GENIE LOGICIEL ORIENTE OBJET

ESISAR

Emmanuel CHENU
Développeur logiciel à THALES Avionics
emmanuel.chenu@fr.thalesgroup.com

Mais au fait …

Pourquoi ce thème n'est pas abordé avec le reste du cours?

Car la rédaction du cours a suivi une **démarche itérative et incrémentale**.

- *« Un tiens vaut mieux que deux tu l'auras. »*
- Il vaut mieux tenir le délai avec un contour réduit que livrer le contour complet en retard.

Objectifs

- Comprendre **l'intérêt** de le démarche de développement orientée objet.
- Identifier les **principales caractéristiques** du développement orienté objet.

Démarche du cours

Tout comme on a dû

en passer par le **cycle en V**

pour comprendre l'intérêt du **cycle itératif et incrémental**,

il faut

parler de la démarche de développement par **découpe fonctionnelle descendante et programmation procédurale**

pour comprendre l'intérêt de la démarche de développement **orientée objet**.

Quelques mises au point

- **Analyse** : investigation du problème et des besoins plutôt que la recherche d'une solution.
  - *« C'est **quoi** le problème? »*
  - *« C'est **quoi** le besoin? »*
  - *« De **quoi** parle-t-on ?»*
- **Conception**: élaboration d'une solution conceptuelle répondant aux besoins plutôt que la mise en œuvre de cette solution.
  - *« **Comment** résout-on le problème? »*
  - *« **Comment** répond-on au besoin? »*
- **Implémentation**: la solution conceptuelle est mise en œuvre en code source.

Découpe fonctionnelle descendante *1/9*

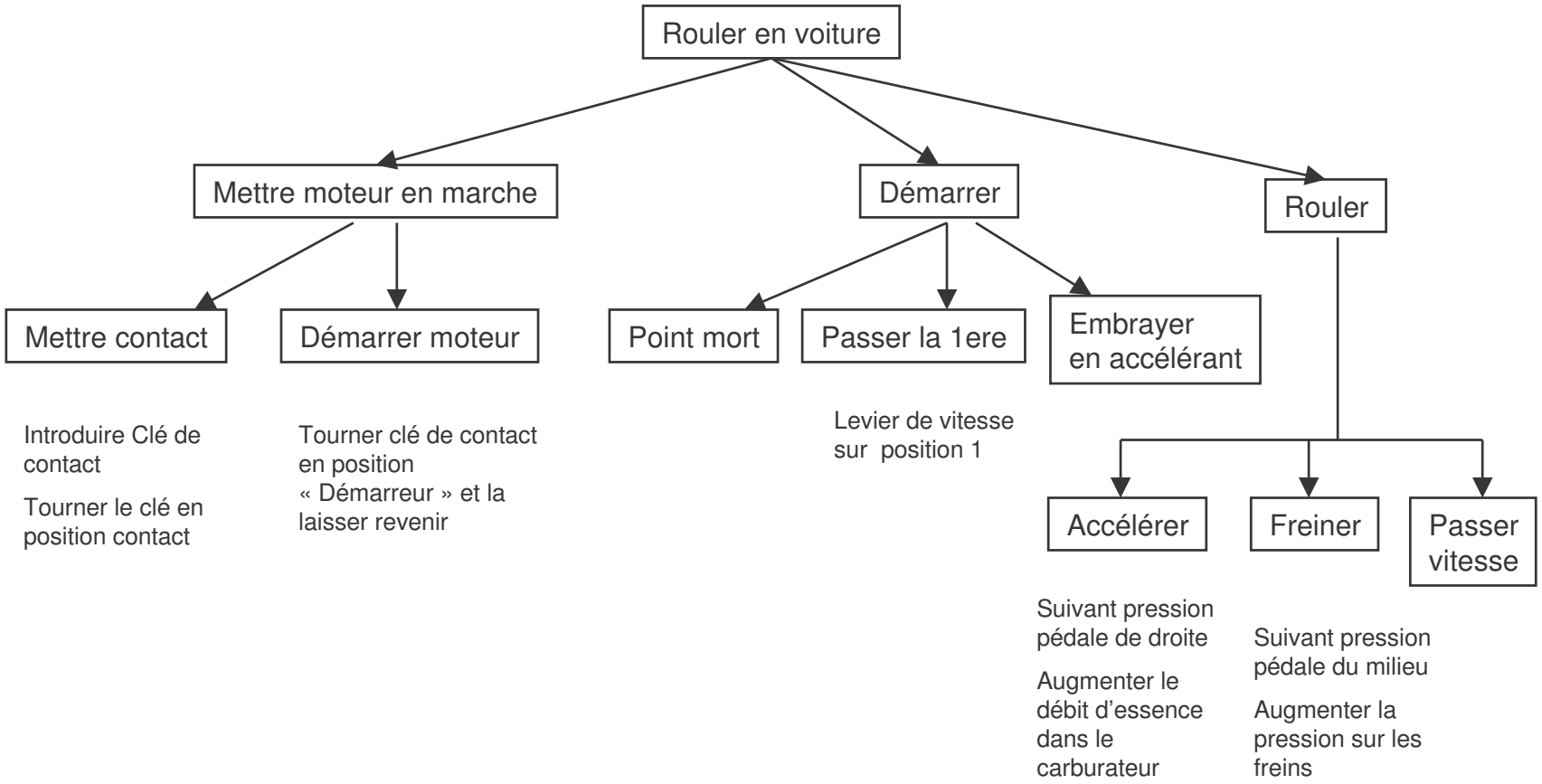
La forme la plus immédiate pour décrire un travail à effectuer est de lister les **actions à réaliser**.

On découpe une tâche complexe à effectuer en une **hiérarchie d'actions** à réaliser de plus en plus simples, petites et précises:

1. **Rouler** En Voiture:
    1. **Mettre** moteur en marche:
        1. **Mettre** Contact
        2. **Démarrer** Moteur
    2. **Démarrer** Voiture:
        1. **Mettre** Point Mort
        2. **Passer** La Première
        3. **Embrayer** En Accélérant
    3. **Rouler**:
        1. **Accélérer**
        2. **Freiner**
        3. **Passer** Vitesse

Pour décrire, on utilise le **verbe**.

Découpe fonctionnelle descendante *2/9*



Découpe fonctionnelle descendante *3/9*

L'implémentation en code source d'une solution décrite en terme d'actions est un **code organisé en fonctions** (= **programmation procédurale**):

```
roulerEnVoiture()
    mettreMoteurEnMarche()
        mettreContact()
        demarrerMoteur()
    demarrer()
        mettrePointMort()
        passerLaPremiere()
        embrayerEnAccelerant()
    rouler()
        accelerer()
        freiner()
        passerVitesse()
```

Découpe fonctionnelle descendante *4/9*

Dans ce cadre de travail:

- L'**analyse** est une découpe fonctionnelle descendante des fonctionnalités à pourvoir.
- La **conception** est une découpe du logiciel en une hiérarchie descendante d'actions permettant de satisfaire les fonctionnalités à pourvoir.
- L'**implémentation** est une programmation procédurale.

Découpe fonctionnelle descendante *5/9*

Dans une programmation procédurale issue d'une découpe fonctionnelle descendante, **les données et les fonctions travaillant sur ces données sont dispersées dans des modules différents.**

```
type typeVitesse est {pointMort, premiere, seconde, troisieme, quatrieme
   cinquieme, marcheArriere};

vitesseCourante : typeVitesse := pointMort;

procedure passerVitesse(
    vitesseCourante : in out typeVitesse;
    vitesseAPasser : in typeVitesse);

procedure mettreAuPointMort(
    vitesseCourante : in out typeVitesse);
```

Découpe fonctionnelle descendante *6/9*

Dispersion données/fonctions:

- Lorsque le logiciel évolue, il faut faire évoluer les structures de données et les fonctions en parallèle *(probablement dans des modules différents).*
- Maintenir cette cohérence est laborieuse parce que données et fonctions sont dispersées.

La dispersion données/fonctions nuit à l'extensibilité!

Découpe fonctionnelle descendante *7/9*

Une découpe fonctionnelle descendante **privilégie une manière d'utiliser le logiciel**: celle qui a donnée naissance à sa découpe fonctionnelle descendante.

Privilégier une manière d'utiliser le logiciel va le rendre:

- **plus difficile à réutiliser tel quel** dans un autre contexte et
- **plus difficile à faire évoluer** pour d'autres contextes d'utilisation.

De plus, un contexte d'utilisation n'est **pas quelque chose de stable** sur le long terme car les **besoins évoluent**.

Découpe fonctionnelle descendante *8/9*

Autre contexte d'utilisation:

```
type typeVitesseAutomatique est {marcheAvant, pointMort, marcheArrière};
vitesseAutomatiqueCourante : typeVitesseAutomatique := pointMort;
procedure passerVitesseAutomatique(
    vitesseCourante : in out typeVitesseAutomatique ;
    vitesseAPasser : in typeVitesseAutomatique );
```

Possibilité de réutiliser le code existant? Possibilité de faire cohabiter les deux utilisations? Facilité d'évoluer d'une utilisation vers l'autre?

La découpe fonctionnelle initiale d'un logiciel privilégie le contexte d'utilisation initial ce qui nuit à son extensibilité et sa réutilisation!

Couplage/Cohésion *1/6*

A RETENIR ABSOLUEMENT

Pour qu'un logiciel soit **extensible** et **réutilisable**, il faut qu'il soit **découpé en modules**

- **faiblement couplés** et
- à **forte cohésion**.

Couplage/Cohésion *2/6*

Couplage

*« Une entité (fonction, module, classe, package, composant) est couplée à une autre si elle **dépend** d'elle. »*

Couplage faible

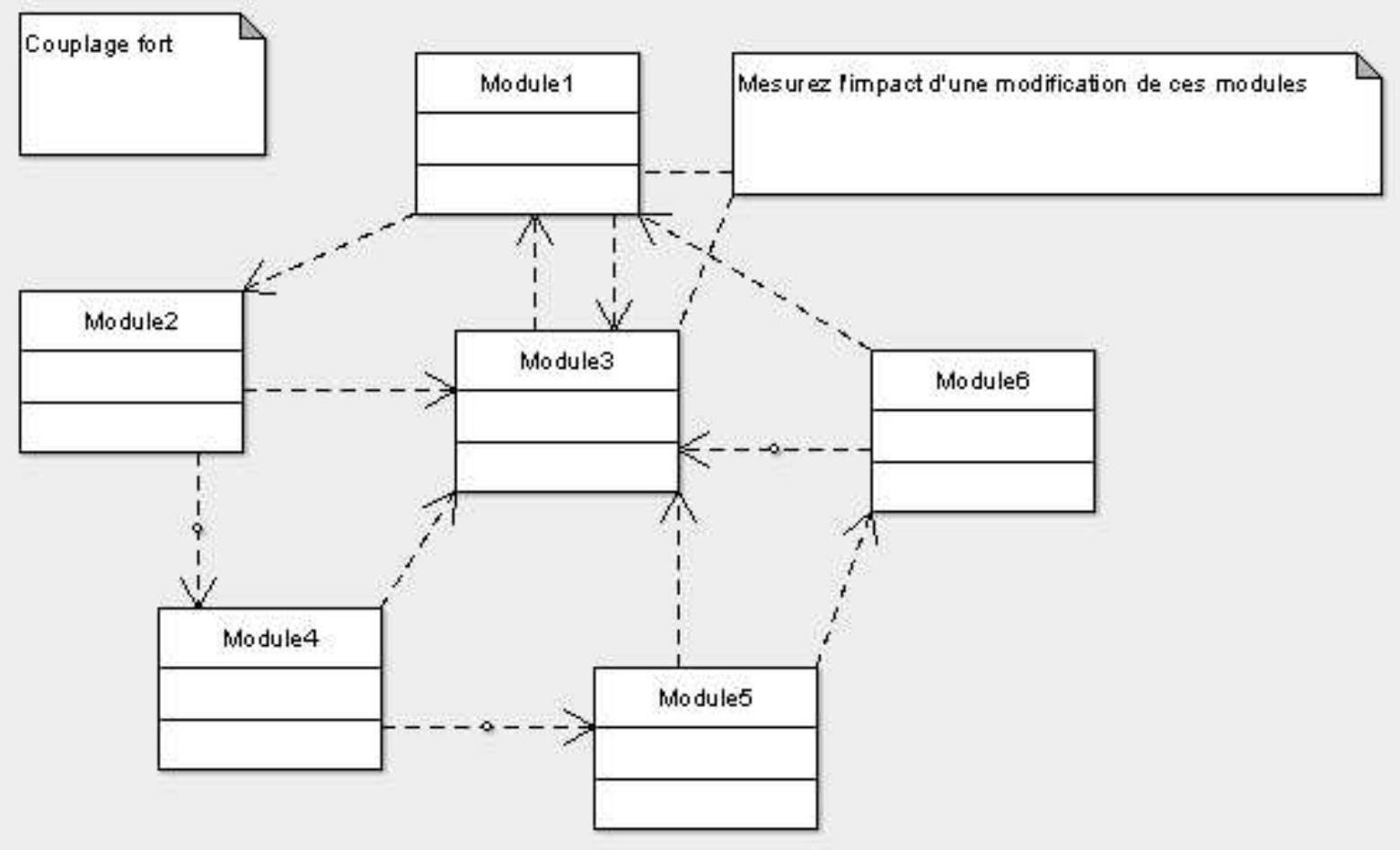
« Désigne une relation faible entre plusieurs entités (classes, composants), permettant une grande souplesse de programmation, de mise à jour.

