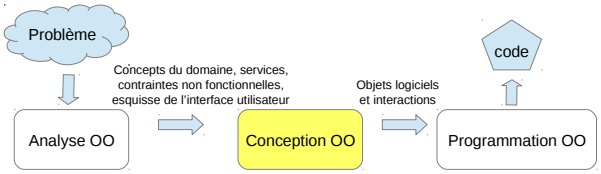
M1 informatique
Module Génie Logiciel

CM2 : Design Patterns (DP), automatisation et Intégration Continue (IC)

Céline ROUDET
Celine.Roudet@u-bourgogne.fr
http://ufrsciencestech.u-bourgogne.fr/~roudet/teaching.html
http://ufrsciencestech.u-bourgogne.fr/master1/GenieLogiciel/

Phase de conception logicielle OO

- penser et repenser le produit à réaliser sous une **forme abstraite**, avant sa production effective,
- construire des modèles de conception objet (avec UML) pour résoudre un problème et satisfaire les besoins identifiés et décrits.



Bonne conception (extensibilité, flexibilité, réutilisabilité) liée au nombre et à la nature des **dépendances** entre les éléments de l'application

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception
- **Dépendance (couplage) et cohésion**
- Principes de conception : SOLID et autres
- Intérêt des interfaces

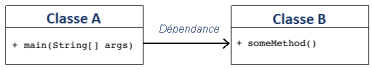
2- Patrons de conception

3- Automatisation en Java avec Ant et Maven

4- Intégration continue avec Jenkins

Dépendance ou couplage

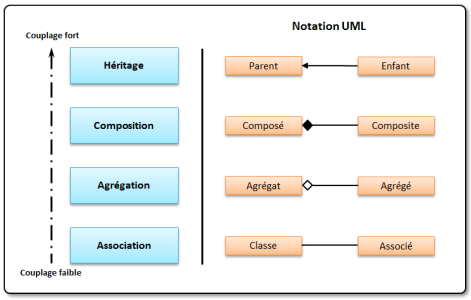
En POO, un objet de type A **dépend** d'un objet de type B (**A → B**) si au moins une de ces conditions est vérifiée :
- A est de **type** B (dépendance par héritage) ;
- A possède un **attribut** de type B (par composition) ;
- A dépend d'un objet de type C qui dépend d'un objet de type B (par transitivité) ;
- **une méthode de A appelle une méthode de B** :




```
public class A {
    public static void main(String[] args)
        B b = new B();
        b.someMethod();
    }
}
```

Tiré de Wikipedia et http://igm.univ-mlv.fr/~dr/XPOSE2010/guicespring/di_presentation.html

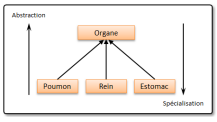
Couplage et liens entre classes



Conseil du Gof : *« favorisez la composition (délégation) par rapport à l'héritage »*

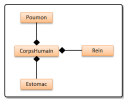
Tiré de : https://blog.developpez.com/rawsrc/p10377/conception/dependances_et_couplage_des_classes_poo
Design Patterns – Elements of Reusable Object-Oriented Software.E. Gamma, R. Helm, R. Johnson & J. Vlissides. 1995

Couplage et liens entre classes (2)

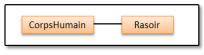


1) Héritage ou dérivation
- classes dérivées de **même nature** que classe mère
- l'**héritage fonctionnel** est à proscrire

2) Composition et agrégation (délégation)
- classes **composites** ne peuvent exister en dehors de la classe composée.
- classe (client) qui **délègue** à une autre classe une partie de son activité : **attribut** du client contient une référence vers la classe serveuse



3) Association
- paramètre typé que l'on passe à une méthode d'une classe,
- la classe qui exécute la fonction se sert du paramètre mais n'en garde **aucune trace**.




```
public class CorpsHumain {
    public raser(Rasoir rasoir){
      //on ne garde pas trace du rasoir utilisé
    }
}
```

Tiré de : https://blog.developpez.com/rawsrc/p10377/conception/dependances_et_couplage_des_classes_poo

Cohésion et indépendance fonctionnelle *(wikipedia)*

- **Cohésion** : métrique mesurant le respect des principes
  - d'**encapsulation** des données (séparer interface et implantation : favorise la modularité),
  - de **masquage de l'information** (cacher les détails d'implantation pour offrir une interface simple à comprendre et à utiliser),
  - et d'utilisation de **patrons de conception** reconnus.
- **Indépendance fonctionnelle** :
  - évaluée par le **rapport cohésion / couplage**
  - mesure l'autonomie et la possibilité de réutilisation d'un composant logiciel

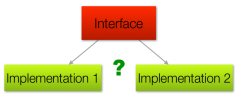
Principes de conception : généralités et SOLID

- Facilitent la conception générale de systèmes faciles à **maintenir** et à **étendre** (code fiable et robuste)
- Principes **SOLID** : identifiés par Robert Martin (« Uncle Bob »)
  - **S** (Single responsability) : un objet fait une seule chose et est le seule à le faire
  - **O** (Open-Close) : classe ouverte aux extensions et fermées aux modifications
  - **L** (Liskov substitution) : classe fille doit pouvoir être utilisée à la place de sa mère (par une autre classe) en toute transparence
  - **I** (Interface Segregation) : plusieurs petites interfaces plutôt qu'une grosse
  - **D** (Dependency Inversion) : dépendre d'abstractions et pas d'implantations → essentiel pour permettre le test unitaire

Robert Martin. « Design Principles and Design Patterns ». objectmentor.com (2000)

Intérêt des interfaces

- Évoluer **par ajout** *vs* par modification
  - usage des **interfaces**, qui évoluent peu
  - pour rapidement changer de stratégie



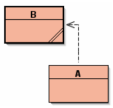
- Intérêt des interfaces
  - déf. abstraite d'un type, indépendamment de la façon dont il est implanté,
  - forme de **contrat** (ensemble de services rendus),
  - les classes qui l'implémentent ont un **paradigme commun** et doivent **respecter ce contrat** (s'y plier).

