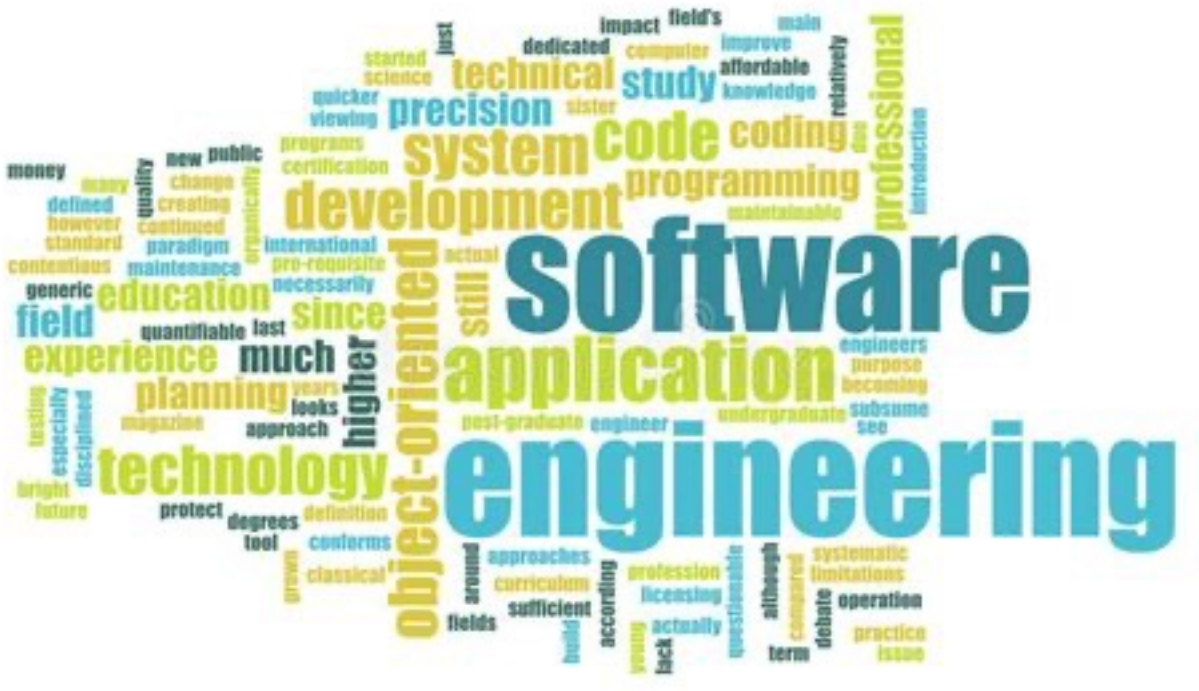
Chapitre 6 – Qualité Logiciel / Clean Code



Véronique DESLANDRES - LP DevOps, IUT Univ. Lyon1

IUT Lyon 1
l'excellence technologique

1- Evolution des applications

Croissance continue de la complexité des systèmes

- **Couverture élargie et fonctionnalités** plus **évoluées**
  - Ex. passage d'un progiciel Achat… à un ERP (Enterprise Ressource Planning)
  - Ex.: systèmes auto-adaptatifs (capables de s'administrer et de réparer des problèmes seul)
- Explosion des services tiers et du cloud
  - Problèmes de sécurité
- 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

➔ Mode agile, DevOps, DevSecOps

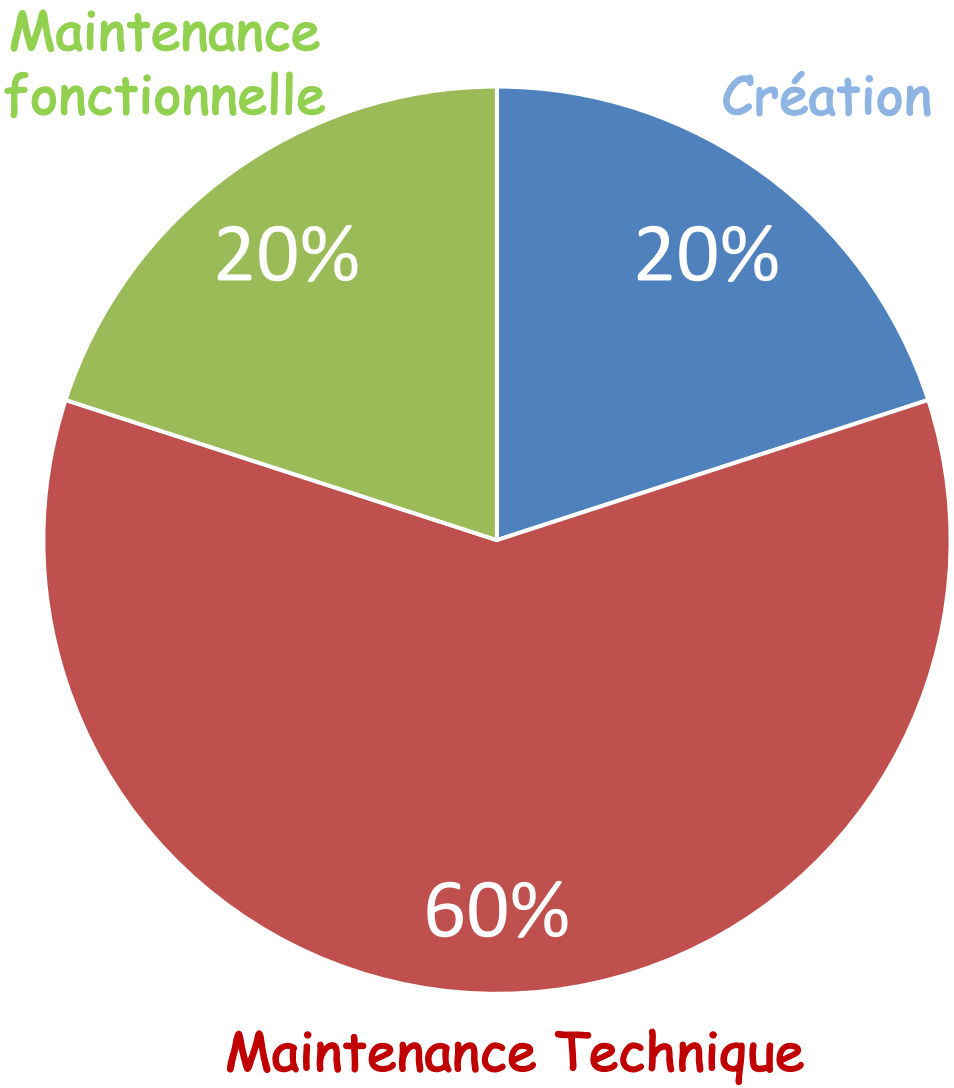
2- Évolution des matériels et technologies

- 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 (Adobe) et HTML5 (open-source, W3C)** pour une UX évoluée (2010/2015)

- Des besoins sans cesse élargis
  - Hétérogénéité, ouverture, sécurité, étendabilité (*scalability, passage à l'échelle*), gestion des défaillances, concurrence, intégration, transparence, communication, interopérabilité, *responsive design*

