



Ingénierie Logicielle

Mesurer la qualité du logiciel

Licence Pro SIL

Département informatique – IUT de Nantes

2012 - MAJ 2014

Arnaud Lanoix

arnaud.Lanoix@univ-nantes.fr



Qualité



Qualité (définition de l'AFNOR) :

« Ensemble des **propriétés** et des **caractéristiques** d'un produit ou d'un service qui lui confère l'aptitude à **satisfaire** des **besoins** exprimés ou implicite »

Un logiciel est **de qualité**, s'il...

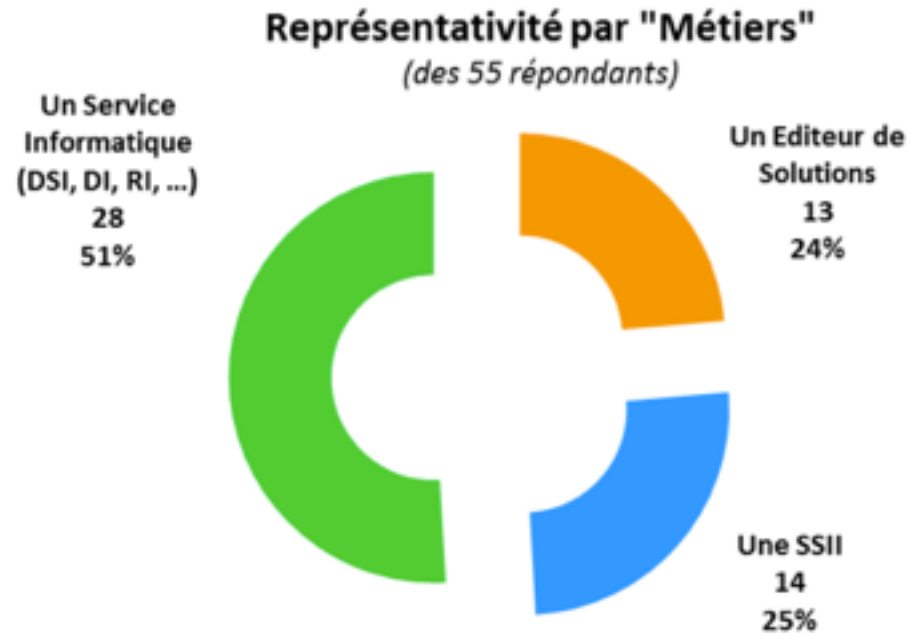
- *répond aux attentes du client ?*
- *est « correct » ?*
- *réalise les spécifications ?*
- *son coût de développement est contrôlé ?*
- *sa maintenance est facilitée ?*

⇒ **cela dépend du point de vue ;-)**

Critères difficiles à "mesurer"

ADN'Ouest > BaroQL

- Sources : 55 entreprises interrogées en juin 2012 / 42 en avril 2014
- <http://www.adnouest.fr> / <http://a2jv.fr/baroql/>

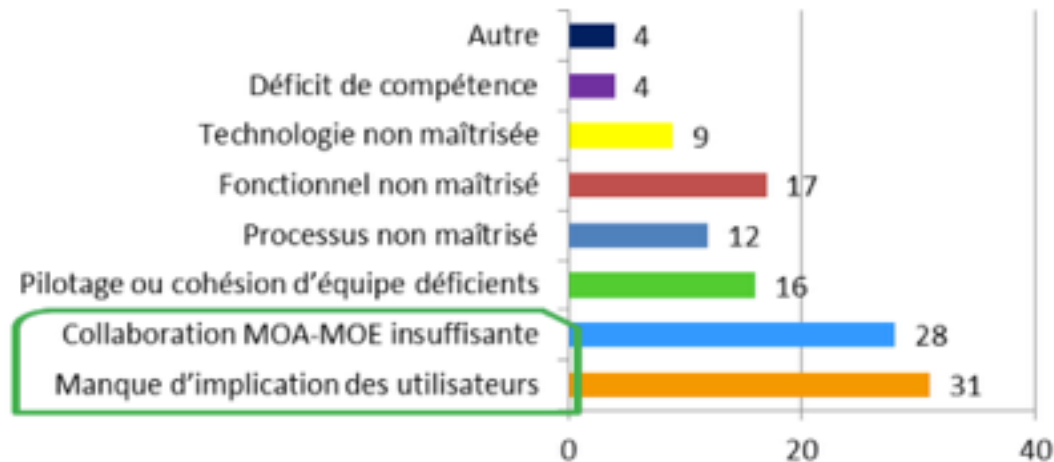


Source BaroQL'2012

Quels sont les Facteurs majeurs d'ECHEC de vos projets ?

