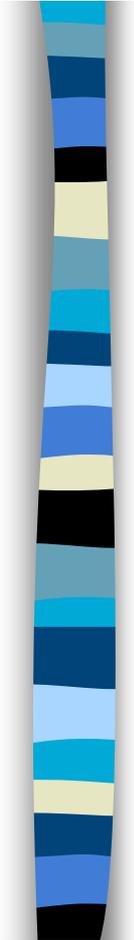
Deux catégories de responsabilités

■ Faire

- faire quelque chose soi-même (ex. créer un autre objet, effectuer un calcul)
- déclencher une action d'un autre objet
- contrôler et coordonner les activités d'autres objets

■ Savoir

- connaître des données privées encapsulées
- connaître les objets connexes
- connaître des éléments qu'il peut calculer ou dériver



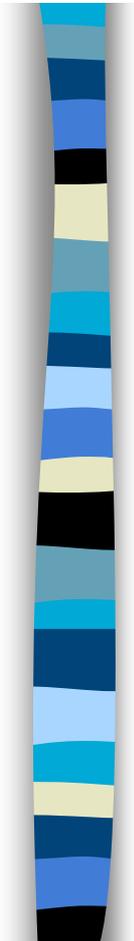
Exemples (bibliothèque)

■ Faire

- *Adhérent* est responsable de la vérification du retard sur les livres prêtés

■ Savoir

- *Livre* est responsable de la connaissance de son numéro *ISBN*
- *Abonné* est responsable de savoir si il reste la possibilité d'emprunter des livres



GRASP

■ *General Responsibility Assignment Software Patterns*

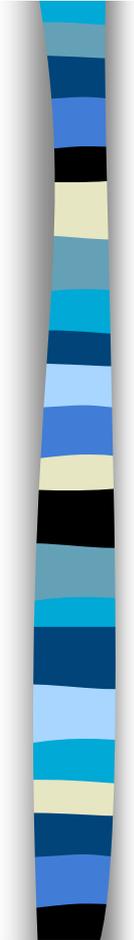
■ Ensemble de patterns généraux d'affectation de responsabilité pour aider à la conception orientée-objet

- raisonner objet de façon méthodique, rationnelle, explicable

■ Utile pour l'analyse et la conception

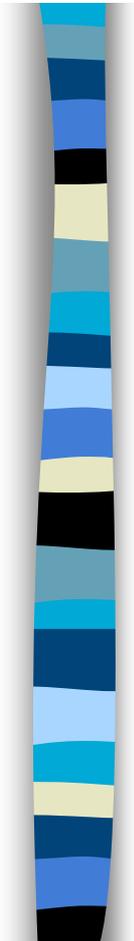
- réalisation d'interactions avec des objets

■ Référence : Larman 2004



9 patterns GRASP

- Créateur
- Expert en information
- Faible couplage
- Contrôleur
- Forte cohésion
- Polymorphisme
- Indirection
- Fabrication pure
- Protection des variations



Un principe général

- Toujours chercher à *réduire le décalage des représentations* entre
 - la façon de penser le domaine (humaine)
 - « un échiquier a des cases »
 - les objets logiciels correspondants
 - un objet *Echiquier* contient des objets *Case*
 - vs. un objet *Echiquier* contient 4 objets *16Cases*
 - vs. un objet TYR43 contient des EE25.

Expert (GRASP)

■ Problème

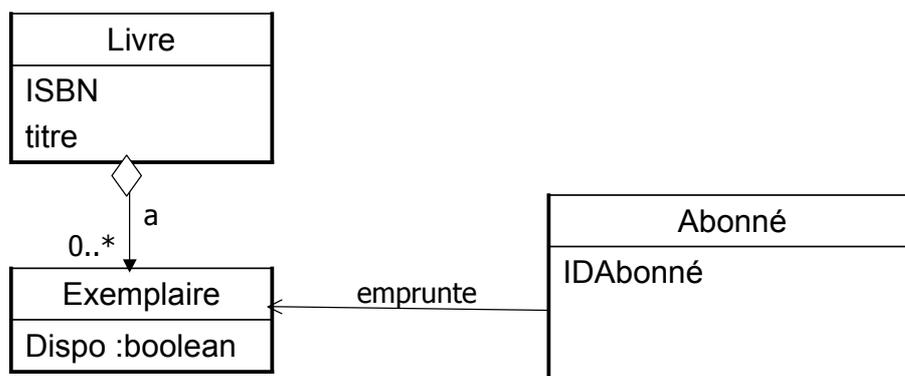
- Quel est le principe général d'affectation des responsabilités aux objets ?

■ Solution

- Affecter la responsabilité à l'*expert* en information
 - la classe qui possède les informations nécessaires pour s'acquitter de la responsabilité

Expert : exemple

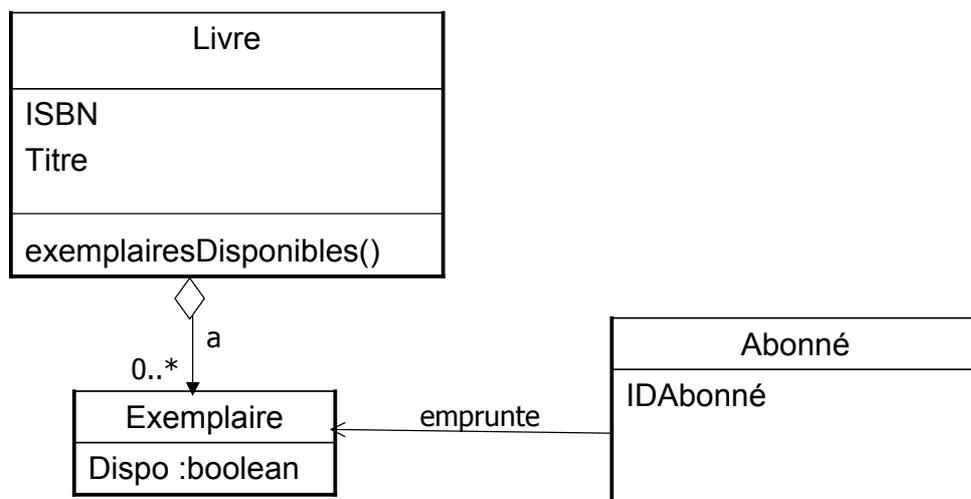
- Bibliothèque : qui doit avoir la responsabilité de connaître le nombre d'exemplaires disponibles ?



Expert : exemple (suite)

- Commencer avec la question
 - De quelle information a-t-on besoin pour déterminer le nombre d'exemplaires disponibles ?
 - *Disponibilité de toutes les instances d'exemplaires*
- Puis
 - Qui en est responsable ?
 - *Livre est l'Expert pour cette information*

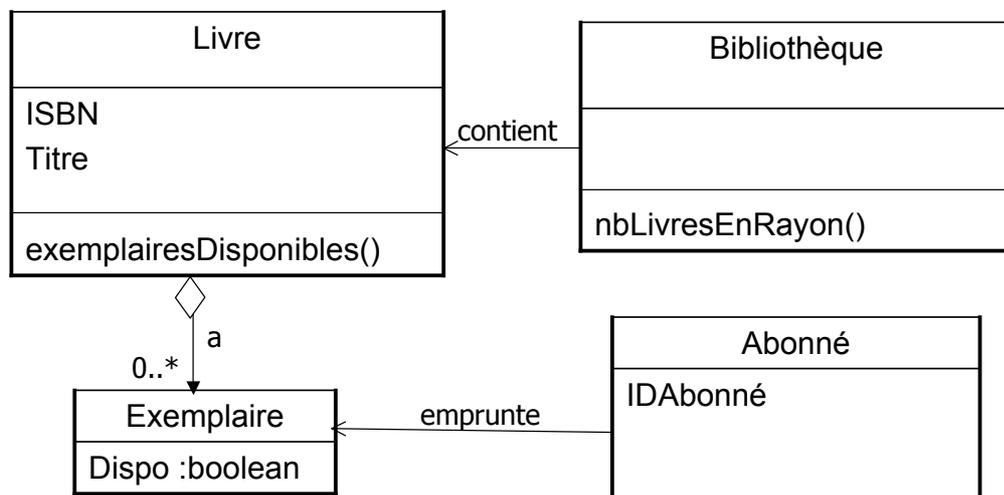
Expert : exemple (suite)

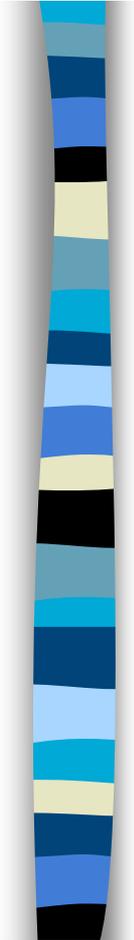


Expert : discussion

- Le plus utilisé de tous les patterns d'attribution de responsabilité
- Un principe de base en OO
- L'accomplissement d'une responsabilité nécessite souvent que l'information nécessaire soit répartie entre différents objets
 - des experts qui collaborent dans la tâche

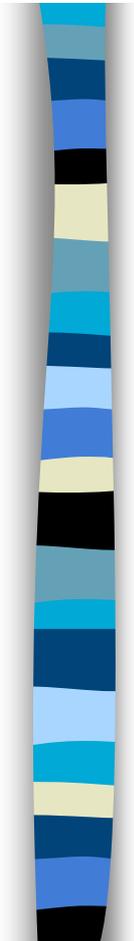
Expert : exemple (suite)





