Chapitre 2 – Introduction au Génie Logiciel

Véronique DESLANDRES - Licence DEVOPS, IUT Univ. Lyon1

IUT Lyon 1
l'excellence technologique

1- Evolution des applications

Croissance continue de la complexité des systèmes

- **Couverture élargie**
  - Ex. passage d'un progiciel Achat… à un ERP (Enterprise Ressource Planning)
- Des **fonctionnalités** plus **évoluées**
  - Ex.: systèmes auto-adaptatifs (capables de s'administrer et de réparer des problèmes seul)
- Une **temporalité de plus en plus élevée**
  - Ex.: le *push trading* en bourse, avec des ordres lancés **à la nano seconde près**
  - Des demandes d'amélioration fonctionnelle plus fréquentes

2- Évolution des matériels et technologies

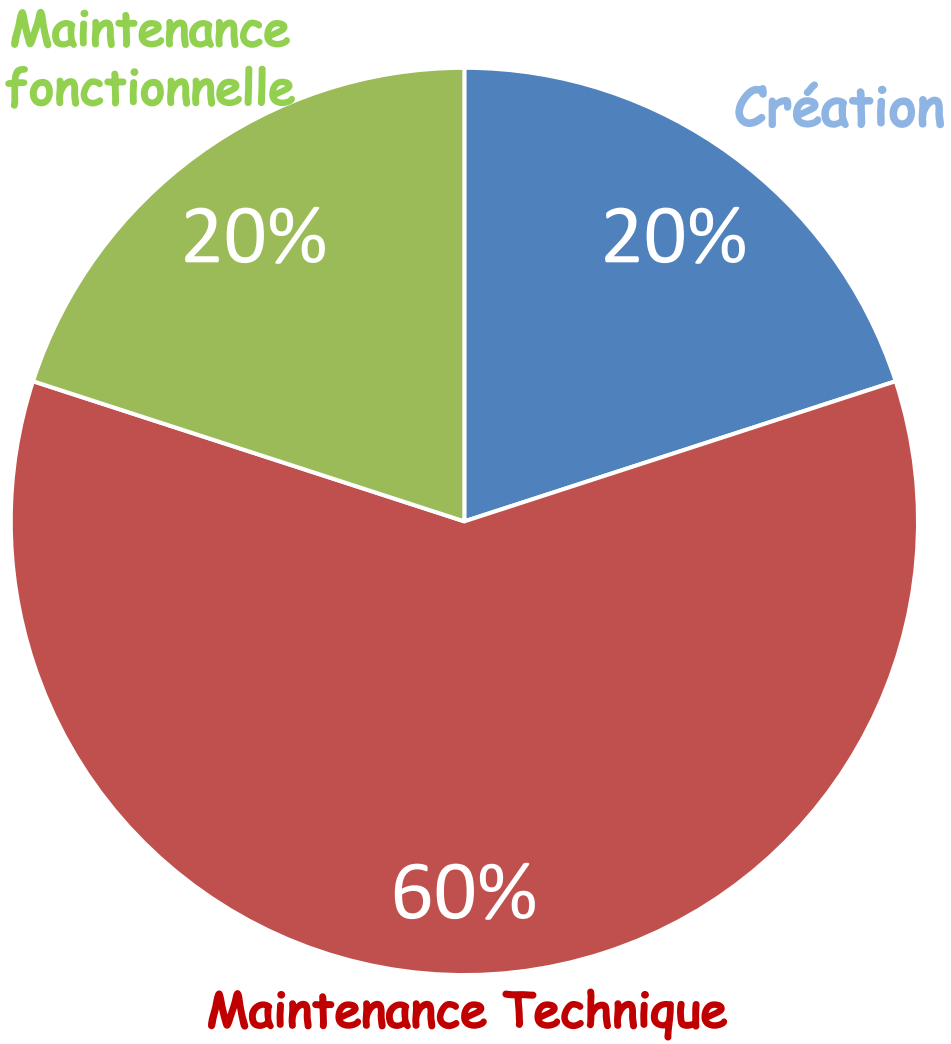
- Complexité soutenue par l'évolution en parallèle :
  - Parallélisme
  - Puissance des machines
  - Mise à disposition de vastes volumes de données (Web2, Web3, big data)
  - Abstraction des langages de programmation
  - Pertinence des méthodes de développement

- Diversité des technologies possibles
  - **Développement web** : JEE ou js + PHP archi REST ou .NET avec ASP et JSP ?
  - **BlockChain** : y aller ou pas ?
  - Cf choix entre **Flash et HTML5** pour une UX évoluée (2010/2015)

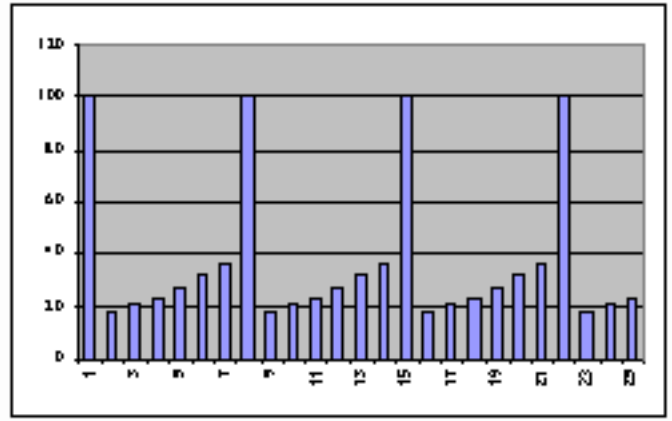
- Considération de nouveaux besoins
  - Hétérogénéité, ouverture, sécurité, étendabilité (*scalability, passage à l'échelle*), gestion des défaillances, concurrence, intégration, transparence, communication, interopérabilité, *responsive design*

Coût du développement logiciel

3 phases dans la vie d'un logiciel :



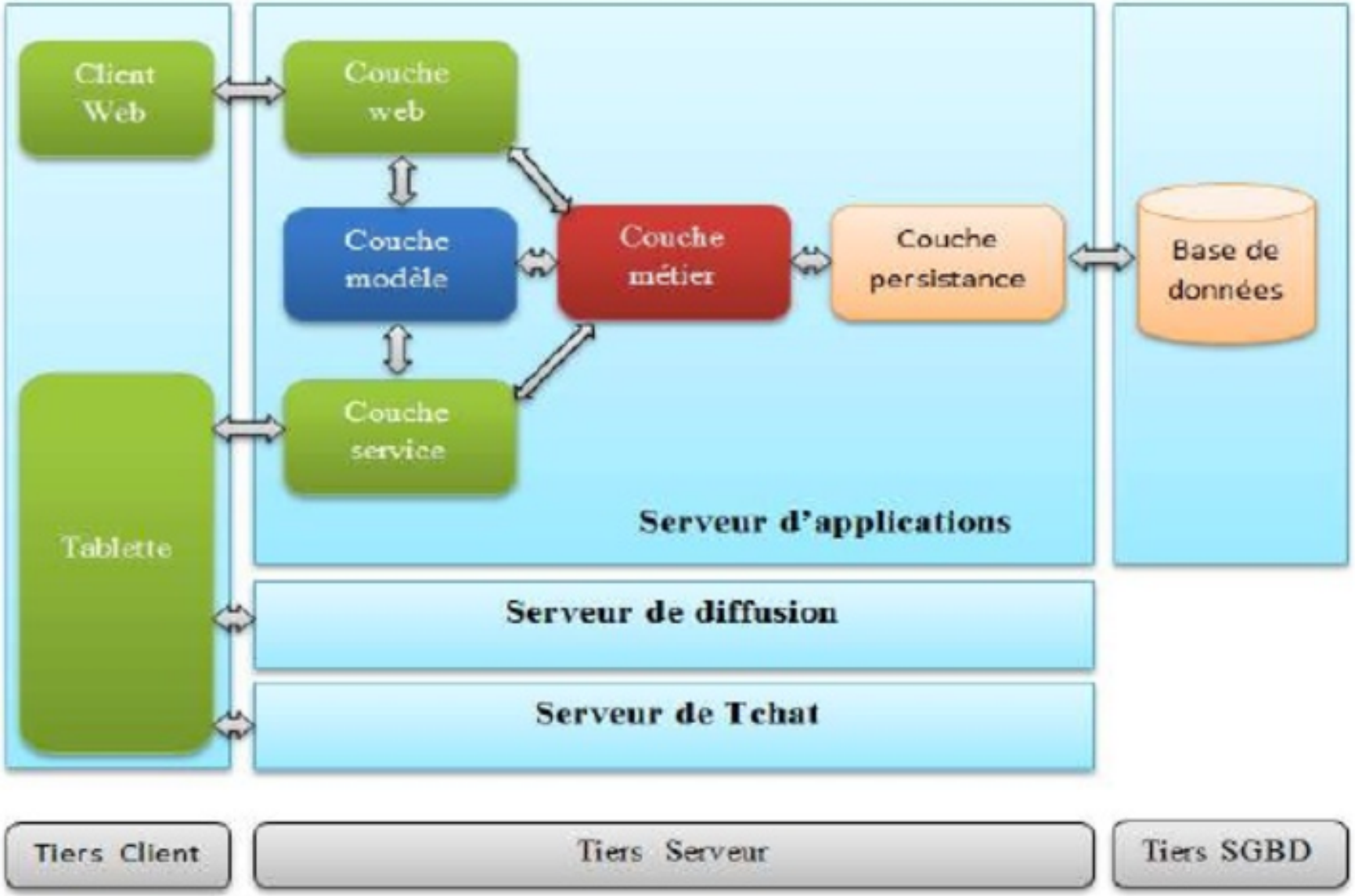
Le coût de maintenance augmente **exponentiellement** avec le temps.