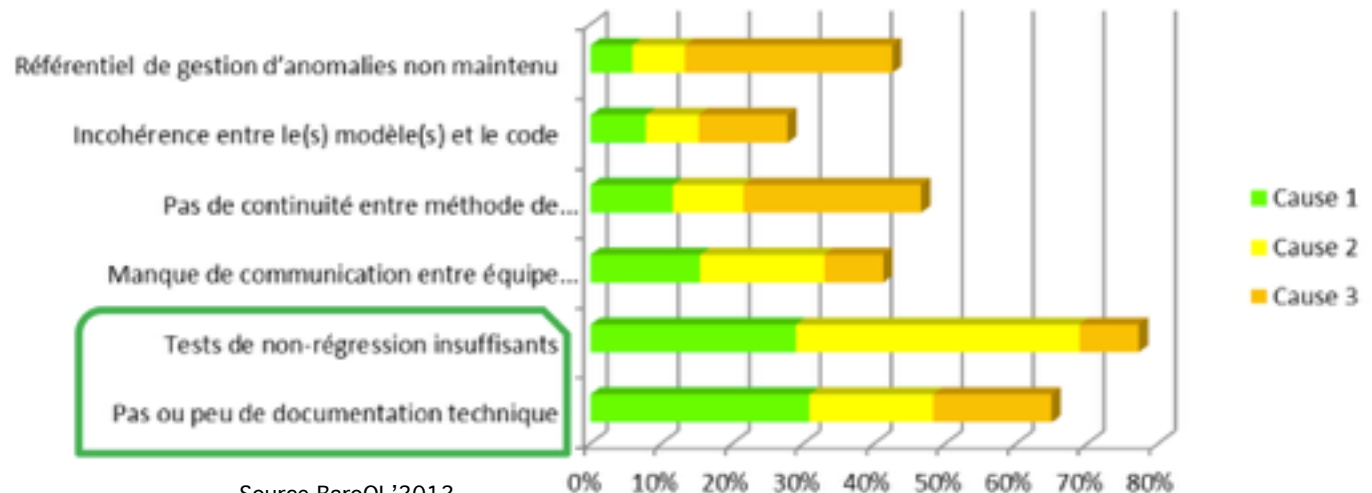
(55 répondants)

Source BaroQL'2012



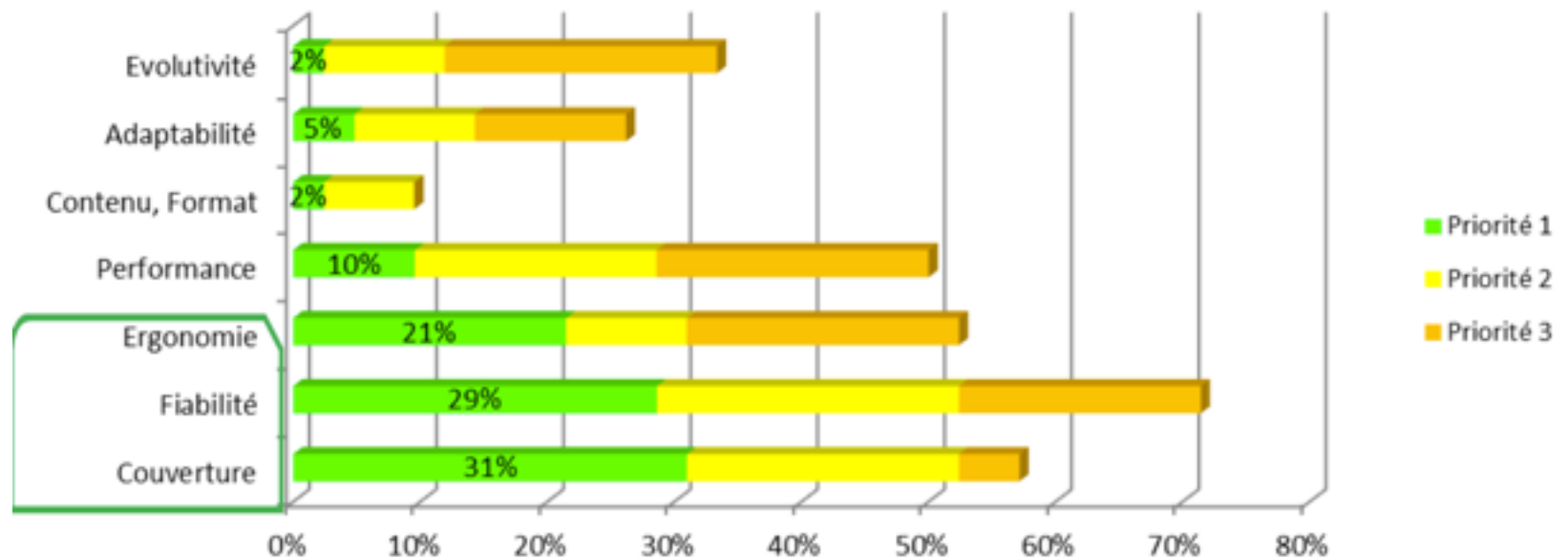
Principales CAUSES des DIFFICULTES rencontrées en MAINTENANCE ?

(pour les 55 répondants)



Source BaroQL'2012

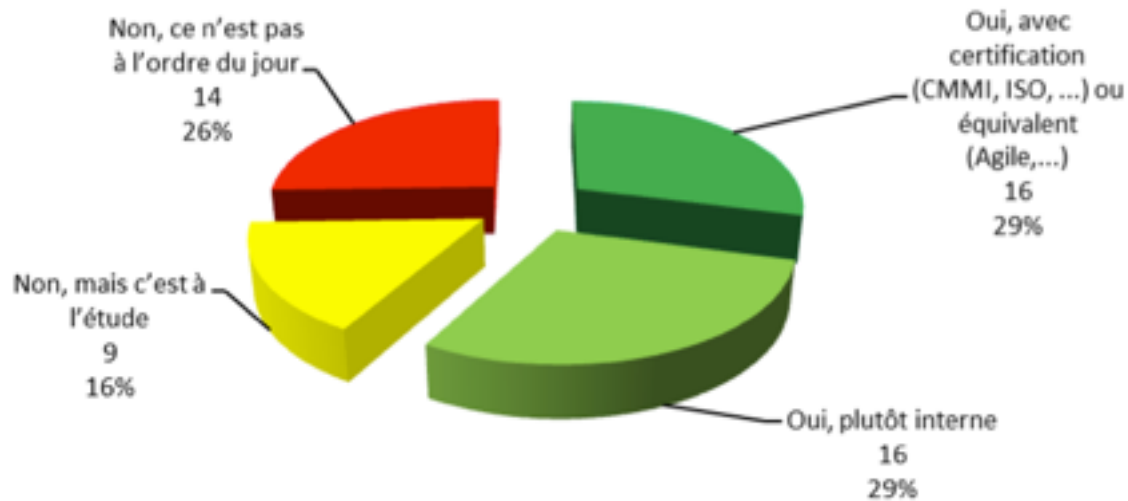
A quels Facteurs êtes-vous le plus SENSIBLE sur la qualité des logiciels que vous utilisez ?