*Ainsi, chaque entité peut être modifiée en **limitant l'impact du changement** au reste de l'application. »*

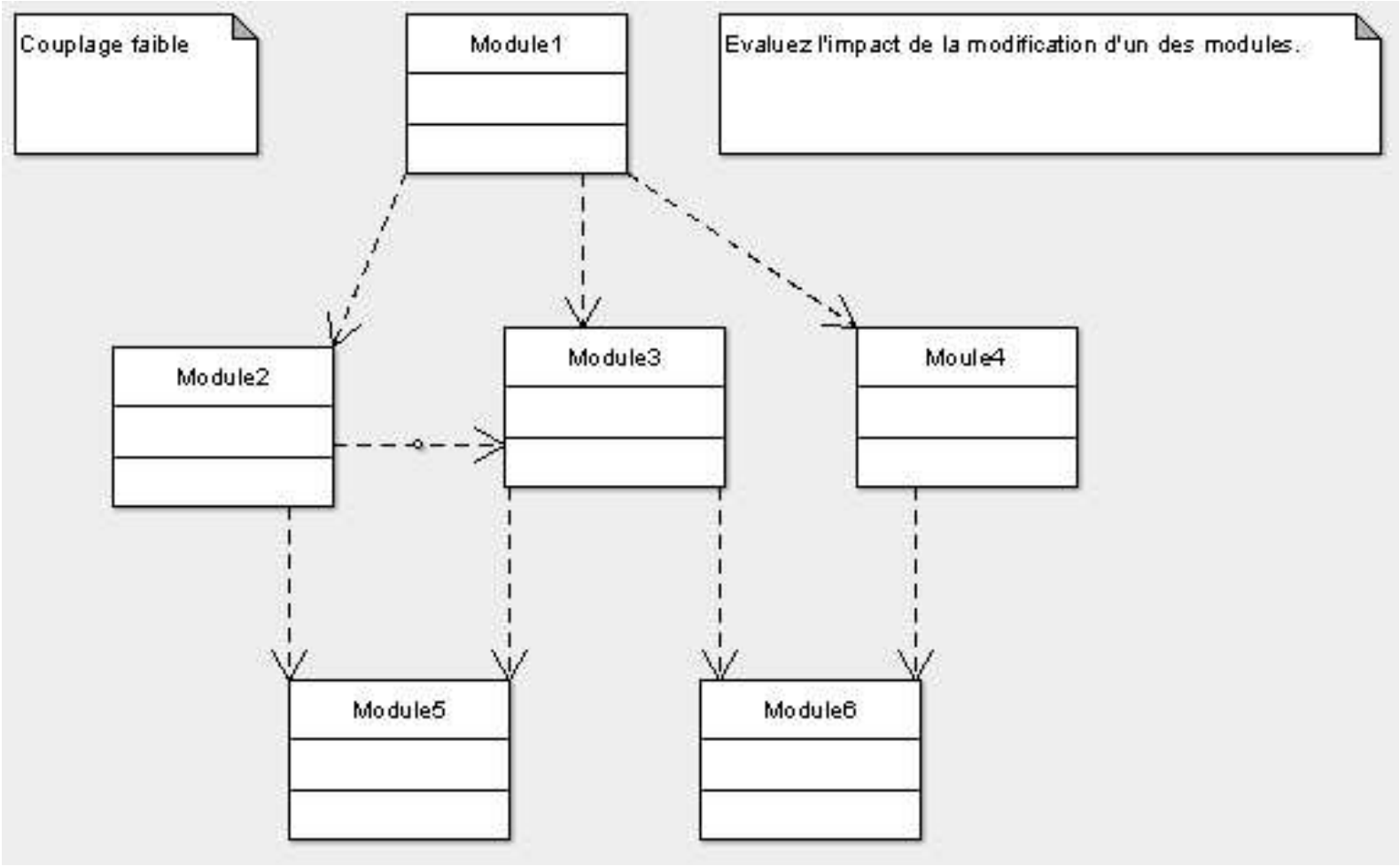
Le couplage fort

« au contraire, tisse un lien puissant qui rend l'application plus rigide à toute modification de code. »

Couplage/Cohésion *3/6*



Couplage/Cohésion *4/6*



Couplage/Cohésion *5/6*

La maitrise du couplage est une **maitrise des dépendances**.

Dépendances en code source:

- C, C++: `#include «…»`
- Ada: `with …;`
- Java: `import …;`

Plus il y a de dépendances, plus la vie du développeur est pénible ...

Couplage/Cohésion *6/6*

Cohésion entre modules dans un composant, entre services dans un module: *« qui se ressemble s'assemble »* ou *« groupir! »*

```
+ tuneFrequency(freq)
+ tuneNextStation()
+ tuneNextFrequency()
+ displayTunedFrequency() // gare à l'intrus
+ getTunedFrequency:freq
```

Il faut réunir ce qui est impacté par une même modification!

Découpe fonctionnelle descendante *9/9*

On y revient …

Comment évaluer la découpe fonctionnelle descendante et la programmation procédurale en termes de **couplage** et **cohésion**?

Séparer données et fonctions sur ces données entraine

⇒ Un **fort couplage** par les données,

⇒ **Perte de cohésion** (par dispersion).

Le code est donc peu extensible et peu réutilisable!

Démarche orientée objet

« Paradigme de programmation informatique qui consiste en la définition et l'assemblage de briques logicielles appelées objet.

Un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. »

⇒ On ne raisonne plus en termes d'**actions** mais plutôt en **concepts du monde physique**. Puisqu'ils appartiennent au monde physique, ces concepts peuvent être **stables** et **réutilisables**.

⇒ On ne raisonne uniquement en **verbes**, mais davantage en **noms**.

Démarche orientée objet

Classe *1/2*

« En programmation orientée objet une classe déclare des propriétés communes à un ensemble d'objets.

*La classe déclare des attributs représentant l'**état** des objets (données) **et** des méthodes (opérations) représentant leur **comportement**. »*

⇒ **Données** et **opérations** traitant les données ne sont pas séparées, mais réunies au sein d'un même module. **Cohésion!**

Démarche orientée objet

Classe *2/2*

*« Une classe représente donc une **catégorie** d'objets. Il apparaît aussi comme un moule ou une usine à partir de laquelle il est possible de créer des objets. On parle alors d'un objet en tant qu'instance d'une classe (création d'un objet ayant les propriétés de la classe). »*

Un **objet** ou une **instance** est une **variable** d'une classe:

```
Voiture maVoiture; // maVoiture est une Voiture;
```

Démarche orientée objet

Le paradigme de la POO **s'oppose** au paradigme de la programmation fonctionnelle:

- En programmation fonctionnelle, les **données** sont séparées des **méthodes** qui les manipulent.

- En programmation OO, les **données** sont **encapsulées** au sein d'un objet. Elles sont manipulées à l'aide de **méthodes** définies dans la classe de l'objet.

Démarche orientée objet

Dans la démarche OO

Avant *(ou en même temps)* qu'on s'occupe de la **résolution du problème**, on doit créer le **monde du problème** *(le domaine)*.

- ⇒ Si ce monde est inspirée du monde physique, alors les classes d'objets seront **stables** et **réutilisables**.
- ⇒ Les objets du monde physique sont plus **stables** que les besoins.
- ⇒ La résolution du problème est une démarche **descendante**.
- ⇒ Créer le monde du problème est une démarche **ascendante**.

Démarches comparées *(1/5)*

Analyse des exigences DigitalTuner par **découpe fonctionnelle descendante** *(1/3)*

- `Incrémenter fréquence captée`
  - `Incrémenter fréquence courante`
  - `Gérer dépassement de la gamme de fréquence`
  - `Envoyer nouvelle fréquence à DigitalRadio`
  - `Afficher fréquence`
  - `Afficher station éventuelle`
  - `Mémoriser fréquence courante`
- `Décrémenter fréquence captée`
  - `Décrémenter fréquence courante`
  - `Gérer dépassement de la gamme de fréquence`
  - `Envoyer nouvelle fréquence à DigitalRadio`
  - `Afficher fréquence`
  - `Afficher station éventuelle`
  - `Mémoriser fréquence courante`

Démarches comparées *(2/5)*

Analyse des exigences DigitalTuner par **découpe fonctionnelle descendante** *(2/3)*

- Capter prochaine station
  - Rechercher prochaine station
    - Gérer dépassement de la gamme de fréquence
  - Envoyer nouvelle fréquence à DigitalRadio
  - Afficher fréquence
  - Lire et afficher station
  - Mémoriser fréquence courante
- Capter précédente station
  - Rechercher précédente station
    - Gérer dépassement de la gamme de fréquence
  - Envoyer nouvelle fréquence à DigitalRadio
  - Afficher fréquence
  - Lire et afficher station
  - Mémoriser fréquence courante

Démarches comparées *(3/5)*

Analyse des exigences DigitalTuner par **découpe fonctionnelle descendante** *(3/3)*