Règle : « *il faut **refaire** le logiciel lorsque le coût de maintenance atteint le **tiers** du coût de réfection totale* »

Des exemples de coûts dus à la non qualité : https://go.univ-lyon1.fr/couts

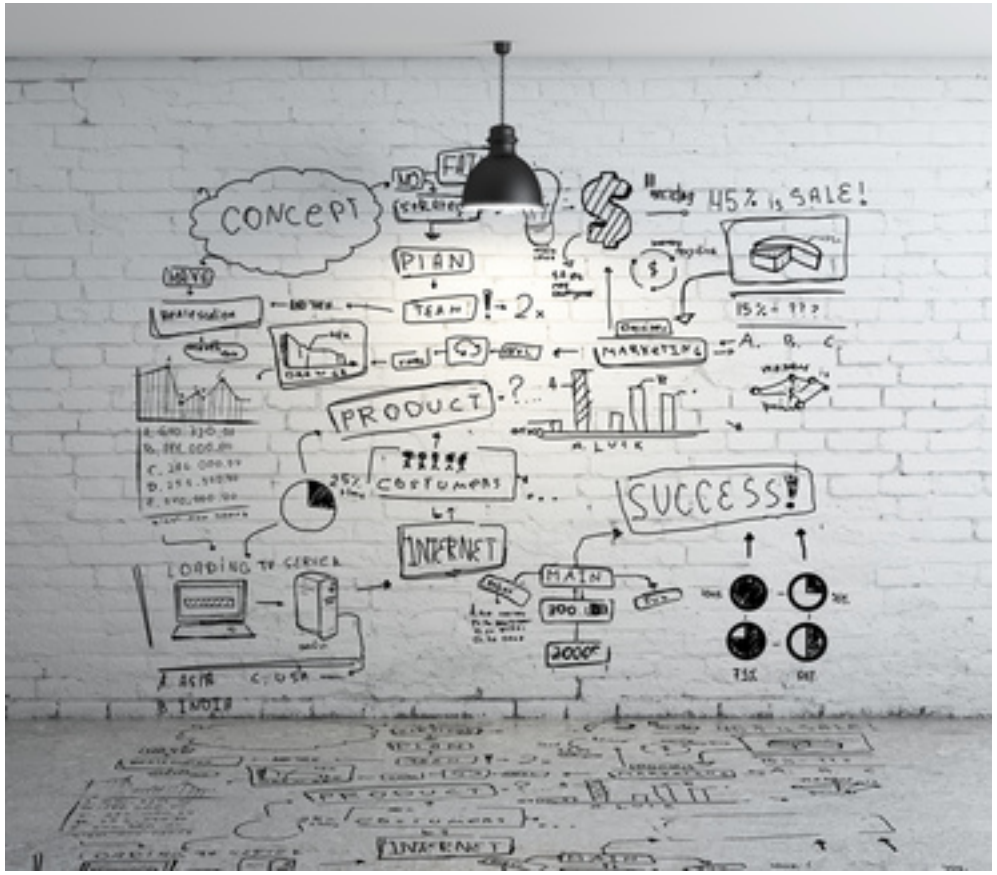
Pour garantir la maintenance (fonctionnelle et technologique), **l'architecture logicielle** est primordiale



Structures de données après *n* modifications, extensions et évolutions technologiques…

L'architecture logicielle, c’est quoi ?

- Découpe de la **solution logicielle**
  - Pour réduire la complexité globale à des sous-ensembles plus faciles à maîtriser
- Repose sur un **paradigme architectural**
  - Architecture orientée **Composants**,
  - Orientée **Données**,
  - Orientée **Services** (web services, micro-services),
  - Orientée Container, etc.



QUALITÉ LOGICIELLE / ASSURANCE QUALITÉ

Définition

AFNOR

Un « bon logiciel » ou logiciel de qualité, s'entend comme un logiciel capable de **répondre parfaitement aux attentes du client**, le tout sans défaut d'exécution

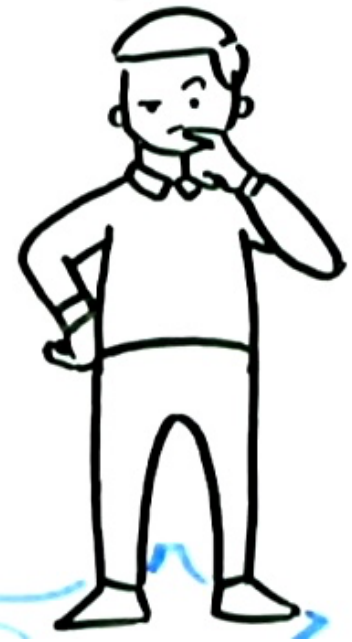
➔ Nécessite de mesurer : métrologie du logiciel

Quelques facteurs de qualité logicielle

- Ergonomie
- Fiabilité
- Intégrité
- Maintenabilité, etc.

Quelques facteurs de qualité logicielle

Existe-t-il une relation de cause à effet entre **ergonomie** et **fiabilité** ?



Réponse : pas nécessairement

Un logiciel non fiable (comportant des erreurs, des bugs fonctionnels) peut par ailleurs être très convivial : il ne cause pas de **problèmes d'utilisation** à ses utilisateurs mais fournit de faux résultats par exemple.

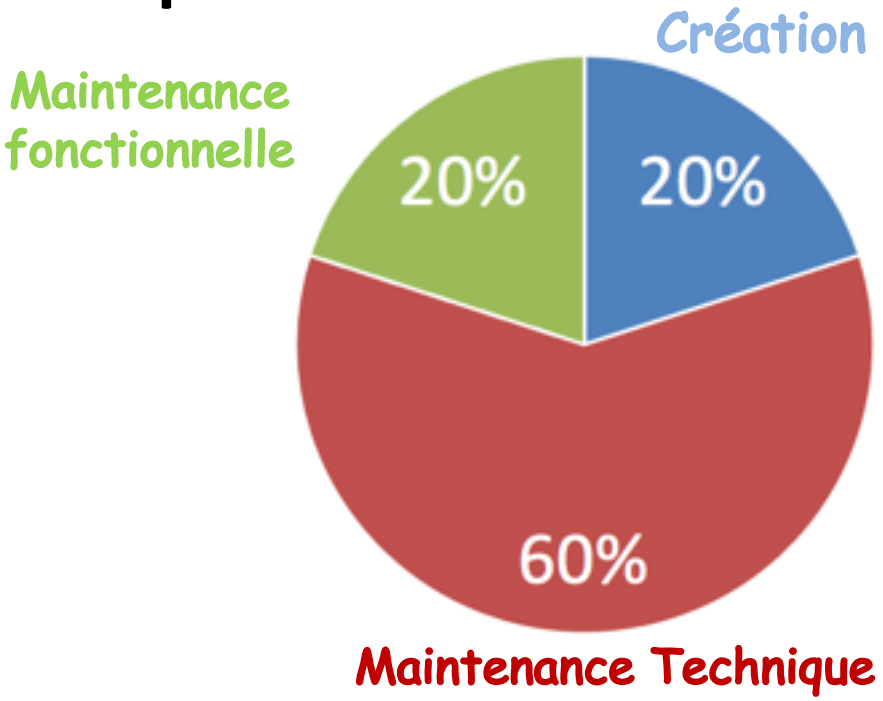
Un logiciel ergonomique évitera en effet les *erreurs de manipulation* de l'utilisateur, mais **pas les défauts de fonctionnement** s'il y en a.

Réciproquement, un logiciel **fiable** peut être **peu ergonomique**.

Métrologie du logiciel / Qualimétrie

- Enjeux = **qualifier** le code, définir le **bug**, **quand** détecter
  - Mesurer le **risque** : vulnérabilité du logiciel aux changements futurs, aux failles
- Quand ?
  - *Revues qualité* pour des parties sous-traitées
  - Analyser la *capacité d'évoluer* d'une application
  - Choix de codes *à réutiliser* : comparaison
- Comment ? 2 types de mesures
  - **Quantitatives** : facile à mettre en œuvre (automatisation), souvent mesure de comptage ; nécessite de bien savoir ce que l'on cherche.
  - **Qualitatives** : c'est la « revue de code », plus complexe.