Source BaroQL'2014

Existence d'une Démarche Qualité Logicielle

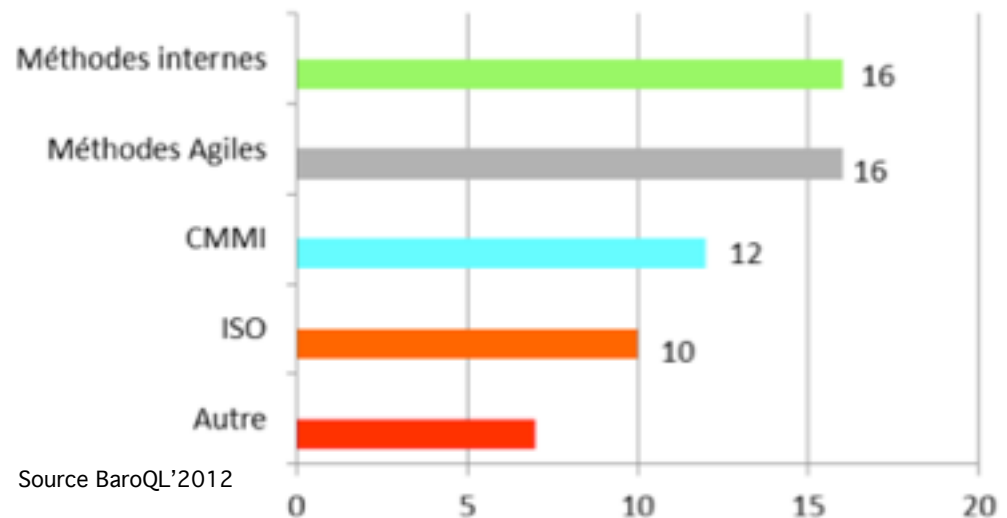
(55 répondants)



Source BaroQL'2012

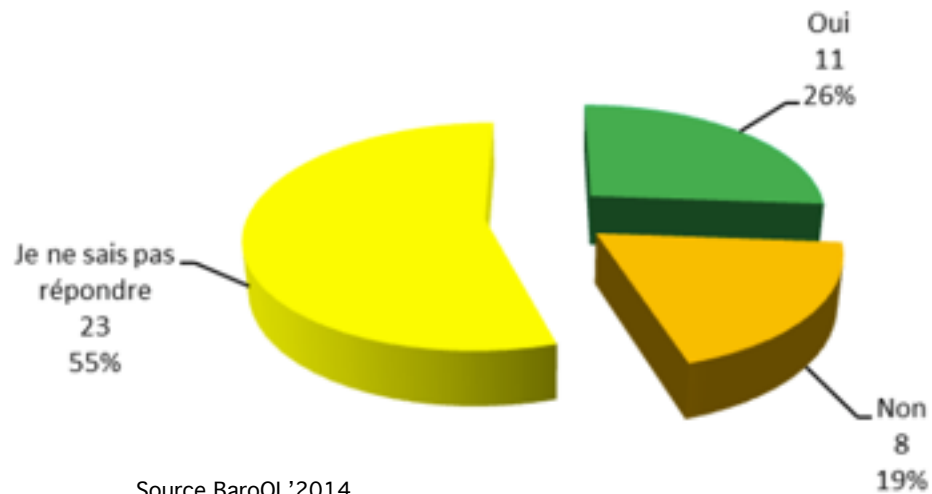
Nature Démarche Qualité Logicielle

(sur les 32 répondants ayant indiqué en avoir une)



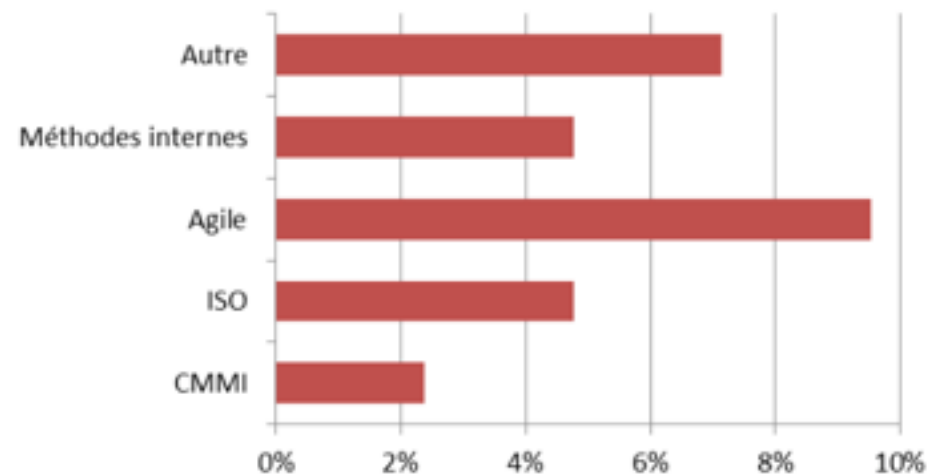
Source BaroQL'2012

Exigence par votre Entreprise d'une Démarche Qualité de ses fournisseurs de Logiciels