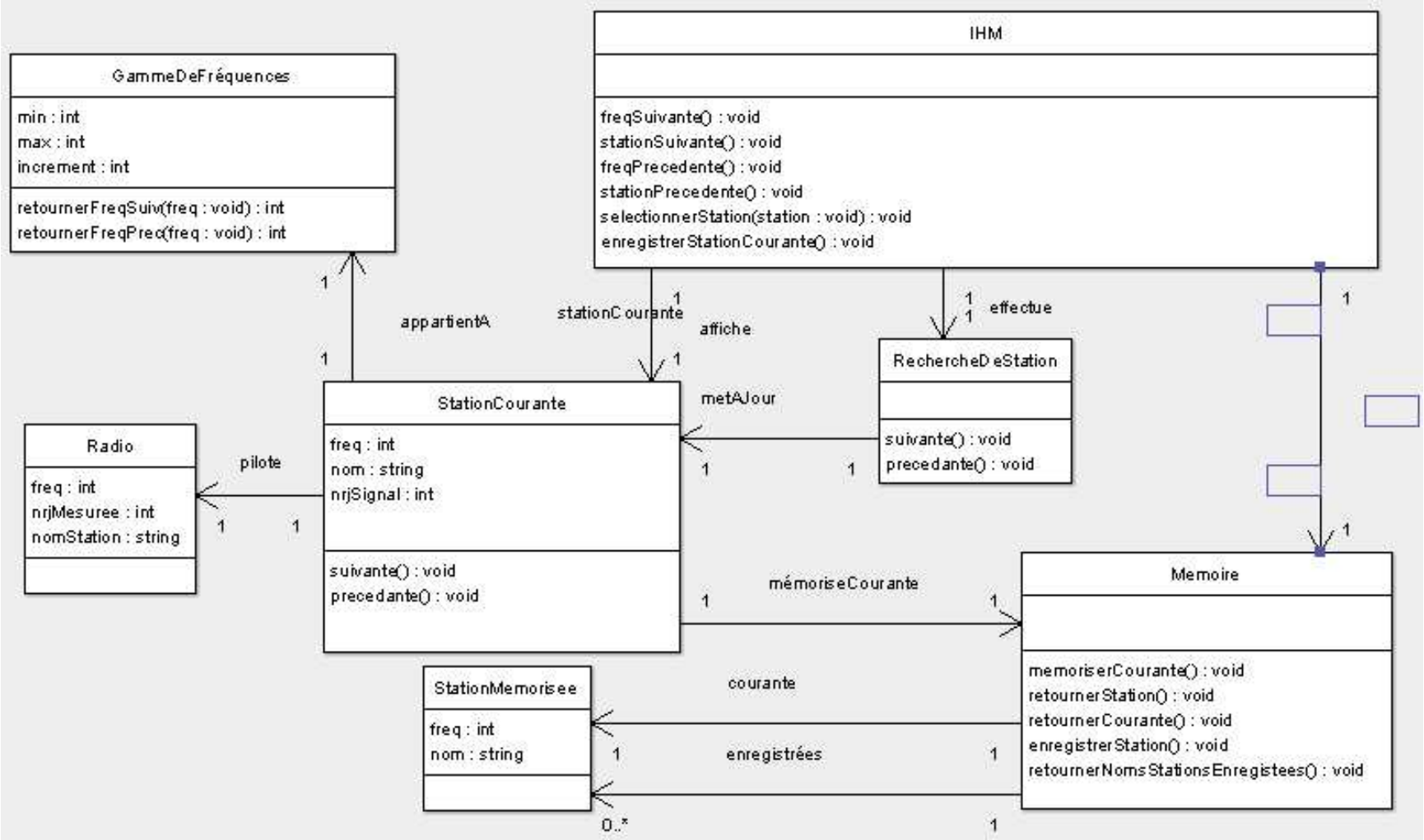
- `Mémoriser station`
  - `Mémoriser station et fréquence courantes`
- `Sélectionner station mémorisée`
  - `Envoyer fréquence mémorisée à DigitalRadio`
  - `Afficher fréquence`
  - `Lire et afficher station`
  - `Mémoriser fréquence courante`

Démarches comparées *(4/5)*

Analyse des exigences DigitalTuner par **démarche orientée objet**

- `Radio`
- `IHM`
- `Requête`
- `Fréquence`
- `GammeDeFrequences`
- `Station`
- `RechercheDeStation`
- `MémoireDeStations`

Démarches comparées *(5/5)*



Démarche orientée objet

Un logiciel orienté objet

- est composé d'objets qui communiquent les uns avec les autres.
- est une **micro-société d'objets communicants**.

Démarche orientée objet

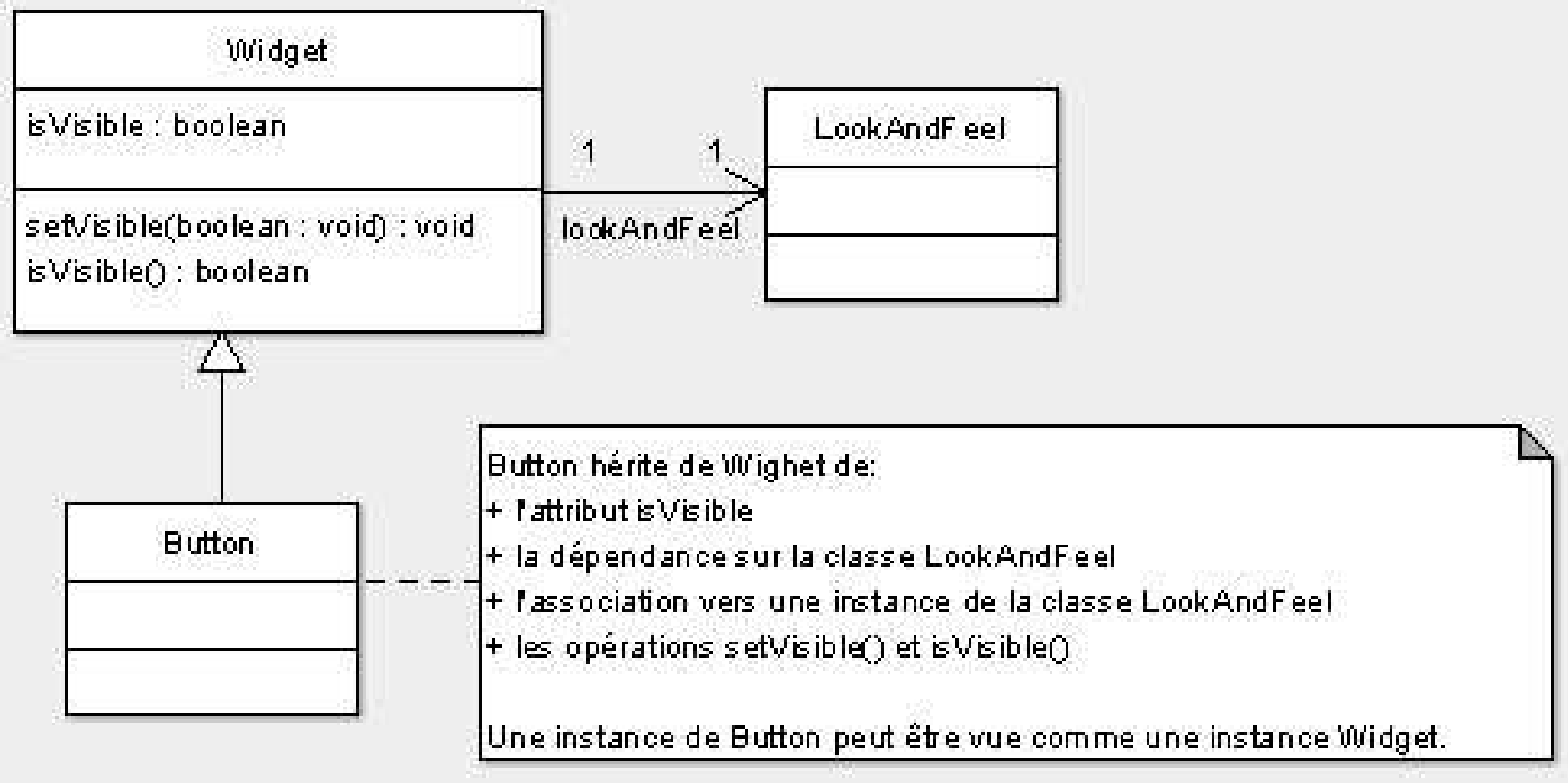
- L'**analyse** orientée objet est davantage tournée vers la description des objets – ou concepts – du domaine du problème.
- La **conception** orientée objet est centrée sur la définition des objets logiciels et sur la façon dont ils collaborent pour satisfaire les besoins.
- L'**implémentation** orientée objet utilise un langage de programmation orienté objet pour traduire en code source la solution issue de la conception orientée objet.

Programmation orienté-objet

3 Principes de base:

1. **Encapsulation**
2. **Héritage**
3. **Polymorphisme**

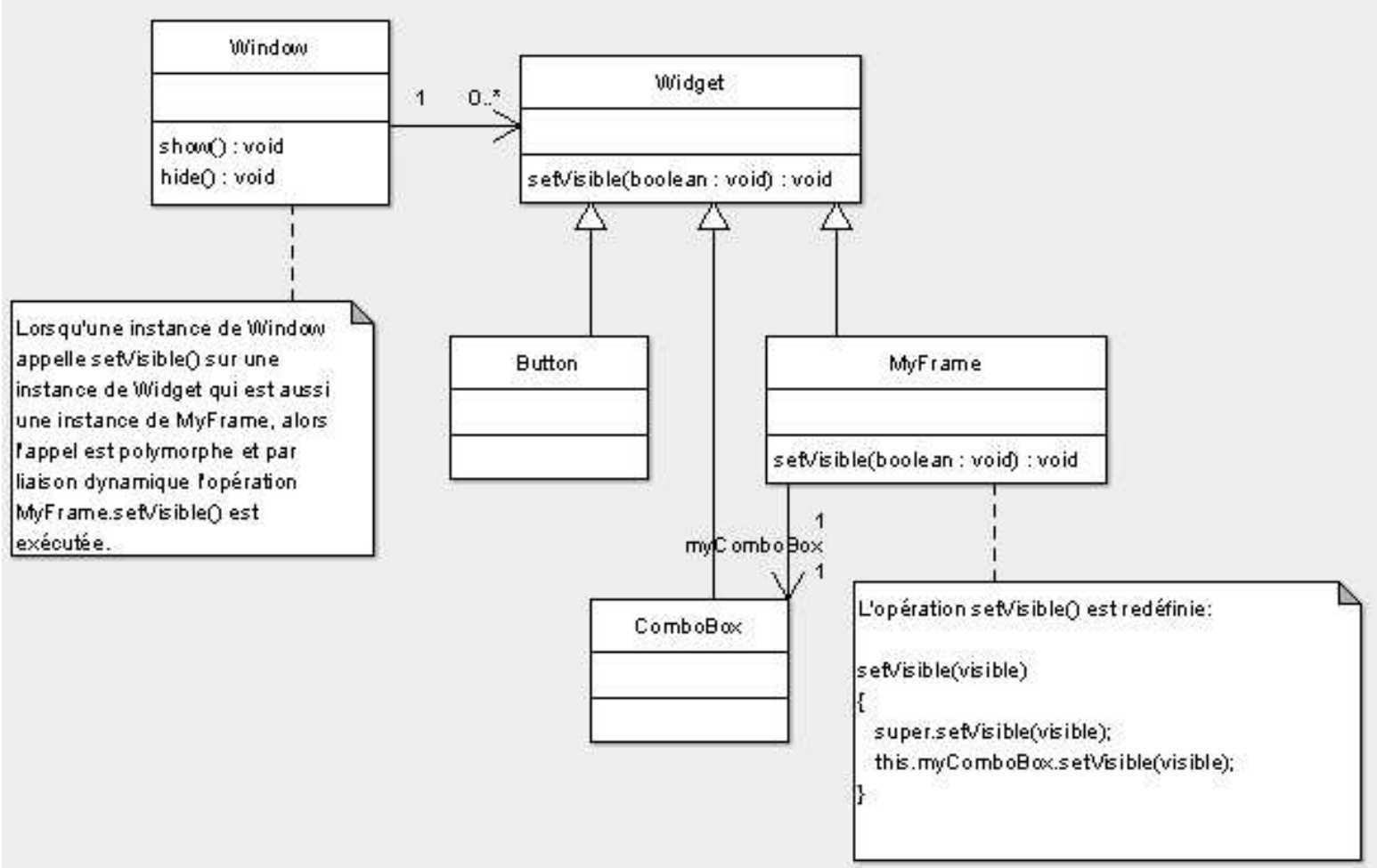
Héritage *(1/2)*


Widget
isVisible : boolean
setVisible(boolean : void) : void
isVisible() : boolean
1
1
lookAndFeel
LookAndFeel
Button
Button hérite de Wighet de:
+ l'attribut isVisible
+ la dépendance sur la classe LookAndFeel
+ l'association vers une instance de la classe LookAndFeel
+ les opérations setVisible() et isVisible()
Une instance de Button peut être vue comme une instance Widget.