Expert : avantages

- **Maintien de l'encapsulation**
 - les objets utilisent leur propre information pour mener à bien leurs tâches
 - supporte le *Couplage faible* (voir plus loin), ce qui conduit à des systèmes plus robustes et plus maintenables
- **Le comportement est distribué à travers différentes classes qui ont l'information nécessaire**
 - encourage des définition de classes plus légères, plus cohésives, plus facile à comprendre et à maintenir
 - supporte *Forte cohésion* (voir plus loin)
- **Autres noms (AKA)**
 - Mettre les responsabilités avec les données
 - Qui sait, fait
 - Faire soi-même

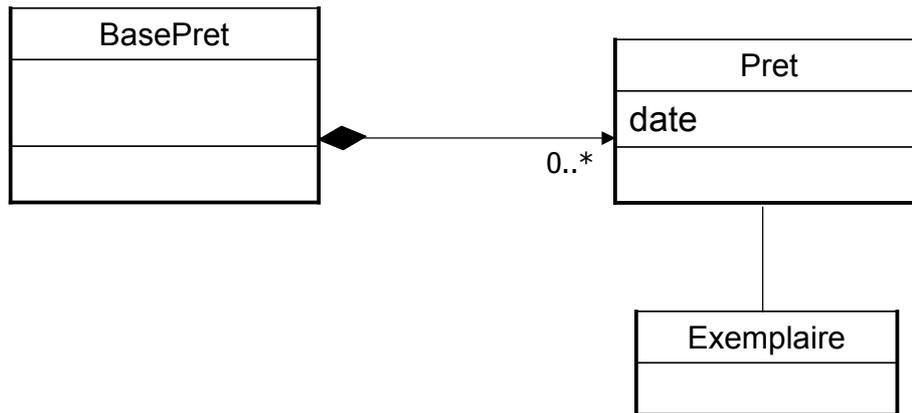


Créateur (GRASP)

- **Problème**
 - Qui doit avoir la responsabilité de créer une nouvelle instance d'une classe donnée ?
- **Solution**
 - Affecter à la classe B la responsabilité de créer une instance de la classe A si une - ou plusieurs - des conditions est vraie :
 - B contient ou agrège des objets A
 - B enregistre des objets A
 - B utilise étroitement des objets A
 - B a les données d'initialisation qui seront transmises aux objets A lors de leur création
 - B est un *Expert* en ce qui concerne la création de A

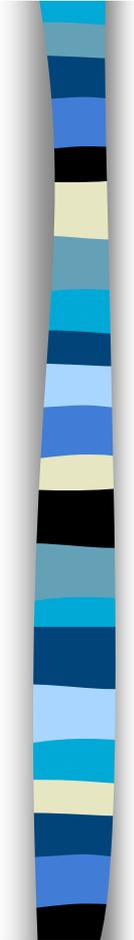
Créateur : exemple

- Bibliothèque : qui doit être responsable de la création de *Pret* ?
- *BasePret* contient des *Pret* : elle doit les créer.



Créateur : discussion

- Guide pour attribuer une responsabilité pour la création d'objet
 - une tâche très commune en OO
- Finalité : trouver un créateur pour qui il est nécessaire d'être connecté aux objets créés
 - favorise le *Faible couplage*
 - Moins de dépendances de maintenance, plus d'opportunités de réutilisation
- Pattern liés
 - *Faible couplage*
 - *Composite*
 - *Fabricant*



Faible couplage (GRASP)

■ Problème

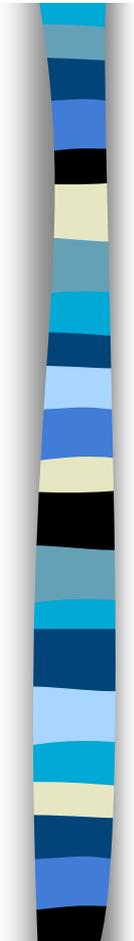
- Comment minimiser les dépendances, réduire l'impact des changements, et augmenter la réutilisation ?

■ Solution

- Affecter une responsabilité de sorte que le couplage reste faible. Appliquer ce principe pour évaluer les solutions possibles.

■ Couplage

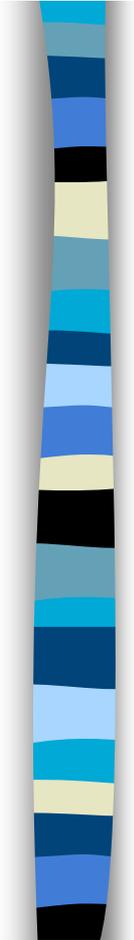
- Mesure du degré auquel un élément est lié à un autre



Couplage

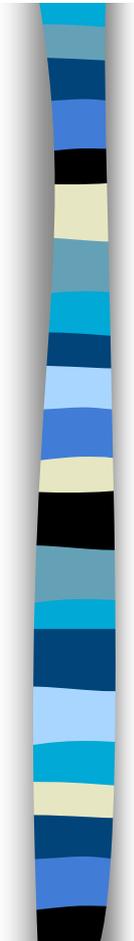
■ Exemples classiques de couplages de *TypeX* vers *TypeY* dans un langage OO

- *TypeX* a un attribut qui réfère à *TypeY*
- *TypeX* a une méthode qui référence *TypeY*
- *TypeX* est une sous-classe directe ou indirecte de *TypeY*
- *TypeY* est une interface et *TypeX* l'implémente



Couplage faible (suite)

- Avoir un couplage faible signifie une faible dépendance aux autres classes
- Problèmes du couplage fort
 - Un changement dans une classe force à changer toutes ou la plupart des classes liées
 - Les classes prises isolément sont difficiles à comprendre
 - Réutilisation difficile : l'emploi d'une classe nécessite celui des classes dont elle dépend
- Bénéfices du couplage faible
 - Exactement l'inverse



Couplage faible : exemple

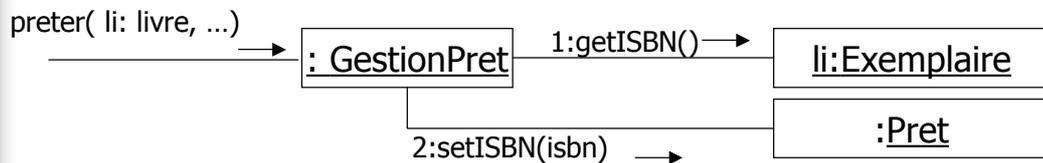
- Pour l'application de bibliothèque, il faut mettre l'ISBN d'un Exemple dans le Prêt.
- Quelle classe en sera responsable ?

IHMPrêt

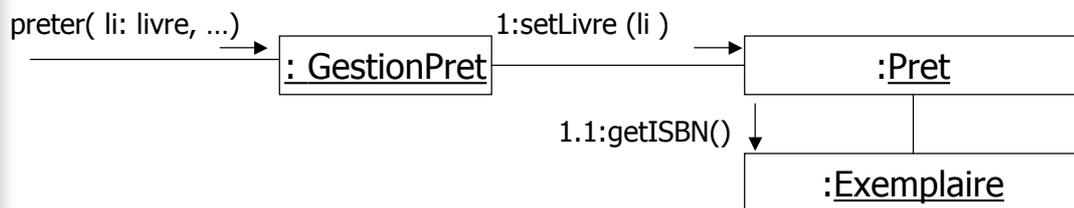
Prêt

Exemple

Couplage faible : exemple (suite)

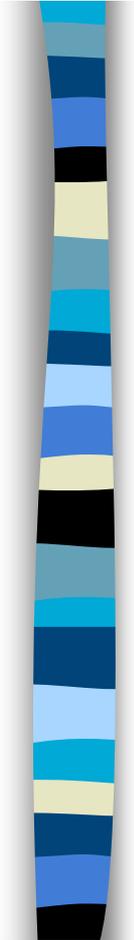


Que choisir ?



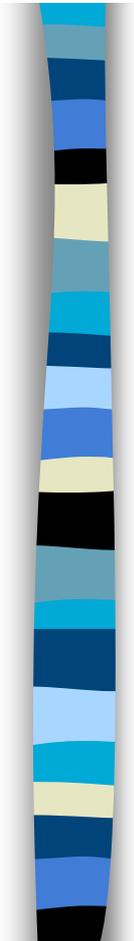
Faible couplage : discussion

- Un principe à garder en tête pour toute les décisions de conception
 - but sous-jacent à ne jamais oublier
- Un principe d'évaluation qu'un concepteur applique lorsqu'il évalue les choix de conception
- Ne peut pas être considéré indépendamment d'autres patterns comme *Expert* et *Forte cohésion*



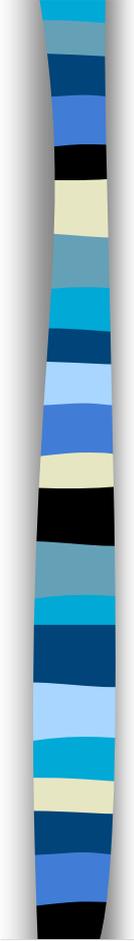
Faible couplage : discussion (suite)

- Pas de mesure absolue de quand un couplage est trop fort
- De façon générale, les classes qui sont très génériques par nature, et très réutilisables doivent avoir un faible couplage
- Un fort couplage n'est pas dramatique avec des éléments très stables
 - Java.util par exemple



Faible couplage : discussion (suite)

- Cas extrême de faible couplage
 - des objets incohérents, complexes, qui font tout le travail
 - des objets isolés, non couplés, qui servent à stocker les données
 - peu ou pas de communication entre objets
 - une mauvaise conception qui va à l'encontre des principes OO (collaboration d'objets)
- Bref
 - un couplage modéré est nécessaire et normal pour créer des systèmes OO



Forte cohésion (GRASP)

■ Problème

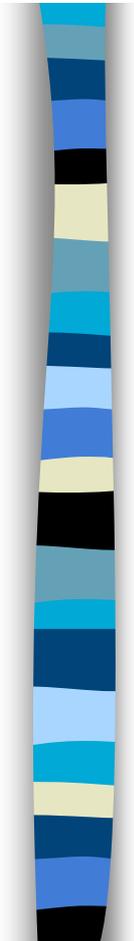
- Comment parvenir à maintenir la complexité gérable ?
 - Comment s'assurer que les objets restent compréhensibles et faciles à gérer, et - bénéfice second - qu'ils contribuent au faible couplage ?

