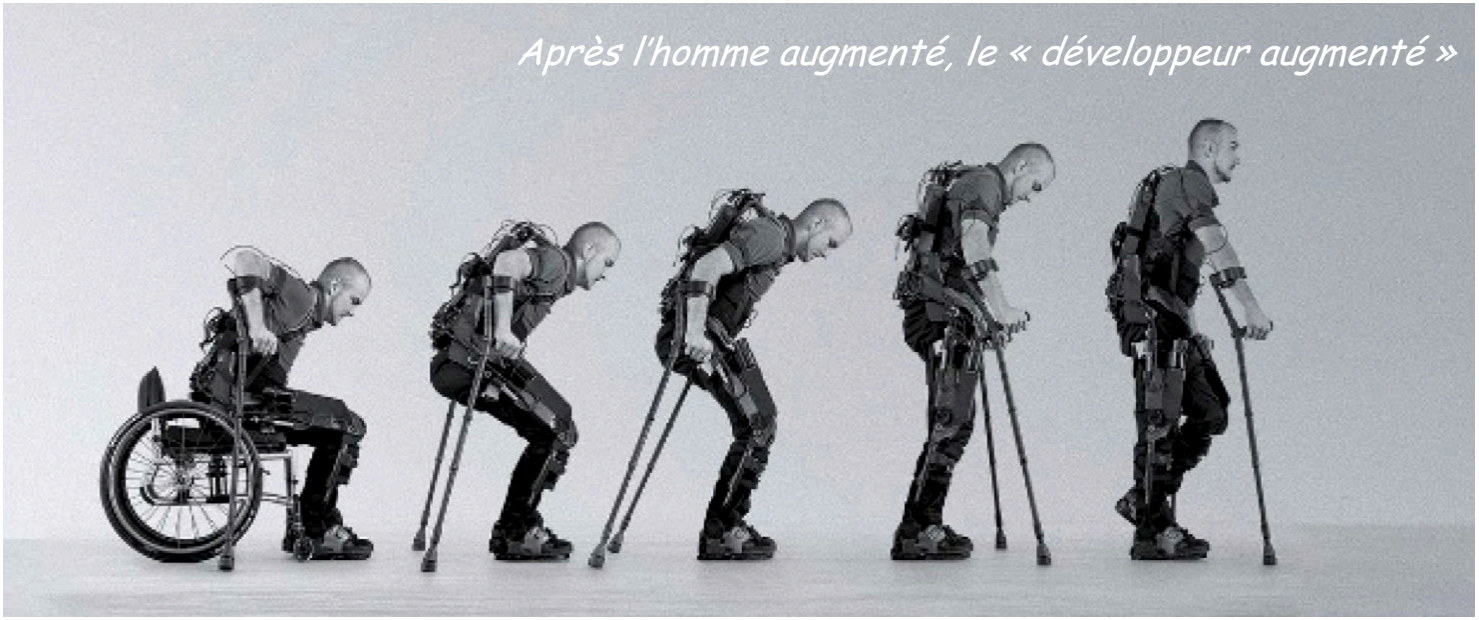
Module Modélisation / Conception de logiciel / Bonnes pratiques Agiles