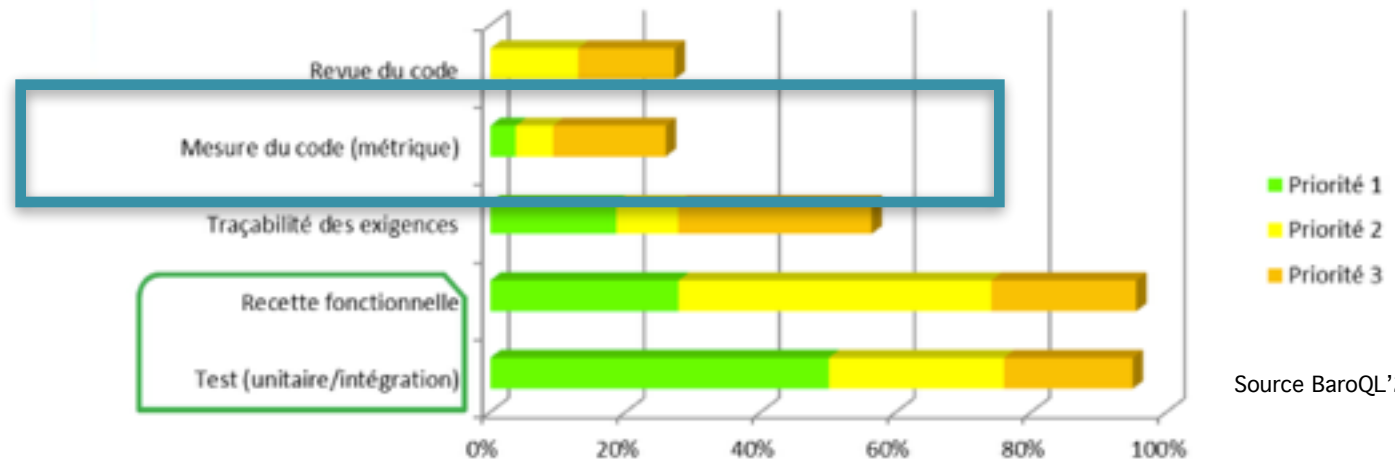


Nature Démarche Qualité Logicielle

Source BaroQL'2014

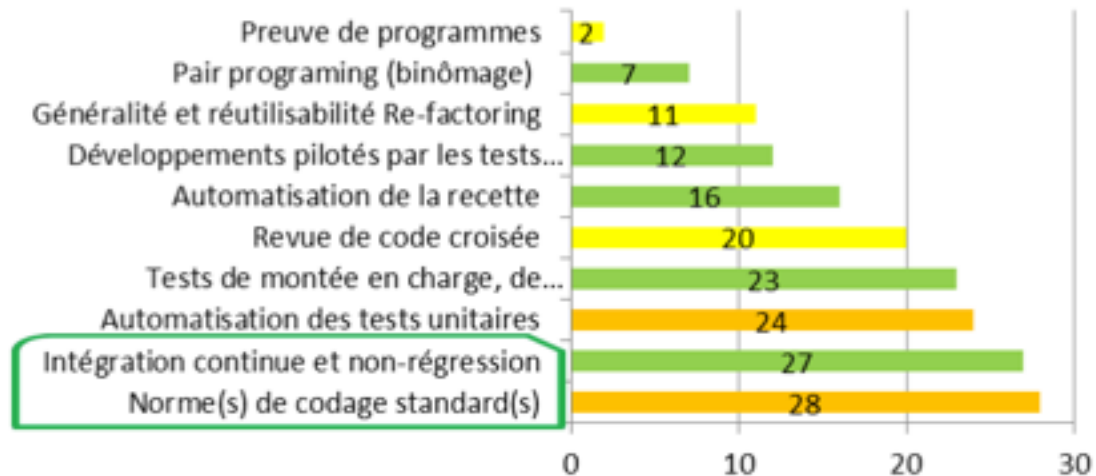


Comment réalisez-vous plutôt votre contrôle de la qualité (55 répondants)