Inversion du contrôle (IoC)

- **Contrôle** = flot d'exécution d'une application
  - prog. **classique** : résulte exclusivement des instruction des programmes
  - prog. **événementielle** (ex : *Swing*) : programme ne le maîtrise plus, mais c'est Swing qui le pilote (**IoC**) et gère la boucle d'attente des événements
- **Inversion de contrôle** (IoC) :
  - caractéristique des **frameworks** (composants actifs) qui les différencie des bibliothèques de code (composants passifs)
  - principe de conception très général, appelé principe « **Hollywood** »
- **Injection de dépendance** :
  - dépendances entre classes non exprimées dans le code de manière statique
  - mais déterminées dynamiquement (**injectées**) à l'exécution,
  - faite à la main ou à l'aide de frameworks (*Spring IoC, Google Guice*),
  - favorise les **architectures testables** et l'utilisation de **doublures** !

Principe d'inversion de dépendance : exemple

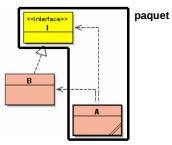
```
public class A{
     private B b;
     public A(){ this.b = new B(); }
}
public class B{...}
```



- Inversion de dépendance (découple A de B) :

- créer une **interface I** qui contiendra toutes les méthodes que A peut appeler sur B,
- indiquer que **B implémente l'interface I**,
- remplacer toutes les **références au type B** par des références à l'interface I dans A.

```
public class A{
     private I i;
     public A(){ this.i = new B(); }
}
public class B implements I{...}
public interface I{...}
```



- Problèmes :

- disposer dans A d'un objet implantant I alors qu'on ne sait pas comment l'instancier,
- à cause du « new », la dépendance A → B subsiste

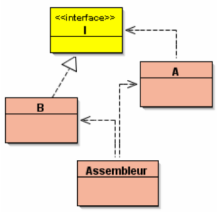
Principe d'injection de dépendance : exemple

```
public class A{    //A ne dépend plus de B
     private I i;

     public A(I i){     //injection de dépendance
          this.i = i; } //via le constructeur

     public setI(I i){
          this.i = i; }   //ou via un mutateur
}

public class B implements I{}
```



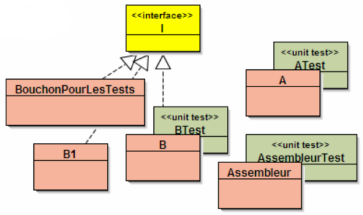
Nécessité d'un assembleur (classe appelante) qui centralise la gestion des dépendances :

```
public class Assembleur{

     public Assembleur(){
          A a = new A( new B() );  //injection de la dépendance via le constructeur

          A a = new A();
          a.setI( new B() );  //ou injection de la dépendance via la méthode set
     }
}
```

Gains de l'injection de dépendance



- **substitution** d'une classe par une autre (dans le code de l'assembleur) :

```
A a = new A( new B());  //on remplace la classe B
A a = new A( new B1()); //par la classe B1
```

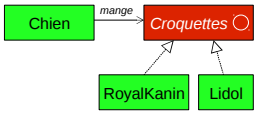
- **« vrais » tests unitaires** : en isolation (plus simples)
- **doublures** pour les tests peuvent être utilisés (en attendant la classe B, tests unitaires de A) :

```
public class ATest extends junit.framework.TestCase{
    A a = new A( new BouchonPourLesTests() );  //injection de la doublure
}
```


Couplage fort : exemple

Exemple d'un chien, à qui on donne des croquettes. En début de mois c'est du Royal Kanin et en fin de mois du Lidol.

```
public class Chien() {
    private Croquettes bouffe;
    public Chien() {
       if debutMois {
           bouffe = new RoyalKanin();
       } else {
           bouffe = new Lidol();
       }
    }
}
```



Que se passe-t-il si on veut donner du **Frolik** ou si **Lidol fait faillite** ?
On doit retoucher la classe **Chien** qui dépend de l'implantation des Croquettes.
Il y a un **fort couplage**.

Tiré de *www.developpez.net*

Injection de dépendance (par mutateur)

Solution : la classe Chien **n'utilise que l'interface**.
Le choix de l'implantation étant donnée au composant appelant :

```
public class Chien {
    private Croquettes bouffe;
    public Chien() {}
    public setCroquettes(Croquettes croquettes) {
        bouffe = croquettes; }
}
```

Chien dépend d'une abstraction (interface) et pas d'une implantation

injection de la dépendance en **argument** du constructeur ou d'une méthode

```
public class Maitre {
    private Chien medor;
    public Maitre() {
        medor = new Chien(); }
    public void nourritChienDebutMois() {
        medor.setCroquettes(new RoyalKanin());  }
    public void nourritChienFinMois() {
        medor.setCroquettes(new Lidol()); }
}
```

c'est **Maitre** qui décide comment nourrir son chien

Tiré de *www.developpez.net*

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception

2- Patrons de conception
- **Généralités sur les patrons**
- Patrons architecturaux : MVC et couche
- DP de construction/création
- DP de structuration
- DP de comportement

3- Automatisation en Java avec Ant et Maven

4- Intégration continue avec Jenkins

Généralités sur les patrons (pattern, modèle)

- **Patron de conception** : solution générale à un problème de conception courant, validé par l'expérience
  - structure et comportement d'une société de classes utilisant des interfaces,
  - description nommée d'un problème et d'une solution éprouvée,
  - avec conseils d'application (expérience de programmeurs).
- Ils se focalisent sur des problèmes **ponctuels** :
  - pas suffisants pour aider à développer des solutions complètes,
  - à combiner avec les **principes de conception** (plus globaux et abstraits).
- Ils visent tous à :
  - renforcer la **cohésion** et diminuer le **couplage,**
  - structurer son code et partager une approche commune.