Héritage *(2/2)*

```
// Test d'un widget:
void testWidget(Widget w)
{
   w.setVisible(true);
   assert(w.isVisible());
   w.setVisible(false);
   assert(w.isVisible() == false);
}
//
Button myButton = new Button(); // Button inherits Widget.
// Test d'un Button:
myButton.setVisible(true); // Button a hérite de setVisible().
assert(myButton.isVisible()); // Button a hérite de isVisible().
myButton.setVisible(false);
assert(myButton.isVisible() == false);
testWidget(myButton); // Réussit car Button hérite de Widget.
```

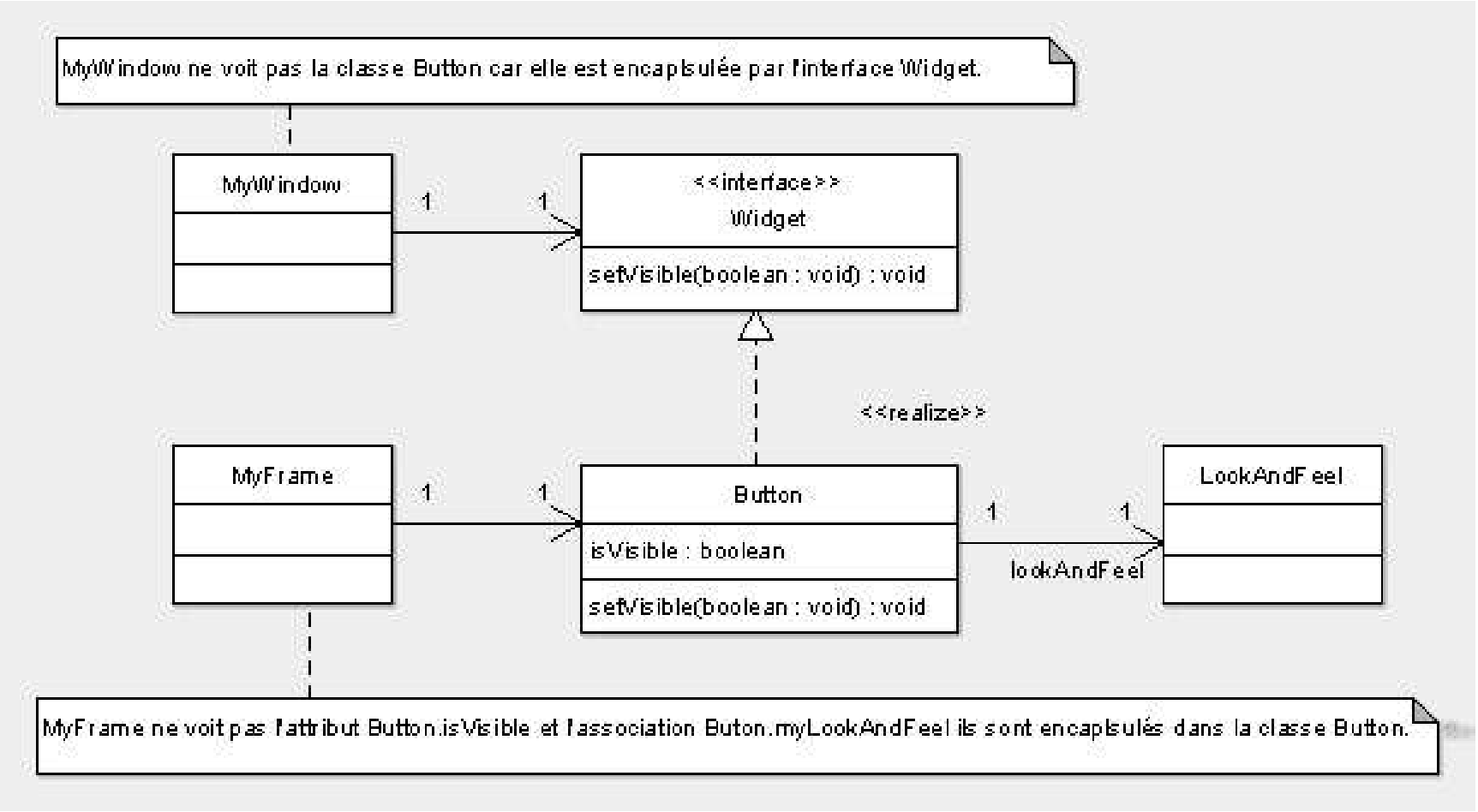
Polymorphisme *(1/2)*



Polymorphisme *(2/2)*

```
Window myWindow = new Window();
Button myButton = new Button(); // Button inherits Widget.
myWindow.add(myButton);
MyFrame myFrame = new MyFrame(new ComboBox()); // inhert
myWindow.add(myFrame);
// extract of class Window:
void add(Widget w) {…}
void hide() {
   Iterator it = this.widgets.iterator();
   while (it.hasNext()) {
      Widget widget = (Widget)it.next();
      widget.setVisible(false); // myButton et myFrame
   }
}
```

Encapsulation



Principes avancés de conception objet

En matière de développement logiciel, on constate aujourd'hui que le design reste principalement une affaire de style personnel et d'expérience :

- Les principes de base de l'objet que sont l'**encapsulation**, l'**héritage** et le **polymorphisme** ne suffisent pas à guider le design.
- Les Design-Patterns définissent des référentiels de plus haut niveau, mais ils ne forment pas un tout suffisamment cohérent pour aider à générer des designs complets.

Principes avancés de conception objet

Il existe pourtant des principes extrêmement utiles en matière de design.

Les principes sont listés ici en trois groupes principaux:

1. **Gestion des évolutions et des dépendances entre classes,**
2. **Organisation de l'application en modules,**
3. **Gestion de la stabilité de l'application.**

Gestion des évolutions et des dépendances entre classes

Principe d'ouverture/fermeture - Open-Closed Principle (OCP)

"Un module doit être ouvert aux extensions mais fermé aux modifications."

Principe de substitution de Liskov- Liskov Substitution Principle (LSP)

"Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir."

Principe d'inversion des dépendances - Dependency Inversion Principle (DIP)

"A. Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions."

"B. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions."

Principe de séparation des interfaces - Interface Segregation Principle (ISP)

"Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas."

Organisation de l'application en modules

Principe de fermeture commune - Common Closure Principle (CCP)

"Les classes impactées par les mêmes changements doivent être placées dans un même package."

Gestion de la stabilité de l'application

Principe des dépendances acycliques - Acyclic Dependencies Principle (ADP)

"Les dépendances entre packages doivent former un graphe acyclique. "

Principe d'ouverture/fermeture *(1/6)*

Evolution sans modification

Les problèmes de rigidité et de fragilité sont liés à une même cause **: l'impact des changements sur de nombreuses parties de l'application**.

Chacun de ces changements oblige à modifier du code existant, ce qui est à la fois coûteux et risqué.

Pour éviter cela, le principe d'ouverture/fermeture stipule que tout module *(package, classe, méthode)* doit être à la fois :

- **ouvert aux extensions** : le module peut être étendu pour proposer des comportements qui n'étaient pas prévus lors de sa création.
- **fermé aux modifications** : les extensions sont introduites sans modifier le code du module.

En d'autres termes, l'ajout de fonctionnalités doit se faire en ajoutant du code et non en éditant du code existant.

Principe d'ouverture/fermeture *(2/6)*

L'abstraction comme moyen d'ouverture/fermeture

L'ouverture/fermeture se pratique en faisant reposer le code "fixe" sur une abstraction du code amené à évoluer.

En d'autres termes, l'OCP consiste à **séparer le stable du variable**, en faisant reposer le stable sur une définition stable du variable.

Deux mécanismes principaux permettent de mettre en place l'abstraction préconisée par l'OCP :

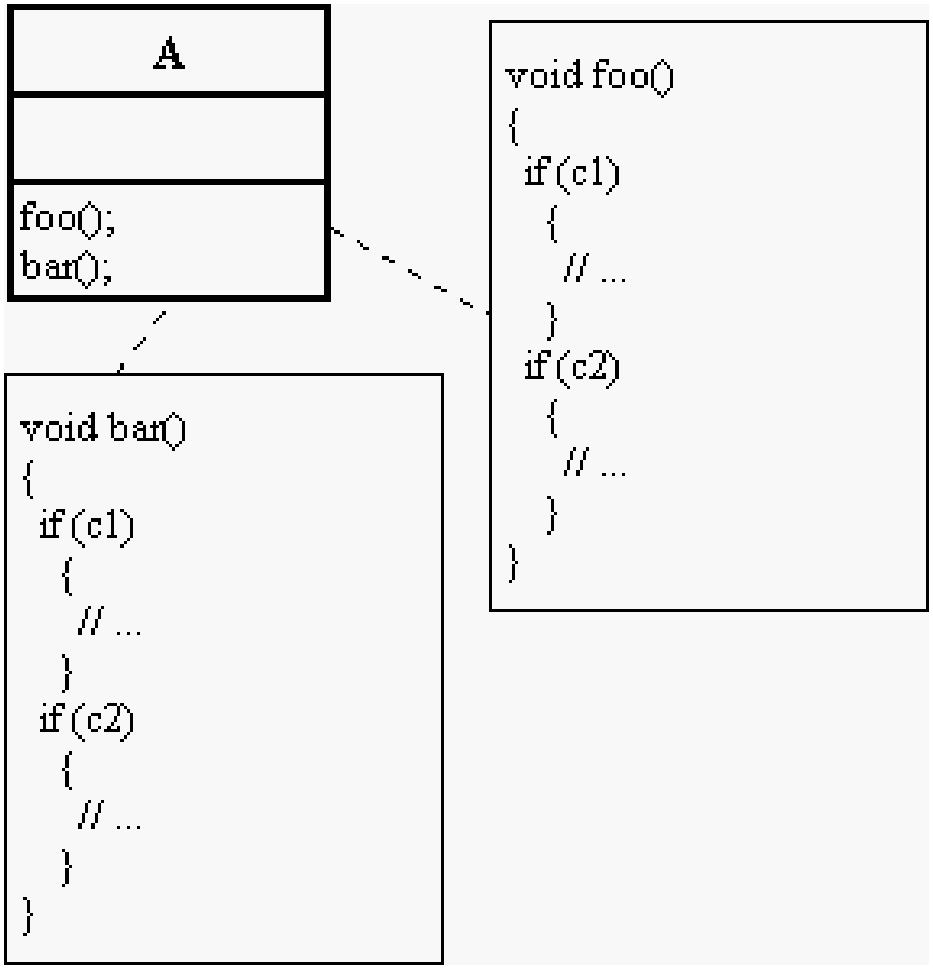
- En **programmation objet**, l'abstraction repose sur l'utilisation de classes d'interface (classes abstraites en C++ et Ada95, interfaces en Java) ,
- En **programmation générique**, l'abstraction repose sur l'utilisation de templates.