Chapitre 1.1 – Introduction au Génie Logiciel



IUT Lyon 1
l'excellence technologique

Véronique DESLANDRES - Licence DEVOPS, IUT de LYON

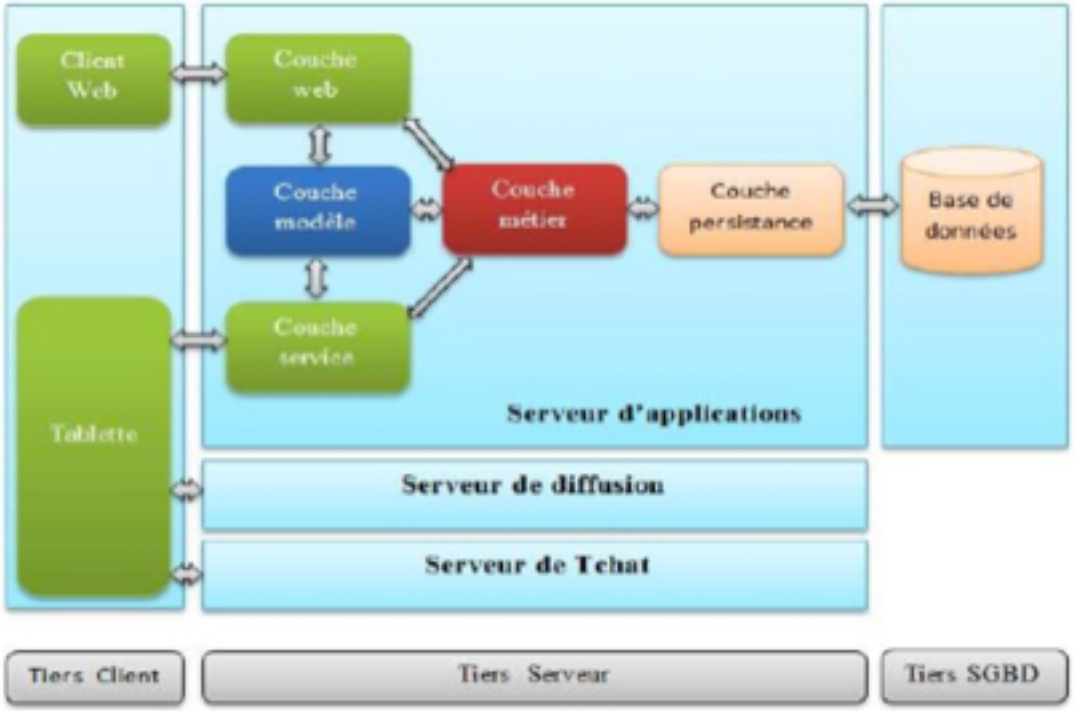
POURQUOI UNE MÉTHODE DE MODÉLISATION ?

Qui plus est : Orientée Objet ?

Les systèmes d'information sont de plus en plus :

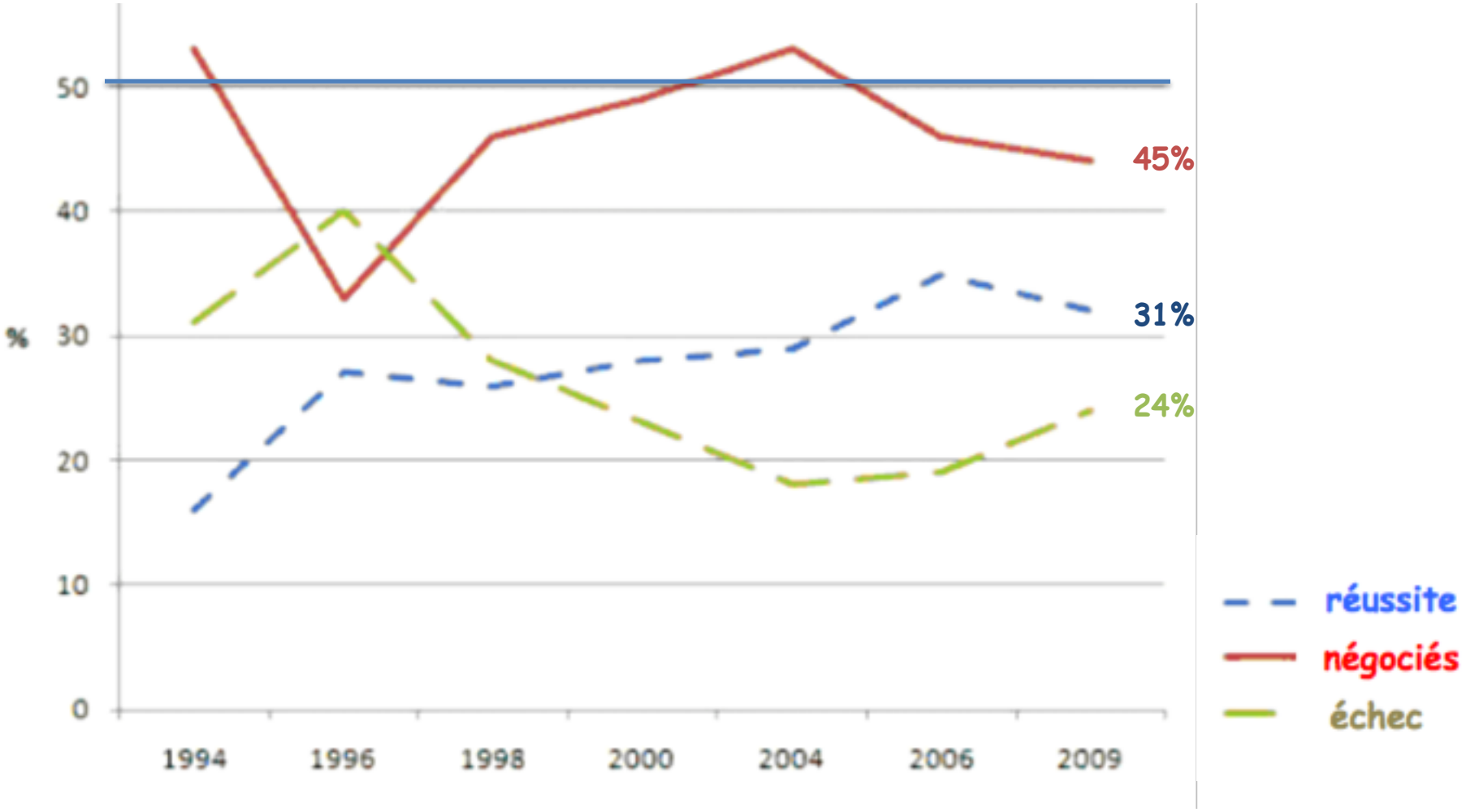
- **Complexes**
  - droits d'accès variés, données complexes, exploitation TR d'entrepôts de données, calculs parallèles élaborés, …
- **Distribués**
  - Stockage des données, présentation et calculs peuvent s'effectuer sur des machines différentes: problèmes de synchronisation et de mise à disposition des éléments