Que mesurent ces métriques ?



En priorité on va chercher à évaluer :

- la **complexité**
- la **clarté**
- **l'uniformité**
- la **couverture par les tests**

Rappel : coder avec 2 principes fondamentaux

- **Less is more :** moins il y a de code, le mieux c'est. Code minimaliste ➔ meilleure factorisation ➔ meilleure évolutivité

- **KISS** - *Keep it simple and stupid*. Le plus simple est le mieux. Code simple ➔ facile à appréhender ➔ facile à maintenir dans le temps.

Ex. de Métriques Logicielles (1)

LOC	**Lines of Code** : le nb total de lignes de code. Les lignes blanches et les commentaires ne sont pas comptabilisés.
NOC	**Number of Classes :** le nombre de classes dans l'élément sélectionné

→ **Règles de Codage de Qualité, par ex. pour le Langage C :**

Règle 1 - La longueur des fonctions devrait être comprise entre 4 et 40 lignes de programme

Règle 2 - La longueur d'un fichier devrait être de 40 à 400 lignes de programme.

Ex. de Métriques Logicielles (2)

NOP	**Number of Packages** : le nombre de packages dans l'élément sélectionné
NOA	**Number of Attributs**: le nombre d'attributs dans l'élément sélectionné
NSF	**Number of Static Features** : le nombre de variables statiques
NOM	**Number of Methods :** le nombre de méthodes
NSM	**Number of Static Methods :** le nombre de méthodes statiques
PAR	**Number of Parameters** : le nombre de paramètres utilisés sur la portion de code sélectionnée

Ex. de Métriques Logicielles (3)

NOI	**Number of Interfaces :** le nombre d'interfaces
RMA	**Abstractness** : le pourcentage de classes abstraites et d'interfaces par package
DIT	**Depth of Inheritance Tree** : profondeur de l'arborescence de classes (niveaux depuis la classe *Object*)

Ces indicateurs dépendent du type de logiciel :

- Si c'est **un logiciel de jeux**, qui nécessite beaucoup de calculs TR, il y aura beaucoup de *méthodes* et des boucles imbriquées.
- Par contre la *profondeur de classes* (**DIT**) sera sans doute plus faible que pour un logiciel de type **Inventaire** (recueil de données stockées en BD), par ex.

CE (ou CBO)	**Efferent Coupling (ou Coupling between Objects)** : le nombre de classes *dans* un package qui dépendent d'une classe d'un autre package.
CA	**Afferent Coupling** : le nombre de classes *hors* d'une package qui dépendent d'une classe dans le package

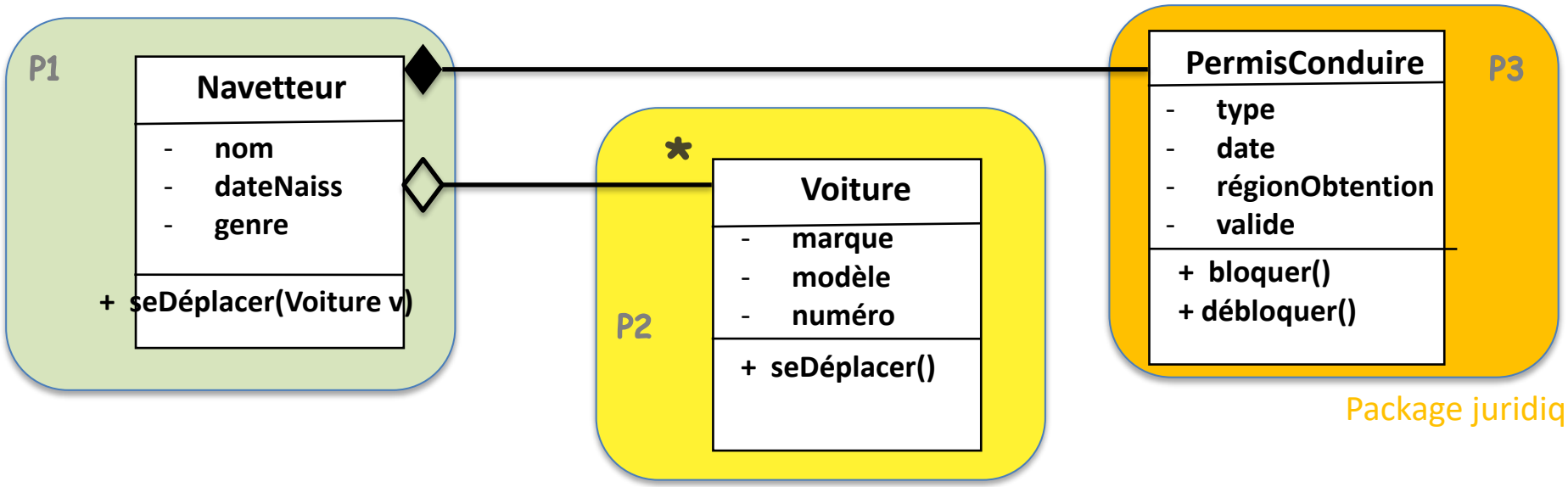
CE : dépendance vers l'Extérieur

Ex.: nb de **types** différents utilisés pour les **attributs** de la classe, les paramètres des méthodes

CE ou CBO (*Couplage between objects)* : nombre de types différents utilisés pour les **attributs** de la classe, les **paramètres** des méthodes, les **variables** locales

CA : responsAbilité vis-à-vis des classes externes

Exemple : un logiciel gérant des conducteurs de voiture



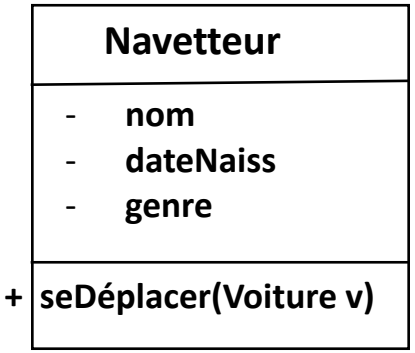
Package juridique

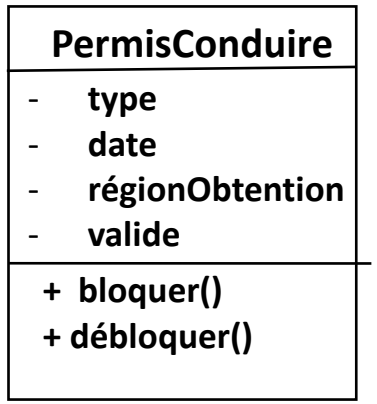
CE(P1) = 2

Code Java de la classe Navetteur

```
public class Navetteur {
    String nom;
    Date dateNaiss;
    Char genre;
    PermisConduire pc;
    List<Voiture> mesVoitures;

    // Constructeur
    Navetteur() {
        …
        pc = new PermisConduire(t,d,r,true);
    }
    void seDéplacer(Voiture v) {
        v.seDéplacer();
        …
    }
}
```





CE (ou CBO)	**Efferent Coupling (ou Coupling between Objects)** : le nombre de classes *dans* un package qui dépendent d'une classe d'un autre package.
CA	**Afferent Coupling** : le nombre de classes *hors* d'une package qui dépendent d'une classe dans le package

CE : dépendance vers l'Extérieur

Ex.: nb de **types** différents utilisés pour les **attributs** de la classe, les paramètres des méthodes

Quand CE > 50, dépendance trop élevée

Ces classes sont sans doute complexes et ont trop de responsabilités → *refactoring*

Indice de Spécialisation : SIX

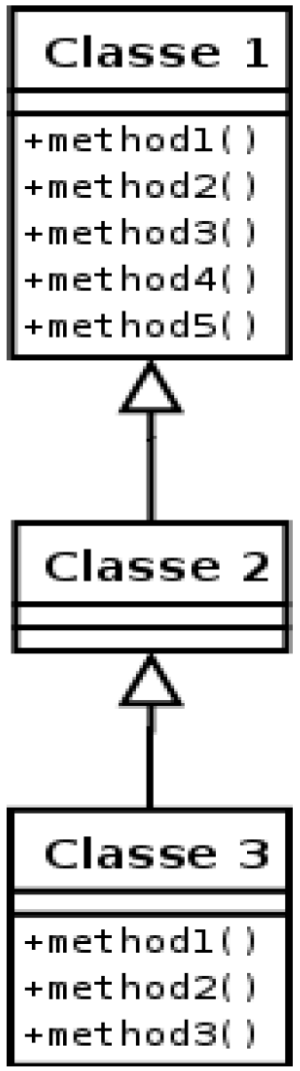
SIX	**Specialization Index : niveau de spécialisation d'une classe. C'est le calcul :** NORM * DIT / NOM. Pour tout le projet : moyenne des index.
NORM	**Number of Overridden Methods** : nombre de méthodes redéfinies

(NOM : Number of Methods)

- SIX mesure jusqu'à quel point les sous-classes réécrivent le comportement des classes parents : plus il est élevé, plus on doit se demander si la spécialisation est bien nécessaire.
- Règle SIX – si plus de 10 méthodes redéfinies dans une sous-classe : mauvaise conception !