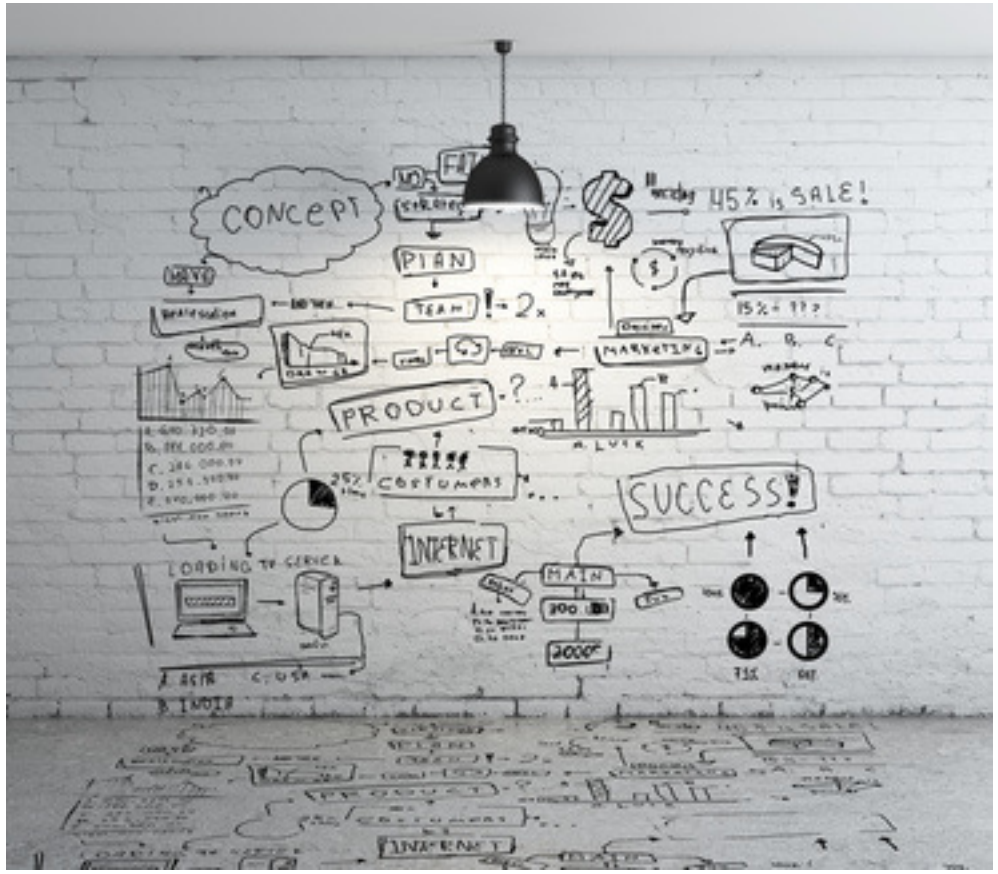
Coût du développement logiciel



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



QUALITÉ LOGICIELLE / ASSURANCE QUALITÉ

Définition

AFNOR

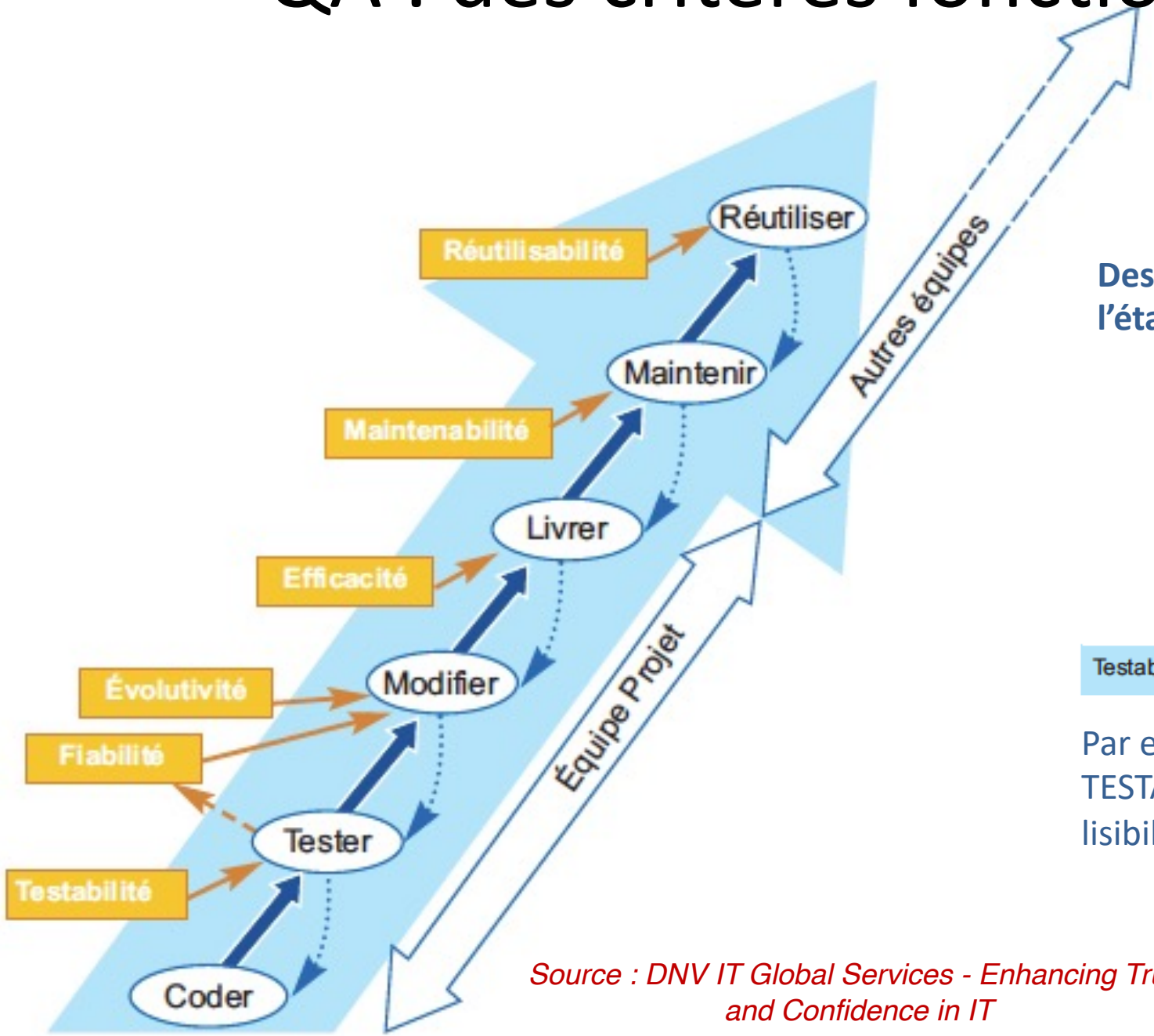
Un « bon logiciel » ou logiciel de qualité, s'entend comme un logiciel capable de **répondre parfaitement aux attentes du client**, le tout sans défaut d'exécution

➔ Nécessite de mesurer : métrologie du logiciel

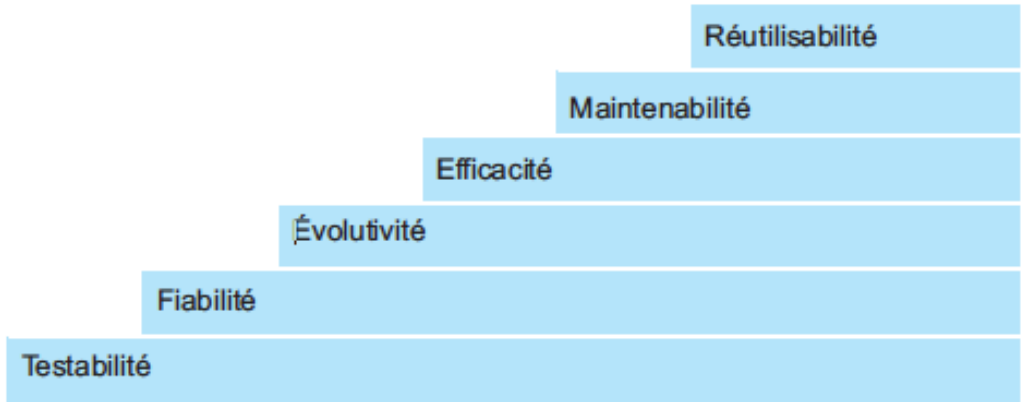
Quelques facteurs de qualité logicielle

- Ergonomie
- Fiabilité
- Efficacité,
- Intégrité
- Maintenabilité,
- Réutilisabilité,
- etc.

QA : des critères fonction de l'étape



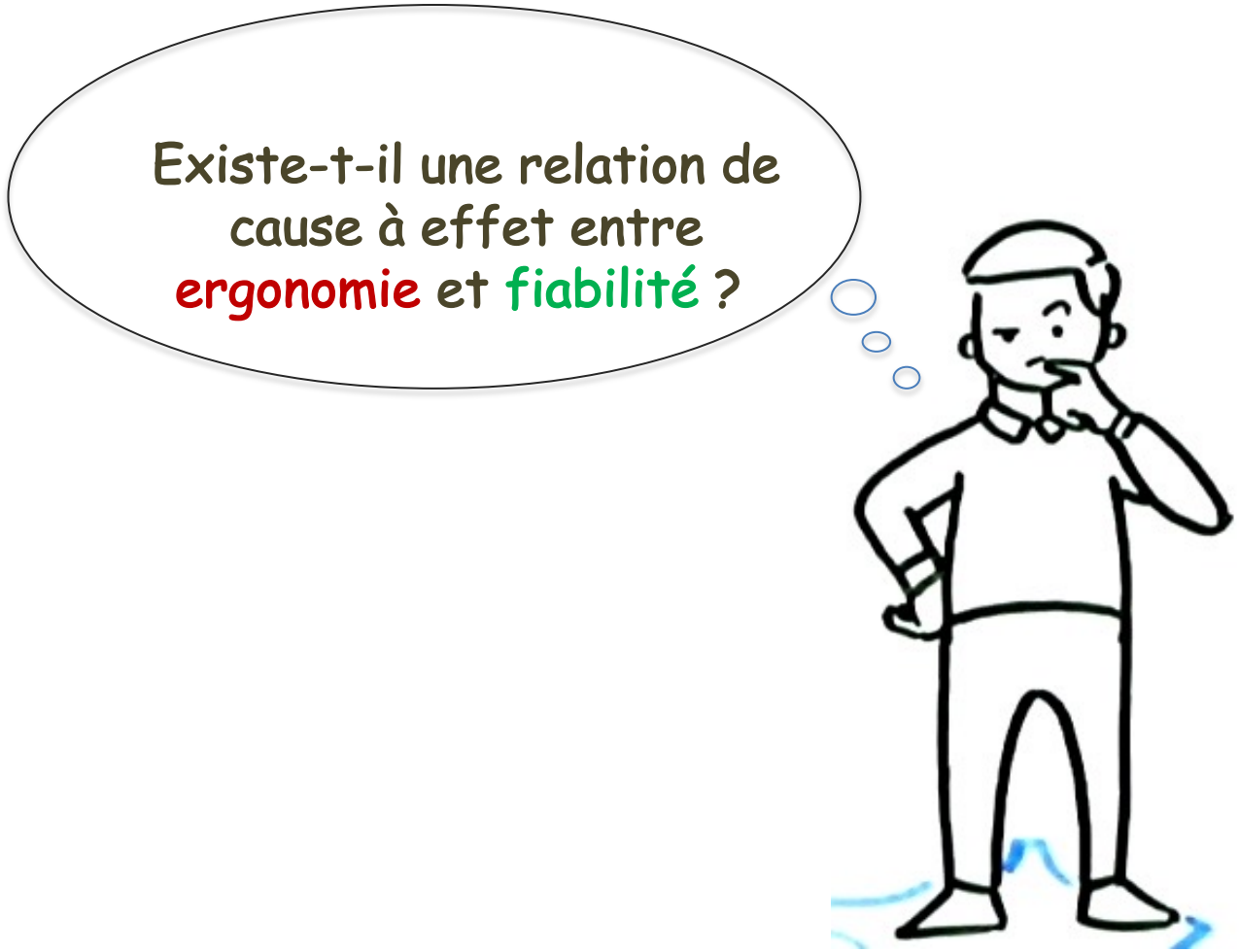
Des exigences Qualité qui diffèrent selon l'étape du cycle de développement :



Par ex. tests unitaires et tests d'intégration au niveau TESTABILITE, la MAINTENABILITE est décomposée en lisibilité et compréhensibilité.

Source : DNV IT Global Services - Enhancing Trust and Confidence in IT

- Tous ces facteurs dépendent aussi du point de vue de la personne (utilisateur, exploitant) et certains sont antinomiques



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.

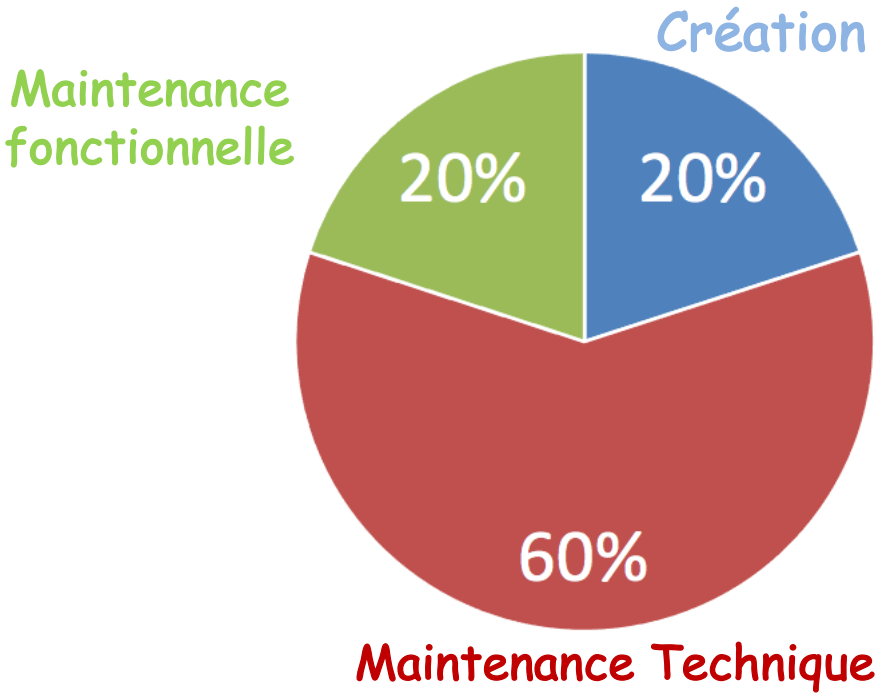
Un logiciel ergonomique évitera en effet les *erreurs de manipulation* de l'utilisateur, mais **pas les défauts de fonctionnement** s'il y en a.

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

Métrologie du logiciel / Qualimétrie

- Enjeux = **qualifier** le code, définir le **bug**, et **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**
- La **couverture par les tests**
- La **facilité de maintenance (extensibilité du code)**

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.

→ 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)

- Le nombre de classes abstraites (AC) et concrètes (CC) est un **indicateur d'extensibilité d'un package**.

- Plus AC est important, plus les entités qu'elles implémentent peuvent être étendues indépendamment les unes des autres.

Ex. de Métriques Logicielles (3)

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

Ex. de Métriques Logicielles (4)

NOI	**Number of Interfaces :** le nombre d'interfaces
A	**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 sont à interpréter en fonction 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.

Critère qualité : **Abstractness**

- Mesure le degré d'abstraction du package.
- Pourcentage de classes abstraites sur le total de classes.
  - Proche de 0 : package concret, proche de 1 : package abstrait.