Or, pour garantir la maintenance (fonctionnelle et technologique), on constate que **l'architecture logicielle** est primordiale.



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

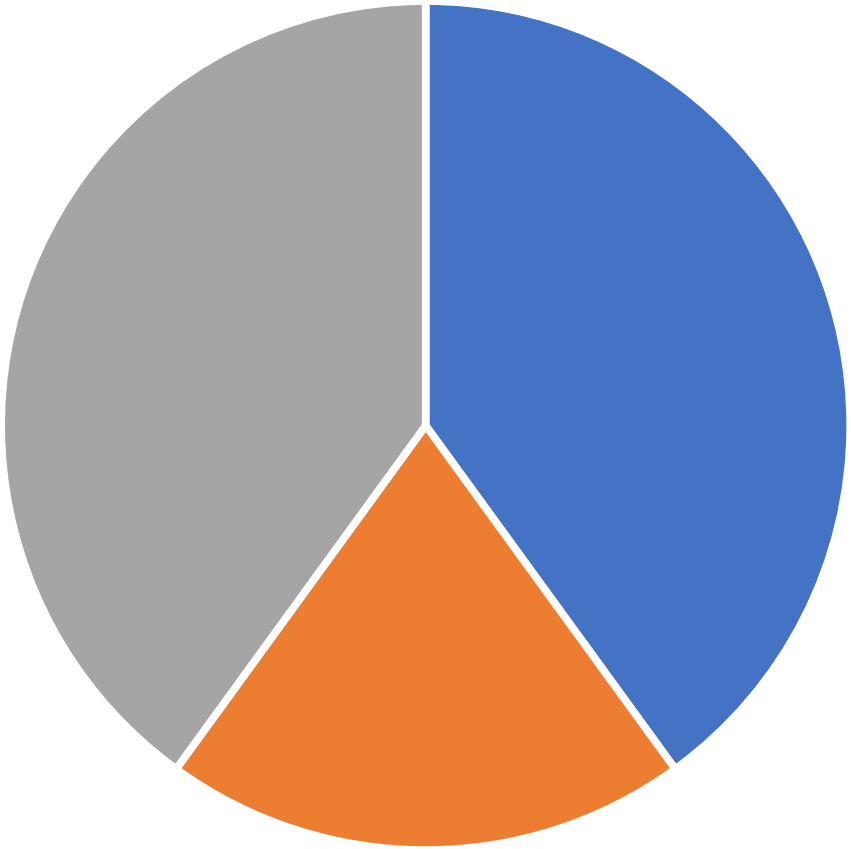
Taux de réussite des projets



Etude du Standish Group (US) - 2010

« Projets négociés » = besoins réalisés mais avec un **délai** et/ou un **budget** supplémentaire

Coût du développement logiciel



Qualimétrie logicielle

- Mesurer la **qualité de code**
- Enjeu = mesurer le risque : vulnérabilité du logiciel aux changements futurs, aux failles
- Quand ?
  - Revues qualité pour des parties sous-traitées
  - Analyse de la capacité à faire évoluer une application
  - Comparaison de codes pour décider duquel choisir

Qualimétrie logicielle: comment ?