Exercice Spécialisation SIX

Calculer l'indice de spécialisation de la Classe3, de la Classe2



Et si Classe3 redéfinit les 5 méthodes ?

Corrigé exercice SIX

SIX = NORM * DIT / NOM

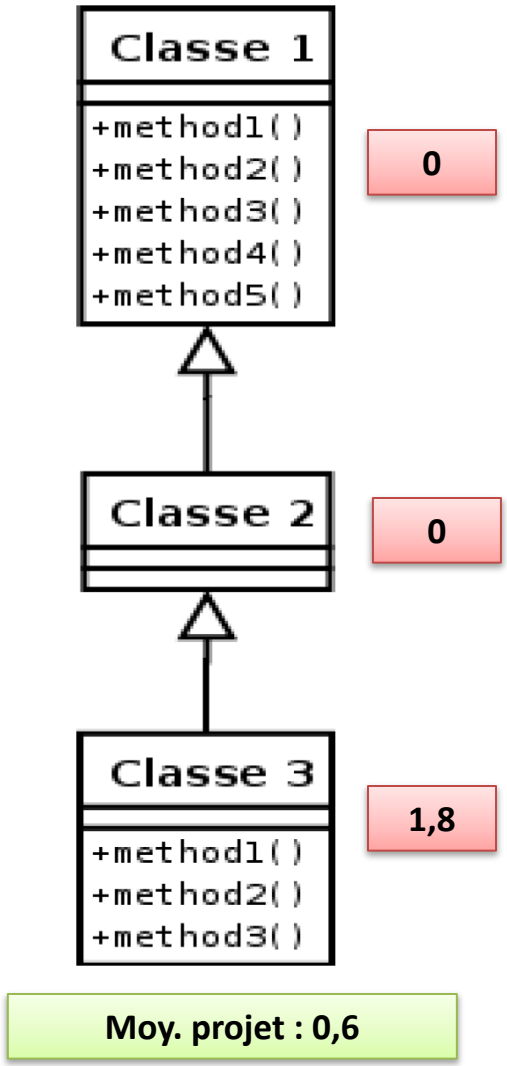
NORM : nb méthodes redéfinies

DIT : profondeur de l'arborescence

NOM : nb de méthodes

- Classe1 : (0 * 1) / 5 = 0
- Classe2 : (0 * 3) / 5 = 0
- Classe3 : (3 * 3) / 5 = 9/5 = 1,8

Moyenne : 0,6



Si Classe3 ne redéfinit aucune méthode :

Classe1 : (0 * 1) / 5 = 0
Classe2 : (0 * 3) / 5 = 0
Classe3 : 0 – **Moy du projet : 0**

Si Classe3 redéfinit les 5 méthodes :

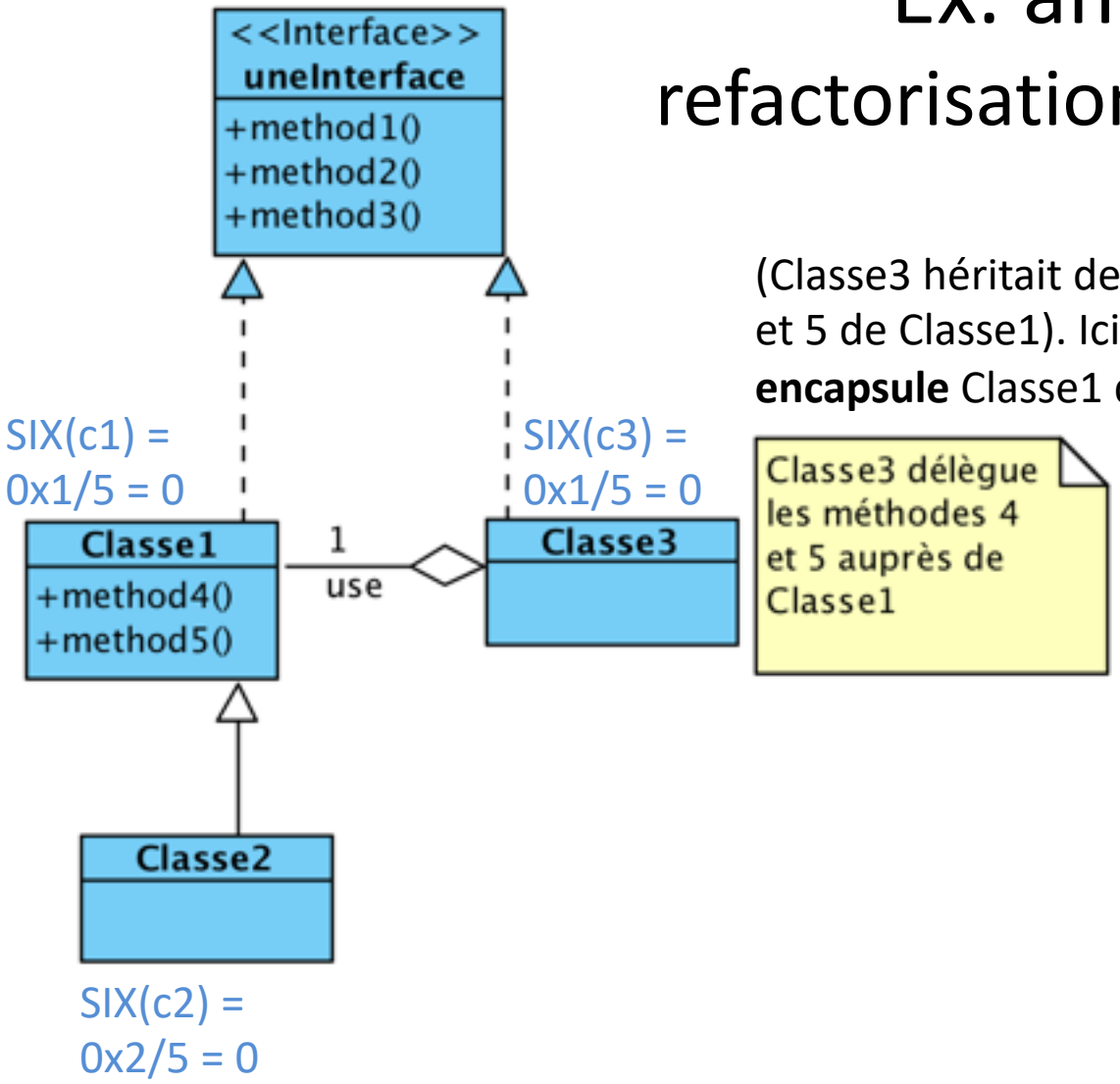
Classe1 : (0 * 1) / 5 = 0
Classe2 : (0 * 3) / 5 = 0
Classe3 : (5 * 3) / 5 = 3 – **Moy du projet : 1**

Interprétation SIX

- Un indice de spécialisation **trop grand (>1.5)** :
  - La classe redéfinit trop de méthodes
    - Une entité hérite d'une autre alors qu'il s'agit d'une classe très spécialisée
  - ou la profondeur est trop importante

➔Essayer de **factoriser** ou d'utiliser des **interfaces**

Ex. amélioration : refactorisation avec une interface



(Classe3 héritait des méthodes4 et 5 de Classe1). Ici on **encapsule** Classe1 dans Classe3 :

SIX(projet) = 0 (meilleure que 0,6)

Ici aucune classe ne REDEFINIT de méthode

RMI	**Instability : CE / (CA + CE)** : ce nombre indique l'instabilité du projet, c'est-à-dire les dépendances entre les paquets

CE (ou CBO)	**Efferent Coupling (ou Coupling between Objects)** : le nombre de classes *dans* un package qui dépendent d'une classe d'un autre package.
CA	**Afferent Coupling** : le nombre de classes *hors* d'une package qui dépendent d'une classe dans le package

Cet indice d'instabilité est toujours compris entre 0 et 1

Bon RMI : proche de 0

le paquetage peut être considéré comme stable

Exercice : Instabilité RMI

RMI = Ce / (Ca + Ce)