Principe d'ouverture/fermeture *(3/6)*

Utilisation de la « délégation abstraite »

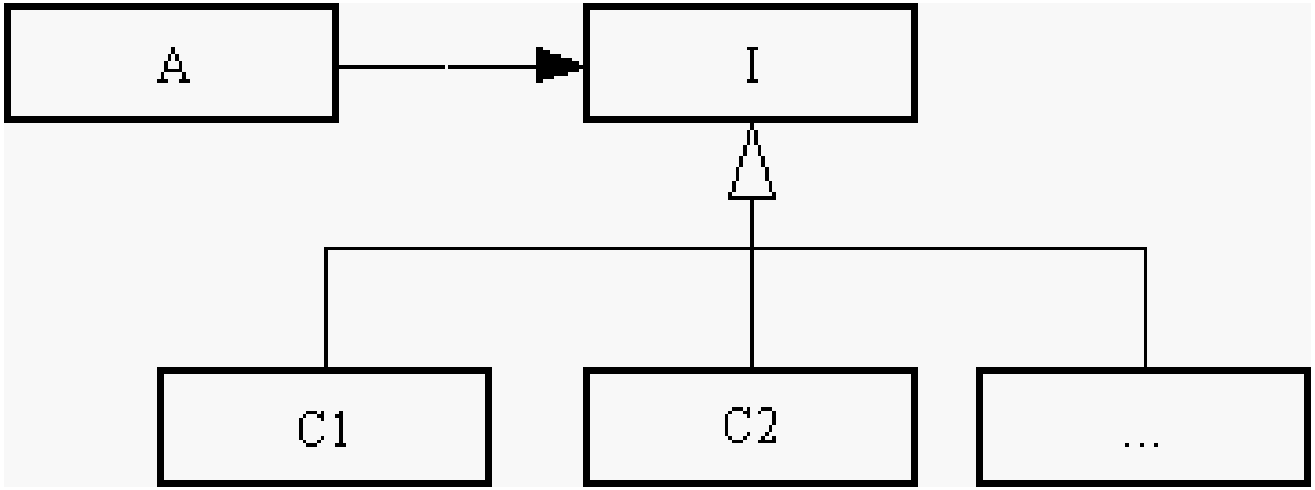
A gère les cas **c1** et **c2**.

Si un nouveau cas **c3** doit être géré, il faut modifier le code de **A** en conséquence (opérations **A.foo()** et **A.bar()**) :



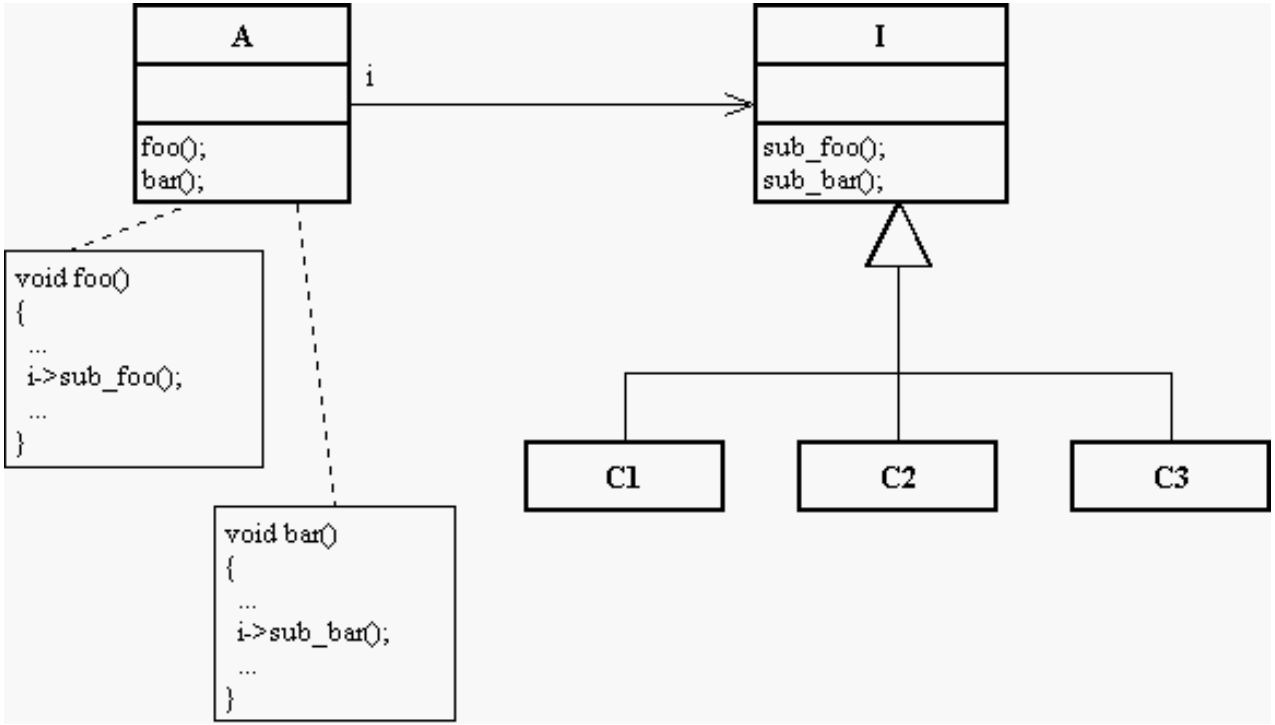
Principe d'ouverture/fermeture *(4/6)*

Le code de **A** peut être ouvert aux extensions et fermé aux modifications en introduisant une classe d'interface **I** dont dérivent des classes **C1** et **C2** correspondant aux cas **c1** et **c2** :



Principe d'ouverture/fermeture *(5/6)*

Puisque **A** repose uniquement sur l'interface **I**, il devient possible d'ajouter un nouveau cas **c3** sous la forme d'une classe **C3** dérivée de **I**, sans avoir à modifier **A** :



Principe d'ouverture/fermeture *(6/6)*

Il convient d'identifier correctement les points d'ouverture/fermeture de l'application, en s'inspirant :

- des besoins d'évolutivité exprimés par le client,
- des besoins de flexibilité pressentis par les développeurs,
- des changements répétés constatés au cours du développement.

= il faut savoir identifier ce qui sera stable et variable pour pouvoir les séparer.

Principe de substitution de Liskov *(1/5)*

Un carré un est rectangle!

```
Class Rectangle
- height_
- width_
+ setHeight(height) {height _ = height;}
+ setWidth(width) {width _ = width;}
+ getSurface():float {return height_ * width_;}

Class Square inherits Rectangle
+ setHeight(height) {height _ = height; width_ = height;} // redéfinition.
+ setWidth(width) {width _ = width; height_ = width;} // redéfinition.
```

Principe de substitution de Liskov *(2/5)*

```
// Test:
testRectangle(Rectangle rectange) {
    rectangle.setHeight(2);
    rectangle.setWidth(3);
    assert(rectangle.getSurface() == 6);
}
// Test:
testSquare(Square square)
{
    square.setHeight(2);
    assert(square.getSurface() == 4); // Ca marche?
    testRectangle(square); // Ca marche?
}
```

Principe de substitution de Liskov *(3/5)*

Il faut hériter pour substituer.

Pour qu'une instance dérivée se substitue à une instance père, il faut:

- Que la classe dérivée n'exige pas plus de son utilisateur.
- Que la classe dérivée ne garantisse pas moins à son utilisateur.

Exiger/garantir = DesignByContract.

Principe de substitution de Liskov *(4/5)*

« Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir. »

Finalement, il est rare de pouvoir hériter!

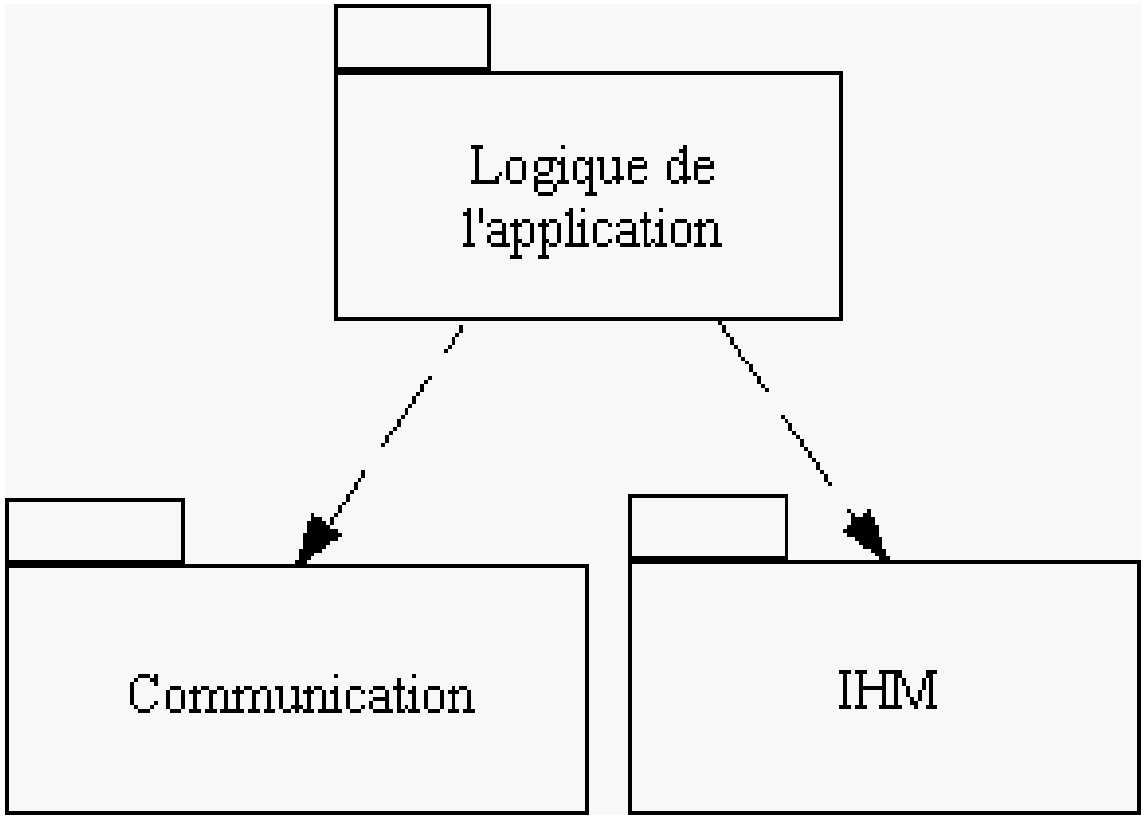
PSL et agréger au lieu d'hériter *(5/5)*

```
Class Square
- Rectangle rectangle_; // agrégation.
+ setHeight(height) {
rectangle_.setHeight(height); restangle_.setWidth(height);} // délégation.
+ setWidth(width)  {
rectangle_.setWidth(width); rectangle_.setHeight(width);} // délégation.
+ getSurface():float  {
return rectangle_.getSurface();} // délégation.
```

Un carré contient un rectangle!

Hériter pour substituer, sinon, agréger.