- Mesures **quantitatives**
  - Métriques (nombre de classes, duplication de code, couverture et taux de réussite des tests, complexité du code,..)
  - Facile à mettre en œuvre ; pertinence ? Nécessite de bien savoir ce que l'on cherche

- Mesures **qualitatives**
  - « revue de code » par ex., respect des règles de codage...
  - Plus complexe; pertinence ?

Risques associés à une piètre qualité logicielle

Lesquels voyez-vous ?

Risques associés à une mauvaise qualité ?

- Des bugs subsistent
  - Peuvent pénaliser plus ou moins fortement son utilisation

- Client non satisfait
  - Ne fera plus appel à nous
  - Refuse de payer le montant fixé
  - Diffuse son mécontentement sur les réseaux

Risques associés à une sur-qualité logicielle

Quels sont-ils ?

Risques associés à la sur-qualité

Surcoût dû à un **délai dépassé**, ce qui induit :

- Client insatisfait
- Coût final plus élevé
- Pénalités de retard

Surcoût dû à l'appel à **ressources supplémentaires** :

- Ressources prises à d'autres projets…
- Pénalise l'activité globale

- Risque « d'erreur de focus » : les dév peuvent avoir travaillé un aspect jugé non prioritaire pour les utilisateurs

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
NOP	**Number of Packages** : le nombre de packages dans l'élément sélectionné
NOF	**Number of Features** : 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
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*)

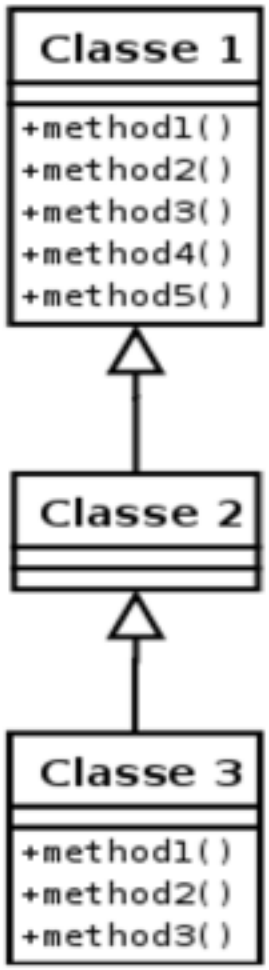
CE	**Efferent Coupling** : le nombre de classes *dans* un package qui dépendent d'une classe d'un autre package.
CA	**Afferent Coupling** : le nombre de classes *hors* d'une package qui dépendent d'une classe dans le package

Quand Ce > 50, dépendance trop élevée. Ces classes sont sans doute complexes et ont trop de responsabilités : refactorer

CE	**Efferent Coupling (dépendances sortantes)** : le nombre de classes *dans* un package qui dépendent d'une classe d'un autre package.
CA	**Afferent Coupling (entrantes)** : le nombre de classes *hors* d'une package qui dépendent d'une classe dans le package
NORM	**Number of Overridden Methods** : nombre de méthodes redéfinies
SIX	**Specialization Index** : indice de spécialisation d'une classe. C'est le calcul : NORM * DIT / NOM. Pour tout le projet : moyenne des

Exercice SIX

- Calculer l'indice de spécialisation de la Classe3, de la Classe2 et donc du projet



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
  - *Recalculer par ex. avec Classe2 et 3 sans redéfinition de méthode*

➔Essayer de **factoriser** ou d'utilis **interfaces**

CE	**Efferent Coupling (dépendances sortantes)** : le nombre de classes *dans* un package qui dépendent d'une classe d'un autre package.
CA	**Afferent Coupling (entrantes)** : le nombre de classes *hors* d'une package qui dépendent d'une classe dans le package
NORM	**Number of Overridden Methods** : nombre de méthodes redéfinies
SIX	**Specialization Index** : indice de spécialisation d'une classe. C'est le calcul : NORM * DIT / NOM. Pour tout le projet : moyenne des index.
RMI	**Instability** : CE / (CA + CE) : ce nombre indique l'instabilité du projet, c'est-à-dire les dépendances entre les paquets

Interprétation RMI

- Cet **indice d'instabilité** est toujours compris entre 0 et 1
  - Traduit l'effort nécessaire pour modifier le PKG sans impliquer les autres
  - Score proche de 0 : considéré comme stable