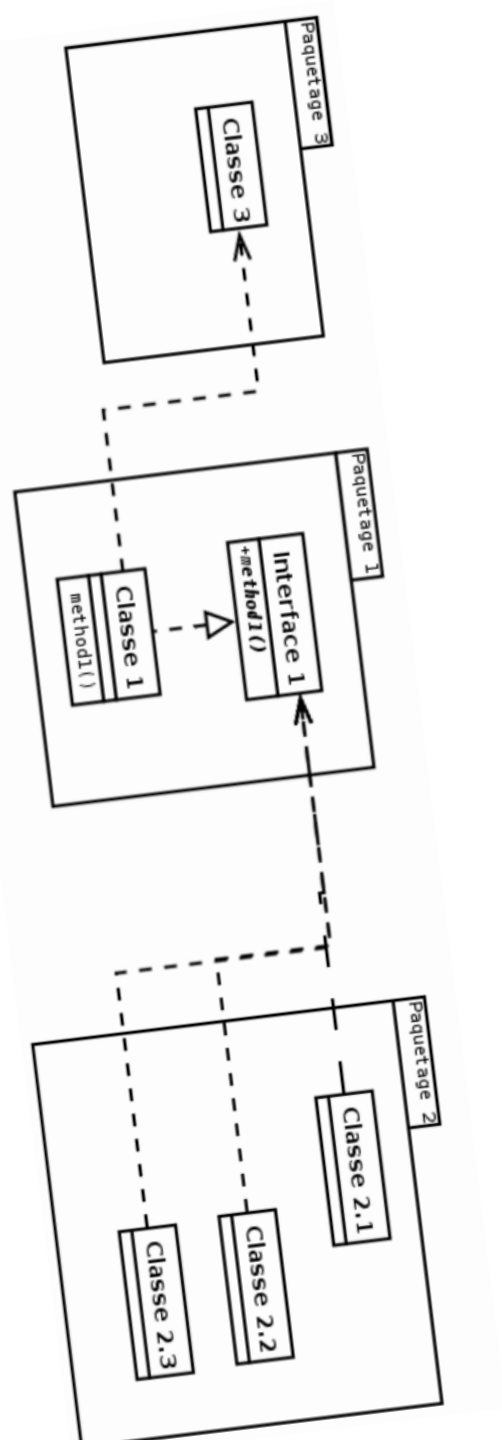
Ce (efferent coupling) =

dépendance vers extérieur

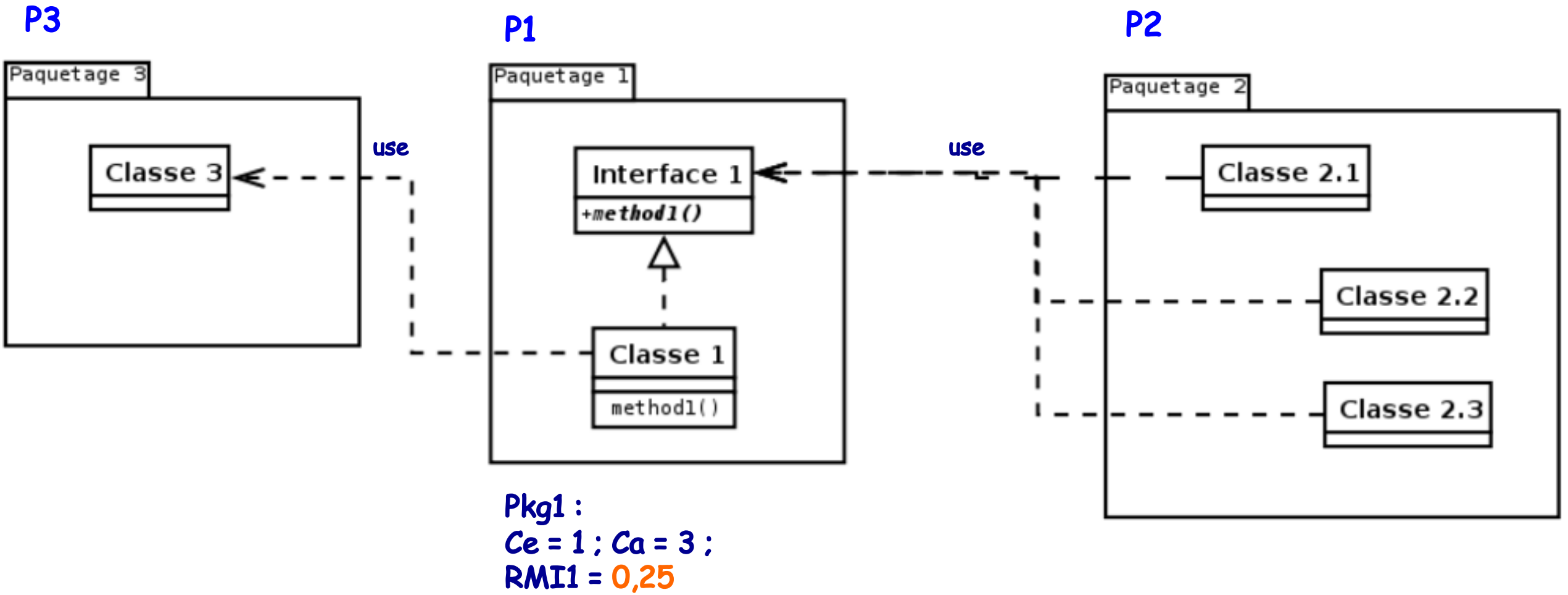
Ca (afferent coupling) =

responsabilité

instabilité



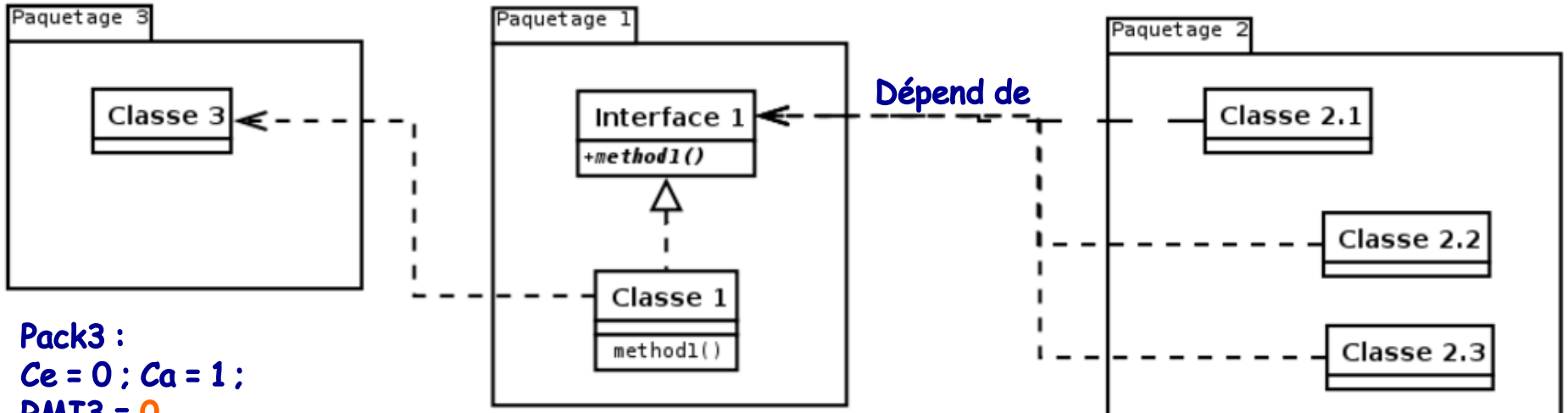
Exercice Instabilité



Pkg1 :
Ce = 1 ; Ca = 3 ;
RMI1 = 0,25

Corrigé exercice RMI - Instabilité

0 ≤ RMI ≤ 1



Pack3 :
Ce = 0 ; Ca = 1 ;
RMI3 = 0

stable

Pack1 :
Ce = 1 ; Ca = 3 ;
RMI1 = 0,25

Assez stable

Pack2 :
Ce = 3 ; Ca = 0 ;
RMI2 = 1

instable

RMI global =
Moy des RMIi = 0,42

Interprétation RMI

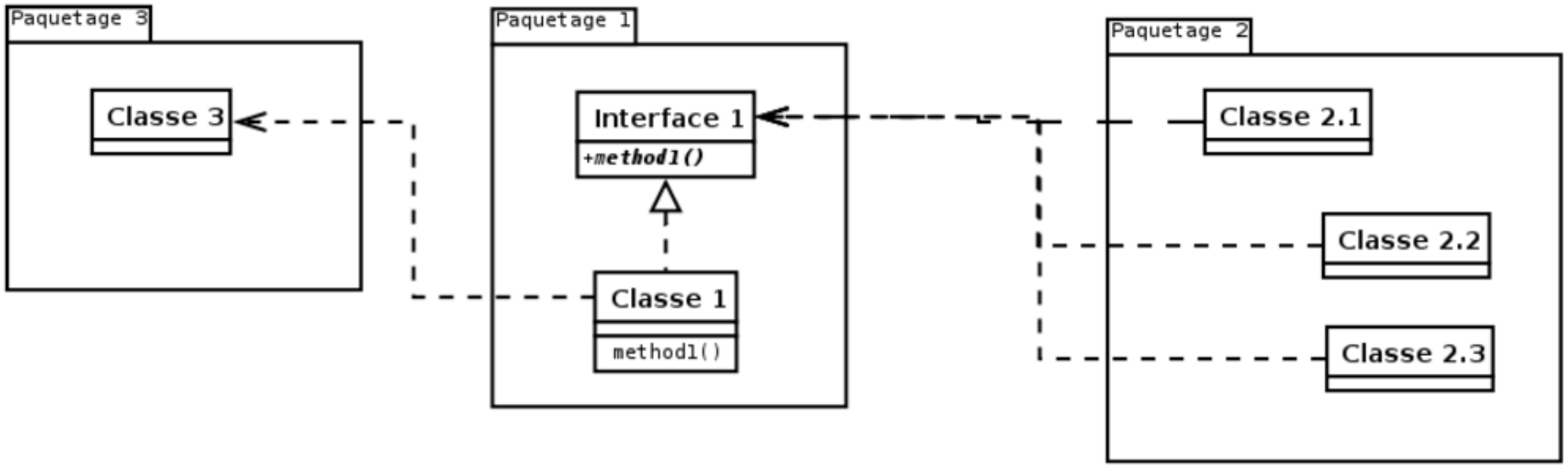
- RMI > 0 : fait ressortir les paquetages **qui dépendent plus des autres** que les autres ne dépendent d'eux.
  - Ces packages peuvent poser des pbs de fiabilité, puisqu'une modification dans un des paquetages dont ils dépendent impacte potentiellement leur fonctionnement
  - (on ne maîtrise pas leur fonctionnement)

RMI : bonnes pratiques

- C'est toujours mieux d'avoir **un seul package instable** et les autres stables
- Sur les packages « à risque »
  - On mettra un maximum de tests (unitaires + intégration)
  - Les premiers à vérifier en cas de problème

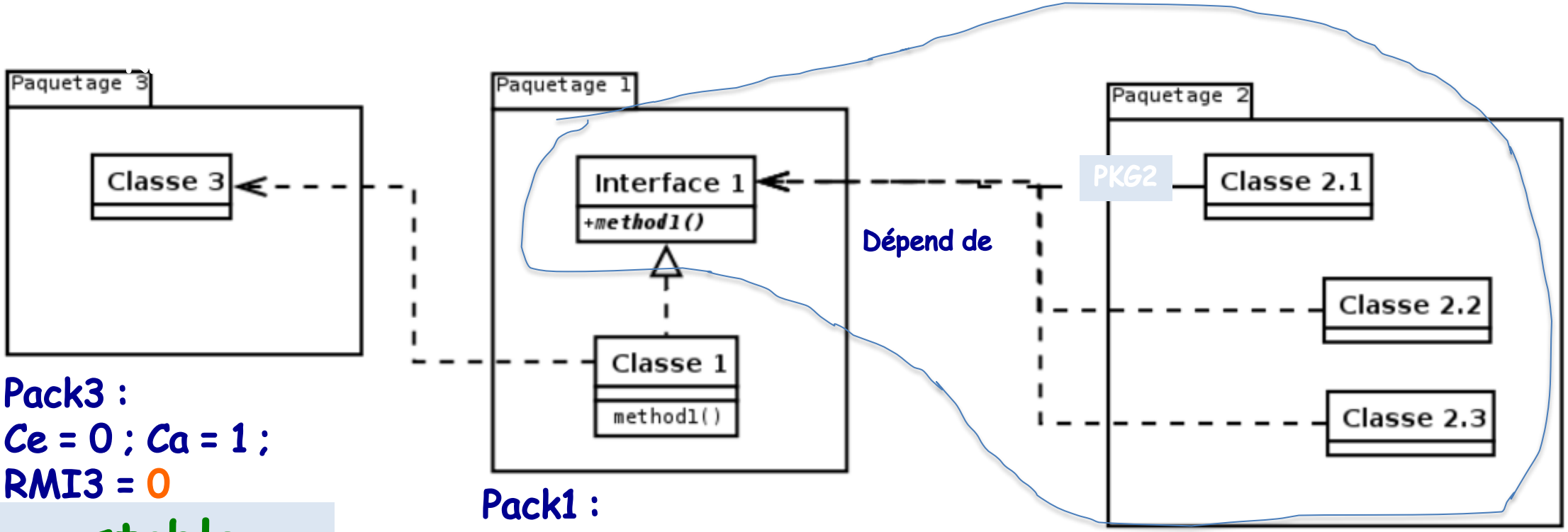
- Il faut ÉVITER les **dépendances cycliques** entre packages
  - Cas où un *packA* référence un *packB* qui référence un *packC* qui référence à son tour le *packA*...