- **Le degré d'abstraction d'un package doit tendre vers l'une ou l'autre des deux borne : 0 ou 1.**
- Une valeur proche de 0.5 traduirait une mauvaise écriture du code.

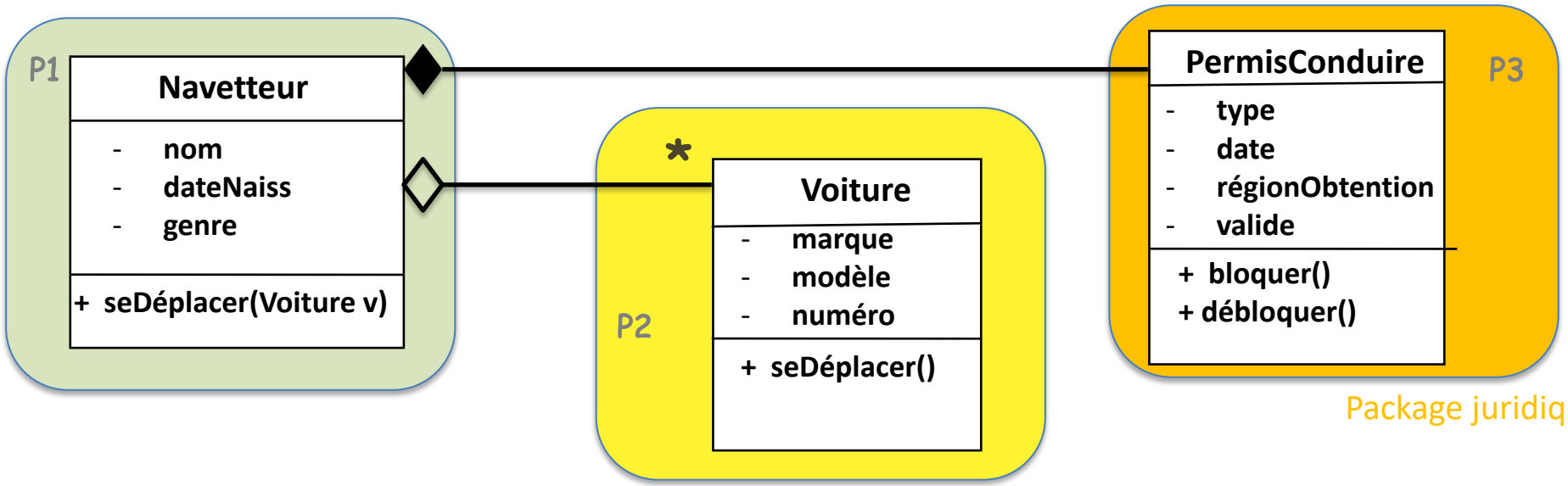
CE (ou CBO)	**Efferent Coupling (ou Coupling between Objects)** : le nombre de classes du package qui dépendent de composants externes. Lien « Utilise »
CA	**Afferent Coupling** : le nombre de classes *hors* d'un package qui dépendent d'une classe du package. « Est utilisé par »

CE : dépendance vers l'Extérieur, dEscendante

CA : responsAbilité vis-à-vis des classes externes, dépendance Ascendante

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

Exemple : un logiciel gérant des conducteurs de voiture



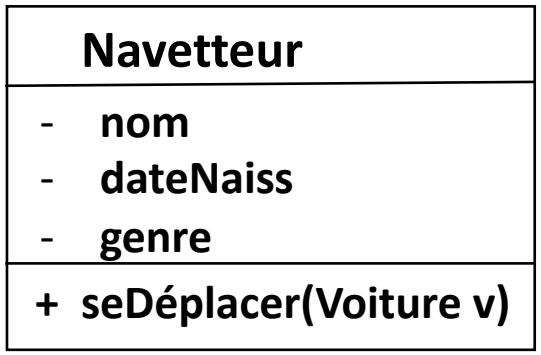
Package juridique

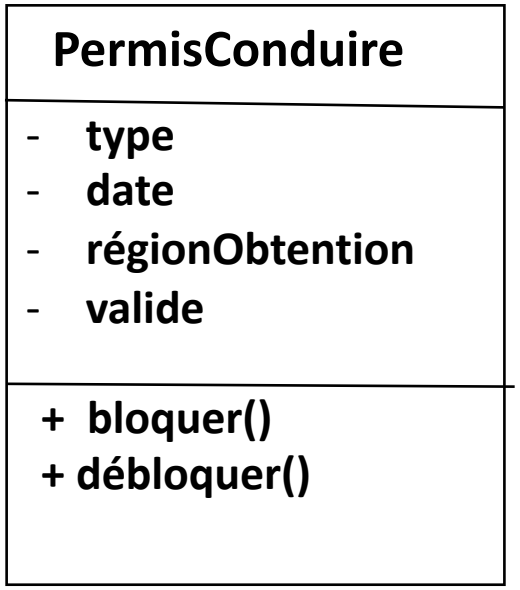
CE(P1) = 2

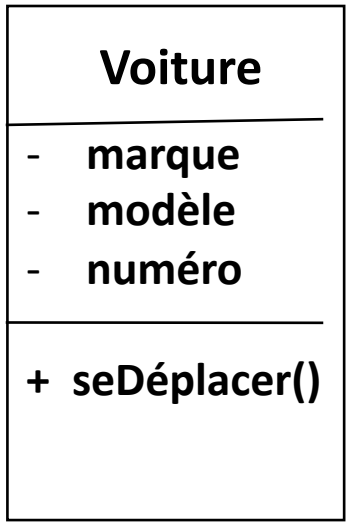
CA(P1) = 0

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 du package qui dépendent de composants externes. Lien « Utilise »
CA	**Afferent Coupling** : le nombre de classes *hors* d'une package qui dépendent d'une classe du package. « Est utilisé par »

- Le nombre de dépendances efférentes (descendantes) est un indicateur d'indépendance du code. Plus ce nombre est faible, mieux c'est.

- CE élevé : les classes sont sans doute complexes et ont trop de responsabilités → *refactoring*

CA : dépendance ascendante

- CA élevé :
  - soit mauvaise gestion des dépendances,
  - soit le package est le centre de l'application.

- Plus CA est grand, plus il faut se demander s'il n'est pas nécessaire de fragmenter le package.

Indice de **Spécialisation** : **SIX**

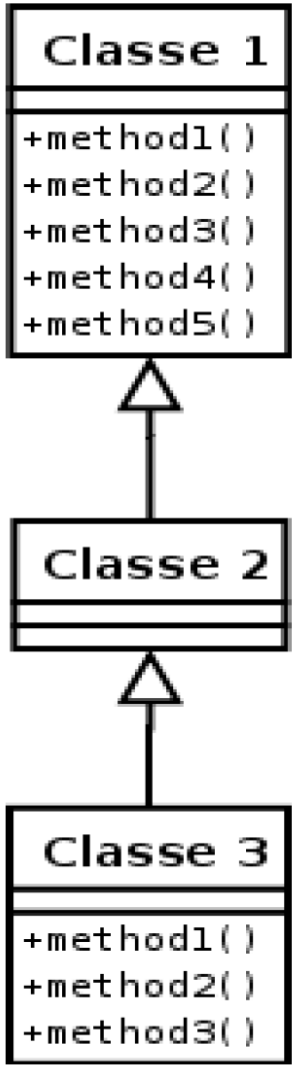
SIX	**Specialization Index : niveau de spécialisation d'une classe. C'est le calcul :** NORM * DIT / NOM. Pour tout le projet : moyenne des index des classes.
NORM	**Number of Overridden Methods** : nombre de méthodes redéfinies

(NOM : Number of Methods)

- Cet index permet de **savoir si une classe est à sa place dans l'arbre d'héritage**. L'idéal est d'avoir SIX = 0.
- 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, l'héritage n'est pas nécessaire.

Exercice Spécialisation SIX

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



Et si Classe3 redéfinit les 5 méthodes ?

Métrique SIX

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 * 2) / 5 = 0
- Classe3 : (3 * 3) / 5 = 9/5 = 1,8

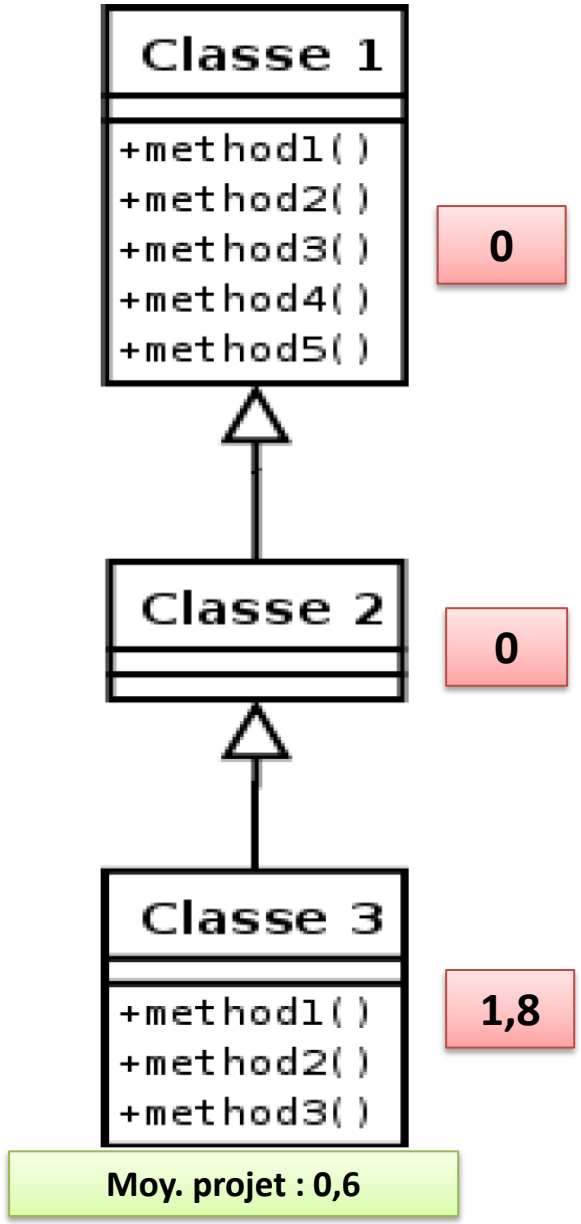
Moyenne : 0,6

Si Classe3 ne redéfinit aucune méthode :

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

Si Classe2 et 3 redéfinissent les 5 méthodes :

Classe1 : (0 * 1) / 5 = 0
Classe2 : (5 * 2) / 5 = 2
Classe3 : (5 * 3) / 5 = 3 - **Moy du projet : 1,7**