Généralités sur les patrons (2)

- Propriétés :
  - **Pragmatisme** : solutions existantes 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
- Éléments d'un patron :
  - **Nom** (1 ou 2 mots) : évocateur, concis
  - **Problème** : comment il survient, points bloquants que le patron cherche à résoudre
  - **Description abstraite** de la solution, **exemple** d'utilisation
  - **Discussion** : conseils sur la façon de l'appliquer, variantes, …

Patrons de conception GRASP *(C. Larman 1995)*

General Responsability Assignment Software Patterns

- relèvent du « bon sens de conception », sont intuitifs,
- patrons généraux d'**affectation de responsabilités**
  - faible **couplage**, forte **cohésion**, **polymorphisme** :

```
if(mode_paiement.equals(« en liquide »)) { … }
else if(mode_paiement.equals(« carte de credit »)) { … }     // choix multiples
else if ...
```

→ plutôt affecter la responsabilité de l'enregistrement du paiement à des classes telles que `PaiementEnLiquide, PaiementParCarte, …` qui implantent une interface `Paiement` avec une méthode polymorphe `enregistrerPaiement()`.

  - **indirection** : découpler des classes à l'aide d'une classe intermédiaire (*MVC*)
  - **protection des variations** : identifier les points de variation ou d'instabilité prévisibles (*associer aux objets des interfaces stables*)

Patrons de conception GRASP et anti-patrons

- **Quatre derniers patrons GRASP** = situations plus précises d'attribution des responsabilités
  - **expert en information** : « celui qui sait fait le travail »
  - **contrôleur** : réception/traitement des événements → classes contrôleurs
  - **créateur** : responsabilité de créer les instances d'une classe
  - **fabrication pure** : entité fabriquée de toutes pièces
- **Anti-patrons** : erreurs logicielles faites fréquemment
  - **objet divin** : composant logiciel assurant trop de fonctions essentielles
  - **prog. spaghetti** : impossible de modifier une petite partie sans altérer le fonctionnement de beaucoup d'autres composants
  - **réinventer la roue** : (mal) réinventer une solution standard
  - **erreur de copie/coller** : duplication → factoriser les parties communes
  - **action à distance** : emploi massif de **variables globales**
  - **coulée de lave** : partie de code encore immature mise en production, forçant la lave à se solidifier en empêchant sa modification
  - …

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception

2- Patrons de conception
- Généralités sur les patrons
- **Patrons architecturaux : MVC et couche**
- DP de construction/création
- DP de structuration
- DP de comportement

3- Automatisation en Java avec Ant et Maven

4- Intégration continue avec Jenkins

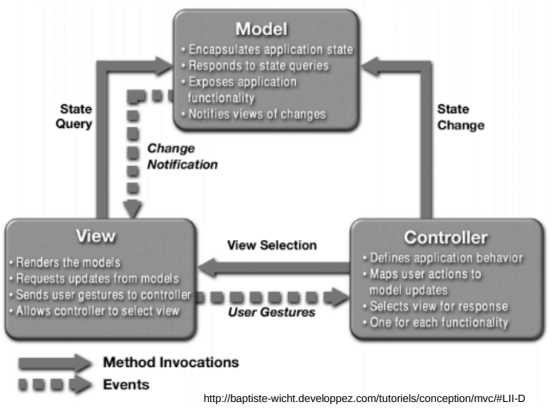
Patrons architecturaux

- **But :** conception de systèmes, organisation d'une application
- Exemples :
  - architecture multi-tiers (en couches), MVC (application interactives),
  - patrons d'authentification, d'autorisation, de sécurité, ...

MVC :

- **But :** séparer traitements, données et présentation pour faciliter l'évolution des IHM et en proposer plusieurs versions
- **Problème** : comment rendre le modèle (domaine métier) indépendant des vues (interface utilisateur) qui en dépendent ?
  → réduire le **couplage** entre modèle et vue
- **Solution** : insérer une couche supplémentaire (contrôleur) pour la gestion des événements utilisateur et le choix de la vue appropriée

Pattern MVC



http://baptiste-wicht.developpez.com/tutoriels/conception/mvc/#LII-D

Pattern MVC : exemple

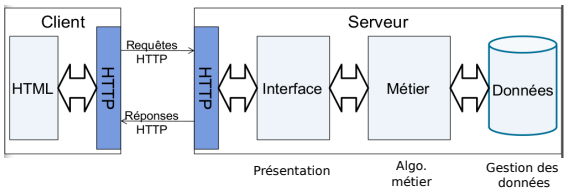
- l'utilisateur effectue une action (clic sur un bouton) ;
- l'action est captée par le contrôleur, qui :
  - vérifie la cohérence des données,
  - peut les transformer afin que le modèle les comprenne.
- le contrôleur peut aussi demander à la vue de changer ;
- le modèle reçoit les données et change d'état ;
- le modèle notifie la/les vue(s) qu'il faut se mettre à jour ;
- l'affichage dans la/les vue(s) est modifié en conséquence en allant chercher l'état du modèle.

Avantages :
- rend les modifications plus simples,
- toutes les vues qui montrent la même chose sont synchronisées.

https://openclassrooms.com/courses/apprenez-a-programmer-en-java/mieux-structurer-son-code-le-pattern-mvc

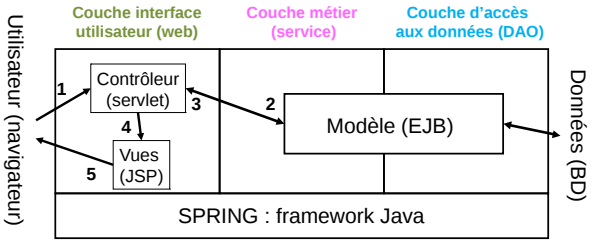
Architecture multi-tiers (n-tiers)

- **Objectif** : découpler les différentes fonctionnalités d'un programme (séparation des préoccupations)
- **Fonctionnement** : concevoir séparément chacune de ces fonctionnalités
- Les isoler les unes des autres (autant que possible)



Pattern MVC et architecture web à 3 couches

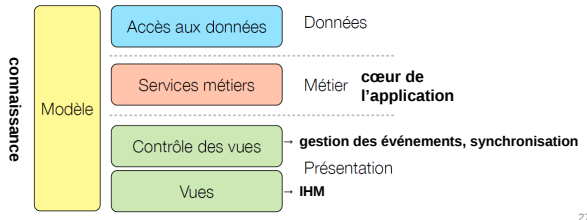
- **3 couches** : indépendantes grâce à l'utilisation d'interfaces
- intégration des différentes couches : réalisée avec **Spring**



Pattern MVC et architecture web à 3 couches (2)

Séparation des responsabilités :