■ Solution

- Attribuer une responsabilité de telle sorte que la cohésion reste forte. Appliquer ce principe pour évaluer les solutions possibles.

■ Cohésion

- La cohésion (la cohésion fonctionnelle) est une mesure de l'étroitesse des liens et de la spécialisation des responsabilités d'un élément (d'une classe)
- Une classe qui a des responsabilités étroitement liées et n'effectue pas un travail gigantesque est fortement cohésive



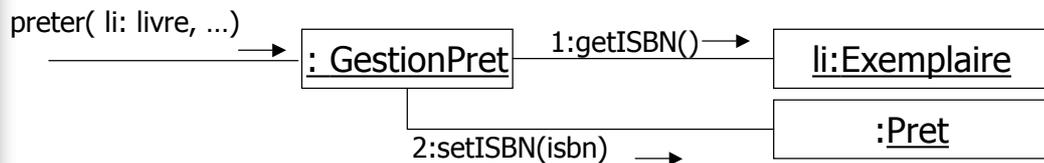
Forte cohésion (suite)

- Les classes ayant une faible cohésion effectuent des tâches sans liens entre elles ou effectuent trop de tâches

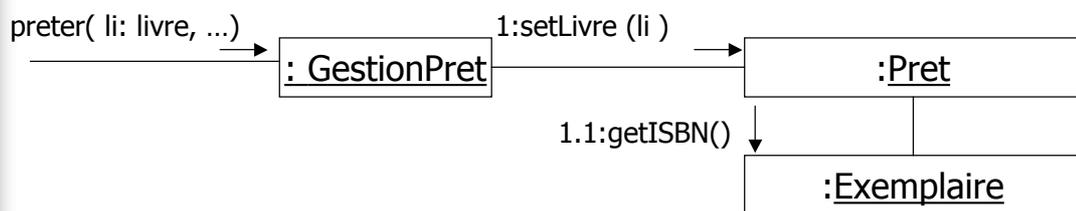
■ Problème des classes à faible cohésion :

- Difficiles à comprendre
- Difficiles à réutiliser
- Difficiles à maintenir
- Fragiles, constamment affectées par le changement

Forte cohésion : exemple



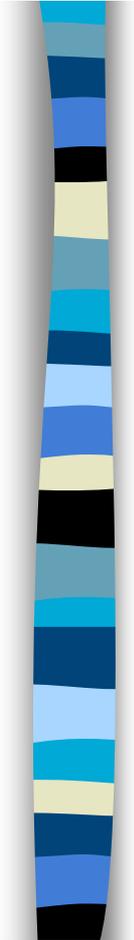
- On rend GestionPret partiellement responsable de la mise en place de l'ISBN
- *GestionPret* sera responsable de beaucoup d'autres fonctions



- On délègue la responsabilité de mettre l'ISBN au prêt

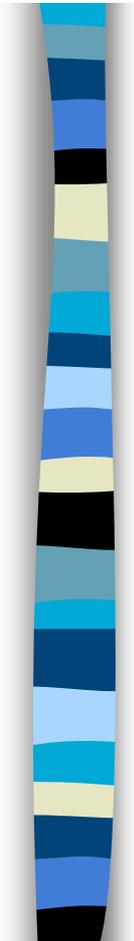
Forte cohésion : discussion

- Comme *Couplage faible*, un pattern d'évaluation à garder en tête pendant toute la conception
- [Booch] : Il existe une cohésion fonctionnelle quand les éléments d'un composant (eg. Les classes) « travaillent toutes ensemble pour fournir un comportement bien délimité »



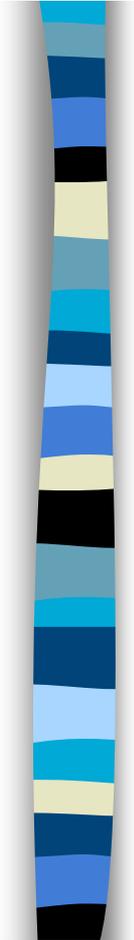
Forte cohésion : discussion

- Une classe de forte cohésion a un petit nombre de méthodes, avec des fonctionnalités hautement liées entre elles, et ne fait pas trop de travail
- Un test
 - décrire une classe avec une seule phrase.
- [Booch] : la modularité est la propriété d'un système qui a été décomposé en un ensemble de modules cohésifs et peu couplés
- Bénéfices de la forte cohésion
 - Augmentation de la clarté et de la compréhension de la conception
 - Maintenance et améliorations simplifiées
 - Signifie en général couplage faible
 - Meilleur potentiel de réutilisation



Contrôleur (GRASP)

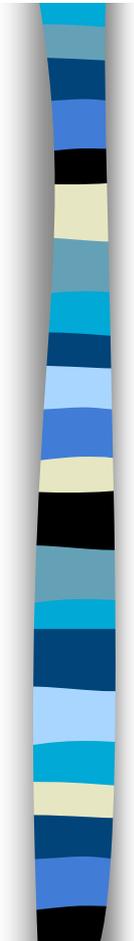
- Problème
 - Quel est le premier objet au delà de l'IHM qui reçoit et coordonne (contrôle) une opération système
 - opération système : événement majeur entrant dans le système
 - contrôleur : objet n'appartenant pas à l'IHM ayant la responsabilité de recevoir ou de gérer un événement système
- Solution
 - Affecter cette responsabilité à une classe qui correspond à l'un des cas suivants
 - elle représente le système global, un sous-système majeur, un équipement sur lequel le logiciel s'exécute (eq. à des variantes d'un contrôleur *Façade*)
 - elle représente un scénario de cas d'utilisation dans lequel l'événement système se produit (contrôleur de CU ou contrôleur de session)



Contrôleur: exemple

- Au cours de la détermination du comportement du système (besoins, CU, DSS), les opérations système sont déterminées et attribuées à une classe générale *Système*
- A l'analyse/conception, des classes contrôleur sont mises en place pour prendre en charge ces opérations

Bibliothèque
preterLivre()
enregistrerMembre()
.....

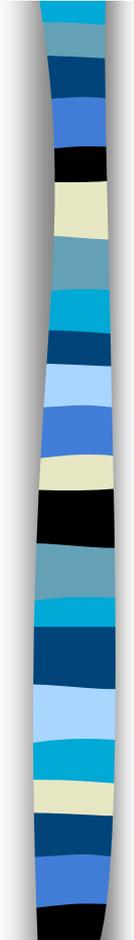


Contrôleur : exemple(suite)

- Pour la gestion d'une bibliothèque, qui doit être contrôleur pour l'opération système *emprunter* ?
- Deux possibilités

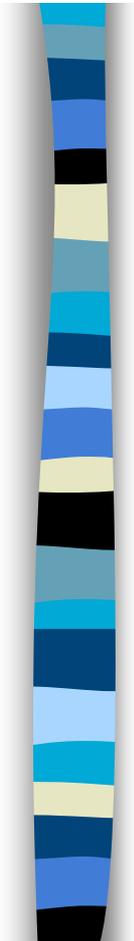
Système global	<u>:BiblioControleur</u>
Opérations systèmes liées au cas d'utilisation <i>emprunter</i>	<u>:GestionPret</u>

- La décision d'utiliser l'une ou l'autre solution dépend d'autres facteurs liés à la cohésion et au couplage



Contrôleur Façade

- Représente tout le système
 - exemples : ProductController, RetailInformationSystem, Switch, Router, NetworkInterfaceCard, SwitchFabric, *etc.*
- A utiliser quand
 - il y a peu d'événements système
 - il n'est pas possible de rediriger les événements systèmes à un contrôleur alternatif