Principe d'inversion des dépendances *(1/4)*



Dans la plupart des applications, les modules de haut niveau *(ceux qui portent la logique fonctionnelle de l'application ou les aspects "métier")* sont construits directement sur les modules de bas niveau *(par exemple les librairies graphiques ou de communication)*

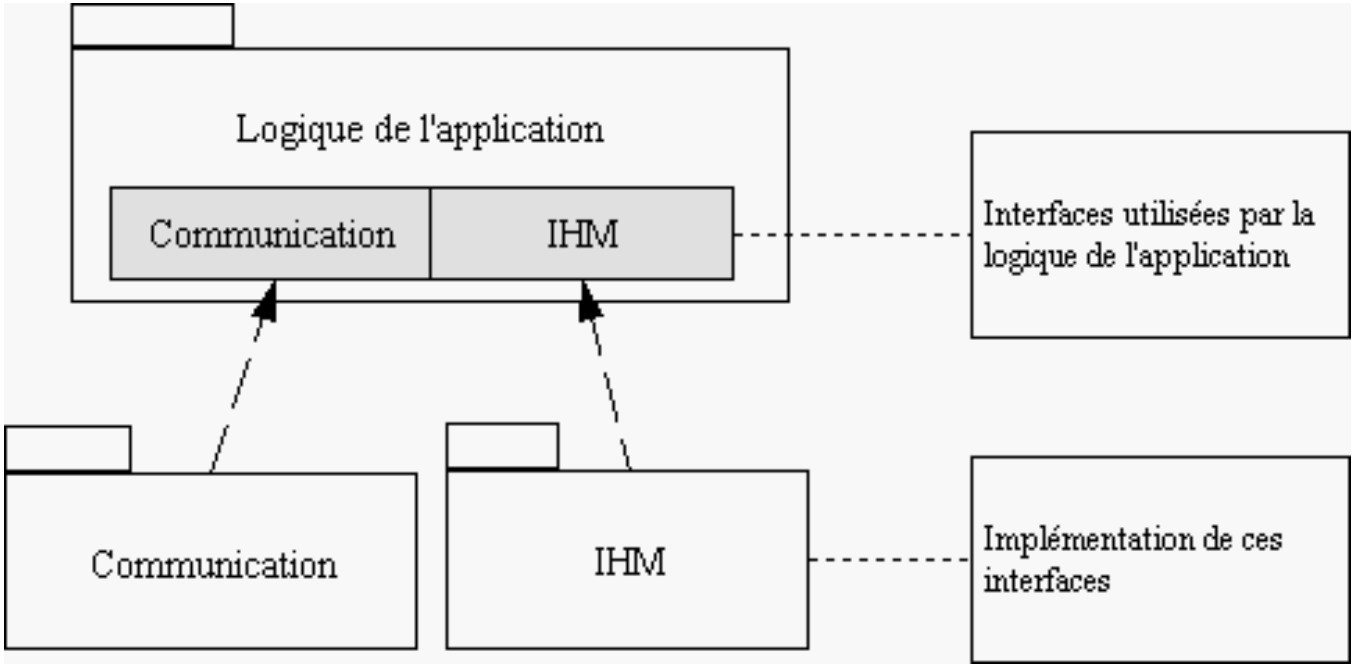
Principe d'inversion des dépendances *(2/4)*

Cela paraît naturel au premier abord mais pose en réalité deux problèmes essentiels :

- Les modules de haut niveau doivent être modifiés lorsque les modules de bas niveau sont modifiés.
- Il n'est pas possible de réutiliser les modules de haut niveau indépendamment de ceux de bas niveau. En d'autres termes, il n'est pas possible de réutiliser la logique d'une application en dehors du contexte technique dans lequel elle a été développée.

Principe d'inversion des dépendances *(3/4)*

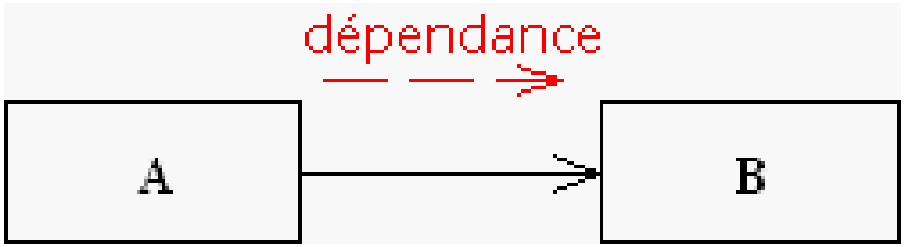
Selon ce principe, la relation de dépendance doit être inversée : **les modules de bas niveau doivent se conformer à des interfaces définies et utilisées par les modules de haut niveau.**

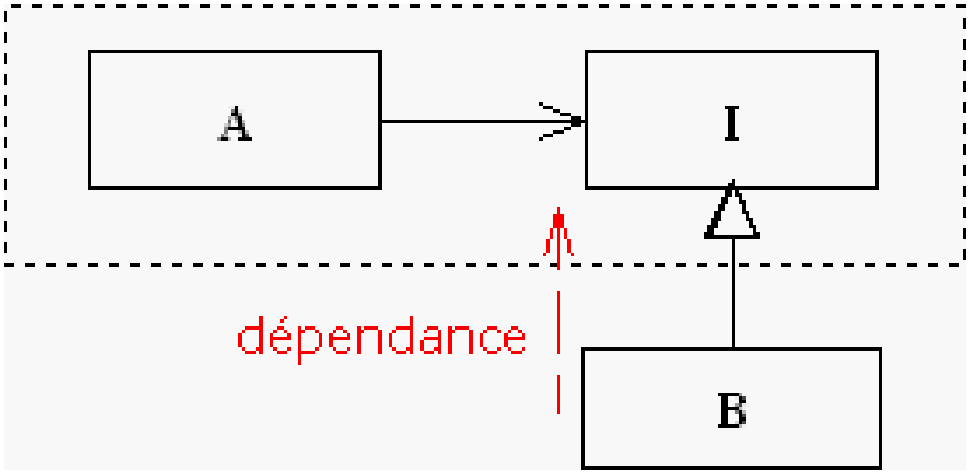


Principe d'inversion des dépendances *(4/4)*

L'abstraction comme technique d'inversion des dépendances.

Pour inverser la dépendance de A vers B, on introduit une classe d'interface I dont dérive B.

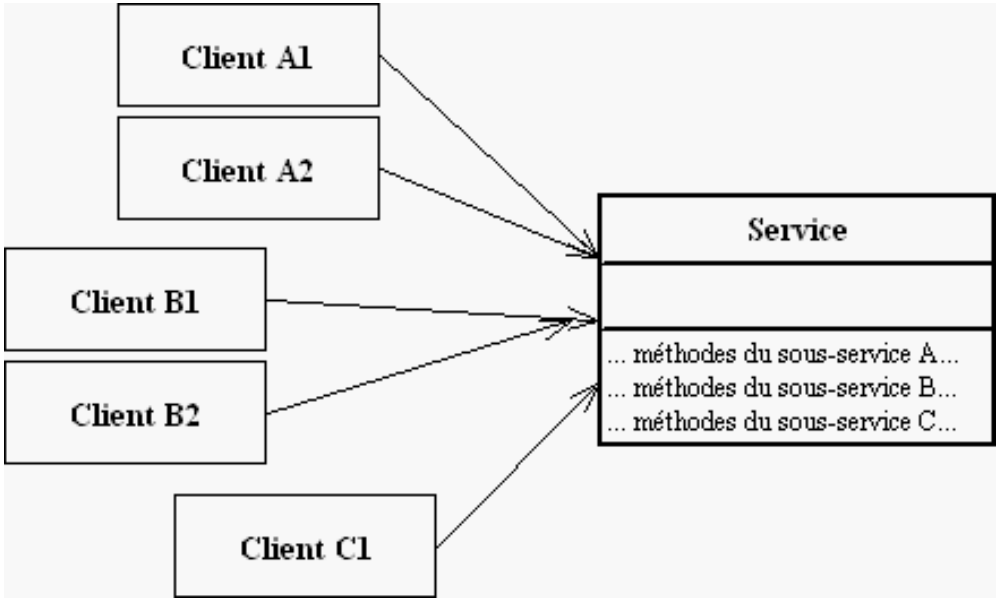




Principe de séparation des interfaces *(1/4)*

Pollution d'interface par agrégation de services

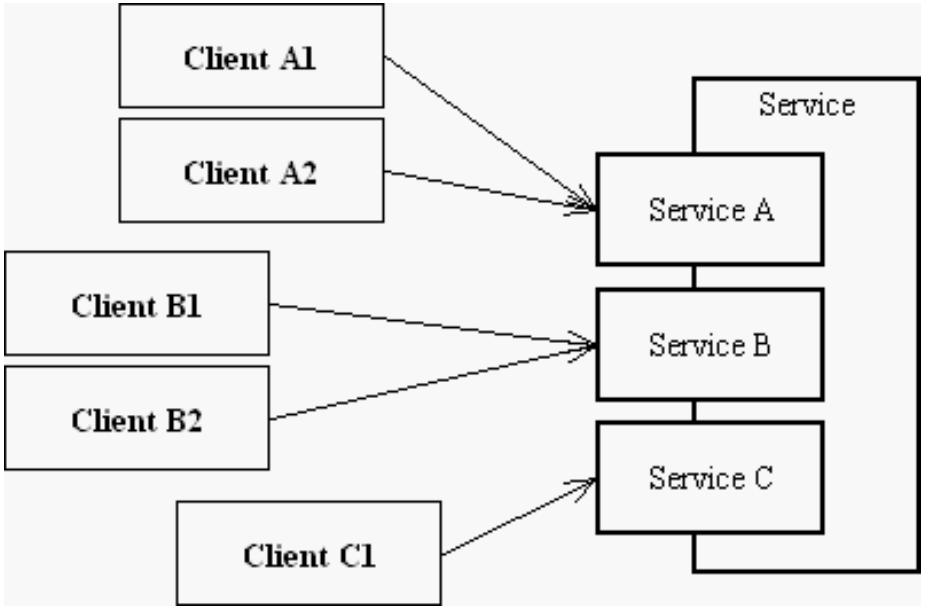
On retrouve dans la plupart des designs quelques classes qui rendent plusieurs services simultanément, comme l'illustre le schéma ci-dessous :



Principe de séparation des interfaces *(2/4)*

Solution : séparation des services de l'interface

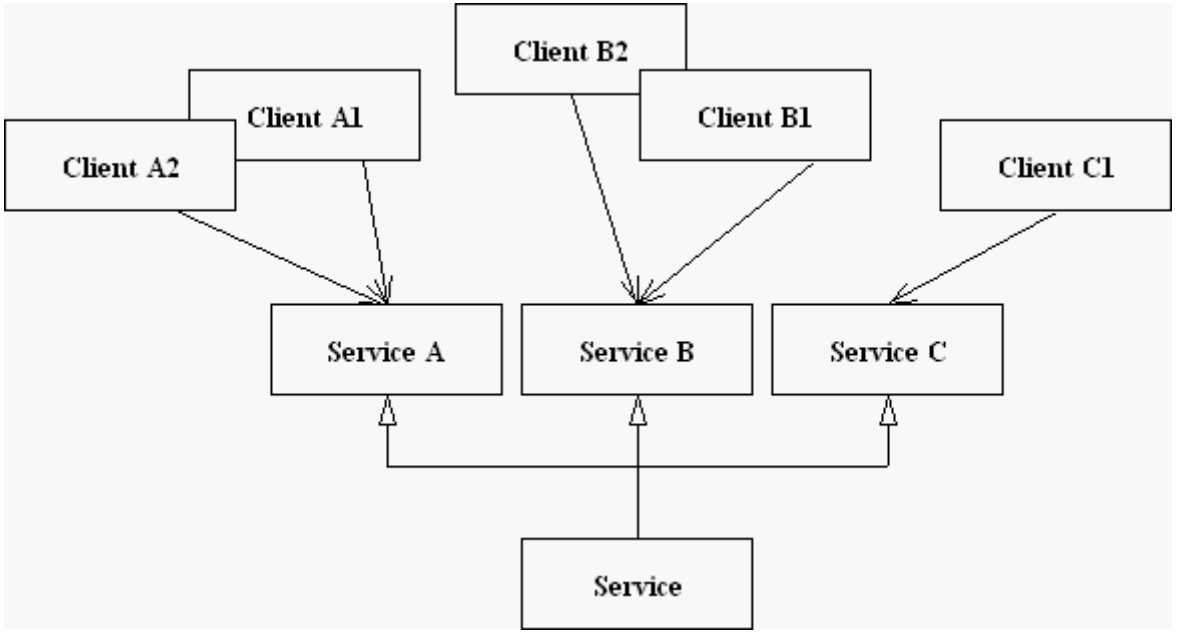
Le principe de séparation des interfaces stipule que chaque client ne doit "voir" que les services qu'il utilise réellement :