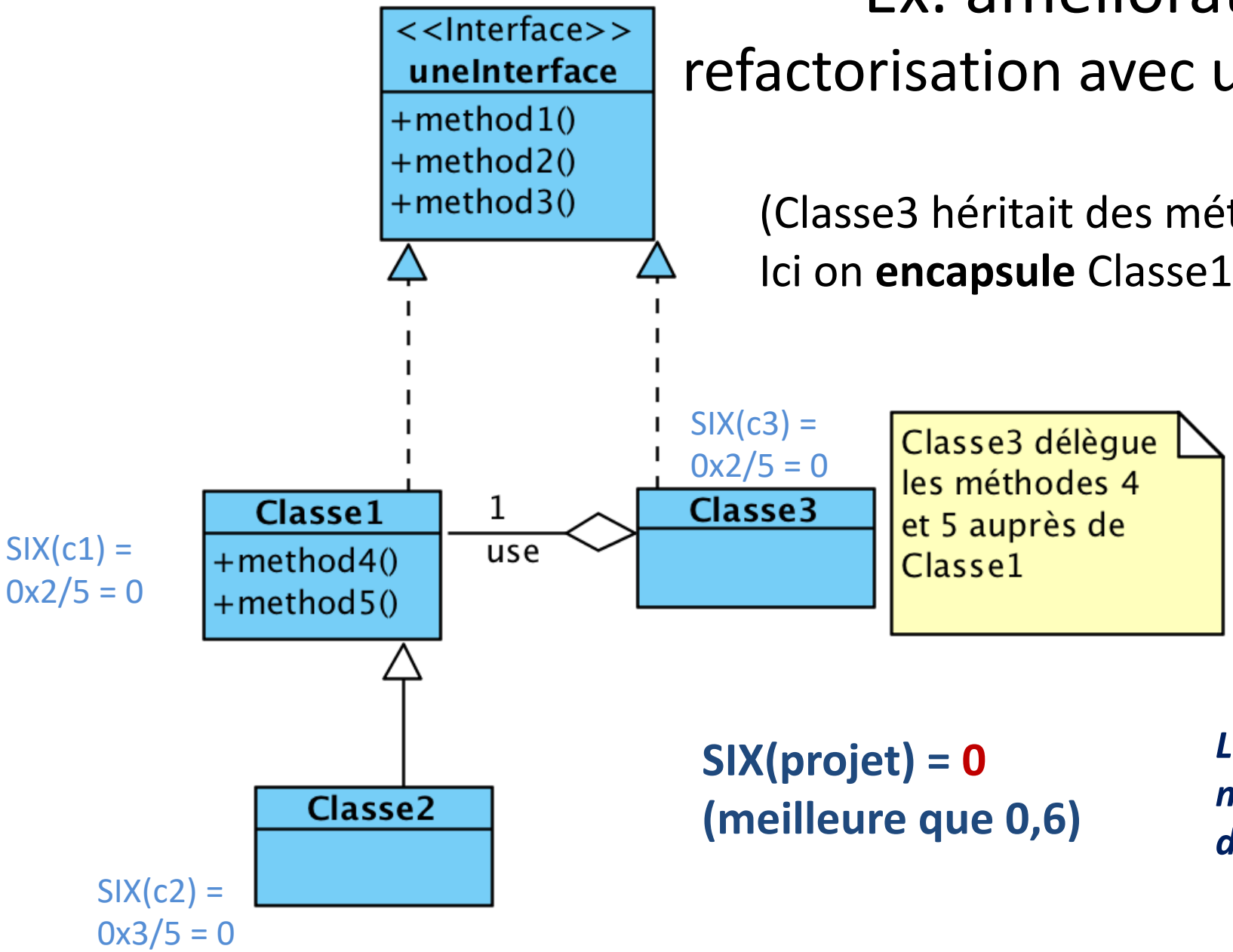
Interprétation SIX

- Un indice de spécialisation **trop grand (>1.5)** :
  - Soit les classes redéfinissent trop de méthodes
    - Une entité hérite d'une autre alors qu'il s'agit d'une classe très spécialisée
  - Soit 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)

La profondeur augmente ici, mais aucune classe ne REDEFINIT de méthode

(Exemple d'implémentation de Classe3)

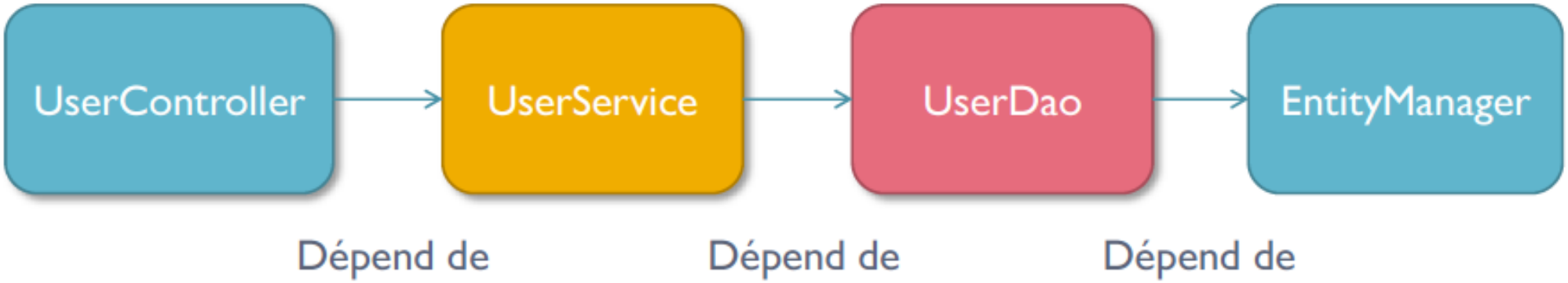
```
public class Classe3 implements uneInterface {

   private Classe1 maClasse1;

      // usage de method4() et method5 : par le délégué

      Classe3() {
         uneVar = maClasse1.method4() ;
         uneAutreVar = maClasse1.method5();
         .....
       }
}
```

Je retiens :

Les métriques CE / CA
La métrique SIX

Une autre métrique

INDICE D'INSTABILITÉ : I OU RMI



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** : le nombre de classes du package qui dépendent de composants externes. Lien « Utilise »
CA	**Afferent Coupling** : le nombre de classes *extérieures* qui dépendent du package. Lien « est utilisé »

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

Bon RMI (ou I) : proche de 0

le paquetage peut être considéré comme stable

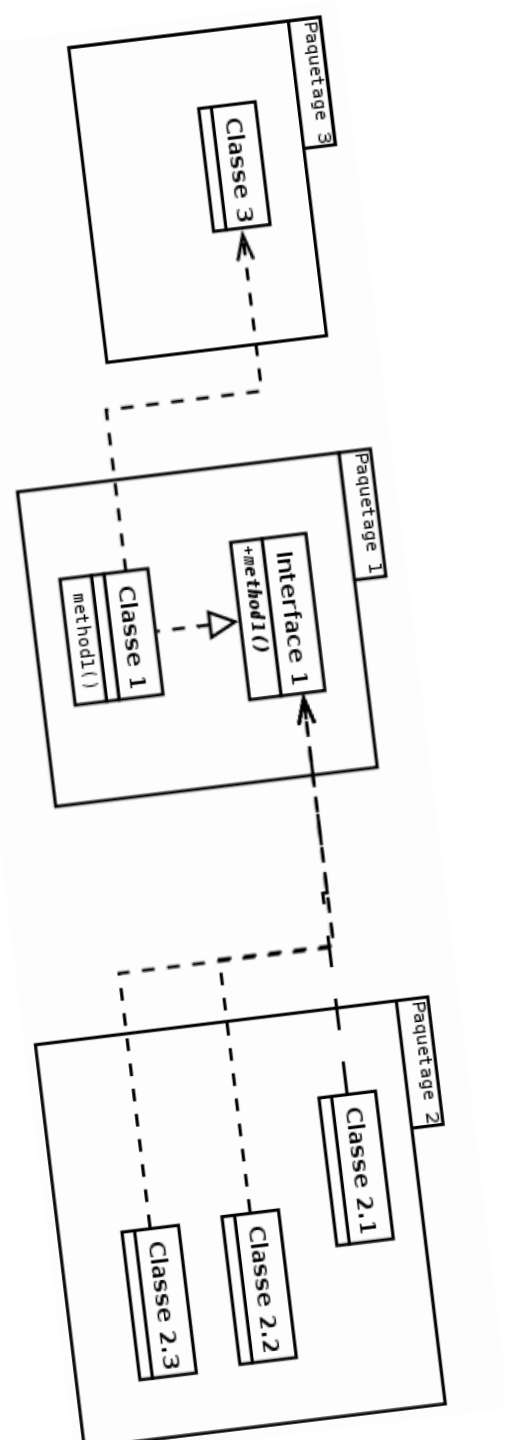
Exercice : Instabilité RMI

RMI = Ce / (Ca + Ce)

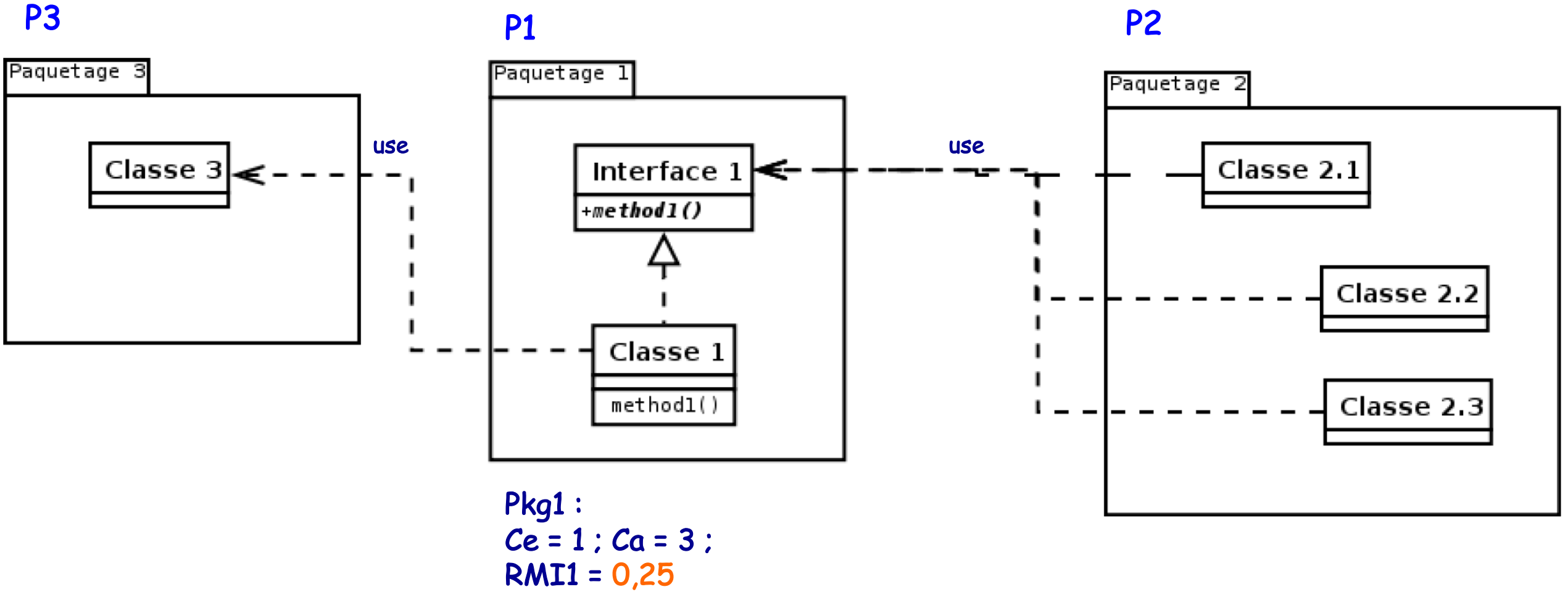
Ce (efferent coupling) = dépendance vers extérieur