Contrôleur de cas d'utilisation

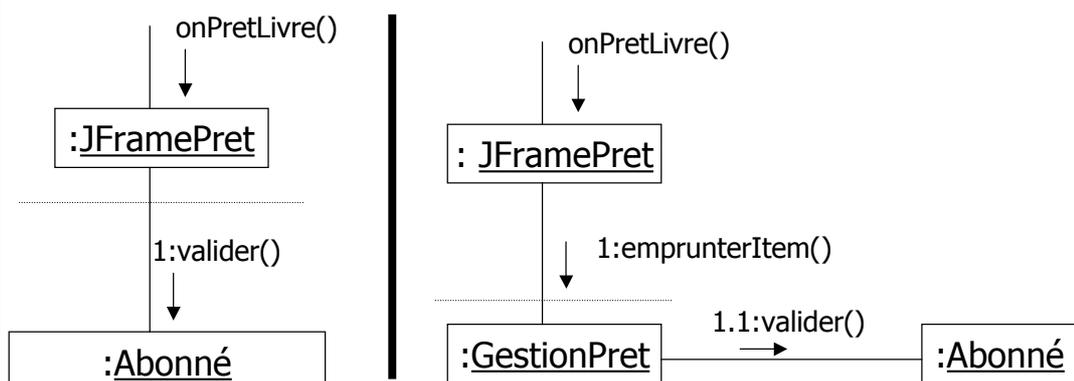
- Utiliser le même contrôleur pour tous les événements d'un cas d'utilisation
- Un contrôleur différent pour chaque cas d'utilisation
 - contrôleur artificiel, pas objet du domaine
- A utiliser quand
 - les autres choix amènent à un fort couplage ou à une cohésion faible (contrôleur *trop chargé - bloated*)
 - il y a de nombreux événements systèmes qui appartiennent à plusieurs processus
 - répartit la gestion entre des classes distinctes et faciles à gérer
 - permet de connaître et d'analyser l'état du scénario en cours

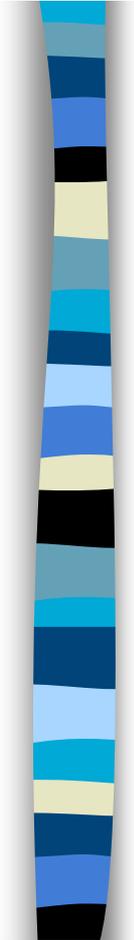
Contrôleur trop chargé (pas bon)

- Pas de focus, prend en charge de nombreux domaines de responsabilité
 - un seul contrôleur reçoit tous les événements système
 - le contrôleur effectue la majorité des tâches nécessaires pour répondre aux événements systèmes
 - un contrôleur doit déléguer à d'autres objets les tâches à effectuer
 - il a beaucoup d'attributs et gère des informations importantes du système ou du domaine
 - ces informations doivent être distribuées dans les autres objets
 - ou doivent être des duplications d'informations trouvées dans d'autres objets
- Solution
 - ajouter des contrôleurs
 - concevoir des contrôleurs dont la priorité soit de déléguer

Remarque : couche présentation

- Les objets d'interfaces graphique (fenêtres, applets) et la couche de présentation ne doivent pas prendre en charge les événements système
 - c'est la responsabilité de la couche domaine





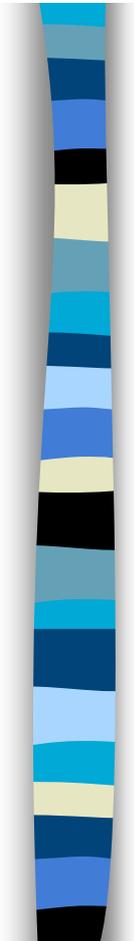
Contrôleur (fin)

■ Avantages

- Meilleur potentiel de réutilisation
 - moyen de séparer les connaissances du domaines de la couche présentation/IHM
- Capture l'état d'un CU
 - Permet de s'assurer que les opérations système se produisent dans le bon ordre

■ Patterns liés

- Commande, Façade, Couches, Fabrication pure



Polymorphisme (GRASP)

■ Problème

- Comment gérer des alternatives dépendantes des types ?
Comment créer des composants logiciels « enfichables » ?

■ Solution

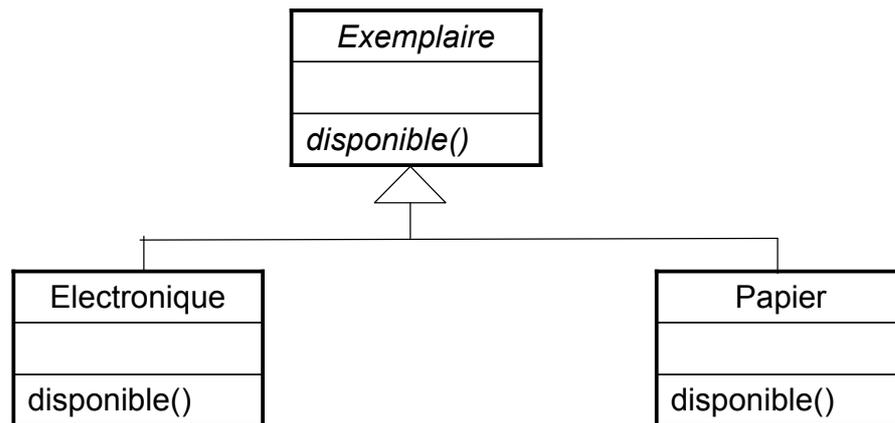
- Quand des fonctions ou des comportements connexes varient en fonction du type (classe), affectez les responsabilités - en utilisant des opérations polymorphes - aux types pour lesquels le comportement varie
 - ne pas utiliser de test sur le type d'un objet ou une logique conditionnelle (if/then/else) pour apporter des alternatives basées sur le type

■ Polymorphisme

- donner le même nom à des services dans différents objets

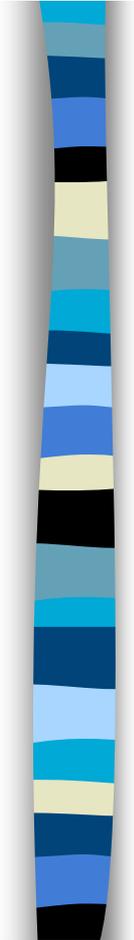
Polymorphisme : exemple

- Bibliothèque : qui doit être responsable de savoir si un exemplaire est disponible ?



Polymorphisme

- Implique en général l'utilisation de classes abstraites et d'interfaces
- Avantages
 - Les points d'extension requis par les nouvelles variantes sont faciles à ajouter
 - On peut introduire de nouvelles implémentations sans affecter les clients



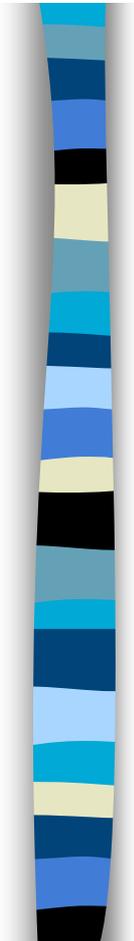
Fabrication pure (GRASP)

■ Problème

- Que faire quand les concepts du monde réel (objets du domaine) ne sont pas utilisables en respectant le Faible couplage et la Forte cohésion ?

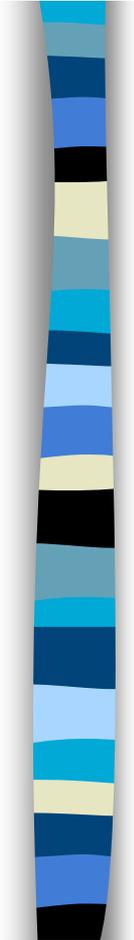
■ Solution

- Affecter un ensemble fortement cohésif à une classe artificielle ou de commodité, qui ne représente pas un concept du domaine
 - entité fabriquée de toutes pièces



Fabrication pure : exemple

- Pour la bibliothèque, les instances de Prêt seront enregistrées dans une BD relationnelle.
- D'après Expert, Prêt a cette responsabilité, mais cela aura des conséquences :
 - la tâche nécessite un grand nombre d'opération de BD
 - non spécialement liées aux fonctionnalités de Prêt
 - Prêt devient donc non cohésif
 - Prêt doit être lié à une BD relationnelle
 - le couplage augmente pour Prêt
 - l'enregistrement d'objet dans une BD relationnelle est une tâche générique utilisable par de nombreux objets
 - pas de réutilisation, beaucoup de duplication



Fabrication pure : exemple (suite)

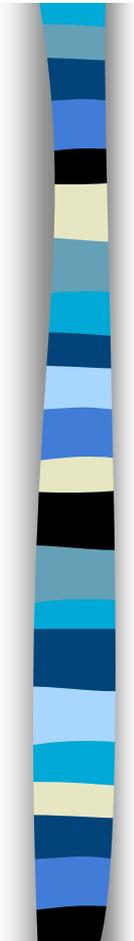
■ Solution

- créer une classe artificielle
`PersistentStorageBroker`

■ Ainsi

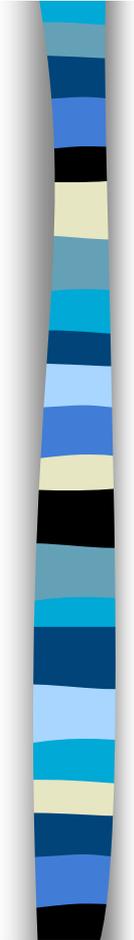
- *Pret* garde une forte cohésion et un couplage faible
- *PersistentStorageBroker* est relativement cohésif
- *PersistentStorageBroker* est générique et réutilisable

PersistentStorageBroker
<code>insert(Object)</code> <code>update(Object)</code> ...