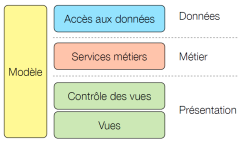
- Classes **modèle** (VO) : représentation logique des données métier
- Interfaces d'**accès aux données** (DAO) *stockées dans 1 BD par ex.*
- Classes en charge d'appliquer les règles de **gestion métier**
- Classes en charge de la **synchronisation** entre les vues et le modèle
- Classes de **vue** en charge de l'affichage



Packages Java : principes

```
src
    main
        java : Java source path
    test
        java : Java source path
target
    classes : default output folder
```

- Ensemble de dossier et sous-dossiers contenant une ou plusieurs classes
- Nom soumis à 1 **convention de nommage** (*ex : fr.univ.tp*)
  - entièrement écrit en minuscule
  - arborescence « fr/univ/tp » créée dans tous les dossiers
- Classes déclarées **« public »** visibles depuis l'extérieur du package qui les contient
- Première instruction d'une classe contenue dans un package : **package fr.univ.tp;**
- S'utilisent pour catégoriser les **classes par couches**



Patterns MVC et couche : exemple d'un blog

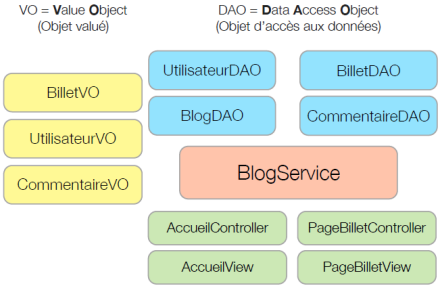


Image tirée du cours de H. Labas

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception

2- Patrons de conception

- Généralités sur les patrons
- Patrons architecturaux : MVC et couche
- **DP de construction/création**
- DP de structuration
- DP de comportement

3- Automatisation en Java avec Ant et Maven

4- Intégration continue avec Jenkins

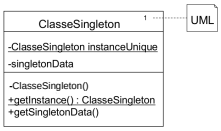
Les patrons du GoF : « Design Patterns » (DP)

- **23 patrons de conception** (interactions de composants)
- **Objectifs :** éviter les erreurs classiques de conception, limiter la complexité du code, ...
- **Catégories de Design Patterns :**
  1) **modèles de création :** politique de création des instances d'objets (déléguée à 1 classe particulière), *ex : Singleton, Factory, Builder, …*
  2) **modèles de structuration :** concevoir ses classes pour qu'elles assument un rôle particulier, *ex : Adapter, Proxy, Composite, …*
  3) **modèles de comportement :** comment les classes communiquent / se répartissent les responsabilités, *ex : State, Strategy, Observer, …*

*Design Patterns – Elements of Reusable Object-Oriented Software.*E. Gamma, R. Helm, R. Johnson & J. Vlissides. 1995

DP de construction/création

- **Abstraire** les mécanismes de création d'objets
- Ces patrons :
  - encapsulent l'utilisation des classes concrètes,
  - favorisent l'utilisation des interfaces dans les relations entre objets,
  - augmentent les capacités d'abstraction dans la conception globale du système.
- Ex : pattern **Singleton**
  - classe possédant au maximum une instance,
  - accès unique/global à l'instance : encapsulé dans la classe,
  - *ex. : connexion au serveur de BD (application web dynamique)*




```
public class DataAccess {
     private static DataAccess _instance = null;

     public static DataAccess getInstance() {
          if(_instance == null)
               _instance = new DataAccess();
          return _instance; //renvoie la seule instance de la classe
     }
}
```


DP Factory Method (Fabrique) : problème

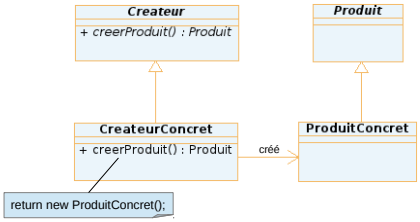
- Classe qui instancie différents types d'objets suivant un paramètre
  ex. : objets dérivés de la classe Animal en fonction de son nom :

```
public class FabriqueAnimal {
  Animal create(String typeAnimal) throws AnimalCreationException {
    if(typeAnimal.equals("chat")) { return new Chat(); }          //succession de conditions
    else if(typeAnimal.equals("chien")) { return new Chien(); }
    throw new AnimalCreationException("Impossible de créer un " + typeAnimal);
  }
}
```

- **Pb :** classe fortement **couplée** à tous les produits qu'elle peut instancier (fait appel à leur type concret)
  - ce code va évoluer (ajout de nouveaux animaux à fabriquer ou suppression de certains animaux obsolètes),
  - instanciation des différents produits réalisée dans d'autres classes (ex. présenter un catalogue des animaux fabriqués) → code dupliqué.

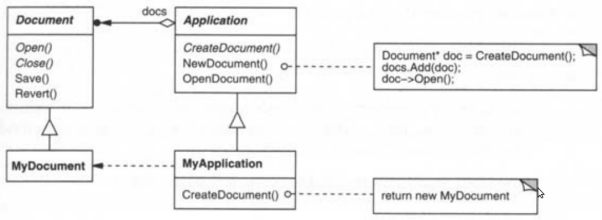
DP Factory Method (Fabrique) : solution

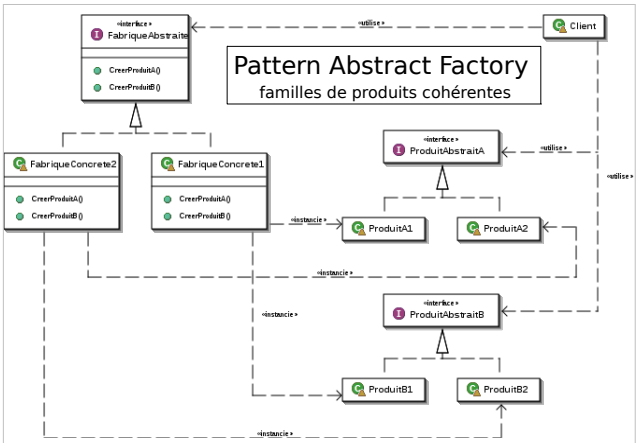
- **But** : introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective.
- **Intérêt** :
  - découpler les clients des classes concrètes à instancier,
  - si l'on ne connaît pas à l'avance toutes les classes à instancier