Ca (afferent coupling) = responsabilité

instabilité



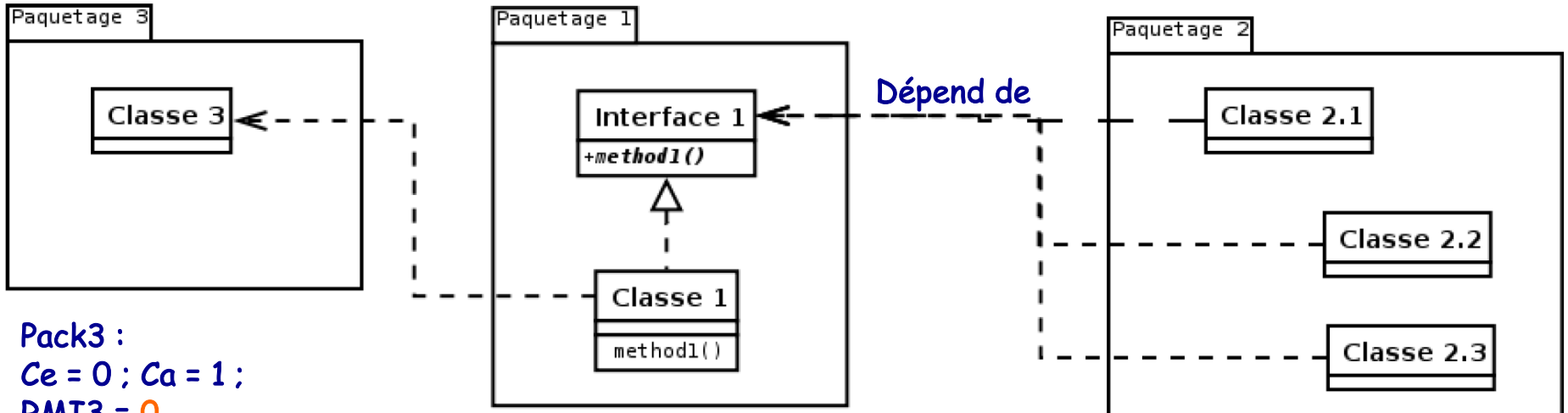
Exercice Instabilité



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

Corrigé 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

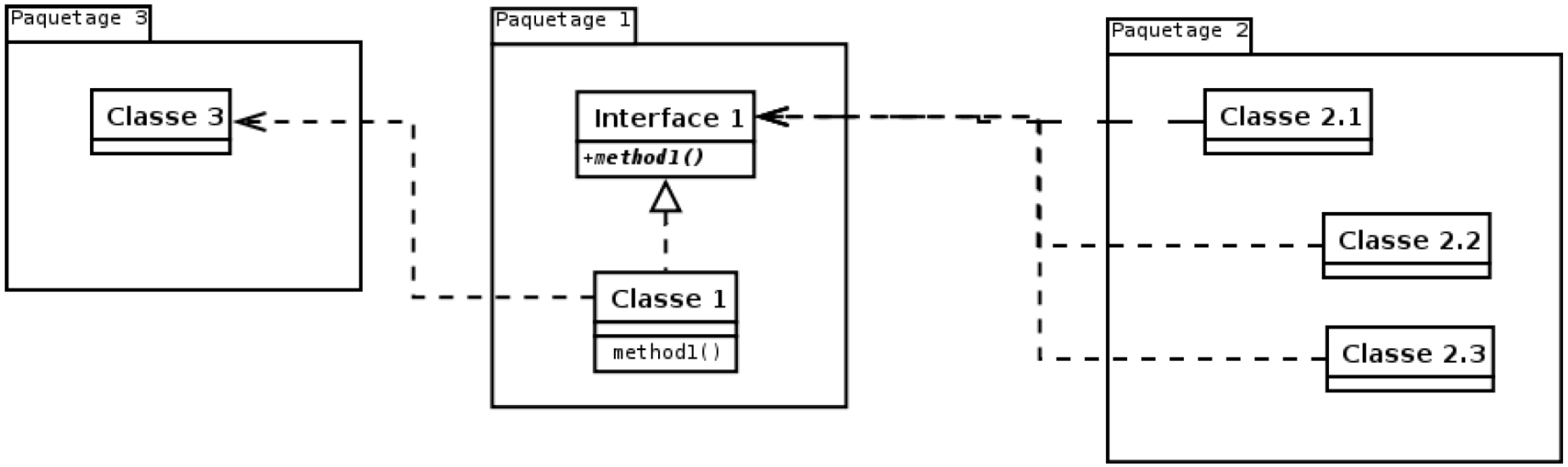
- RMI > 0 : fait ressortir les paquetages **qui dépendent plus des autres** que les autres ne dépendent d'eux.
  - Ces packages peuvent poser des pbs de fiabilité, puisqu'une modification dans un des paquetages dont ils dépendent impacte potentiellement leur fonctionnement
  - (on ne maîtrise pas leur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

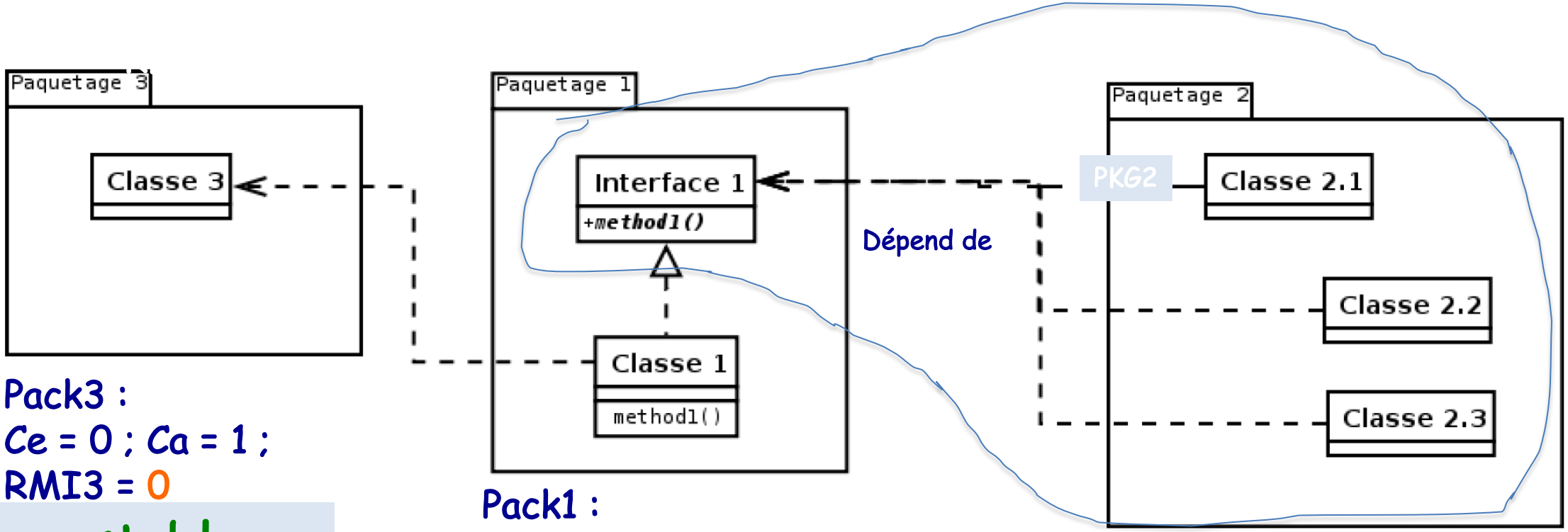
- Il faut ÉVITER les **dépendances cycliques** entre packages
  - Cas où un *packA* référence un *packB* qui référence un *packC* qui référence à son tour le *packA*...
  - Les cycles de vie sont liés et tous ces éléments ne peuvent ni être utilisés ni être modifiés séparément.

- 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 : A + 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**, c'est donc normal (package pur d'implémentation)
- Un certain nb d'autres paquetages dépendent de ce paquetage, mais celui-ci possède **beaucoup d'interfaces** (package central, abstrait).

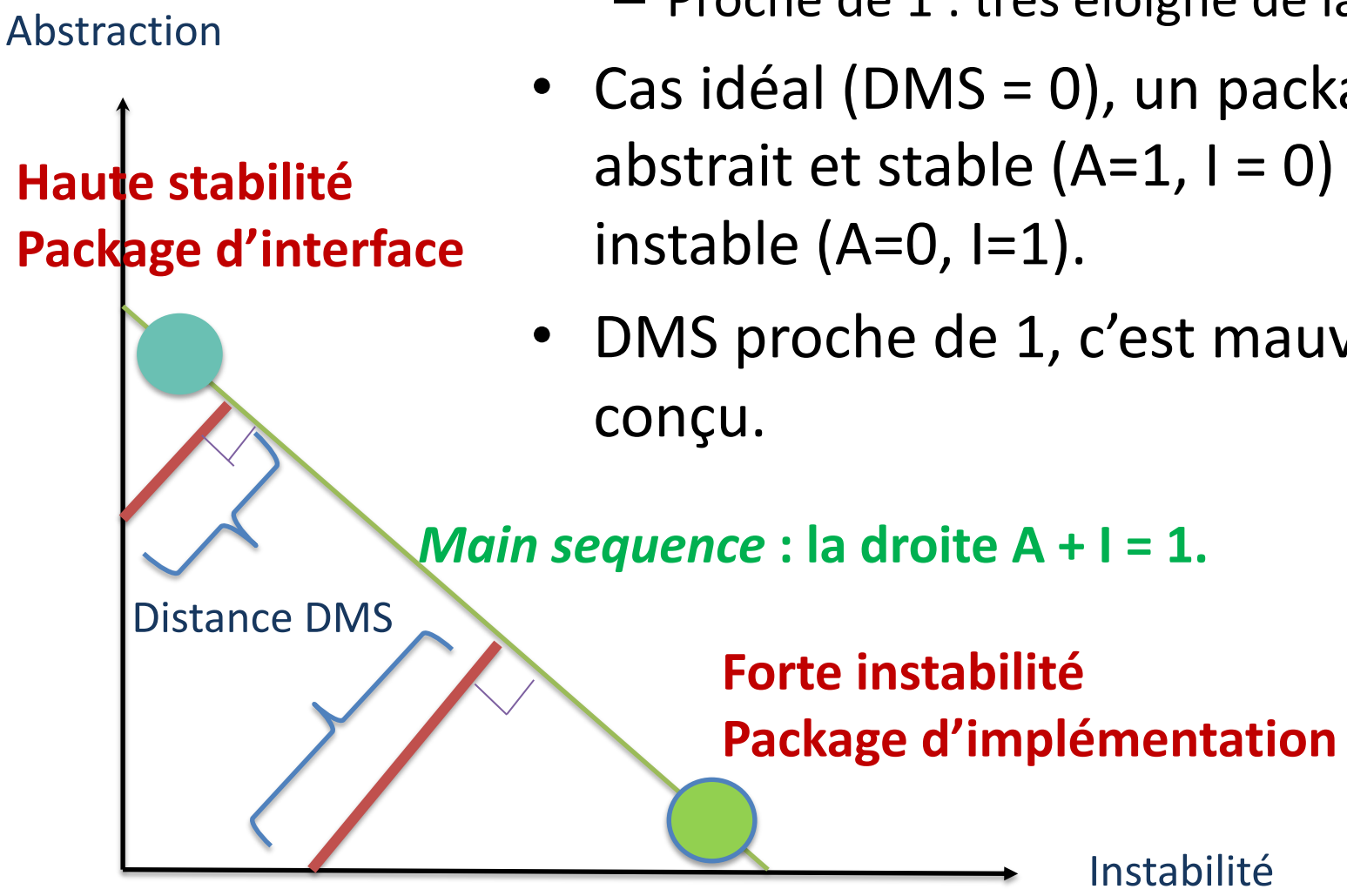
Distance from de Main Sequence (DMS)