  - Les cycles de vie sont liés et tous ces éléments ne peuvent ni être utilisés ni être modifiés séparément.

- Quelle solution proposer pour améliorer ce RMI global ?



- **Ex.: déporter l'interface1 dans le PKG2**

Amélioration RMI ?



Pack3 :
Ce = 0 ; Ca = 1 ;
RMI3 = 0

stable

Pack1 :
Ce = 2 ; Ca = 0 ;
RMI1 = 1

instable

Pack2 :
Ce = 0 ; Ca = 1 ;
RMI2 = 0

stable

RMI global =
Moy des RMIi = 1/3 = 0,33 au lieu de 0,42 : c'est MIEUX

RMI : bonnes pratiques (2)

- Attention, dans une architecture logicielle, certains paquetages **doivent** être instables.
- Pour ces paquetages, il faut alors considérer un autre indicateur: la **DMS** (Distance from the Main Sequence)

DMS	**Normalized Distance : RMA + RMI – 1.** Ce nombre devrait être petit (proche de zéro) pour indiquer une bonne conception des paquets

DMS

La DMS représente en fait l'équilibre qui doit résider entre *le niveau d'abstraction* et *l'indice d'instabilité*.

- Le paquetage est instable mais **possède peu d'interfaces**,
- Un certain nb d'autres paquetages dépendent de ce paquetage, mais celui-ci possède **beaucoup d'interfaces**.

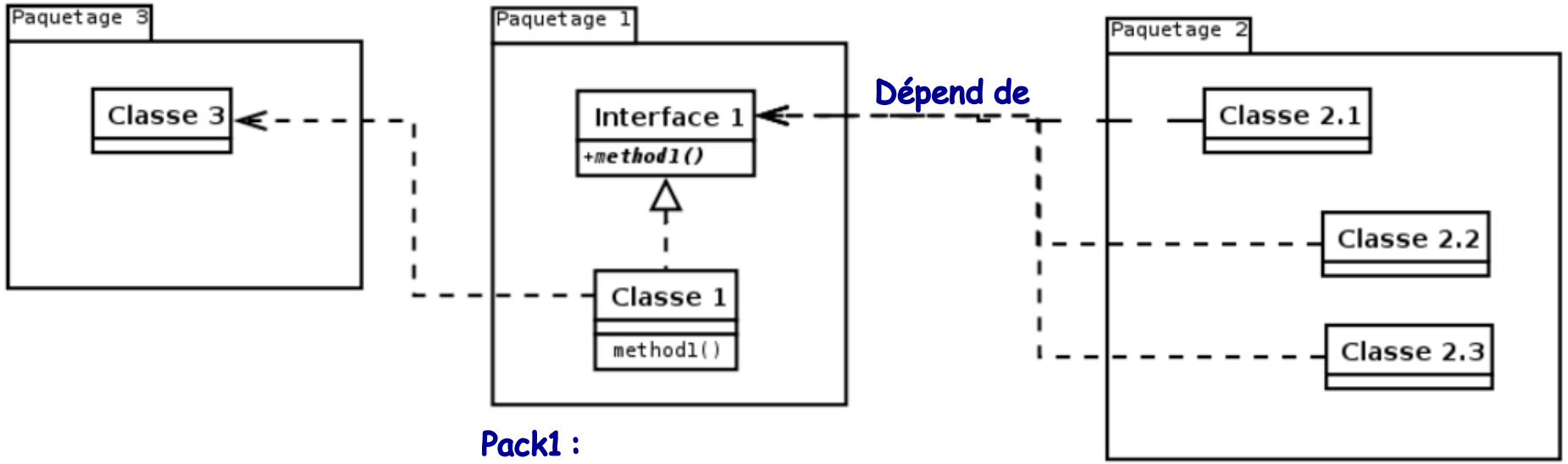
Distance from de Main Sequence (DMS)

$$DMS = | RMA + RMI - 1 |$$

(% de classes abstraites et d'interfaces + Instabilité -1)