DP Factory Method (Fabrique) : exemple

Ex. : une application veut manipuler des documents





Source : http://fr.wikipedia.org/wiki/Fabrique_abstraite_(patron_de_conception)

DP Abstract Factory

Fabrique abstraite

en fonction de l'OS, créer la fabrique

Produits concrets A1 et A2

créerProduit

Fabriques concrètes

Produit abstrait A

```
class WinButton extends Button
{
    public void paint()
    {
        System.out.println("I'm a WinButton: "+ getCaption());
    }
}

class OSXButton extends Button
{
    public void paint()
    {
        System.out.println("I'm an OSXButton: "+ getCaption());
    }
}

public class Application
{
    public static void main(String[] args)
    {
        GUIFactory aFactory = GUIFactory.getFactory();
        Button aButton = aFactory.createButton();
        aButton.setCaption("Play");
        aButton.paint();
    }

    //output is
    //I'm a WinButton: Play
    //or
    //I'm a OSXButton: Play
}
```

```
public abstract class GUIFactory
{
    public static GUIFactory getFactory()
    {
        int sys = readFromConfigFile("OS_TYPE");

        if (sys == 0)
            return(new WinFactory());

        return(new OSXFactory());
    }
    public abstract Button createButton();
}

class WinFactory extends GUIFactory
{
    public Button createButton()
    {
        return(new WinButton());
    }
}

class OSXFactory extends GUIFactory
{
    public Button createButton()
    {
        return(new OSXButton());
    }
}

public abstract class Button
{
    private String caption;

    public String getCaption()
    {
        return caption;
    }

    public void setCaption(String caption)
    {
        this.caption = caption;
    }

    public abstract void paint();
}
```

Source : http://fr.wikipedia.org/wiki/Fabrique_abstraite_(patron_de_conception)

DP Prototype et Builder

- **Prototype :** créer de nouveaux objets par duplication d'objets existants (prototypes) qui disposent de la capacité de clonage
  - `Client` demande à un ou plusieurs `Prototypes` de se dupliquer eux-mêmes,
  - mis en œuvre en Java via l'interface *Cloneable* et la méthode `clone()` de la classe **Object.**
- **Builder :** abstraire la construction d'objets complexes (créés en plusieurs parties) de leur implantation
  - segmenter un processus de création complexe en traitements unitaires, pouvant servir dans différents contextes,
  - isoler le code de construction d'un objet complexe de la représentation de ses constituants (pour les faire évoluer indépendamment).

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception

2- Patrons de conception
- Généralités sur les patrons
- Patrons architecturaux : MVC et couche
- DP de construction/création
- **DP de structuration**
- DP de comportement

3- Automatisation en Java avec Ant et Maven

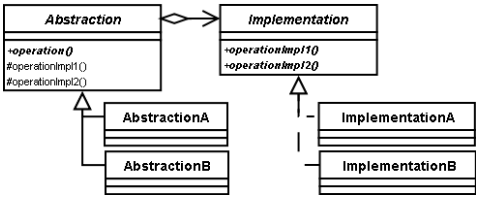
4- Intégration continue avec Jenkins

Patrons de structuration

- **Indépendance** de l'interface d'un objet ou d'un ensemble d'objets vis-à-vis de son **implantation**
  - objet(s) indépendant(s) de la hiérarchie des classes,
  - **encapsulation de la composition** des objets :
    - transférer la **structuration** de l'objet à un second objet
    - premier objet détient l'**interface** vis-à-vis des clients et gère la relation avec le second objet
    - second gère la **composition** et n'a aucune interface avec les clients
  - souplesse de la **composition** qui peut être modifiée dynamiquement
- Séparation des classes alors qu'on aurait tendance à tout mettre dans une seule !

Séparer contrat et implantation : DP Bridge

- Découpler l'abstraction d'un concept de son implantation
- Permettre à l'abstraction et l'implantation de varier indépendamment

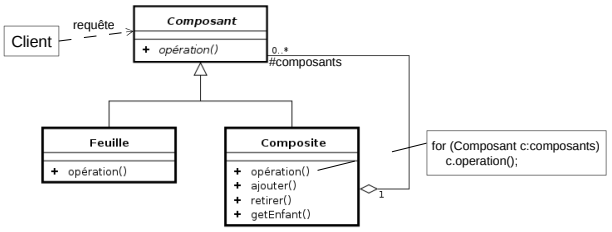


- ex. : Application de dessin géométrique avec une abstraction `FormeGeometrique` et une interface `TechniqueDeColoriage`

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/?page=page_3#LV-B

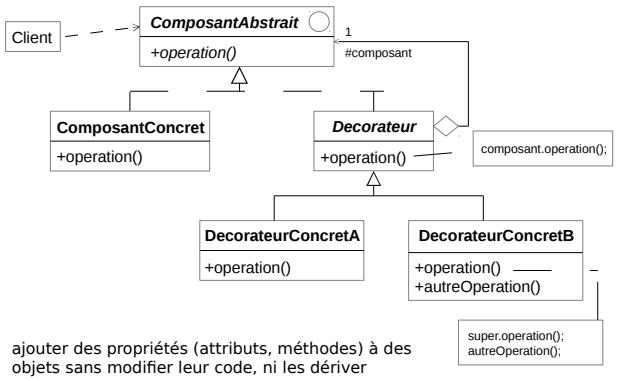
Représenter une hiérarchie : DP Composite

Organiser les objets en structure arborescente :
- `Client` manipule les objets unique et composé de la même manière,
- lorsqu'un composant reçoit une requête, il réagit en fonction de sa classe.



Source : http://fr.wikipedia.org/wiki/Objet_composite

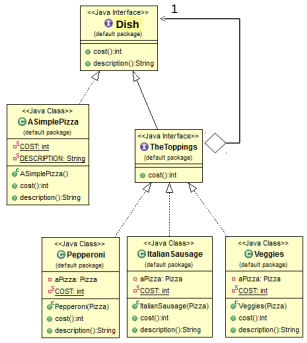
DP Decorator : alternative souple à l'héritage



ajouter des propriétés (attributs, méthodes) à des objets sans modifier leur code, ni les dériver

DP Decorator : exemple

- différents types de pizzas
- **héritage** : créer énormément de classes qui correspondent à toutes les combinaisons possibles de garnitures
- préférable d'utiliser un **décorateur** pour ajouter une ou plusieurs garnitures sur une pizza de base → mais peut conduire à créer beaucoup d'objets !