Fabrication pure : discussion

- Concevoir des objets fabrications pures en pensant à ce qu'ils soient très réutilisables
 - s'assurer qu'ils ont des responsabilités limitées et cohésives
- Une fabrication pure est généralement partitionnée en fonction de ses fonctionnalités
 - sorte d'objet « centré-fonction » (sorte de décomposition comportementale)



Pure Fabrication

■ Avantages

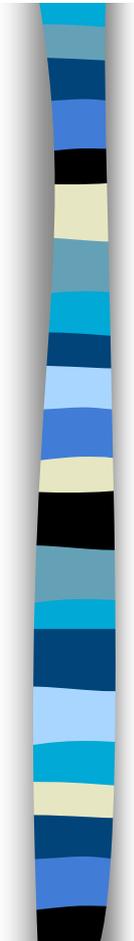
- Supporte Faible couplage et Forte cohésion
- Amélioration de la réutilisabilité

■ Attention

- L'esprit de la conception OO design est centré sur les objets, pas sur les fonctions
- Ne pas abuser des Fabrications pures

■ Patterns liés

- Couplage faible, Forte cohésion, Adaptateur, Observateur, Visiteur



Indirection (GRASP)

■ Problème

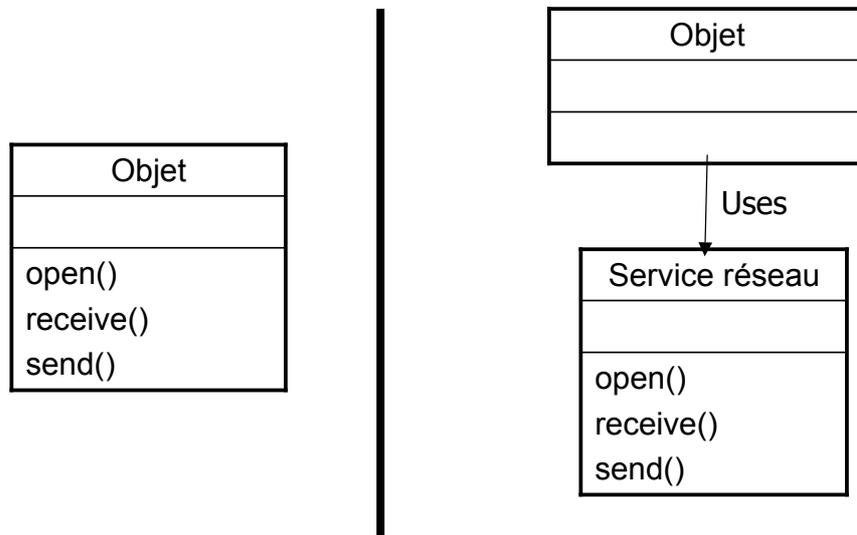
- Où affecter une responsabilité pour éviter le couplage entre deux entités (en général dans deux couches différentes) ?

■ Solution

- Donner la responsabilité à un objet qui sert d'intermédiaire entre d'autres composants ou services pour éviter de les coupler directement
 - l'intermédiaire crée une indirection entre les composants

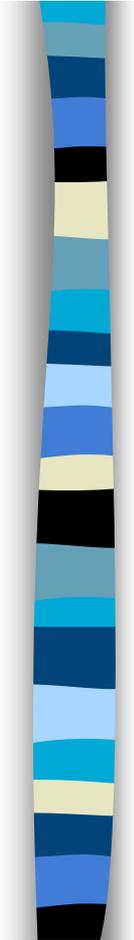
Indirection : exemple

- Supposons qu'un objet doit utiliser une connexion réseau (ex. socket TCP)



Indirection : discussion

- « En informatique, on peut résoudre la plupart des problèmes en ajoutant un niveau d'indirection »
- Beaucoup de fabrication pures sont créées pour des raisons d'indirection
- Objectif principal de l'indirection : faible couplage



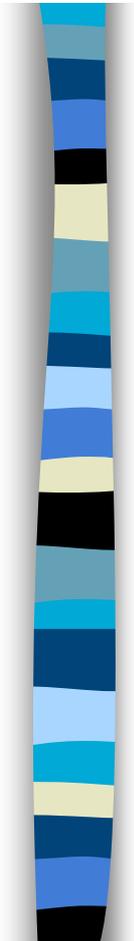
Protection des variations (GRASP)

■ Problème

- Comment concevoir des objets, sous-systèmes, systèmes de telle façon que les variations ou l'instabilité de ces éléments n'ait pas d'impact indésirable sur d'autres éléments

■ Solution

- Identifier les points de variation ou d'instabilité prévisibles. Affecter les responsabilités pour créer une interface stable autour d'eux.



Protection des variations : discussion

■ Ne pas se tromper de combat

- Ne pas passer des jours à préparer des protections qui ne serviront jamais

■ Prendre en compte les points de variation

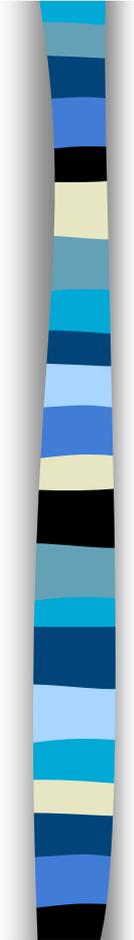
- identifiés dans les besoins

■ Gérer de la bonne manière les points d'évolution

- identifiés mais non obligatoire

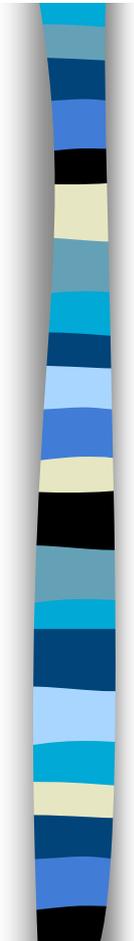
■ Différents niveaux de sagesse

- le novice conçoit fragile
- le meilleur programmeur conçoit tout de façon souple et généralisante
- l'expert sait évaluer les combats à mener



Ne pas parler aux inconnus

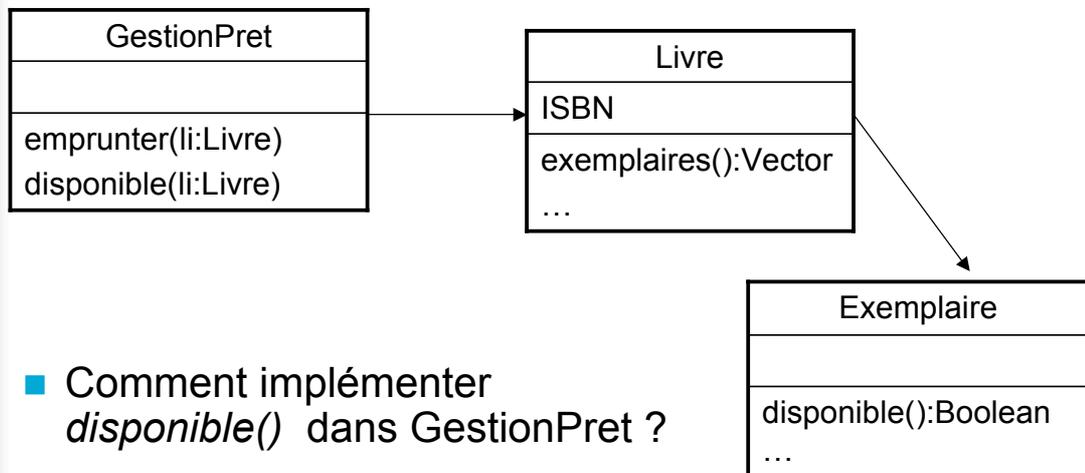
- Cas particulier de Protection des variations
 - variations liées aux évolutions de structures
- Problème
 - comment éviter de connaître la structure d'autres objets indirectement
 - si un client utilise un service ou obtient de l'information d'un objet indirect, comment le faire sans couplage ?
- Solution:
 - Donner la responsabilité à un objet que le client connaît directement de collaborer avec l'objet indirect, de telle sorte que le client n'ait pas besoin de connaître ce dernier.
 - Connue aussi comme « Loi de Demeter »



Ne pas parler aux inconnus (suite)

- Objectif : éviter le couplage entre le client et la connaissance d'objets indirects et les connexions entre objets
 - objets direct : familiers du client
 - objets indirects : inconnus
- Contrainte induite : depuis une méthode, n'envoyer des messages qu'aux objets suivants
 - l'objet *this* (self)
 - un paramètre de la méthode
 - un attribut de *this*
 - un élément d'une collection qui est un attribut de *this*
 - un objet créé à l'intérieur de la méthode
- Implication
 - ajout d'opérations dans les objets directs pour servir d'opérations intermédiaires

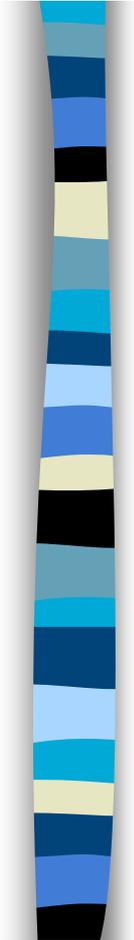
Ne pas parler aux inconnus : exemple



- Comment implémenter *disponible()* dans **GestionPret** ?
- Autre exemple à éviter
Toto t = a.getA().getB().getC() ;

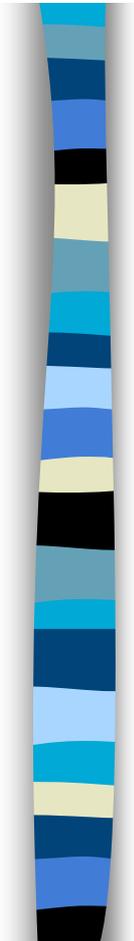
Les patterns GRASP et les autres

- D'une certaine manière, tous les autres patterns sont des applications, des spécialisations, des utilisations conjointes des 9 patterns GRASP, qui sont les plus généraux.



Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks



Généralités

- Origine dans l'architecture
 - ouvrages de Christopher Alexander (77)
- Propriétés
 - pragmatisme
 - solutions existantes et éprouvées
 - récurrence
 - bonnes manières de faire éprouvées
 - générativité
 - comment et quand appliquer, indépendance au langage de programmation
 - émergence
 - la solution globale émerge de l'application d'un ensemble de patrons

Types de patrons informatiques

- Patrons de conception
 - architecture
 - conception de systèmes
 - conception
 - interaction de composants
 - comportement
 - structure
 - création
 - Idiomes de programmation
 - Techniques, styles spécifiques à un langage
- Patrons d'analyse
- Patrons d'organisation
- ...

Références

- Ouvrage du « Gang of Four »
 - Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994), *Design patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 395 p. (trad. française : *Design patterns. Catalogue des modèles de conception réutilisables*, Vuibert 1999)
- Plus orienté architecture
 - Martin Fowler (2002) *Patterns of Enterprise Application Architecture*, Addison Wesley
- Sites
 - <http://www.hillside.net/patterns>
 - ...

Éléments d'un patron

- **Nom**
 - évocateur, référence concise
- **Problème**
 - objectifs que le patron cherche à atteindre
- **Contexte**
 - domaine d'application du patron : précise comment le problème survient, et quand la solution fonctionne.
- **Forces/contraintes**
 - forces et contraintes interagissant au sein du contexte. Détermination des compromis.
- **Solution**
 - comment mettre en œuvre la solution. Point de vue statique (structure) et dynamique (interactions). Variantes de solutions.

Éléments d'un patron (suite)

- **Exemples**
 - exemples d'applications.
- **Contexte résultant**
 - description du contexte résultant de l'application du patron au contexte initial. Conséquences positives et négatives.
- **Justification**
 - raisons fondamentales conduisant à l'utilisation du patron. Réflexions sur la qualité du patron.
- **Patrons associés**
 - similaires ou possédant des contextes initial ou résultant proche.
- **Utilisations connues**
 - exemples d'applications réels.

Les patrons ne sont pas

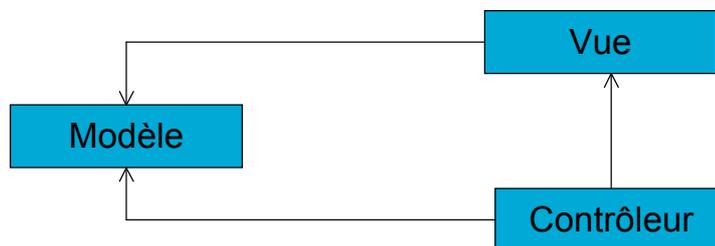
- Limités au domaine informatique
- Des idées nouvelles
- Des solutions qui n'ont fonctionné qu'une fois
- Des principes abstraits ou des heuristiques
- Une panacée

Les patrons sont

- Des solutions éprouvées à des problèmes récurrents
- Spécifiques au domaine d'utilisation
- Rien d'exceptionnel pour les experts d'un domaine
- Une forme littéraire pour documenter des pratiques
- Un vocabulaire partagé pour discuter de problèmes
- Un moyen efficace de réutiliser et partager de l'expérience

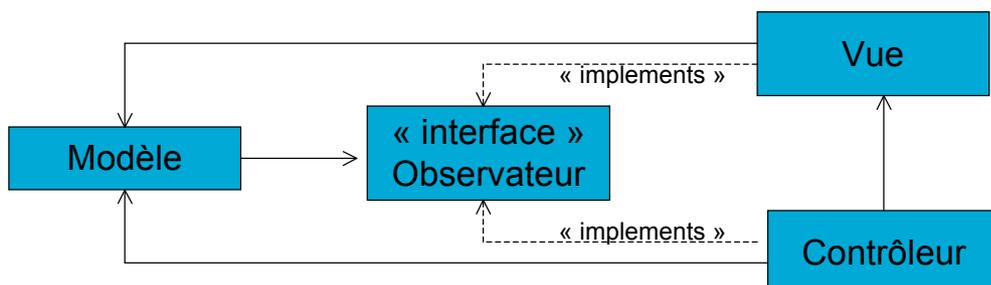
Principes MVC

- Pour rendre le modèle indépendant des vues (utilisateur) qui en dépendent
- Version modèle passif
 - la vue se construit à partir du modèle
 - le contrôleur modifie le modèle les changements que l'utilisateur spécifie dans la vue
 - le contrôleur informe la vue que le modèle a changé et qu'elle doit se reconstruire



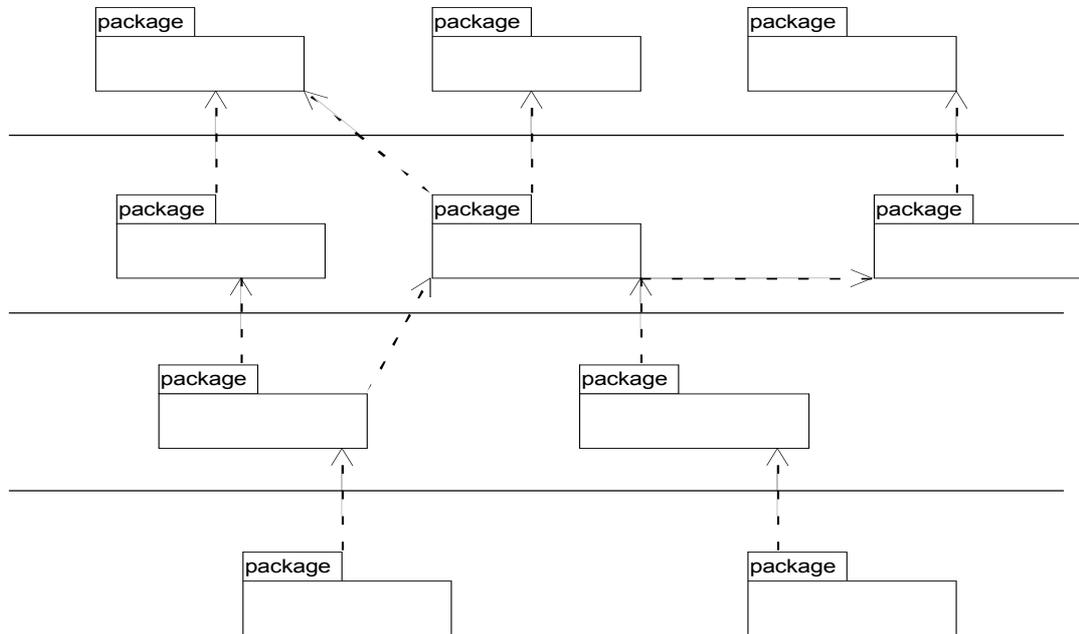
Principe MVC (suite)

- Version modèle actif
 - quand le modèle peut changer indépendamment du contrôleur
 - le modèle informe les abonnés à l'observateur qu'il s'est modifié, ceux-ci prennent l'information en compte (contrôleur et vues)



Pattern Couches

Présentation
(Application)
Domaine
Service
Middleware
Fondation



Fabrique concrète

- Pour être responsable de la création d'objets lorsque la logique de création est complexe, ou bien qu'il convient de séparer les responsabilités de création ?
 - Fabrique concrète = objet qui fabrique des instances
- Avantages par rapport à un constructeur
 - la classe a un nom
 - permet de gérer facilement plusieurs méthodes de construction avec des signatures similaires
 - peut retourner plusieurs types d'objets

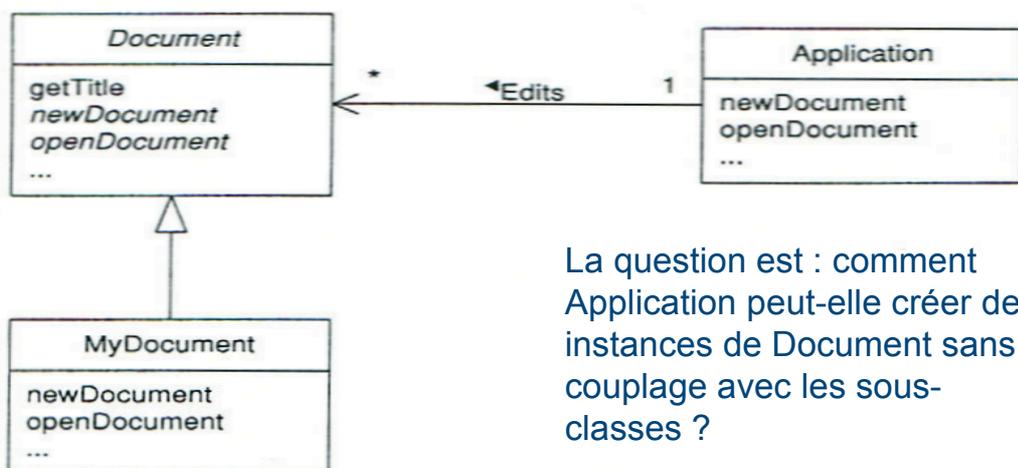
Factory method (GoF)

■ Factory

- un objet qui fabrique des instances conformes à une interface ou une classe abstraite
- par exemple, une *Application* veut manipuler des document, qui répondent à une interface *Document*
 - ou une *Equipe* veut gérer des *Tactique*...