$$DMS = | A + RMI - 1 |$$ *(Abstraction+ Instabilité -1)*

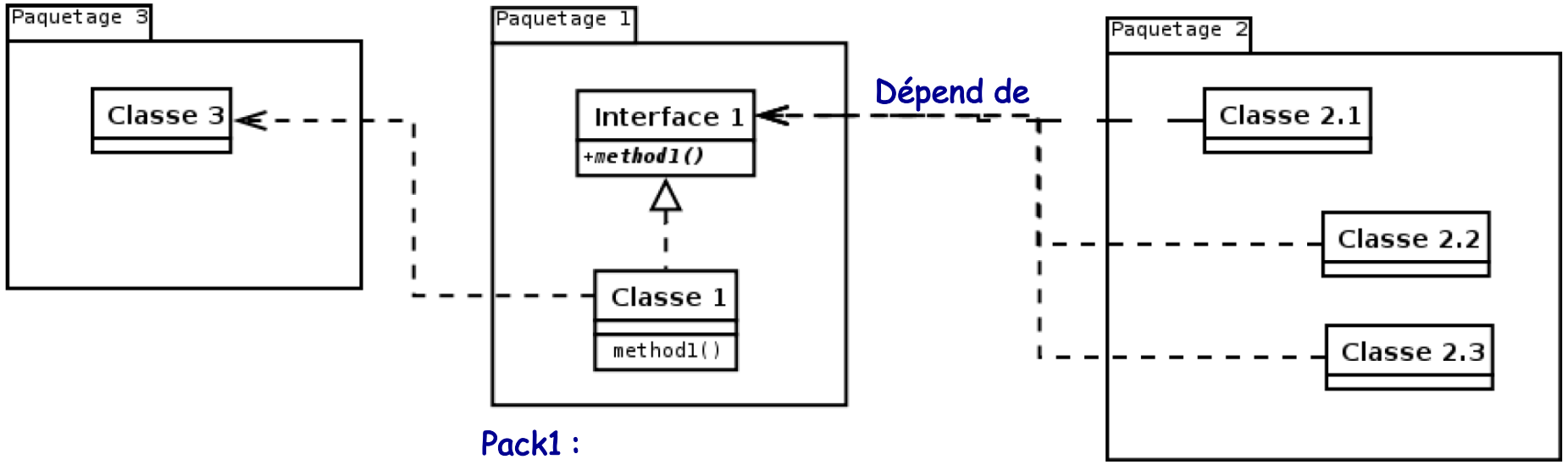
Bonne DMS = valeur proche de 0

DMS

- Distance normale (perpendiculaire) à la droite A + I = 1.
  - Proche de 0 : le package coïncide avec la « Main sequence »
  - Proche de 1 : très éloigné de la « Main séquence ».
- Cas idéal (DMS = 0), un package est soit complètement abstrait et stable (A=1, I = 0) ou complètement concret et instable (A=0, I=1).
- DMS proche de 1, c'est mauvais : sans doute un package mal conçu.



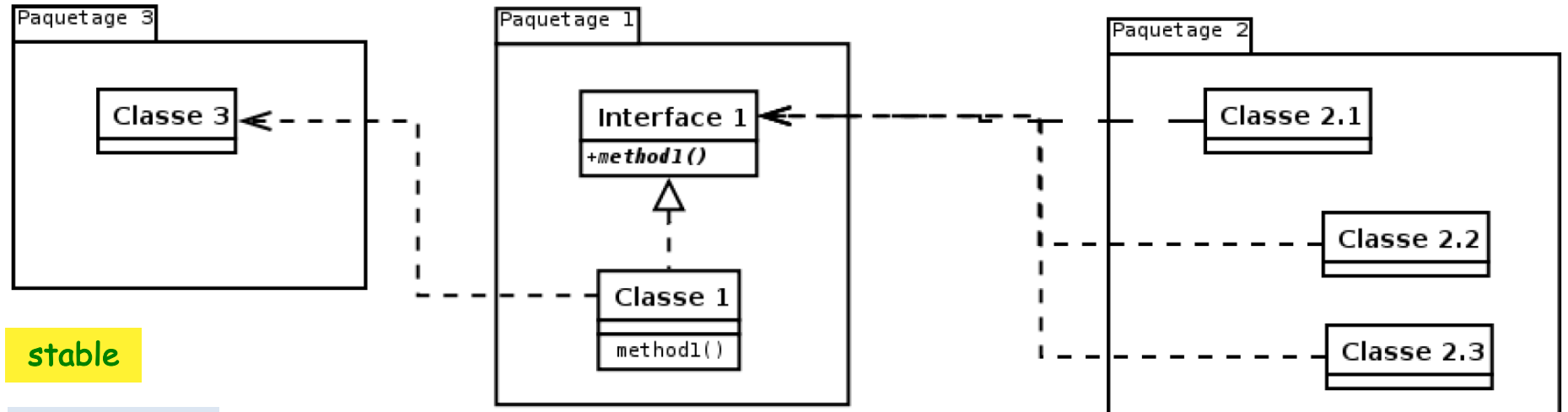
Exercice précédent : DMS



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 + A1 -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 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 (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,**
  - est **moins facile** à maintenir.

- Le calcul de VG est égal au nombre de points de décision + 1.
  - Points de décision : if, while , do, for, ?:, catch , switch, etc.
- En pure théorie, un VG de 1-4 est considérée comme facile à tester, 5-7 OK, 8-10 devrait conduire à refactorer pour faciliter les tests, et 11+ re-factorer car les tests seront impossibles à faire.

Calcul de VG

- Méthode de calcul **pour les méthodes uniquement :**
  - On part de **1**
  - Puis chaque bloc condition (if, expression ternaire*, etc.),
    chaque opérateur d'une condition booléenne (&&, ||, &, | and ^)
    chaque itération (for, while),
    chaque try/catch,
    chaque switch et chaque case,
    chaque return hormis la dernière instruction,

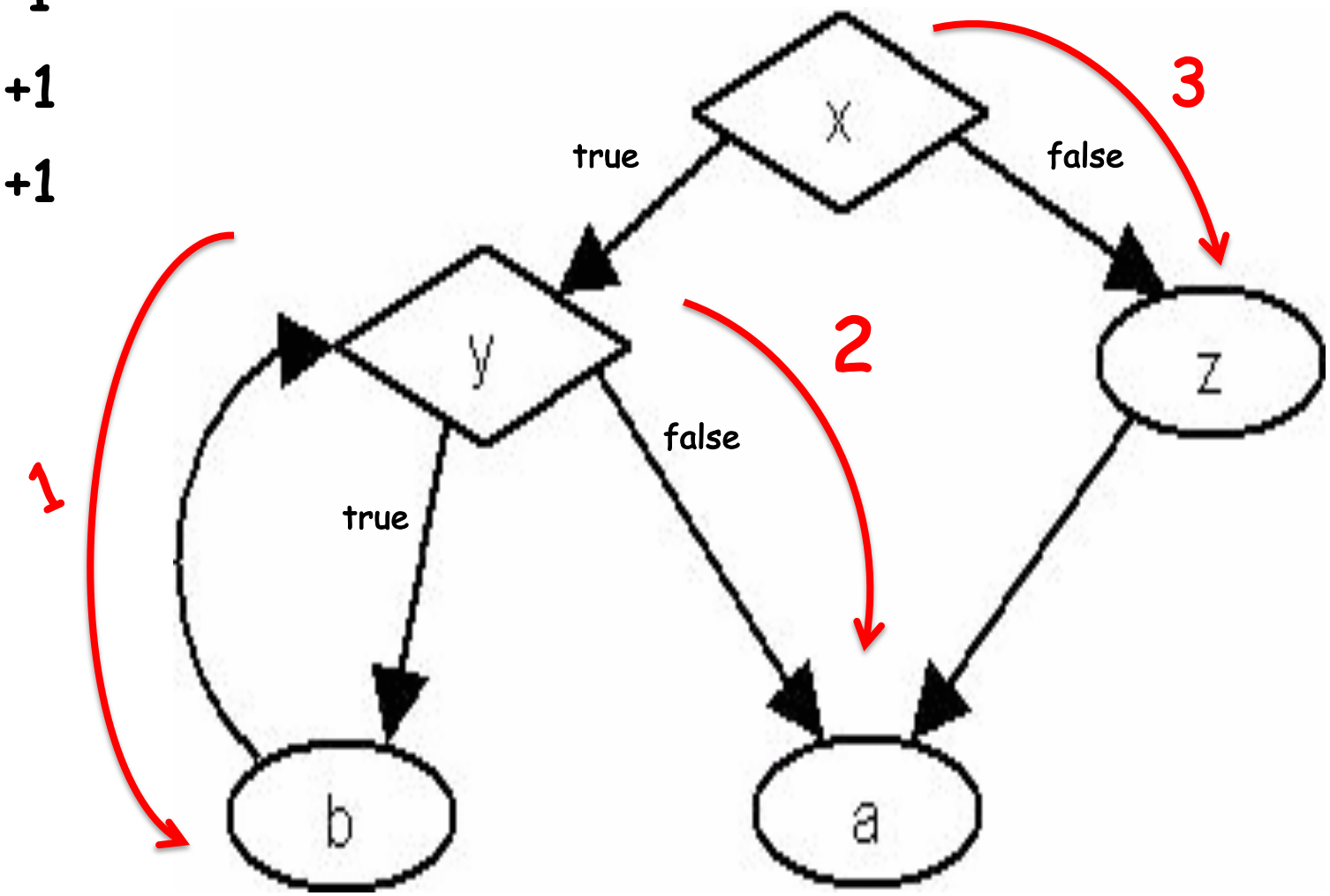
incrémente VG de 1

**Expression ternaire : test ? expr1 : expr2*

Illustration VG

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

1
+1
+1



VG = 3

Ou plus rapidement :

Vg = e – n +2, e nombre d'arêtes (edge), n nombre de sommets (node)

Calculer la VG de ce code

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

}
```

+1

+1

+1

+1

VG = 5

Amélioration : la condition, toujours !