Autres DP de structuration

- **Adapter** : convertit l'interface d'une classe existante en l'interface attendue par des clients
  - élément non compatible avec l'architecture (ex : transformateur de courant)
  - problème d'incompatibilité d'interfaces (API)
- **Facade** : regroupe les interfaces d'un ensemble d'objets en une interface unifiée rendant l'ensemble plus simple à utiliser
  - accès à un sous-système en cachant son fonctionnement
  - ex. : `DateFrancaise` spécialisée dans la génération des dates en français
- **Flyweight** : facilite le partage d'un ensemble important d'objets à fine granularité (ex. : un texte contient des milliers de caractères)
- **Proxy** : fournit à un objet un remplaçant pour gérer l'accès à cet objet (accès indirect pour des besoins d'accès distant, optimisé, sécurisé)

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception

2- Patrons de conception

- Généralités sur les patrons
- Patrons architecturaux : MVC et couche
- DP de construction/création
- DP de structuration
- **DP de comportement**

3- Automatisation en Java avec Ant et Maven

4- Intégration continue avec Jenkins

DP Strategy : ne plus dépendre d'un algorithme

- Si un objet peut effectuer plusieurs traitements différents (dépendant d'une variable ou d'un état),
- et qu'on veut pouvoir permuter dynamiquement ces traitements dans une application (ex. machine à café)
- ex. : outil de compression de fichiers gérant divers algo. de compresion (zip, rar, ...)

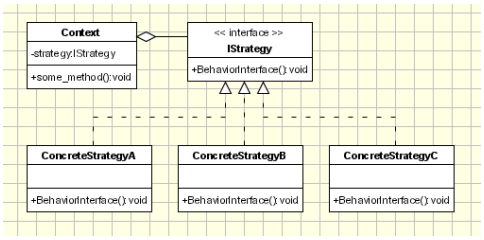
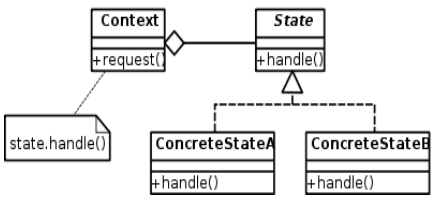


Schéma tiré de : http://www.oodesign.com/strategy-pattern.html

DP State : comportement lié à l'état d'un objet

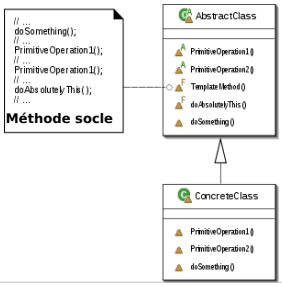
- Permet à un objet d'adapter son comportement en fonction de son état interne (ex. : distributeur automatique = machine à état)
- Se rapproche du pattern **Strategy**
  - ex. : classe `Commande` dont les instances peuvent avoir les états `EnCours`, `Validée` et `Livrée`
  - plutôt que d'avoir de **multiples conditions** dans le corps des méthodes



DP Template Method : délégation d'étapes

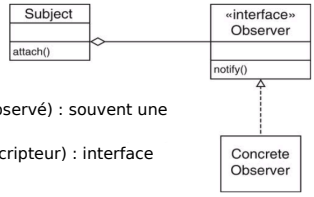
Reporter dans des sous-classes certaines étapes d'une opération d'un objet, ces étapes étant alors décrites dans les sous-classes :

- ex. : calcul du montant TTC d'une commande en tenant compte des différentes TVA (en fonction du pays),
- factorise du code qui serait redondant s'il se trouvait répété dans chaque sous-classe.



Être à l'écoute de ses objets : DP Observer

- Dépendance entre un **sujet** et des **observateurs**
  - chaque modification du sujet est notifiée aux observateurs
  - afin qu'ils puissent mettre à jour leurs états
- Utilisé pour les IHM (**modèle événementiel** en Java)
- Le sujet peut enregistrer dynamiquement les souscripteurs intéressés par un événement et le leur signaler



Sujet (diffuseur / observé) : souvent une classe abstraite

Observateur (souscripteur) : interface

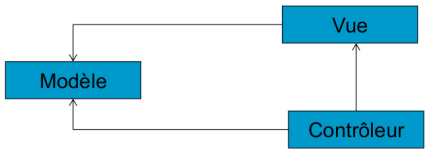
Patron MVC : différentes versions

- la vue connaît ou non le modèle (pour récupérer l'état du système)
- le contrôleur connaît ou non la vue (pour sélectionner la plus pertinente)
- le vue connaît ou non le contrôleur (pour lui transmettre les événements)
- utilisation du pattern **Observer**
- utilisation du pattern **Strategy** (plusieurs contrôleurs), ...

- Choix d'une solution dépend :
  - des caractéristiques de l'application,
  - des autres responsabilités du contrôleur.
- Autre vision appelée « Model-View-Adapter »
  - le contrôleur agit comme un **médiateur** (DP),
  - aucune interaction directe entre la vue et le modèle.

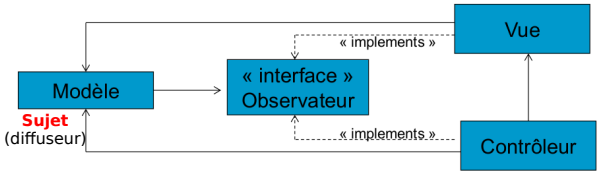
MVC : version modèle passif

- la vue se construit à partir du modèle
- le contrôleur notifie le modèle des changements que l'utilisateur spécifie dans la vue
- le contrôleur informe la vue que le modèle a changé et qu'elle doit se reconstruire



MVC : version modèle actif

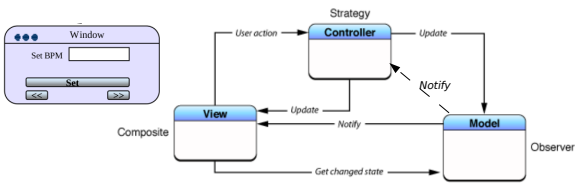
- le modèle peut changer indépendamment du contrôleur
- le modèle informe les abonnés à l'observateur qu'il s'est modifié
- ceux-ci prennent l'information en compte (contrôleur et vues)



MVC : patron composé

- DP **Observer** : vues observateurs des modèles (et contrôleurs)
- DP **Strategy** : contrôleur = stratégie de la vue pour prévenir les risques de changement dans la logique de contrôle.