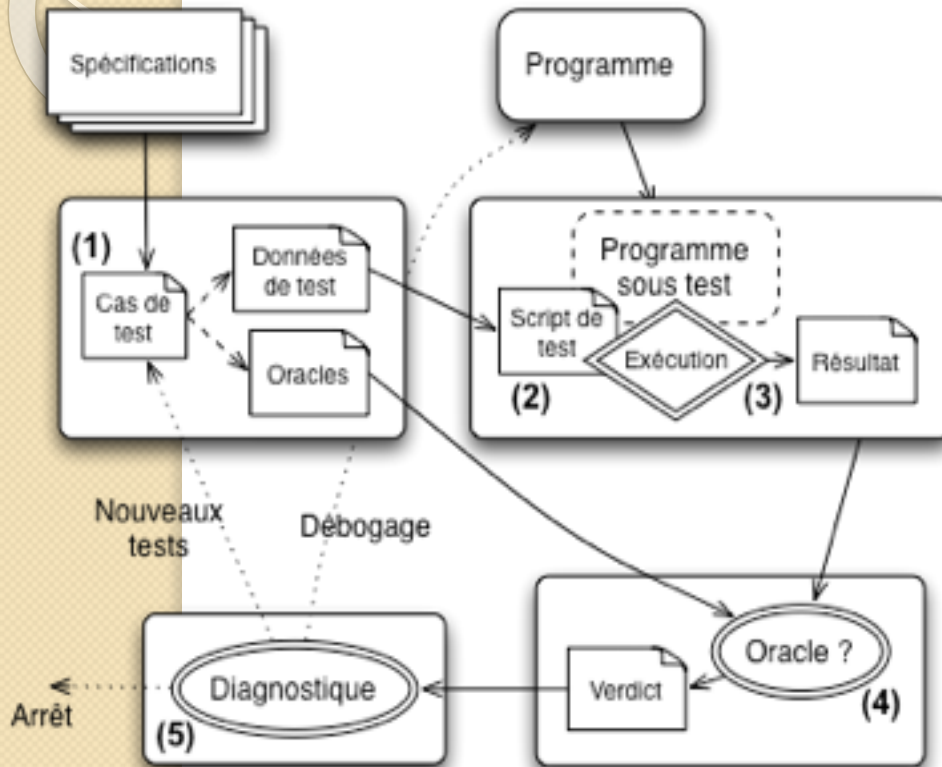
Source BaroQL'2012

PRATIQUES de conception/développement utilisées principalement pour améliorer vos applications ?



Source BaroQL'2012

Test logiciel vs. qualité logicielle



Test logiciel : définition

« Les test logiciel est un moyen d'assurer la **qualité des logiciels**, en vérifiant sur un ensemble de **cas pertinents** (car potentiellement capables de déclencher une faute) que le comportement du programme à l'exécution est **conforme** au comportement attendu »

(AFNOR)

Mesurer la qualité du code source

= donner des **indicateurs (= métriques)** permettant de "mesurer" la qualité

"Une métrique logicielle est une mesure d'une propriété d'une partie d'un logiciel ou de ses spécifications". (Wikipedia)

Il n'existe pas de métrique "ultime" : la pertinence de chaque métrique dépend de l'interprétation qui en est faite et du projet

Métriques Qualité logicielle

- Couverture du code par les tests (= Test coverage)

- **EclEmma**
- **Cobertura**
- Clover

- Complexité(s) du code

- JDepend
- **JavaNCSS**
- **Metrics**
- Crap4j

- "Bonnes pratiques" de développement

- **Checkstyle**
- **PMD**
- **Findbugs**

- (Performances)



Sonar
XRadar
QALab
Analytix



Couverture de code

- Mesure décrivant le taux de code source réellement exécuté (par les tests)
- **Plusieurs critères de couverture**
 - *Function coverage* : nombre de fonctions du programme appelées ?
 - *Statement coverage* : nombre de lignes du programme exécutées ?
 - *Condition coverage* : nombre de point d'évaluation exécuté et vérifié ?
 - *Path coverage* : tous les chemins d'exécution ont-ils été considérés ?
- Notion associée au test logiciel

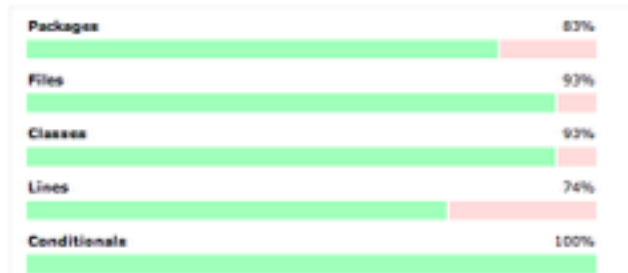
Cobertura / eCobertura

- Calcule le pourcentage de code couvert par les tests
- Indique les portions de code non exécutées
- <http://cobertura.github.io/cobertura/>

Code Coverage

Cobertura Coverage Report

Trend

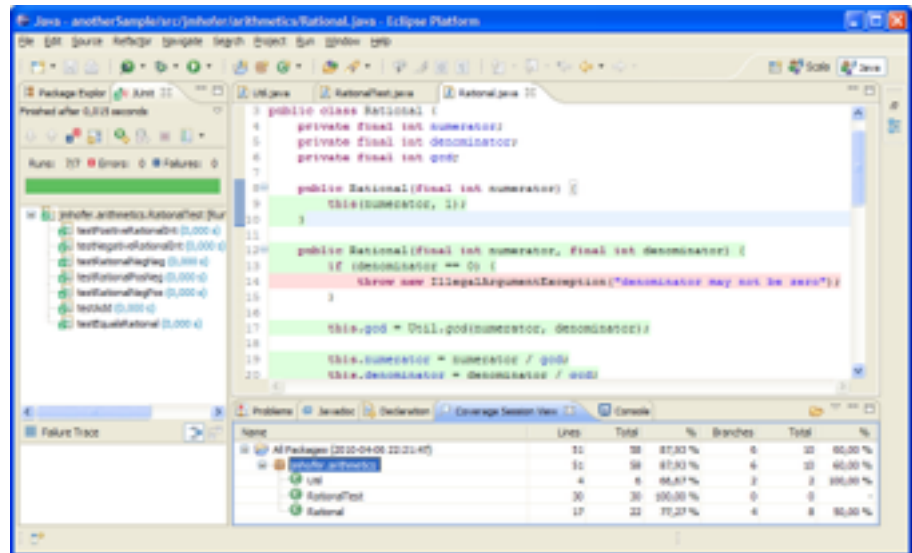


Project Coverage summary

Name	Packages	Files	Classes	Lines	Conditionals
Cobertura Coverage Report	83% 5/6	93% 37/40	93% 37/40	74% 977/1317	100% 0/0