(T. Horton, CS494)

Factory - Fabrique (GoF)

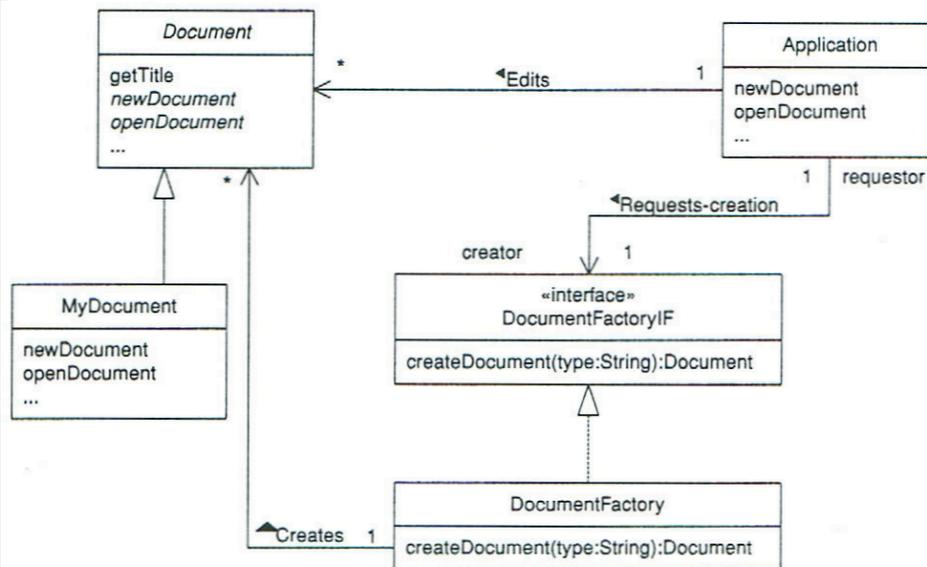


La question est : comment Application peut-elle créer des instances de Document sans couplage avec les sous-classes ?

FIGURE 5.1 Application framework.

(From Grand's book.)

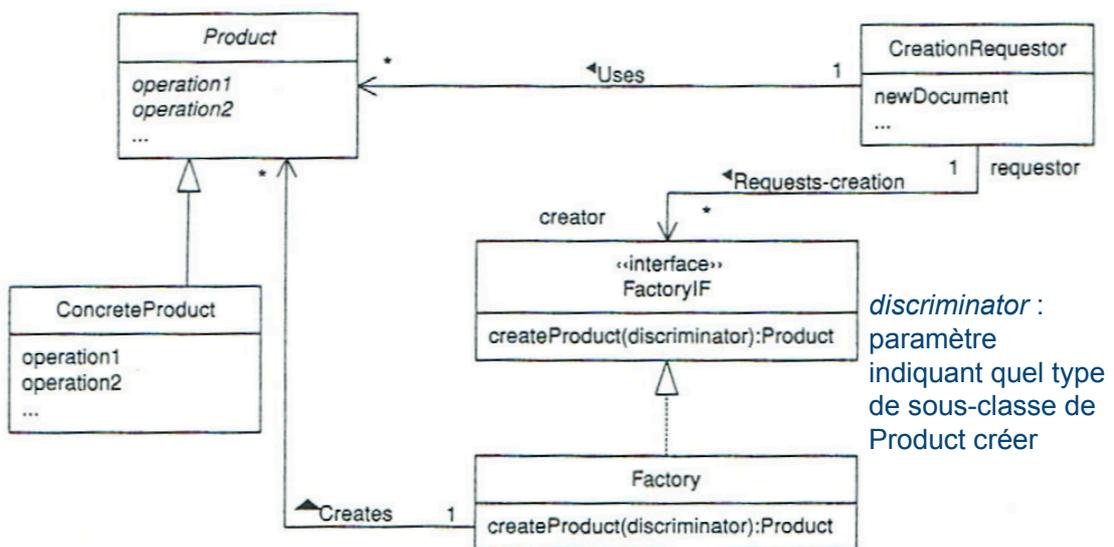
Solution : utiliser une classe DocumentFactory pour créer différents types de documents



(From Grand's book.)

FIGURE 5.2 Application framework with document factory.

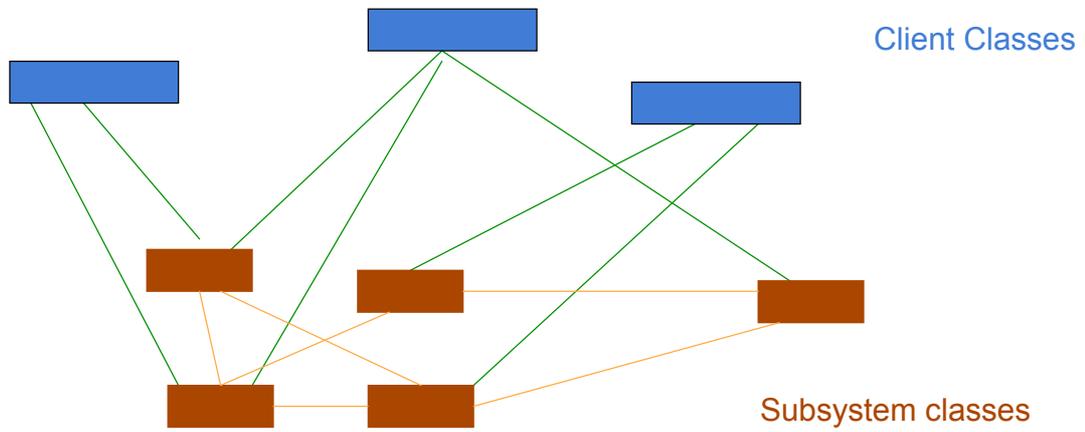
Factory Method Pattern : structure générale



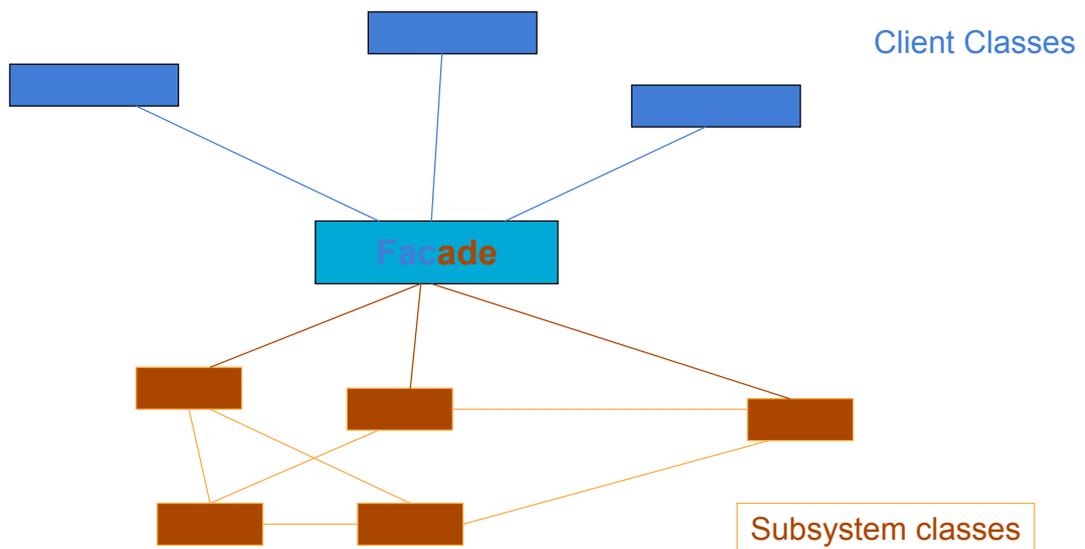
(From Grand's book.)

FIGURE 5.3 Factory method pattern.

Façade (GoF) : problème

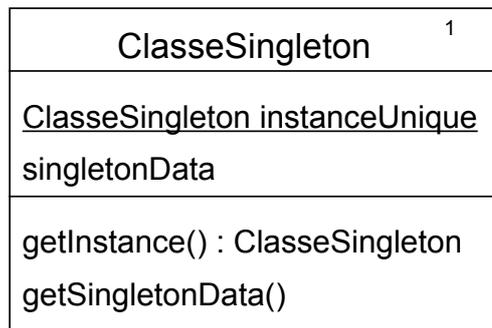


Façade (GoF) : solution



Singleton (GoF)

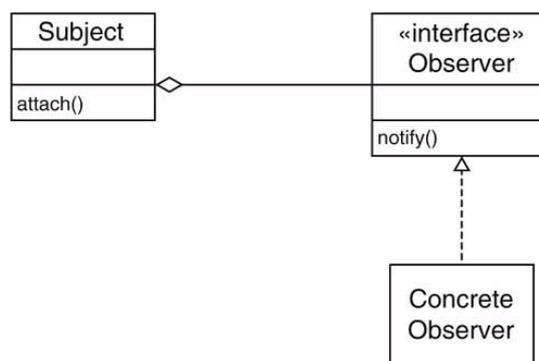
- Quand on a besoin d'une instance unique d'une classe (ex. Factory), avec un point d'accès unique et global pour les autres objets
- Singleton
 - utiliser une méthode statique de la classe qui retourne l'instance