  En Java, contrôleurs = listeners des événements générés sur les vues (ajoutés aux vues, aux composants graphiques)
- DP **Composite** : vues souvent décomposées hiérarchiquement



Autres DP de comportement

- **Chain of responsability** (chaîne d'objets) : si l'un d'eux ne peut pas répondre à une requête, il la transmet à ses successeurs
- **Command** : transforme une requête en un objet, facilitant les opérations d'annulation, de mise en file des requêtes, de suivi
- **Interpreter** : donne une représentation par objet de la grammaire d'un langage afin d'évaluer des expressions écrites dans ce langage
- **Iterator** : fournit un accès séquentiel à une collection d'objets sans que les clients se préoccupent de l'implantation de cette collection
- **Mediator** : construit un objet dont la vocation est la gestion et le contrôle des interactions au sein d'un ensemble d'objets (sans que ses éléments se connaissent mutuellement)
- **Memento** : sauvegarde et restaure l'état d'un objet (conservation de points de reprise, *ex. : jeux vidéo*)
- **Visitor** : classe dont le rôle est de parcourir un ensemble d'éléments et d'y opérer des modifications en fonction de leur type (ou du contexte d'utilisation)
  ex. : fleurs et insectes qui n'en butinent que certaines

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception

2- Patrons de conception

3- Automatisation en Java avec Ant et Maven

4- Intégration continue avec Jenkins

Automatisation : pourquoi ?

```
#!/bin/sh
javac ...
cp ...
echo "Terminé !"
```

```
src
  main
    java
    resources
  test
    java
pom.xml
```

- Accélérer les tâches répétitives
- Réduire le risque d'erreurs
- Différentes approches :
  - **Scripts** : *shell / batch, Ant, Grunt.js, Gradle ...*
    - *Avantage :* aucune limite (on fait ce qu'on veut !)
    - *Inconv. :* dépendant de l'OS, pas de standard, complexe à échanger
  - **Métadonnées** : *Maven, Gradle ...*
    - *Avantages :* flexible, extensible, cadre **standard** (organisation prédéfinie), génération automatique des tâches en s'appuyant sur l'organisation
    - *Inconv. :* temps d'apprentissage

Ant : script de build au format XML

- **Caractéristiques :**
  - **Open Source** (*Apache Public Licence*), **simple**,
  - écrit en **Java** (multi-plateforme),
  - **intégré** dans les **principaux IDE**
- **Principes :**
  - déclarer 1 **projet** (project) composé de **«cibles»** (target)
  - **cible** = enchaînement de tâches unitaires (**tasks** : *compile, jar, javadoc, ftp, junit, cvs, ...*)
  - cibles et leurs dépendances décrites dans un **fichier de config. XML**
  - **extensible :** on peut ajouter ses propres commandes / tâches (*codées sous forme de classes Java*)

Maven 2 et 3 : outil Open Source de build

Automatiser la **gestion/construction** de projets Java :

- **extensible** (nombreux plugins), écrit en Java,
- utilisé par la majorité des projets d'entreprise,
- approche **déclarative** (basée sur le cycle de vie du projet),
- propose un **cadre standard** (approche par métadonnées)
  - organisation des sources, tests unitaires, exécutables
  - description de l'équipe, du référentiel de sources
  - gestion des dépendances et dépendances transitives

Maven : structure de répertoires

src

- **main**
  Principales sources du programme
  - **java**
    Sources Java
  - **resources**
    Fichier non Java utilisé dans le cadre de l'exécution du programme
- **test**
  Fichiers relatifs aux tests unitaires
  - **java**
    Sources des tests unitaires
  - **resources**
    Fichiers non Java utilisés uniquement dans le cadre des tests unitaires

target
Résultat de la compilation

Maven : comment l'utiliser ?

pom.xml : là où est décrit le projet maven

<build> : décrit les fonctions relatives à la construction d'1 projet (classpath, version de java...)

<profiles> : profils de construction, pour personnaliser la totalité du **pom** selon des identifiants de profil

<dependencies> : liste les dépendances selon 1 format « qualifié »

- **groupId** : nom d'un groupe (en général : préfixe du package principal)
- **artifactId** : nom de la dépendance (en général : nom du projet)
- **version** : version du projet
- **type** : format (jar, pom...)
- **scope** (périmètre/portée de la dépendance) :
  - à la compilation (compile), uniquement à l'exécution (runtime),
  - pendant la compilation et l'exécution des tests (test), ...

pom.xml basique (POM = Project Object Model)

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

cœur d'un projet Maven : description détaillée du projet

SNAPSHOT : à 1 moment donné du développement

Dépendances : ex. tests unitaires
Maven gère les autres dépendances liées à JUnit

Placé dans le répertoire de base de votre projet

Phases du cycle de vie du projet

Utilitaire en ligne de commande, à lancer depuis l'emplacement du fichier *pom.xml* : **mvn [goal]**

avec [goal] = (invoque les plugins nécessaire pour faire le travail)

- **generate-sources** : génère le code source supplémentaire nécessité par l'appli *(accompli par les plugins appropriés)*
- **clean** : supprimer les éléments précédemment construits
- **compile** : compile le code source du projet
- **test-compile** : compile les tests unitaires du projet
- **test** : lancer/exécuter les tests unitaires (avec Junit)
- **package** : packager le code compilé (un WAR, un JAR)
- **Install** : installer le JAR sur le dépôt local
- **deploy** : déployer l'application sur le serveur cible (pour être partagé avec d'autres développeurs)
- ...