```
                                                    1
public void process(Car myCar) {

    if ( myCar.isMine() ){                          +1
        car.paint("red");
        car.changeWheel();                          +1
+1  while ( car.hasGazol() && car.getDriver().isNotStressed() ){
        car.drive();
    }
}
```

VG = 4

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*

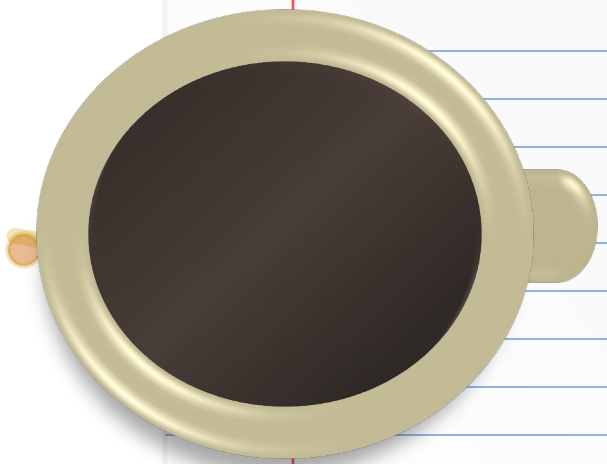
Complexité et **couverture de tests**

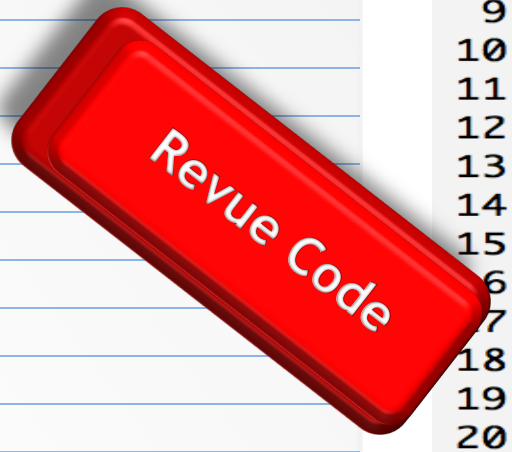
- Notion de “Code Coverage” :
  - Pourcentage de chemins couvert par les tests.
  - Une couverture de tests de 100% signifie que le nombre de tests unitaires d'une méthode est égal à son indice de complexité cyclomatique
    - Tous les chemins possibles sont testés

La complexité aide donc à définir le code coverage, par ex. ici Règles Crap4J

Complexité Cyclomatique	Pourcentage de couverture par les tests requis
0 – 5	0%
6 - 10	42%
11-15	57%
16-20	71%
21-25	80%
26-30	100%
31+	-

Revue de code C





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

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void extract(char* char1, char*

int eval1(char* ch)
{
     int i;
     int valeur1, valeur2;
     int lgval2;
     char *val1, *val2;
     char operation;
     int resultat;
     /* Recherche d'un opérateur
     for( i=0 ; *(ch+i) !='+' &&
     {
     }
     /* Traitement des erreurs */
     if(i==0) /* Le premier opèran
     {
          printf("erreur : pas de
          exit(0);
     }
     else if(i==strlen(ch)-1) /*
     {
          printf("erreur : pas de
          exit(0);
     }
     else if(i==strlen(ch)) /* Il
     {
          printf("erreur : pas de
          exit(0);
     }
     /* char Extraction de la chaî
     premier opérande */
     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.

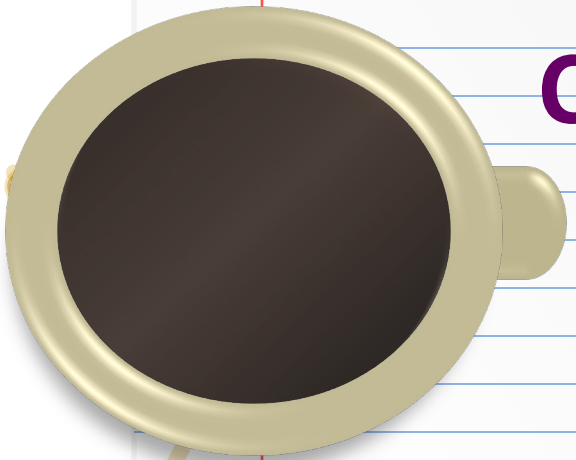
Cette métrique concerne **l'encapsulation**. Elle permet de détecter les méthodes qui n'ont pas leur place dans cette classe. Plus le LCOM est grand, et plus c'est mauvais.

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

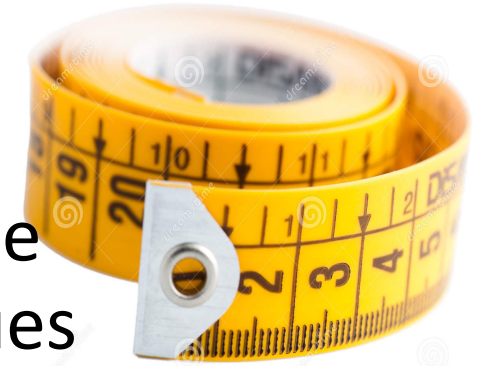
```
<?php
class Example {
  private int $a;

  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.

- Un package faiblement couplé, abstrait et stable est d’excellente qualité.

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

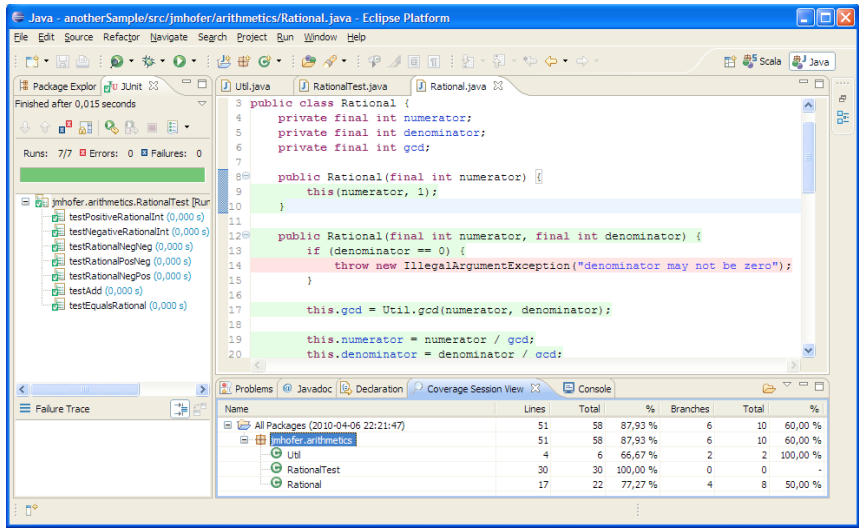
Moyenne globale des métriques

- Pour être utile, un modèle de qualité doit être un modèle d'évaluation et un guide pour augmenter la qualité.
- Un **développeur** doit pouvoir connaitre les composants à améliorer et un **manager**, les points faibles du projet.
- Une simple moyenne ne pointe pas les mauvais composants et même pire : elle les masque.
- Mieux vaut obtenir la **liste des valeurs des métriques** et laisser les spécialistes interpréter.

Outils avec métriques

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

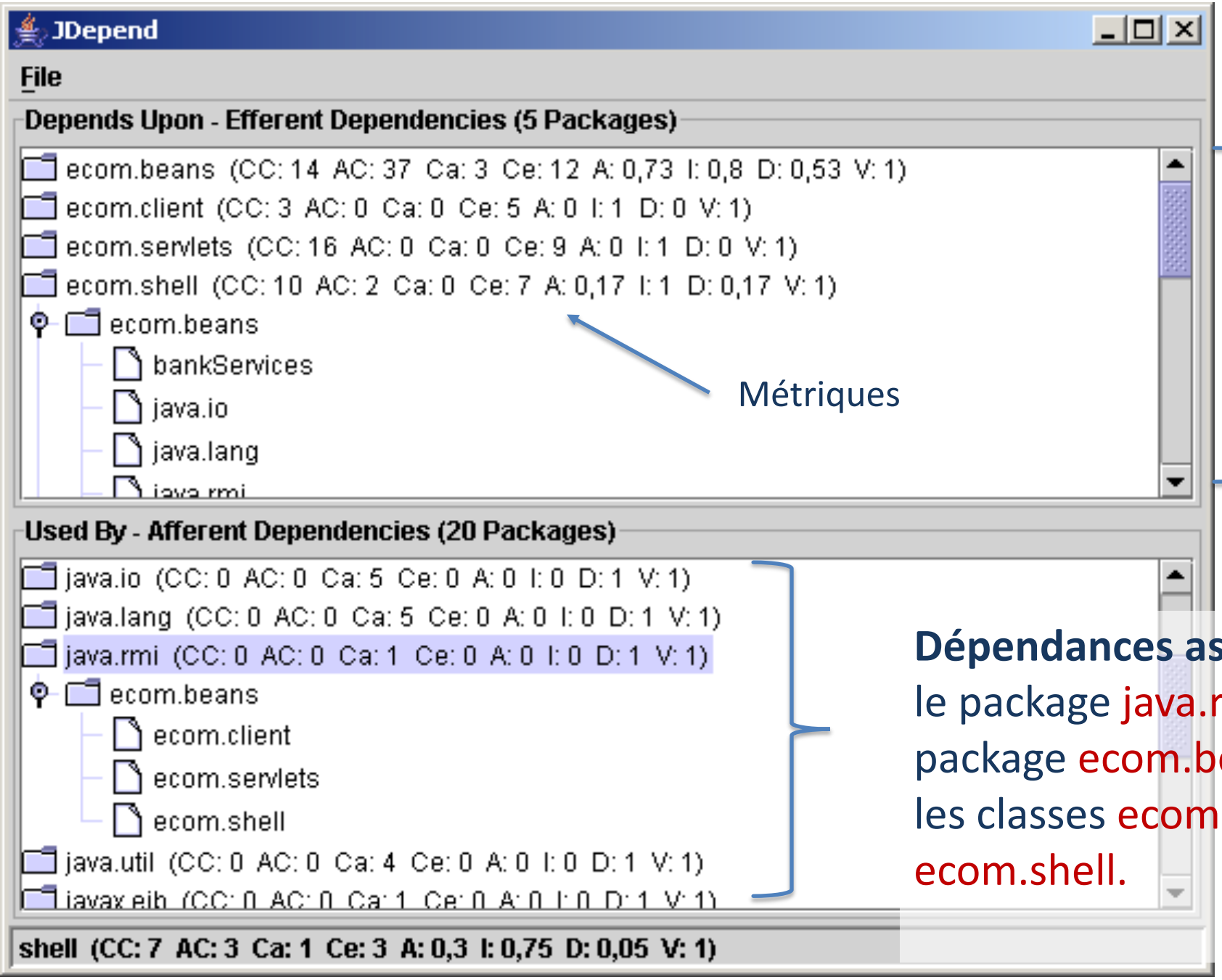
- En Java, un grand nombre d'outils libres sont disponibles :
  - **SonarQube**
  - Cobertura
  - Crap4J
  - PMD
  - FindBugs
  - Eclipse *plug-in* Metrics1-3-6
  - JDepend



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

JDepend

(Hugo Etiévant, Developpez.com)