Principe de séparation des interfaces *(3/4)*

Séparation par héritage multiple

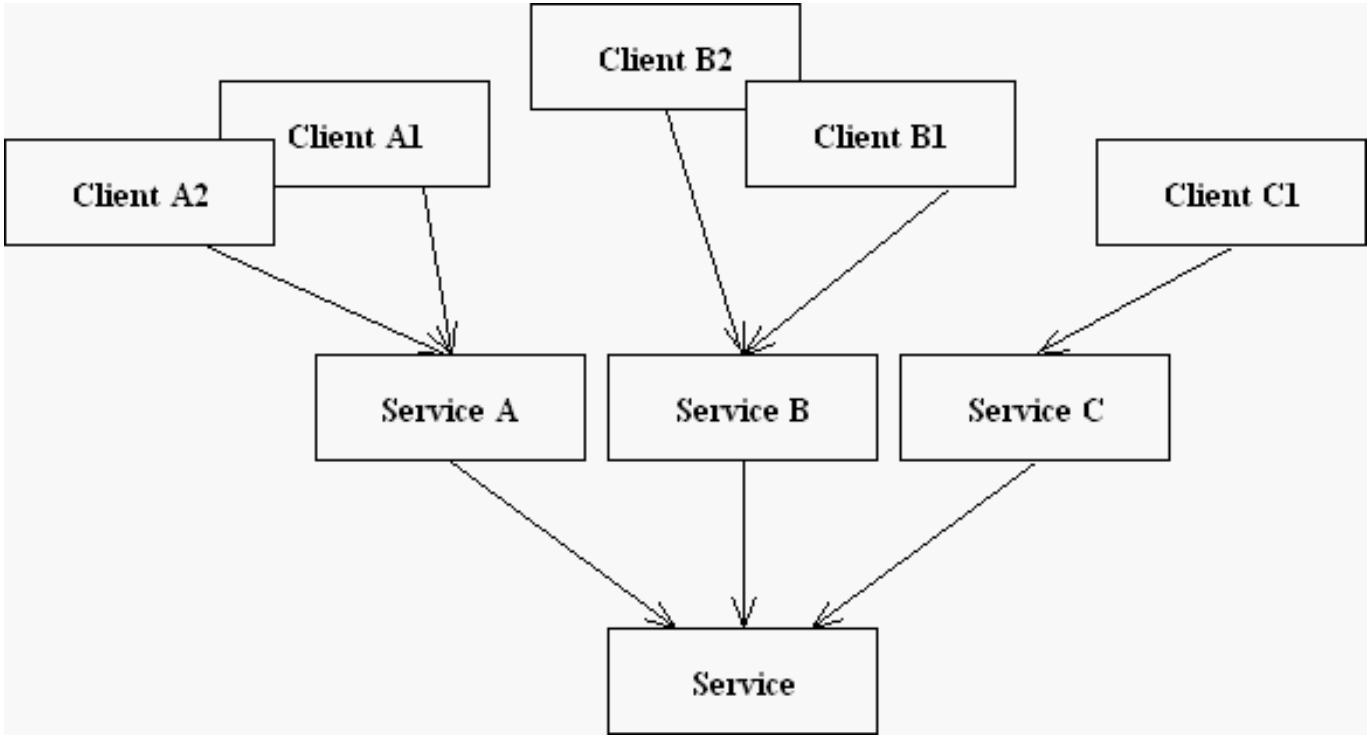
Dans cette approche chaque service est représenté par une classe d'interface dont dérive la classe qui implémente les services. Les clients ne voient les services qu'au travers de ces classes d'interface comme le montre le schéma suivant :



Principe de séparation des interfaces *(4/4)*

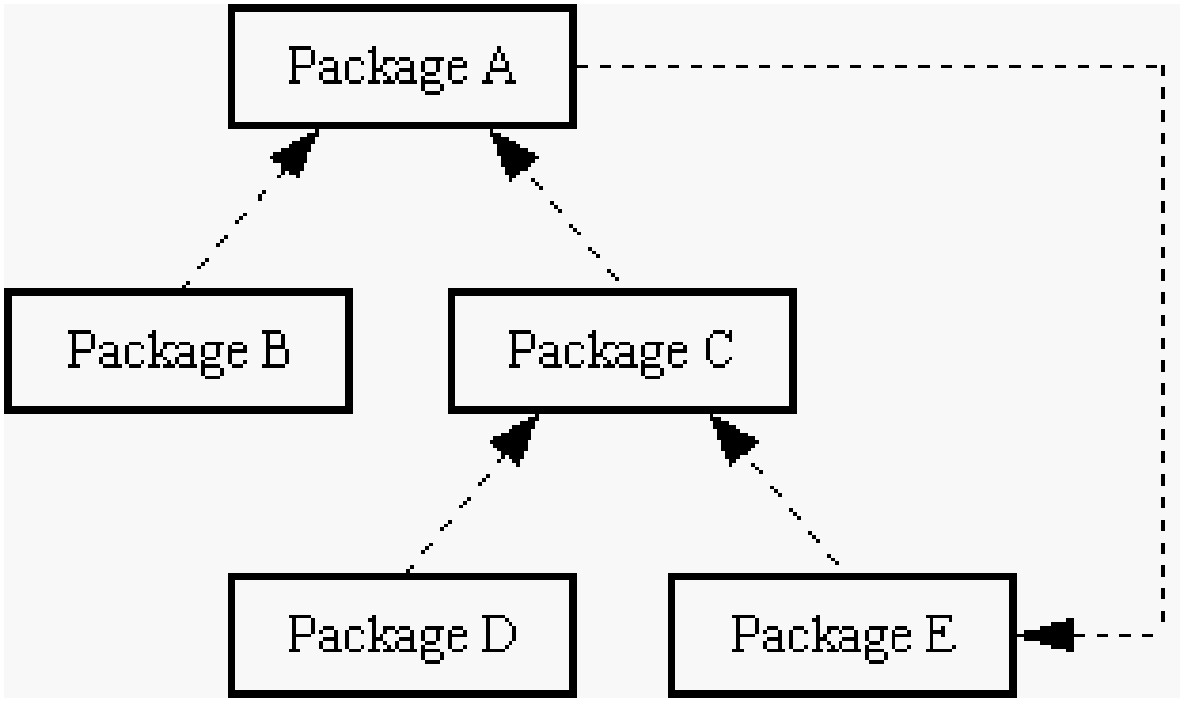
Séparation par Adapter

Lorsque l'héritage multiple n'est pas possible, les services peuvent être représentés par des classes d'adaptation :



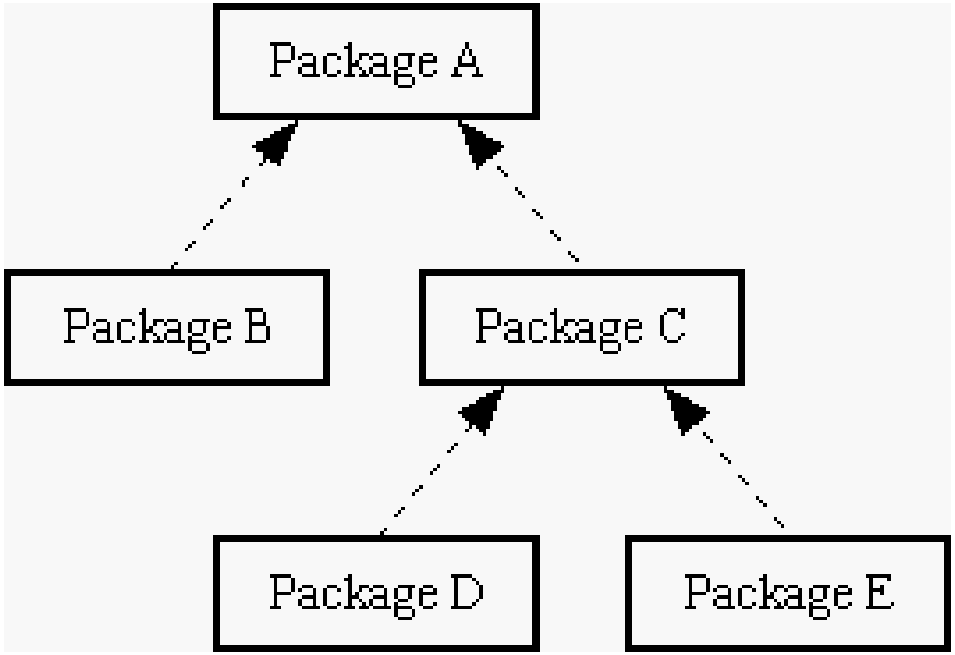
Principe des dépendances acycliques *(1/3)*

Dépendances cycliques entre packages.



Principe des dépendances acycliques *(2/3)*

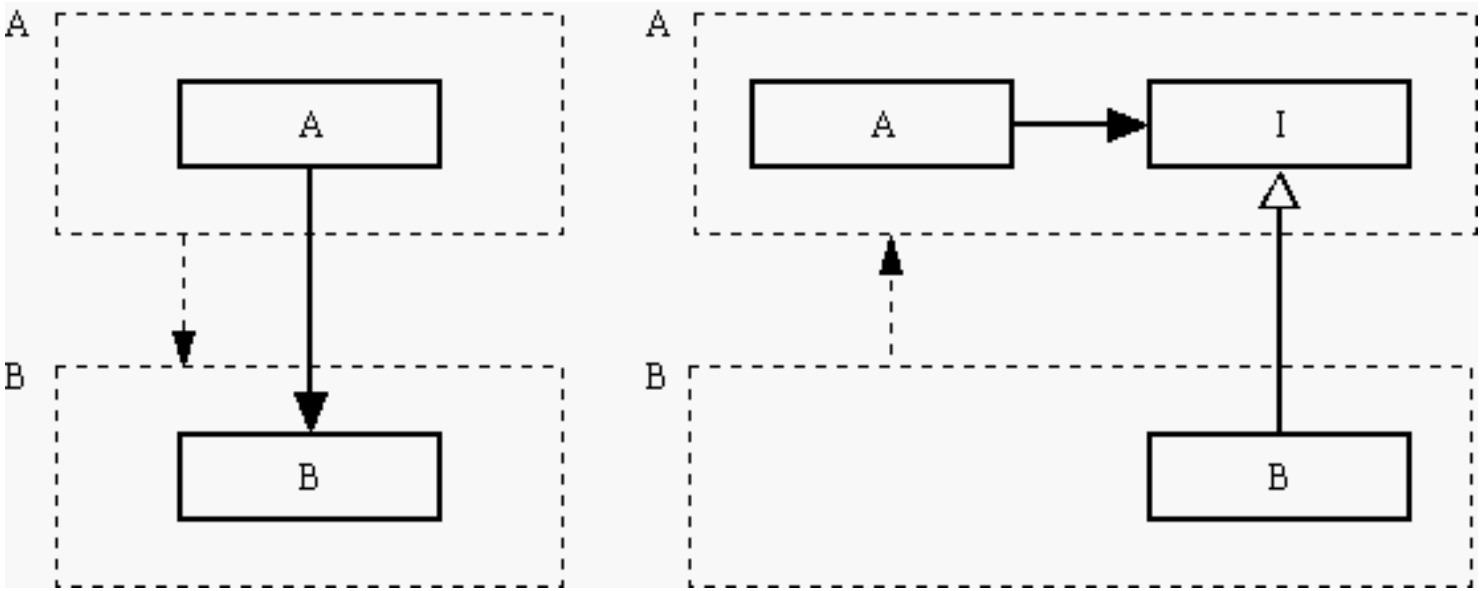
Dépendances acycliques entre packages



Principe des dépendances acycliques *(3/3)*

Technique d'inversion des dépendances

La technique d'inversion de dépendances est celle présentée dans le principe d'inversion des dépendances, et repose sur l'introduction d'une classe d'interface :



Gestion des évolutions et des dépendances entre classes

Il faut :

- Isoler les parties génériques/réutilisables de l'application en les faisant reposer uniquement sur des classes d'interface,
- Considérer l'héritage comme une implémentation d'interface, la classe dérivée pouvant se "brancher" dans n'importe quel code qui utilise cette interface (l'interface forme alors un contrat entre le code utilisateur et les classes dérivées),
- Utiliser des classes d'interfaces pour créer des pare-feu contre la propagation des changements,
- Construire les parties "techniques" de l'application sur les parties "fonctionnelles", et non l'inverse,
- Utiliser l'héritage multiple pour décomposer les interfaces complexes en interfaces simples correspondant chacune à un service spécifique. Une classe donnée peut ensuite proposer plusieurs services simultanément en implémentant les interfaces correspondantes.