Coverage Breakdown by Package

Name	Files	Classes	Lines	Conditionals
com.example.project	100% 18/18	100% 18/18	93% 575/724	N/A
com.example.project.test	100% 3/3	100% 3/3	64% 42/65	N/A
com.example.project.util	0% 0/2	0% 0/2	0% 0/208	N/A
com.example.project.service	100% 6/6	100% 6/6	77% 310/343	N/A
com.example.project.controller	88% 7/8	88% 7/8	80% 331/354	N/A
com.example.project.model	100% 3/3	100% 3/3	100% 19/19	N/A



Emma / eclEmma

- Calcule le pourcentage de code couvert
- Indique les portions de code non exécutées
- <http://emma.sourceforge.net/>
- <http://eclEmma.org/>

The screenshot shows the Eclipse IDE with the Emma coverage plugin. The left pane displays a project hierarchy for 'TestAllPackages'. The right pane shows the source code of 'CursorableLinkedList.java'. The bottom pane displays a table of coverage data for the 'TestAllPackages' project.

Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13730
org.apache.commons.collections	74,1 %	3642	5103
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

Complexité(s) > indicateurs triviaux

- Nombre total de lignes de codes
- Nombre de lignes de codes par objet
- Nombre de méthodes par objet
- Nombre total de méthodes
- Nombre total de classes
- Nombre total de paquets
- **Ratio lignes de codes/nombre de méthodes**
- Ratio lignes de codes/nombre d'objets
- **Ratio lignes de commentaires/lignes de codes**
- ...

Complexité cyclomatique

(Mesure de McCabe)

= *nombre de chemins linéairement indépendants qu'il est possible d'emprunter dans cette méthode*

= *nombre de points de décision de la méthode (if, case, while, ...) + 1 (le chemin principal)*

Si la complexité cyclomatique augmente, la difficulté à comprendre la méthode également

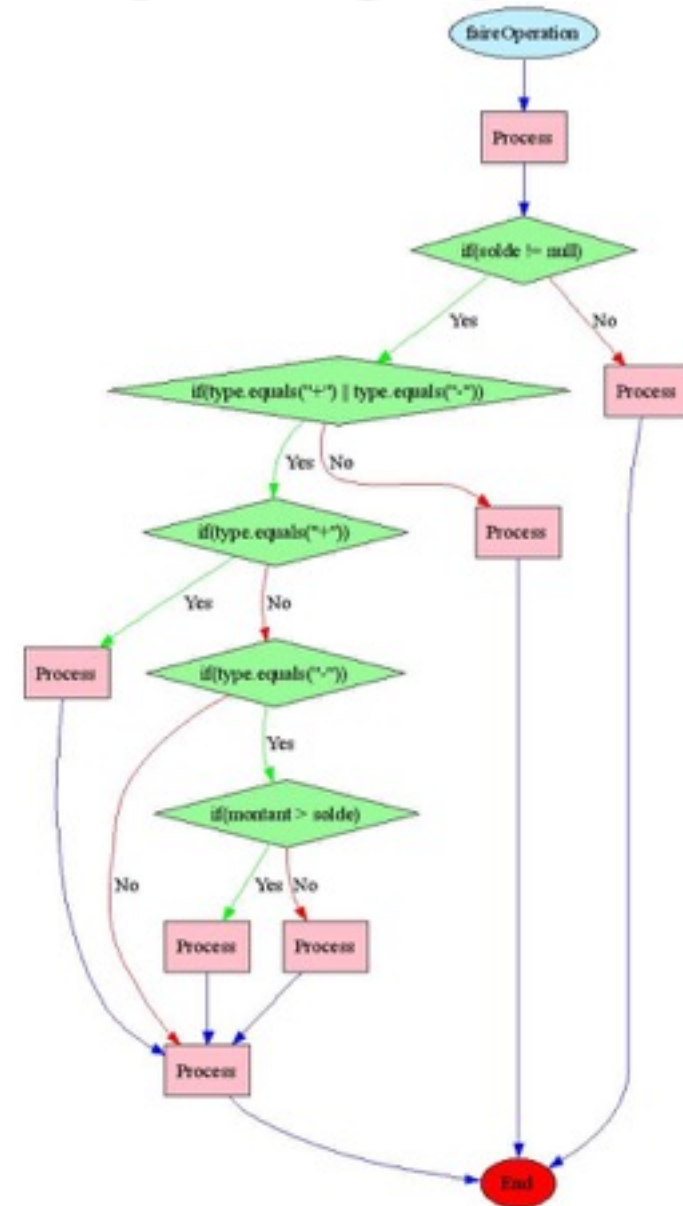


**Complexité cyclomatique =
nombre minimum de cas de tests unitaires nécessaires**

Complexité cyclomatique (exemple)

```
public void faireOperation(String type, double montant) {
    System.out.println("Début d'opération.");
    if(solde != null) {
        if(type.equals("+") || type.equals("-")) {
            if(type.equals("+")) {
                3 solde += montant;
            }
            if(type.equals("-")) {
                if(montant > solde) {
                    4 System.err.println("Solde insuffisant !");
                }
                5 else {
                    solde -= montant;
                }
            }
        }
    }
    else {
        System.err.println("Type d'opération invalide.");
    }
    else {
        System.err.println("Solde non initialisé.");
    }
    System.out.println("Fin d'opération.");
}
```

Complexité cyclomatique
= 5 "if ... else ..." + 1 = 6



Complexité > Indice de spécialisation

$$= \frac{\text{NORM} \times \text{DIT}}{\text{NOM}}$$

avec

- NORM : nombre de méthodes redéfinies.
- DIT : profondeur dans l'arbre d'héritage.
- NOM : nombre de méthodes de la classe

Cet indice **augmente** si

- le nombre de méthodes redéfinies augmente,
- la profondeur d'héritage augmente.