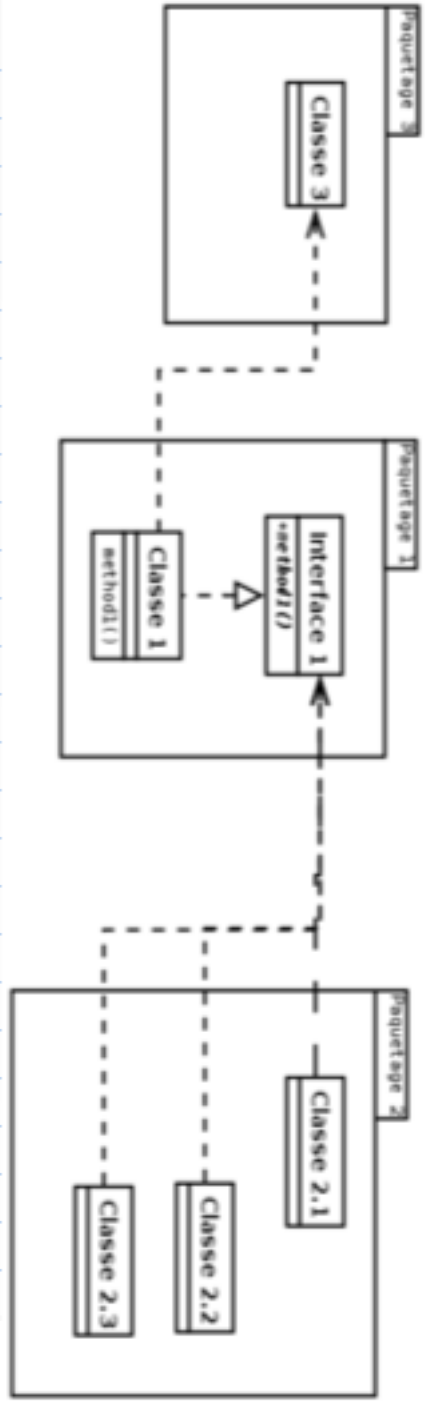
- Fait ressortir les paquetages **qui dépendent plus des autres** que les autres ne dépendent d'eux.
  - Ces packages sont à risque, puisqu'une modification dans un des paquetages dont ils dépendent peut impacter leur fonctionnement
  - (on ne maîtrise donc pas leur bon fonctionnement)

Exercice RMI
Instabilité

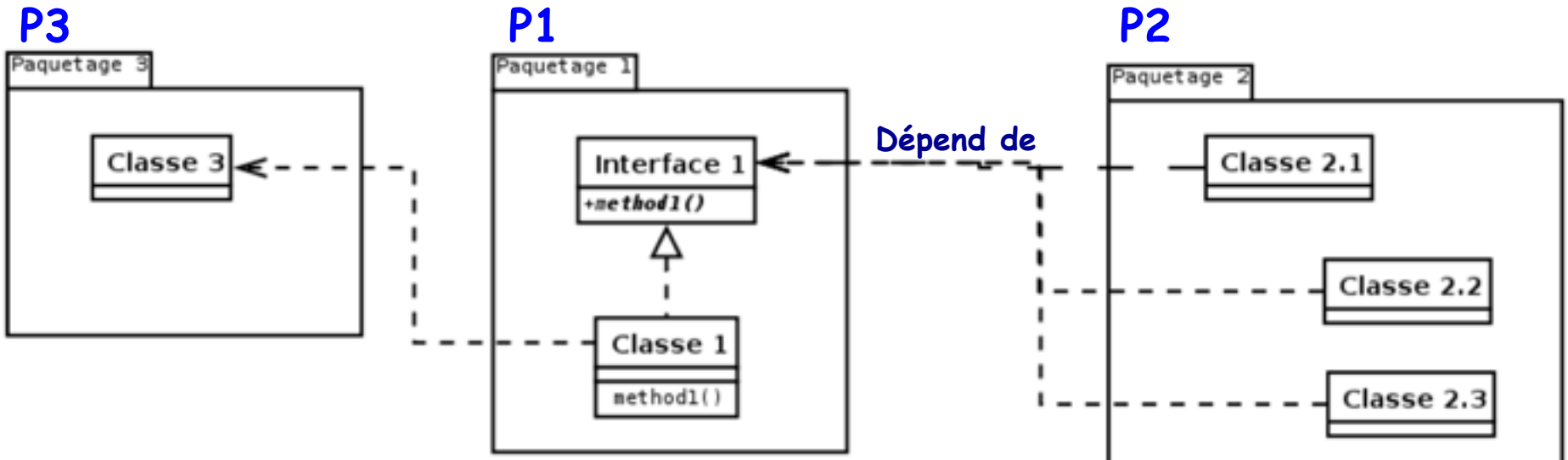
- **RMI = Ce / (Ca + Ce)**

Ce (efferent coupling) =
dépendance vers l'Extérieur

Ca (afferent coupling) =
responsAbilité du pkg



Exercice RMI Instabilité



Pack1 :
Ce = 1 ; Ca = 3 ;
RMI1 = $\frac{1}{4}$ = 0,25

Interprétation RMI

- C’est toujours mieux d’avoir un seul package instable et les autres stables
- On sait exactement lequel est « à risque »
  - dépend des évolutions de code des autres package
  - C’est sur lui qu’on mettra un max de tests (unitaires + intégration)
  - C’est le premier qu’on ira vérifier en cas de pb