CM5 : DP, automatisation et IC

1- Principes et règles de bonne conception

2- Patrons de conception

3- Automatisation en Java avec Ant et Maven

4- Intégration continue avec Jenkins

Intégration continue (IC) : moyens et outils

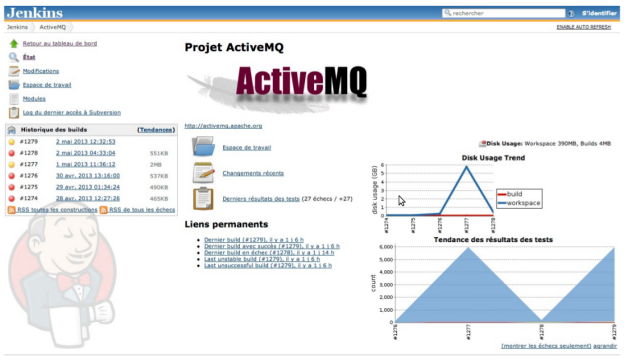
- Pour que l'IC puisse se faire correctement, il faut :
  - partager les sources du projet (SVN, Git)
  - « commiter » régulièrement les modifications (plusieurs fois / jour)
  - disposer de tests unitaires (vérifier la non régression)
- « A la main » *(non recommandé)* : *script / Ant*
- Basique : *Maven* (deploy = déploiement), *Gradle*
- Outillé *(via une IHM)* :
  - **Hudson** : ancêtre de Jenkins
  - **Jenkins** : standard actuel du marché, multi-langages, simple d'accès, à installer, à comprendre
  - **CruiseControl** : moins «user-friendly» que Jenkins, mais répandu
  - **Continuum** : porté par Apache

Principales étapes

déclencher un processus de construction, déploiement

1. Déclencher le processus	1. cron ...
2. Mise à jour des sources	2. svn update ...
3. Compilation	3. javac ...
4. Tests unitaires	4. java -cp junit.jar ...
5. Construction	5. jar -cvf monAppli.war ...
6. Déploiement	6. copy ...

Jenkins : détail d'un job (file d'attente des builds)



Refactoring régulier

- **Objectif :** réécrire/restructurer du code sans en modifier le comportement
- Transformations par petites étapes :
  - éliminer la **duplication** de code → Factoriser
  - améliorer la **lisibilité** → Renommer
  - **raccourcir** les méthodes trop longues → Pliage
  - **extraire** des constantes à partir des valeurs **codées en dur**
  - déplacer du code, changer la signature d'une méthode, ...
- **Attention :** réexécuter les tests après chaque étape

Ex. de plugins : outil Sonar (logiciel libre)

- Outil de suivi de la qualité d'un logiciel (analyse du code) **:**
  - identifie : **code dupliqué, non utilisé, sous-optimisé,**
  - mesure le **niveau de documentation du code** (*Javadoc*),
  - respect des **règles de programmation** (bonnes pratiques de codage),
  - détecte des **bugs** potentiels (*FindBugs*),
  - évalue le niveau de **succès des tests,** la **couverture de code** par les tests unitaires, le **niveau de complexité (taille)** des classes et méthodes,
  - analyse le **design** et l'**architecture** d'une application pour en faire ressortir des **métriques** orientées objet.
- Résultats de l'analyse : stockés dans une BD
- Niveau de criticité des alertes indiqué dans des rapports

Outil Checkstyle : contrôle de code

- Vérifier le **style d'un code source** écrit en Java
- Ensemble de **modules**, contenant des **règles** *(= notification, avertissement ou erreur*) à configurer
- Exemples de règles :
  - vérifier la présence de **commentaires Javadoc** pour les classes, les attributs et les méthodes,
  - conventions de **nommage** des attributs et des méthodes,
  - limitation du **nb de paramètres** de méthodes, longueur des lignes,
  - **bonnes pratiques** d'écriture de classe,
  - sections de **code dupliqué**,
  - diverses **mesures de complexité** (notamment des expressions), …

Outils de gestion du cycle de vie (Tuleap)

Solution d'**ingénierie logicielle** (« forge »)

Modules disponibles :

- système de **suivi de problèmes** *(exigences, risques, bugs, ...)*
- outil de **gestion de projet** *(rapport d'activité, tableau de bord)*
- gestionnaire de **configuration** logicielle : *CVS, SVN, Git*
- **intégration continue** : *Hudson/Jenkins*
- gestionnaire de **documents**
- **wiki** pour l'écriture collaborative
- messagerie instantanée, listes de diffusion, forums, annonces, ...

Liens : DP, Ant, Maven et Jenkins

- **DP :** http://fr.wikipedia.org/wiki/Patron_de_conception, http://en.wikipedia.org/wiki/Architectural_pattern, http://en.wikipedia.org/wiki/Software_design_pattern, http://www.hillside.net/patterns
- **Pattern MVC :** http://baptiste-wicht.developpez.com/tutoriels/conception/mvc/
- **Ant :** http://ant.apache.org/, http://www.onjava.com/pub/a/onjava/2001/02/22/open_source.html?page=1
- **Maven2 :** http://dcabasson.developpez.com/articles/java/maven/introduction-maven2/
- **IC avec Jenkins (Hudson) :** http://linsolas.developpez.com/articles/hudson/, http://jenkins-le-guide-complet.github.io/, http://blog.fabianpiau.com/2009/07/17/continuous-integration/

Livres :

- Ouvrage du « Gang of Four » : **Design patterns, Elements of Reusable Object-Oriented Software**, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, (1994)
(en français : **Design patterns. Catalogue des modèles de conception réutilisables**, Vuibert 1999)

- **Design Patterns en Java.** Les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java [3e éd.], L. Debrauwer (2013)
- **Conception d'applications en Java/JEE.** Principes, patterns et architectures, J. Lonchamp, DUNOD (2014)
- *Plus orienté architecture :* **Patterns of Enterprise Application Architecture**, Martin Fowler, Addison Wesley (2002)
- **Design Patterns – Tête la première**, E. Freeman, E. Freeman, K. Sierra, B. Bates, O'Reilly Eds. (2005)
- **Apache Maven** - Maîtrisez l'infrastructure d'un projet Java EE, M. GREAU, Epsilon (2011)