Bonne DMS = valeur proche de 0

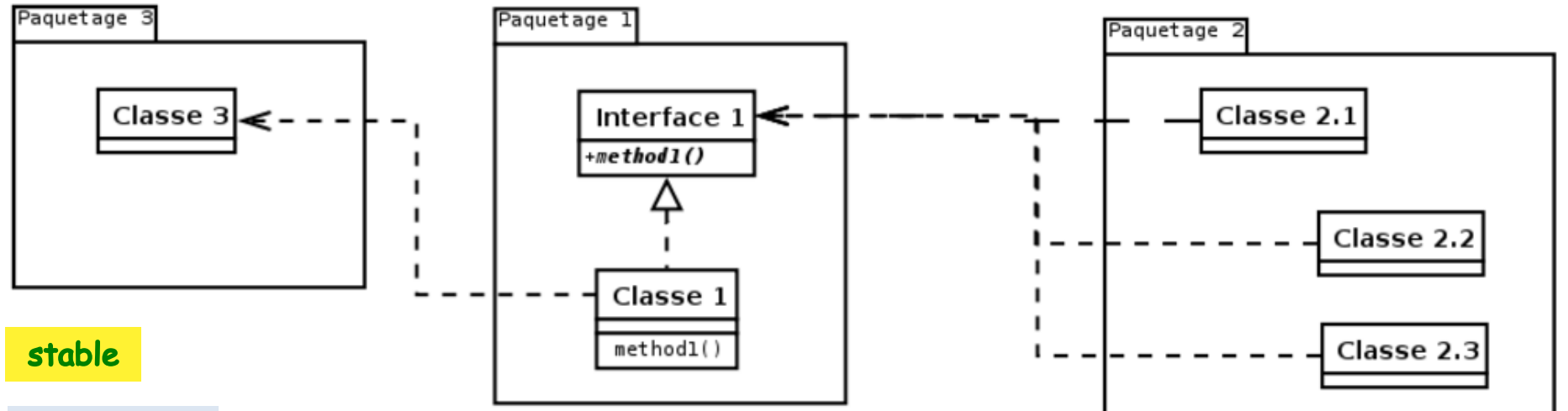
Exercice précédent : DMS



Pack1 :
Ce = 1 ; Ca = 3 ;
RMI1 = 0,25

Calculer la DMS par package

Corrigé DMS



stable

DMS3 : 1
mauvaise

Assez stable

DMS1 : 0,25
correcte

instable

DMS2 : 0 bonne

DMS(P1) = abs(RMI1 + RMA -1) = abs(0,25 + 0,5 -1) → 0,25 : DMS plutôt correcte

DMS(P2) = abs(1 + 0 – 1) → 0 package instable mais DMS OK

DMS(P3) = abs(0 + 0 – 1) → 1 (pas bon)

Interprétation DMS

- Ici l'architecture du paquetage **instable P2** est cohérente avec le besoin que les classes extérieures ont de lui :
  - Il n'est pas sollicité, il n'a donc pas besoin de beaucoup d'interfaces.
  - Lui-même dépend d’une *interface* dans un autre package

Interprétation DMS (suite)

- La DMS du paquetage **stable P3** est mauvaise (elle vaut 1)
- Ce paquetage ne possède pas d'interface alors qu'une autre classe dépend de lui
  - Si on crée une interface, la DMS passe à 0,5

- D'une façon générale, quand un package est stable mais a une mauvaise DMS, il faudrait le **refactoriser en ajoutant des interfaces** pour les classes très utilisées.

Complexité cyclomatique du code (VG)

VG	**McCabe Cyclomatic Complexity : la complexité *cyclomatique* d'une méthode. C'est-à-dire le nombre de chemins possibles à l'intérieur d'une méthode.**
MLOC	**Method Lines of Code** : nombre total de lignes de codes dans les méthodes (idem, les lignes blanches et les commentaires ne sont pas comptabilisés)

Pour info : le site https://www.openhub.net/tools publie quelques-unes de ces métriques concernant de nombreux projets sous licence libre, ou par langage

Complexité cyclomatique VG

- Mesure la **complexité structurelle** du code, très utilisée
  - C'est le nombre de chemins **linéairement indépendants** qu'il est possible de suivre au sein d'une méthode
  - Un code **purement séquentiel** a une VG de 1

- Un programme dont le flux de contrôle est complexe :
  - requiert **beaucoup de tests** pour atteindre une bonne couverture du code
  - est **moins facile** à maintenir.

Calcul de VG

- Pour un bloc de code (par ex., une méthode) :

$$VG = e - n + (2\ p)$$

Formule de McCabe

  - *e* nombre d'arcs
  - *n* nombre de nœuds
  - *p* nombre de composantes connexes du graphe

- Ou encore, autre méthode de calcul pour les méthodes :
  - On part de **1**
  - Chaque bloc condition (if, exprl ? expr2 : expr3), chaque itération (for, while), chaque try/catch, chaque switch/case, chq return si pas dernière instruction :
    - **incrémente VG de 1**

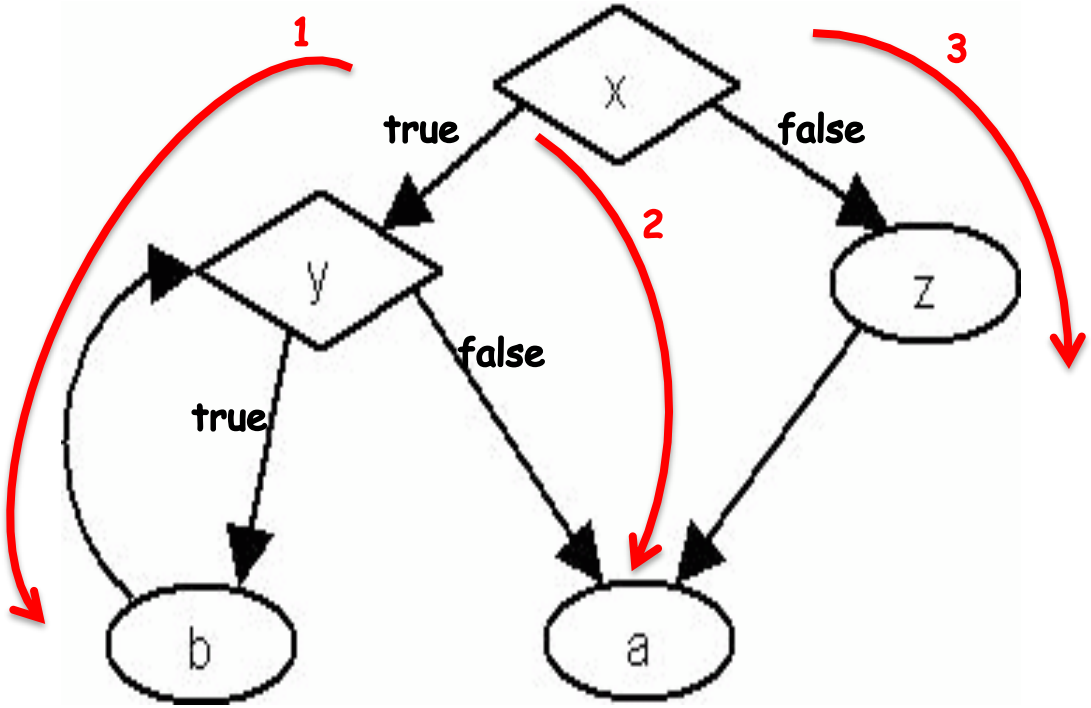
Illustration VG

```
if(x)
    while(y)
        b;
else
    z;
a;
```

1

+1

+1



e nombre d'arcs = 6

n nombre de nœuds = 5

p nombre de parties du graphe connectées, avec le sommet : ici 1

VG = e – n + (2 p) = 6 – 5 + 2x1 = 3

Calculer la VG de ce code

```
public void process(Car myCar) {          // +1
    if ( myCar.isNotMine() ){             // +1
        return;                           // +1
    }
    car.paint("red");
    car.changeWheel();
    while ( car.hasGazol() && car.getDriver().isNotStressed() ){  // +2
        car.drive();
    }
    return;
}
```

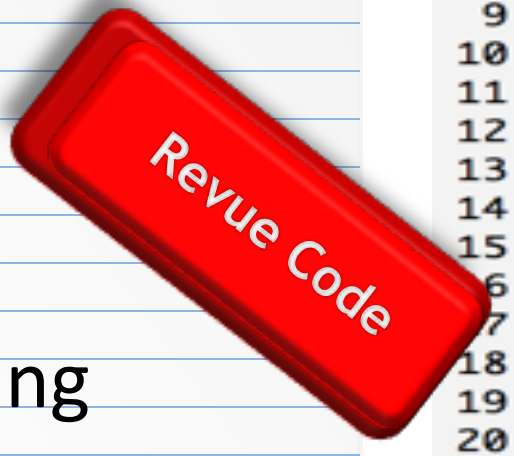
Interprétation VG

- Si une méthode a une complexité cyclomatique **trop élevée (au delà de 10)**
  - elle doit être refactorisée
- Une complexité cyclomatique inférieure à 10 et > 6 :
  - acceptable si la méthode est suffisamment testée

  http://www.mccabe.com/nist/nist_pub.php

- **Weighted Methods per Class (WMC)**: la somme de la complexité cyclomatique de McCabe pour toutes les méthodes de la classe
  - $WMC = \sum c_i$
  - c_i *complexité de la méthode i*