Interprétation RMI (2)

- Attention, dans une architecture logicielle, certains paquetages **sont par construction** dépendant d'autres, donc instables
  - Ex. utilisation d'un framework

- Avec le RMI, on considère souvent un autre indicateur :

  la **DMS** (Distance from the Main Sequence)

DMS

Représente l'équilibre qui doit exister entre *le niveau d'abstraction* et *l'indice d'instabilité*

DEF Distance from de Main Sequence (DMS)

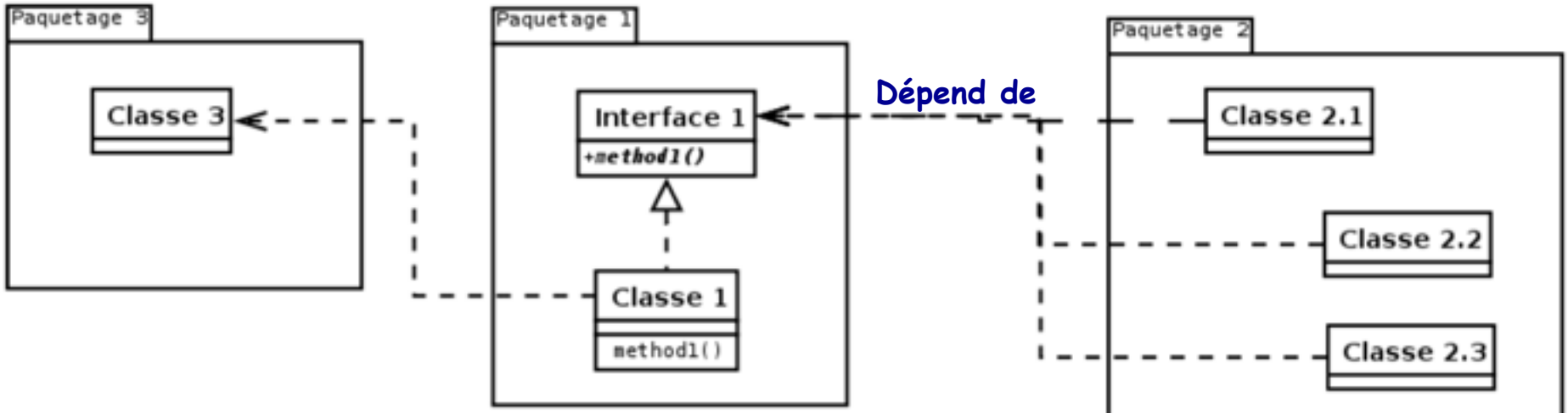
DMS = $|\ RMA + RMI - 1\ |$

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

Bonne DMS = valeur proche de 0 :

- le paquetage est instable mais possède peu d'interfaces,
- d'autres paquetages dépendent de ce paquetage, mais celui-ci possède beaucoup d'interfaces et est stable

Exercice précédent : DMS



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

Calculer la DMS par package et conclure : sont-ils bien **architecturés** ?

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)
NBD	**Nested Block Depth** : La profondeur du code

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 **bcp de tests** pour atteindre une bonne couverture du code
  - est **moins facile à maintenir**.