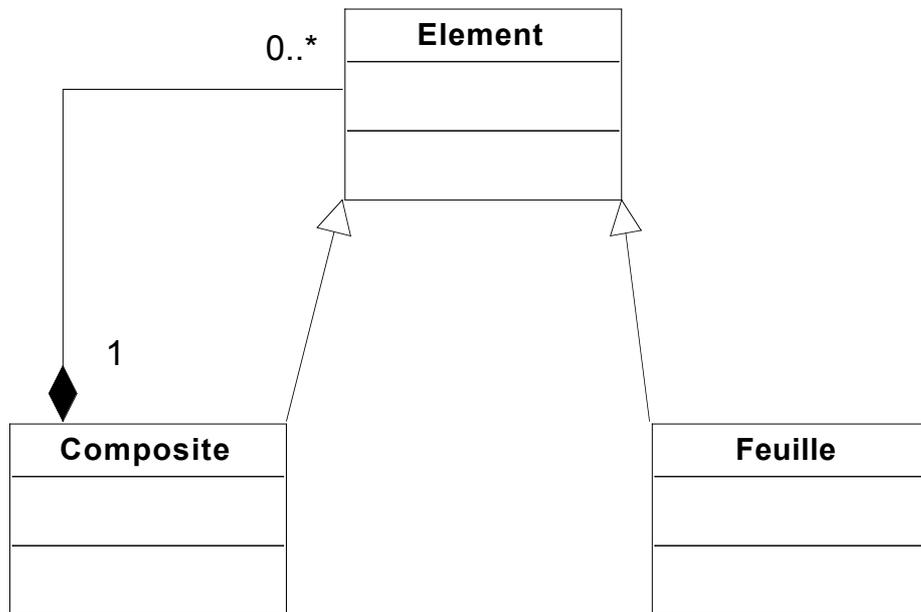
Observateur

- Utile pour présenter de plusieurs manières différentes des informations d'un objet *Sujet*
- Un *Observateur* s'attache à un *Sujet*
- Le sujet *notifie* ses observateurs en cas de changement d'état

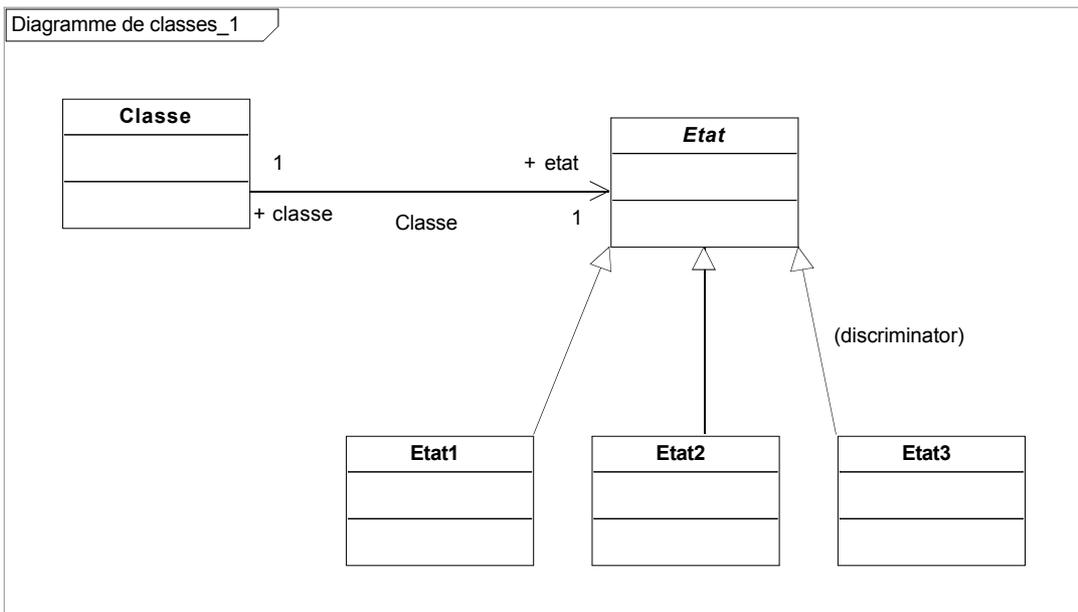
Observer Pattern

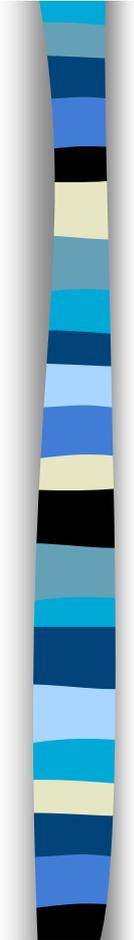


Composite (GoF)



Etat (GoF)

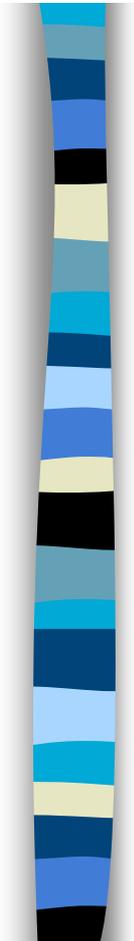




Autres patterns

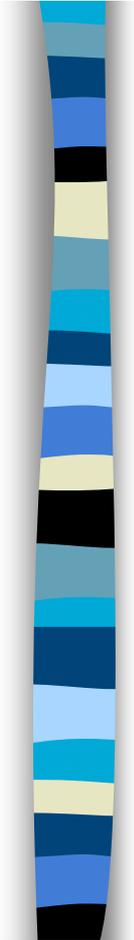
- Bridge (GoF)
- Chain of responsibility (GoF)
- Proxy (Gof)
- Adaptateur (GoF)
- ...

(wikipedia)



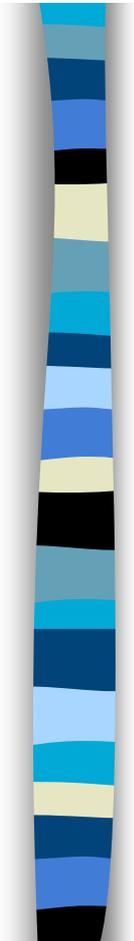
Anti-patterns

- Erreurs courantes de conception documentées
- Caractérisés par
 - lenteur du logiciel, coûts de réalisation ou de maintenance élevés, comportements anormaux, présence de bogues.
- Exemples
 - Action à distance
 - emploi massif de variables globales, fort couplage
 - Coulée de lave
 - partie de code encore immature mise en production, forçant la lave à se solidifier en empêchant sa modification
 - ...



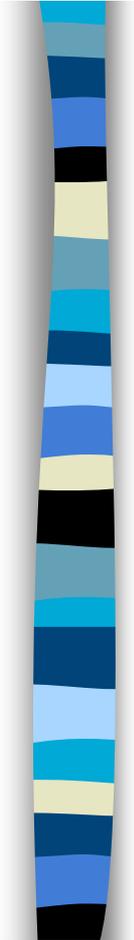
Design patterns et IDE

- Fournir une aide à l'instanciation ou au repérage de patterns
 - nécessite une représentation graphique (au minimum collaboration UML) et le codage de certaines contraintes
- Instanciation
 - choix d'un pattern, création automatique des classes correspondantes
- Repérage
 - assister l'utilisateur à repérer
 - des patterns utilisés (pour les documenter)
 - des « presque patterns » (pour les faire tendre vers des patterns)
- Exemples d'outils
 - Describe + Jbuilder
 - Objecteering
 - ...



Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks



Framework

■ Définition

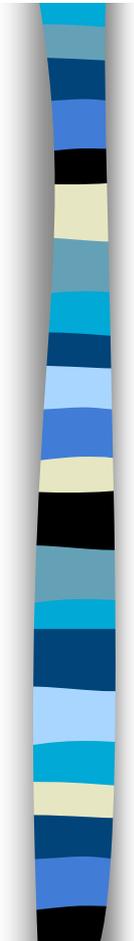
- ensemble de classes qui collaborent à la réalisation d'une responsabilité qui dépasse celle de chacune
- conception générale réutilisable pour une classe d'application donnée

■ Un framework définit

- architecture, classes, respnsabilités, flot de contrôler, *etc.*

■ Un framework doit être spécialisé

- reprise de code de haut niveau (beaucoup de classes abstraites), ajout de code de spécialisation



Exemple de framework : STRUTS

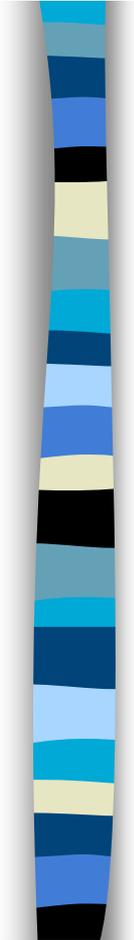
■ Couche IHM d'une application web

- gérer des sessions utilisateur, gérer des actions en fonction des requêtes HTTP

■ Basé sur MVC

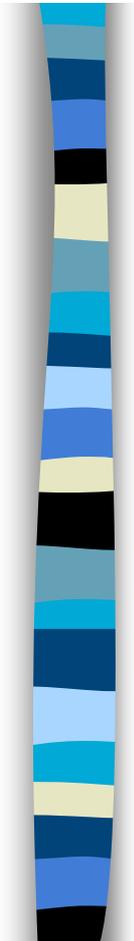
■ Ensemble de classes à spécialiser

- Fabriques
- Contrôleurs
- ...



Conclusion

- On a vu assez précisément les patterns les plus généraux (GRASP)
- On a survolé les autres
 - un bon programmeur doit les étudier et en connaître une cinquantaine



Remerciements

- Olivier Aubert
- Yohan Welikala (Sri Lanka)