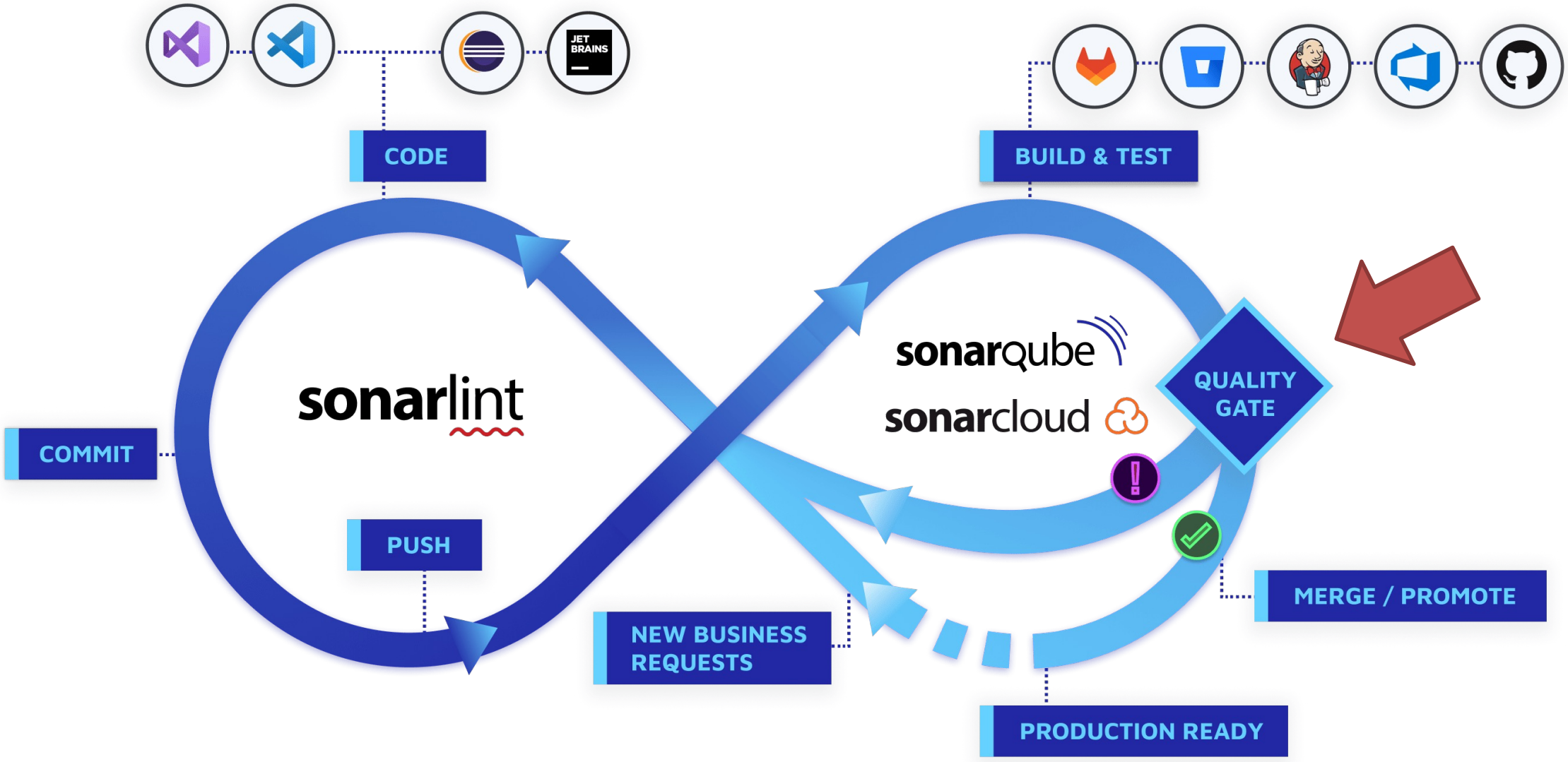
Dépendances descendantes (CE) :

ecom.shell utilise ecom.beans, qui lui-même utilise java.io, etc.

Dépendances ascendantes (CA) :

le package java.rmi est utilisé uniquement par le package ecom.beans (CA = 1), en particulier par les classes ecom.client, ecom.servlets et ecom.shell.

SonarQube


CODE
BUILD & TEST
sonarlint
sonarqube
sonarcloud
QUALITY GATE
COMMIT
PUSH
NEW BUSINESS REQUESTS
MERGE / PROMOTE
PRODUCTION READY

Quelques métriques de SonarQube

- **Complexicity**

  https://docs.sonarqube.org/latest/user-guide/metric-definitions/

  – L'outil utilise sa propre formule de calcul par langage, ainsi pour Java : les mots clefs qui incrémentent la complexité sont : if, for, while, case, catch, throw, &&, ||, ?
- **Cognitive Complexity**
  – Ce qui rend difficile la compréhension du flux de contrôle du code. Basée sur une formule expliquée dans un document à part (white paper)
- **Duplicated blocks :** pas toujours par copier/coller, mais un manque de concertation entre 2 développeurs, qui écrivent la même vérification de données 2x par ex., une à l'appel, une en début de méthode cible…

- **Nombre de *code smells :*** https://pragmaticways.com/31-code-smells-you-must-know/
  – Un *code smell* est un mauvaise code sans gravité : ça peut être un commentaire inutile, du code mort, une classe inutile, du code pour des besoins futurs, plusieurs implémentations de la même fonction, au cas où ça soit mieux…

- **Ratio de dette technique (*Technical Debt ratio*)**
  - Pour cela il faut définir, en se basant sur un projet antérieur :
    - Le temps nécessaire à vos développeurs pour **produire une ligne de code**,
    - Le temps nécessaire **à reprendre le code de base**
  - Exemple :
    - Les développeurs et les chefs de projet ont convenu qu'il a fallu en moyenne 0.2 heures pour produire une ligne de code (12')
    - Et 400h pour reprendre le code de base, qui comptait 30 000 lignes de code
    - Ratio de dette technique = 400 / (30000 * 0,2) = 6,7%
  - Pour un code de plus de 21 jours :

	Tech Debt ratio
Elite Team	< 8%
Normal Team	15%

Source : https://linearb.io/blog/technical-debt-ratio/

  - Comment réduire la dette technique :

https://linearb.io/blog/7-simple-effective-steps-to-reduce-technical-debt/

Dette technique

- Le modèle A2DAM ou [Agile Alliance Debt Analysis Model](), développé avec l'aide de 11 éditeurs de software, qui liste près de 40 règles de base qui, lorsqu'elles ne sont pas respectées, génèrent de la dette technique

- Le jeu 'Dice of Debt' de Tom Grant pour décider d'une stratégie afin de gérer la dette technique sur une période fixée (ex. 10 sprints)
  - https://www.agilealliance.org/dice-of-debt-game/

			SPRINT	1	2	3
Combien de dés vais-je jouer ?	# de dés de création de valeur ajoutée (VA)	(8 au départ)				
	# de dés de dette technique (DT)	(4 au départ)				
	# de dés investis dans une mesure de réduction de la dette					
Choisir à chaque tour	Réduction de la complexité	**COUT :** 2 dés VA pendant 3 tours **EFFET:** Enlever 2 dés du tas de dés TD, les rajouter au tas VA pour le reste de la partie.				
	Revue de code	**COST:** 3 dés VA pendant 2 tours **EFFET:** Enlever 1 dé du tas de dés TD, le rajouter au tas VA pour le reste de la partie.				
	Intégration continue	**COST:** 1 dé VA pendant 2 tours **EFFET:** Rejouer un dé TD (au choix) pour le reste de la partie.				
	Amélioration des tests	**COST:** 1 dé VA pendant 2 tours. **EFFET:** Soustraire 3 du total TD pour le reste de la partie.				
	Nombre total de dés			12	12	12

Valeur Créée ?	VALEUR AJOUTEE CREE : VA	Total des dés "création de valeur"			
	DETTE TECHNIQUE CREE : TD	Total des dés "dette technique"			
	VALEUR AJOUTEE NETTE	VA - TD			
	CUMUL DE VALEUR AJOUTEE	Valeur ajoutée nette cumulée			

Je retiens…

- Les coûts des 3 phases du cycle de vie du logiciel

- Un code de qualité assure la pérennité, la diffusion et
- la maintenabilité du projet. La réussite d'un projet dépend de la qualité du code.

- La **signification** des métriques CE/CA, A, SIX, I ou RMI, VG, LCOM
  - (pas les formules de calcul)

- La problématique de la qualité logicielle
  - Dette technique

Bonnes Pratiques

Voici une liste de certaines BP à considérer. Tenez également compte de l'équilibre entre la **criticité du logiciel** et la **rapidité de livraison**.

- Code/examens par les pairs
  - Détecter les erreurs et les violations de style de codage. Il a été démontré que cette activité accélère et améliore considérablement la qualité du code.
- Robustesse du code
  - Pensez à tous les scénarios : journée ensoleillée et jours de pluie. Soyez très créatif concernant les données invalides.

- **Utiliser ou écrire du code sûr**
  - Pour les applications critiques pour la sécurité, assurez-vous d'appliquer une solution d'analyse statique (MISRA, AUTOSAR pour C/C++) ou d'autres normes de codage qui identifieront l'utilisation de constructions de codage dangereuses (diviser par zéro, utilisation d'un pointeur NULL, etc.)
  - Ex.: https://checkstyle.sourceforge.io/
- **Utiliser ou écrire un code sécurisé**
  - Pour les applications qui doivent être sécurisées, assurez-vous d'appliquer une solution d'analyse statique telle que CERT, OWASP ou d'autres normes de codage qui identifieront les conditions non sécurisées et vulnérables (débordements de tampon, fuite d'informations, injection de script, etc.) pour une attaque.
- **Portabilité des codes**
  - Écrivez du code en pensant à la portabilité. Le code portable, tel que POSIX, ANSI C, etc., peut être facilement et rapidement déplacé vers d'autres plates-formes.

Source : PARASOFT https://fr.parasoft.com/solutions/code-quality/

Exercice Revue de Code

Votre jeune stagiaire décide de construire une API pour la gestion des dates (il ne connaît pas la nouvelle API de Java8, Java Date & Time)

```
public class CalendarUtil {

    private static final Calendar calendar = Calendar.getInstance();

    public static Date getDate(int jour, int mois, int année) {
      // code…
    }
    public static Date getDate() {
        //retourne la date courante
    }
    public static Date incrementer(Date date, int n) {
    // …
    }
    public static boolean equals(Date date1, Date date2) {
```

```
        // indique si deux dates correspondent au même jour calendaire
        // sans tenir compte des heures, minutes, secondes
    }
    public static int difference(Date date1, Date date2) {
        // retourne la différence entre les 2 dates en nombre de jours
    }
    public static boolean isEcheanceEchue(Date date) {
      // permet de savoir si l'échéance est atteinte pour cette date
    }
    public static String format(Date date, String pattern) {
    }
    [...]
}
```

1- Donnez des **illustrations d'utilisation** de cette API

2- Quel problème pose **deux fonctions** de ce code ?

3- Comment le résoudriez-vous ?

Corrigé Exercice Revue Code

Quel problème pose **deux fonctions** de ce code ?

La méthode `public static Date` **`incrementer(Date date, int n)`** *: on ne sait pas ce qui est incrémenté : un jour, un mois ou une année ? Même chose pour* `public static int` **`difference(Date date1, Date date2)`** *qu'est-ce qui garantit que la méthode va retourner un nombre de jours ? Aucune certitude.*

On dit qu'il y a ***conflit de signature*** *pour ces méthodes.*

Comment pensez-vous possible d'**améliorer** le code ?

- *Pour* `incrementer()`*, une solution immédiate serait de* ***renommer le nom du paramètre*** *en nbJours, nbMois ou nbAnnées… pour clarifier le code.*
- *Une solution plus sûre serait de créer 3 méthodes correctement nommées :* `incrementerEnJours(Date d, int nbJours)`, `incrementerEnMois(Date d, int nbMois)` *etc. Les noms explicités diminuent les risques de mauvaise interprétation.*
- *Une dernière solution encore plus robuste serait de définir en plus 3 types de données :* `NbJours`, `NbMois` *et* `NbAnnees` *(ci-dessous), envoyé à la méthode idoine, ce qui permettrait de valider les types à la compilation.*

```
public class NbJours {
    public final int value;

    public NbJour(int n) {
        this.value = n;
    }
}
```

// Cette classe est toute bête, c'est juste un *wrapper* (nouveau nom) vers un *int*.

L'idée est que l'utilisateur de la méthode sait qu'il va ajouter un nb de jours, aucune ambiguité ici

Même principe pour difference(d1,d2)