Calcul de VG

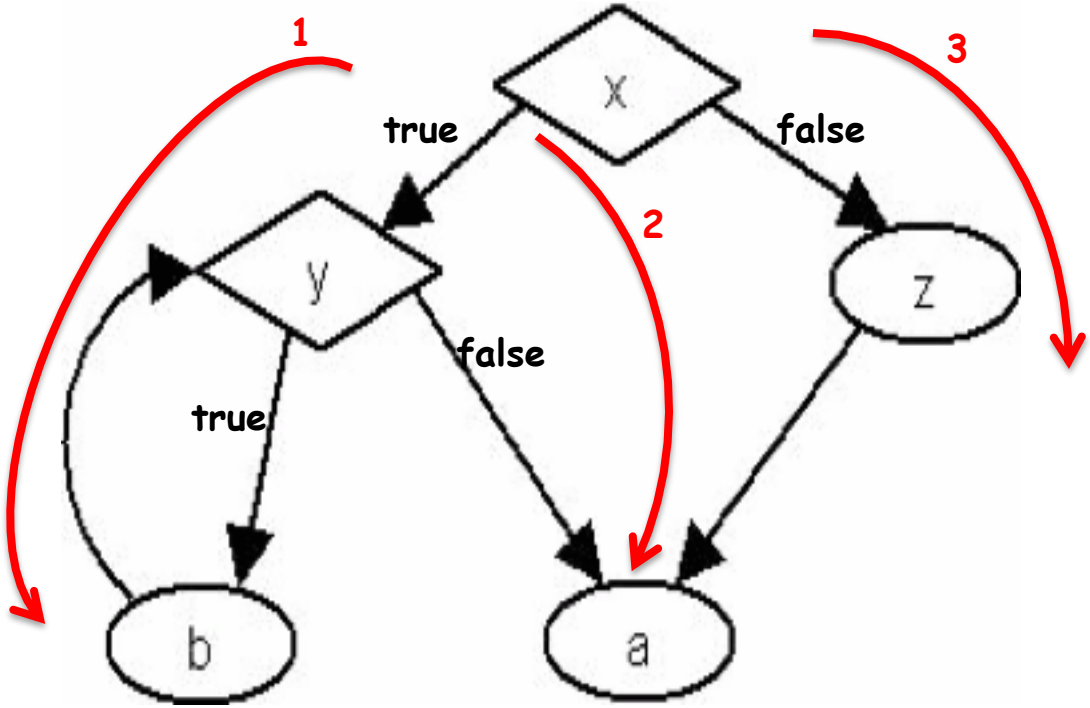
(quasiment indépendant du format du langage)

- Déclaration de méthode : on part de 1
- Chaque bloc condition (*if, exprl ? expr2 : expr3*), chaque itération (*for, while*), chaque *try/catch*, *throw*, chaque bloc *switch/case* (sauf *finally)*, chaque test : *&&, ||, ?* et *return* (sauf si c'est le dernière instruction de la fonction)
  - incrémente VG de 1
  - *else, default, finally* n'incrémentent pas VG

Illustration VG

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

1
+1
+1



Ici complexité VG = 3

3 chemins indépendants dans le graphe des instructions

Quelle complexité ici ?

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

Question : vaut-il mieux avoir une méthode avec un VG de 30 ou 3 méthodes de VG 10 ?

Avantages des petites fonctions

- Il y a un *meilleur **découpage** de la logique* suivie
- Le programme est *plus facile à **maintenir***
- On diminue le risque *d'injecter un **bug***

- En faisant des fonctions petites, on ajoute de *l'aide à la **compréhension*** grâce au nom des fonctions
  - qui est un commentaire explicite

- On *réduit la redondance du code* car les fonctions sont facilement **réutilisables**

Interprétation VG

- Si une méthode a une complexité cyclomatique **trop élevée (au delà de 10)**
  - elle doit être refactorisée
  - On peut notamment utiliser le polymorphisme pour éviter les conditions (à voir en TD)
- **Entre 6 et moins de 10** :
  - La complexité peut-être jugée ‘acceptable’
  - La méthode doit être suffisamment testée

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

Lien VG -> tests

- La complexité cyclomatique permet de définir le nb de tests à effectuer

DEF application complexe : 20 000

domaines administratifs où la réglementation évolue très fréquemment

Complexité	Basse	Moyenne	Elevée	Très élevée
Points VG total	6 000	16 000	50 000	>50 000

Des tests unitaires au niveau de l'équipe de projet devraient suffire pour vérifier la plupart des cas

Des tests unitaires automatisés sont nécessaires, avec répartition par spécialisation des dév, et donc une phase d'intégration (tests d'intégration)

En plus : tests fonctionnels avec une équipe de QA dédiée, avec formalisation des cas de tests basée sur les spécifications

http://qualilogy.com/fr/complexite-et-effort-de-test/

LCOM : cohésion

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

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

Calcul (il existe différentes façons de définir LCOM):

Soit sum(MA), le nombre de méthodes utilisant un attribut de la classe, M : nb de méthodes, A nb d'attributs :

LCOM = 1 – (sum(MA) / M*A)

Si toutes les méth utilisent tous les attributs : sum(MA) = M*A

et LCOM = 0, on dit que la classe est **cohésive**

Exercice Cohésion

Calculer la cohésion de la classe PHP suivante


```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';
   }
}
```

Interprétation LCOM

- LCOM ≥ 0.4 avec NbAttributs < 2 and NbMethods < 10 : problème, classe peu cohesive

- LCOM > 0.8 avec NbAttributs > 10 and NbMethods >10 : problème, classe peu cohesive

- On verra le principe SRP (single responsibility principle) qui dit qu'une classe ne devrait pas avoir plus d'une raison d'être modifiée
- Une telle classe est dite cohésive.