Il **diminue** si

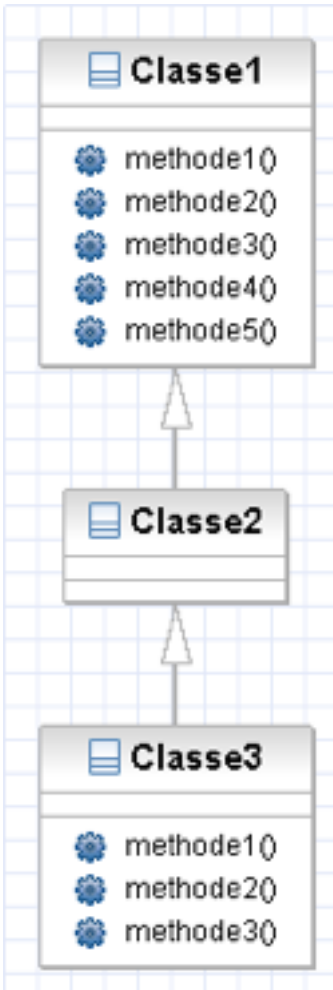
- le nombre de méthodes spécifiques à la classe augmente,
- le nombre de méthodes redéfinies diminue.

Indice de spécialisation > 1.5 => objet trop "spécialisé"



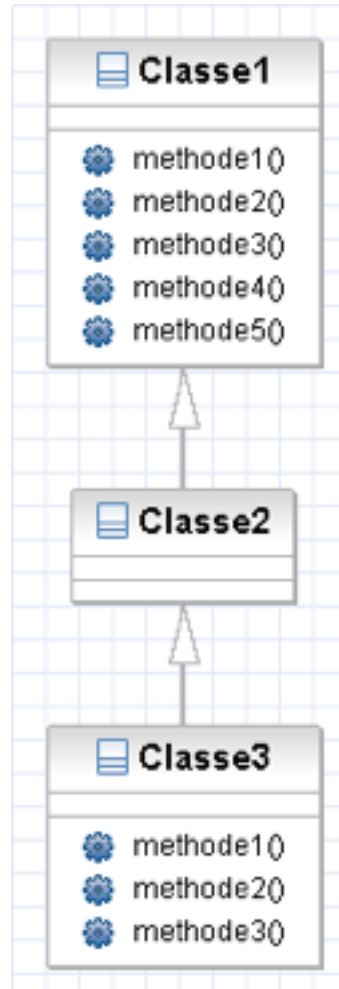
- re-factorisation
- utilisation d'interfaces
- délégation

Indice de spécialisation (exemple)

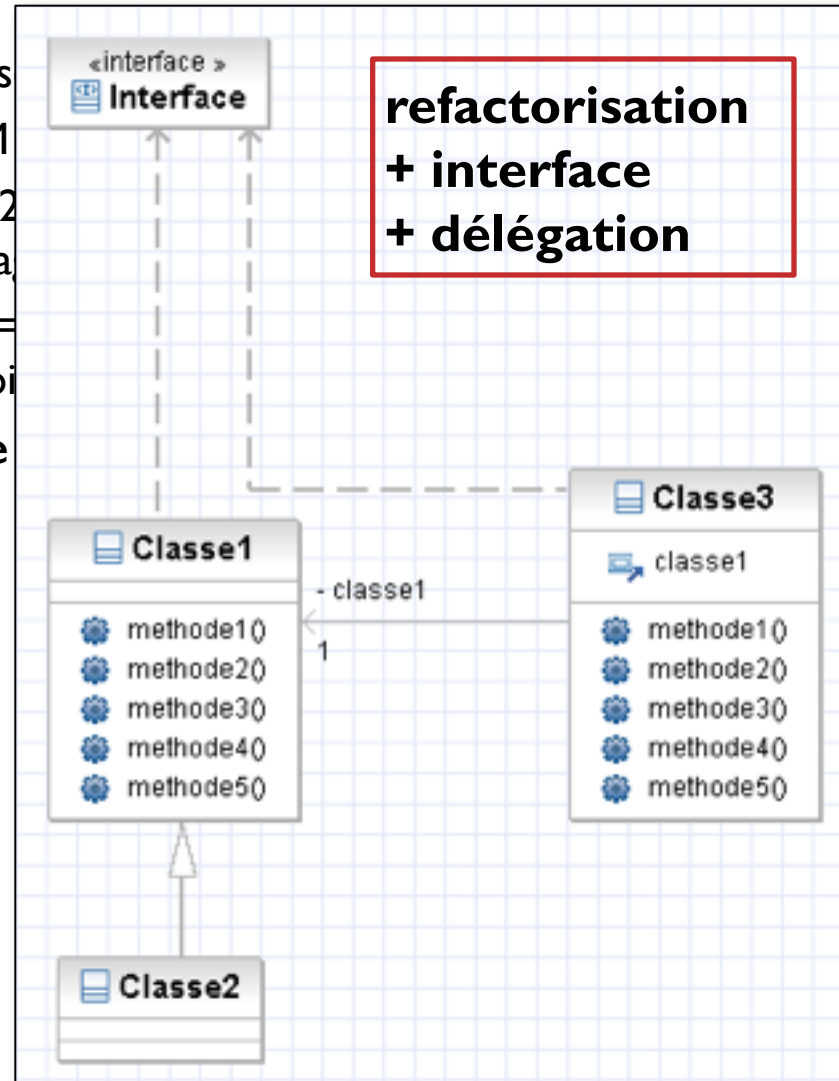


- Pour Classe3,
 - $NORM = 3$, puisqu'on redéfinit trois méthodes.
 - $DIT = 2$, il s'agit de la profondeur de Classe3 dans l'arbre d'héritage.
 - $NOM = 3$, puisqu'au total Classe3 possède trois méthodes (les trois méthodes redéfinies).
- **Indice de spécialisation** = $3 \times 2 / 3 = 2$