Revue de code C



- Analyser le listing fourni (fichier main.c sur le wiki LIRIS) et calculer la complexité Vg du code

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  void extract(char* char1, char*
6
7  int eval1(char* ch)
8  {
9       int i;
10      int valeur1, valeur2;
11      int lgval2;
12      char *val1, *val2;
13      char operation;
14      int resultat;
15      /* Recherche d'un opérateur
16      for( i=0 ; *(ch+i) !='+' &&
17      {
18      }
19      /* Traitement des erreurs */
20      if(i==0) /* Le premier opèra
21      {
22           printf("erreur : pas de
23           exit(0);
24      }
25      else if(i==strlen(ch)-1) /*
26      {
27           printf("erreur : pas de
28           exit(0);
29      }
30      else if(i==strlen(ch)) /* Il
31      {
32           printf("erreur : pas de
33           exit(0);
34      }
35      /* char Extraction de la cha
36      premier opérande */
37      val1=(char*) malloc((i+1)*siz
```

Autres Métriques (fin)

Nested Block Depth (NBD): La profondeur du code

Lack of Cohesion of Methods (LCOM): mesure la cohésion d'une classe. **Plus LCOM est petit et plus la classe est cohérente**.

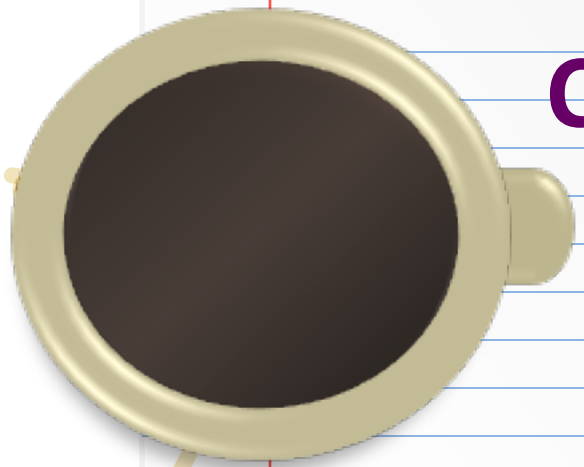
Un nombre proche de 1 indique que la classe pourrait être découpée en sous-classes.

Calcul avec la méthode *d'Henderson-Sellers*

Soit m(A), le nombre de méthodes accédant à un attribut A ; on calcule E(m) = moyenne des $m(A_i)$ pour tous les attributs A_i, et on calcule :

$$LCOM = (m - E(m)) / (m - 1)$$

Exercice LCOM Cohésion



- Que penser de la cohésion de cette classe PHP ? (sans aucun calcul)

```php
<?php
class Example {
  private int $a;

  public function m1() {
      $this->a= a+1;
      $this->m2();
   }
   public function m2() {
      $this->a = a - 67;
   }
   public function m3() {
      $this->a = a/192;
      echo 'On a divisé par 192';
   }
   public function m4() {
      $this->m5();
      echo 'dans m4';
   }
   public function m5() {
      echo 'OK, dans m5';
   }
}
```

Corrigé exercice cohésion LCOM

- m1() appelle m2()
- m2() partage avec m1() et m3() un attribut commun (*a*)
- m4() appelle m5()
- Interprétation ?
  - 2 flux de méthodes, indpts
  - 2 « responsabilités »

- LCOM = 1 – (5-3 méth accédant à *a*) / 5-1)
  = 1-2/4
  = 0,5

Dans l'idéal, LCOM doit être proche de 0

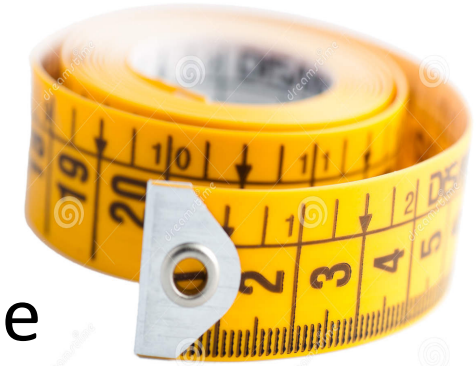
```
<?php
class Example {
  private int $a;

  public function m1() {
     $this->a= a+1;
     $this->m2();
  }
  public function m2() {
     $this->a = a - 67;
  }
  public function m3() {
     $this->a = a/192;
     echo 'On a divisé par 192';
  }
  public function m4() {
     $this->m5();
     echo 'dans m4';
  }
  public function m5() {
     echo 'OK, dans m5';
  }
}
```

Interpréter les métriques

- Telle classe ou tel fichier a une complexité trop élevée, et alors ?
  - Tenir compte du type de logiciel / framework (jeux, inventaire, …)
  - Ce qui est important est de surveiller l'**accumulation d'indicateurs** et **leur orientation générale**
  - Ex. si une seule classe a un peu trop de lignes de code ce n'est pas grave ;
  - si les classes d'un paquet ont une *Complexité Cyclomatique* élevée, une *LCOM* élevée, et un *Indice de maintenabilité* faible, alors il faut agir.

Utilité



- Les métriques de ligne de code, le nombre cyclomatique de McCabe, l'index de maintenabilité et autres métriques (par ex. les métriques Halstead) sont des moyens efficaces pour mesurer :
  - la complexité, la qualité et la maintenabilité d'un logiciel
- Elles peuvent servir à mesurer la qualité d'un code
  - Clients : demandeurs !
- On peut ainsi localiser les modules particulièrement difficiles à tester et maintenir
  - Corriger plutôt que de garder des modules susceptibles d'être cher en maintenance

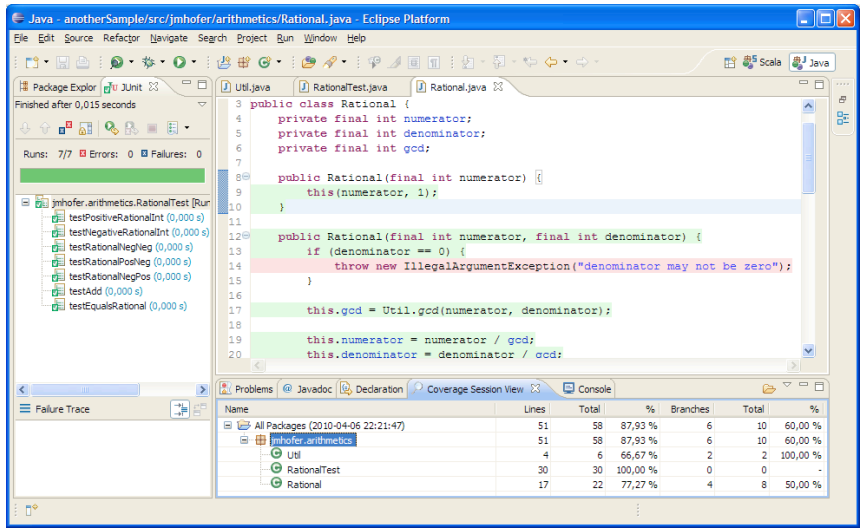
À vous de choisir !

- La qualité d'un logiciel est un sujet qui divise :
  - Certains pensent qu'il s'agit d'un **surcoût** et la voient comme une contrainte,
  - D'autres au contraire pensent qu'il s'agit d'une **opportunité** et voient la qualité comme un guide de travail.
    - Ceux-là opposent, au surcoût induit par la qualité, le coût induit par le *manque de qualité* d'un logiciel
    - **VOC : le manque de qualité logicielle est appelé** « la dette technique »
      - Parce qu'un jour, les défauts vont générer des coûts

Outils avec métriques

http://java-source.net/open-source/code-analyzers

- En Java, un grand nombre d'outils libres sont disponibles :
  - **SonarQube**
  - Cobertura
  - Crap4J
  - PMD
  - FindBugs
  - Eclipse *plug-in* Metrics1-3-6
  - Jdepend



- Nombreux sont intégrés aux **outil d'intégration continue**
  - Comme *Hudson, Jenkins, Bamboo, etc.*

Je retiens…

- Ce qu'est l'architecture logicielle
- La part des étapes dans le cycle de vie du logiciel

- Les métriques SIX, RMI, VG, LCOM

- La problématique de la qualité logicielle
  - Dette technique