BILAN : les métriques

- Interprétation : telle classe ou tel fichier a une complexité trop élevée, …
  - Ce qui est important est de surveiller l'**accumulation d'indicateurs** et **leur orientation générale**
  - Les clients peuvent être **demandeurs**
- On peut localiser les modules particulièrement **difficiles à tester et maintenir**
  - *Refactorer* plutôt que de garder des modules qui vont coûter cher en maintenance

À vous de juger !

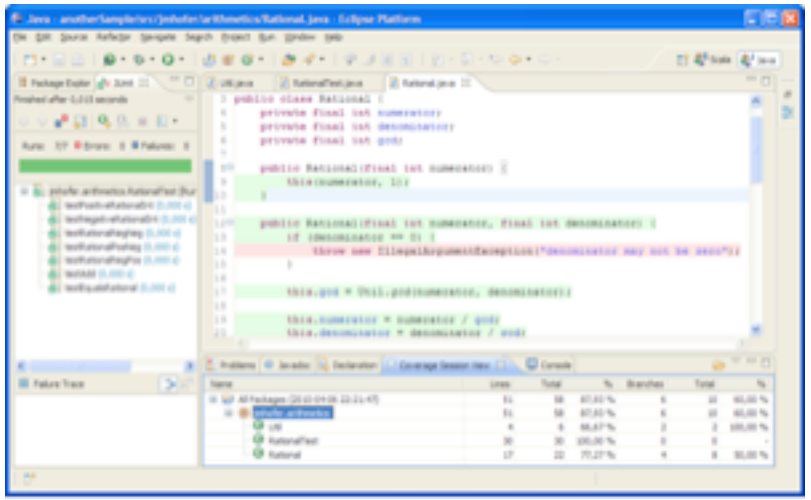
- La qualité d'un logiciel est un sujet qui divise :
  - certains pensent qu'il s'agit d'un **surcoût** et la voient comme une contrainte,
  - d'autres au contraire pensent qu'il s'agit d'une **opportunité** et voient la qualité comme un guide de travail
    - Ceux-là opposent, au surcoût induit par la qualité, le coût induit par le *manque de qualité* d'un logiciel
    - Qualité de code très présente dans le développement Agile

VOC - le manque de qualité logicielle est appelé « la dette technique »

Outils pour les métriques

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

- En Java, un grand nombre d'outils libres sont disponibles :
  - **SonarQube** (free)
  - Cobertura
  - Crap4J
  - PMD
  - FindBugs
  - Eclipse Metrics
  - Jdepend



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

À lire…

Pour aller plus loin : un cours avec beaucoup d'exemples sur les métriques:

http://www-igm.univ-mlv.fr/~dr/XPOSE2008/Mesure%20de%20la%20qualite%20du%20code%20source%20-%20Algorithmes%20et%20outils/metriques-definition.html

La qualité côté dev

- « … un logiciel capable de répondre parfaitement aux **attentes du client**, le tout sans défaut d'exécution »

- Comment bien connaître les attentes du client ?
- Quelles sont les tâches difficiles en program-mation ?

Développeur : 9 tâches les + difficiles

(enquête : 4500 personnes)

- Ecrire les tests
- Estimer le temps nécessaire pour chaque tâche
- Ecrire la documentation
- Expliquer ce que je fais (ou ne fais pas)
- Implémenter une fonctionnalité que je désapprouve
- Travailler avec le code d'un autre
- Nommer les choses
- Concevoir une application
- Négocier avec les autres

Sauriez-vous les classer ?

Capture besoins

- Une activité difficile car
  - Les **utilisateurs** ne connaissent pas vraiment leurs besoins
  - Les développeurs connaissent mal le **domaine** de l'application
  - Utilisateurs et développeurs ont des **langages** différents
  - Les **besoins** évoluent
  - Il faut trouver un **compromis** entre services, coûts et délais

How the customer explained it

D’OÙ LE BESOIN DE MODÉLISER