Organisation de l'application en modules

Il faut :

- Décomposer l'application en packages pour gérer correctement les versions et permettre une réelle réutilisation,
- Regrouper dans un même package les classes qui sont utilisées ensemble et qui sont impactées par les mêmes changements.

Gestion de la stabilité de l'application

Il faut :

- Organiser les modules en un arbre de dépendances (en supprimant donc tout cycle dans le graphe des dépendances),
- Placer les packages les plus stables à la base de l'arbre,
- Placer les interfaces dans les packages les plus stables.

Conclusion *(1/2)*

Les principes énoncés ici placent le **contrôle des dépendances** au cœur de l'activité de design, dans le but de limiter le coût des modifications et ainsi atteindre les objectifs recherchés d'extensibilité et de réutilisabilité.

Ce contrôle repose sur une **utilisation efficace des interfaces**, qu'il s'agisse d'interfaces entre applications, entre packages ou entre classes.

Conclusion *(2/2)*

Ces principes restent bien sûr uniquement des principes :

ils n'apportent pas de recettes miraculeuses ou de règles strictes qui rendraient le design quasiment automatique.

Ils constituent par contre un solide cadre de décision et d'évaluation, qui permet d'aboutir à des applications plus facilement extensibles et réutilisables.

Dernière pratique: s'ouvrir

Parce que les technologies et les méthodes avancent à toute vitesse, il faut:

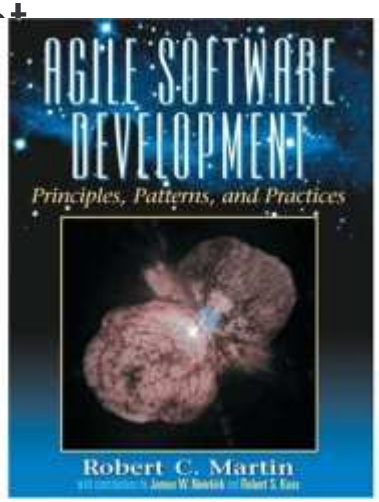
- Se tenir au courant,
- Être curieux
- Lire

Bibliographie

Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin (Uncle Bob)

- Présentation du développement Agile
- Présentation de l'eXtreme Programming
- Présentation des principaux design-pattern
- Présentation des principes de conception objet

500 pages / en anglais



Bibliographie

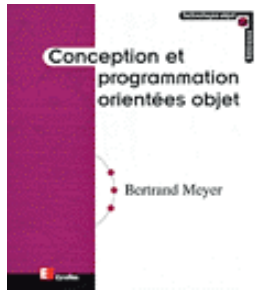
Conception et programmation orientés objet, Bertrand Meyer

LA référence sur le sujet.

Entre autres:

- Conception par contrat
- Principe d'ouverture fermeture
- Principe de séparation commande requête

1200 pages/ en français

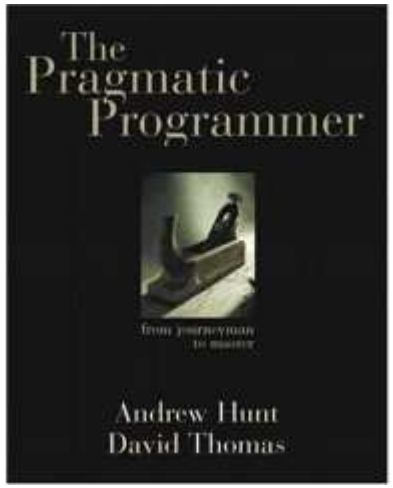


Bibliographie

The Pragmatic Programmer: From Journeyman to Master, Andrew Hunt, David Thomas

- Pratiques pragmatiques pour développeurs

352 pages / en anglais



Bibliographie

Gestion de projet : EXtreme Programming, Jean-Louis Bénard, Laurent Bossavit, Régis Médina, et Dominic Williams

Une très bonne, claire et complète présentation de l'eXtreme-Programming.

300 pages / en français

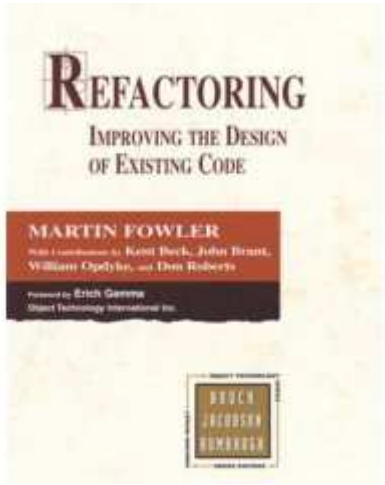
Bibliographie

Refactoring: Improving the Design of Existing Code par Martin Fowler, Kent Beck, et John Brant

LA référence en matière de remaniement de code.

- Catalogue des « Bad-smells in code »
- Explication du pilotage par les tests.
- Catalogue des remaniements.

430 pages / en anglais

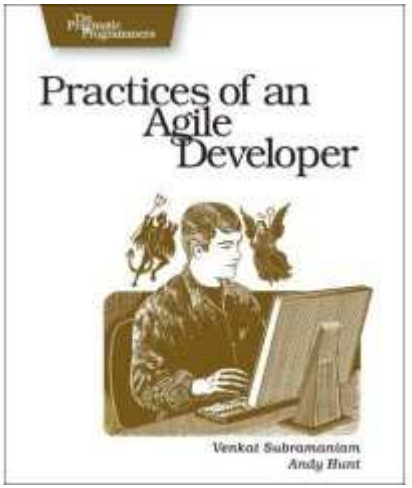


Bibliographie

Practices Of An Agile Developer: Working In The Real World, Venkat Subramaniam, Andy Hunt

- Catalogue de pratiques pragmatiques
- Présentation mauvaise habitude/bonne habitude

184 pages / en anglais



Bibliographie

UML 2 et les design patterns, Craig Larman

Le titre est très mal choisi …

- Présentation de l'analyse et de la conception orientée objet
- Présentation d'un processus de développement itératif et incrémental très pertinent
- Présentation des aspects utiles d'UML
- Présentation des principaux design-patterns
- Présentation de l'agilité

600 pages / en français

Bibliographie

Design patterns. Catalogue des modèles de conception réutilisables, Richard Helm, Ralph Johnson, John Vlissides, Eric Gamma

- LA référence en matière de patrons de conception.

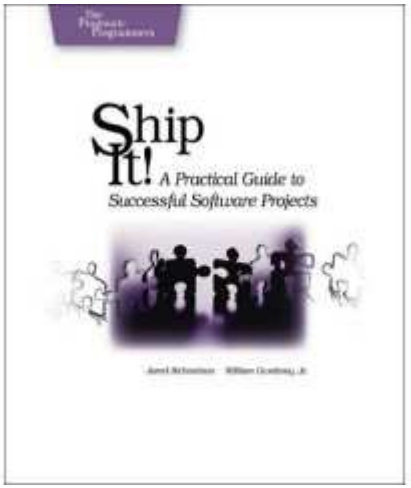
450 pages / en français

Bibliographie

Ship It!: A Practical Guide To Successful Software Projects, Jared Richardson, William A., Jr. Gwaltney

- Catalogue de conseils pragmatiques pour réussir les projets de développement logiciel.

200 pages / en anglais

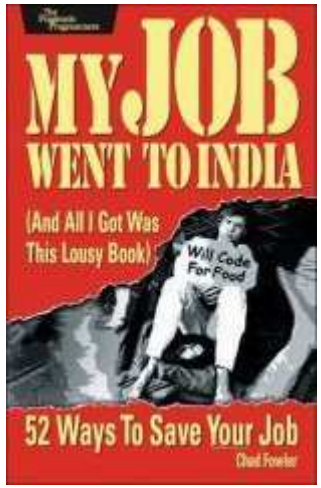


Bibliographie

My Job Went to India: And All I Got Was This Lousy Book, Chad Fowler

- Comment survivre si on compte vivre du développement logiciel.

230 pages / en anglais

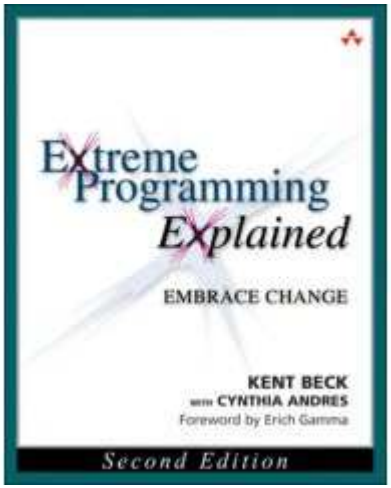


Bibliographie

Extreme Programming Explained: Embrace Change,
Kent Beck et Cynthia Andres

- XP par son créateur.
- Approche philosophique d'XP.

220 pages / en anglais

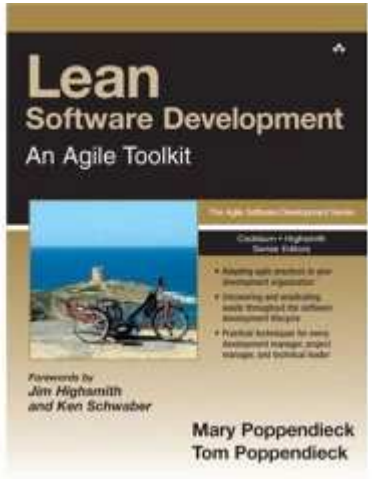


Bibliographie

Lean Software Development: An Agile Toolkit, Mary et Tom Poppendieck, 240 pages / en anglais

Implementing Lean Software Development: From Concept to Cash, Mary et Tom Poppendieck, 300 pages / en anglais

- Approche Lean du développement logiciel



Bibliographie

UML 2.0, Martin Fowler

(pas lu, mais souvent recommandé …)

200 pages / en français

Bibliographie

The Mythical Man-Month: Essays on Software Engineering, Frederick P. Brooks

(pas encore lu …)

- Gestion de projets et ses facteurs d'échecs
- Un classique, toujours cité.

300 pages / en anglais

Traduit en français: **Le mythe du mois-homm**