Indice de spécialisation (exemple)



- Pour Class
 - NORM
 - DIT = 2
 - d'héritage
 - NOM =
 - (les trois)
- Indice de



bre
hodes

D'autres indicateurs de complexité

- Complexité cyclomatique
- Indice de spécialisation
- Indice d'instabilité
 - *dépendance entre paquets*
- Indice d'abstraction
 - *niveau d'abstraction d'un paquetage*
- Indice de maintenabilité
 - *complexité de maintenance*
- ...

JavaNCSS

- Donne certaines métriques
 - Nombre de paquetages,
 - Nombre de classes,
 - Nombre de méthodes,
 - **Nombre d'instructions non commentées (NCSS)**,
 - *Non Commenting Source Statements*
 - Nombre de commentaires Javadoc,
 - **Complexité cyclomatique**

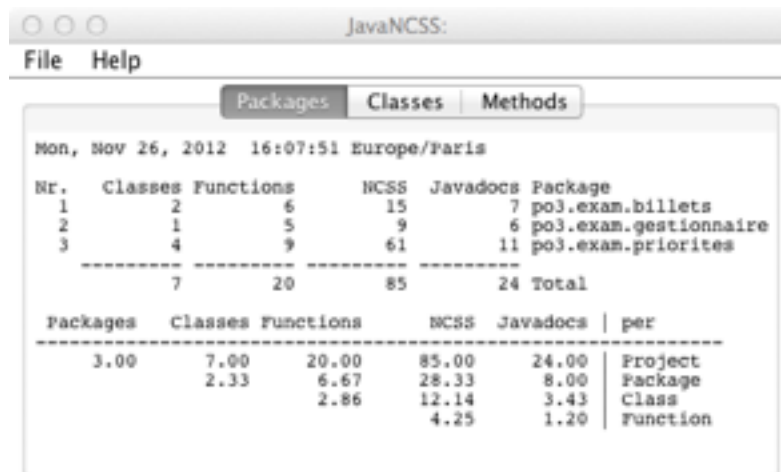
- Usage en ligne de commande

`javanccss [-<option>] <sources>`

`-gui | -xml | -out <fichier> |`

`-recursive | -package | -object | -function`

- **Compatible avec Ant ou Maven**
- <http://www.kclee.de/clemens/java/javanccss/>



JavaNCSS: File Help

Packages Classes Methods

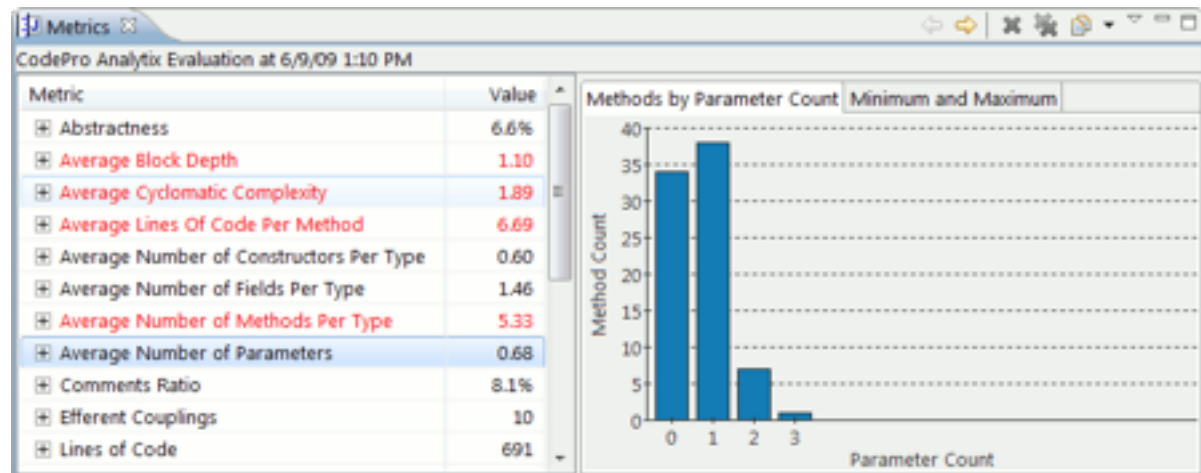
Mon, Nov 26, 2012 16:07:51 Europe/Paris

Nr.	Classes	Functions	NCSS	Javadocs	Package
1	2	6	15	7	poj.exam.billets
2	1	5	9	6	poj.exam.gestionnaire
3	4	9	61	11	poj.exam.priorites
7		20	85	24	Total

Packages	Classes	Functions	NCSS	Javadocs	per
3.00	7.00	20.00	85.00	24.00	Project
	2.33	6.67	28.33	8.00	Package
		2.86	12.14	3.43	Class
			4.25	1.20	Function

Metrics

- Plugin Eclipse
- <http://metrics2.sourceforge.net/>
- Calcule certaines métriques pour un projet, un paquetage ou une classe
 - Indice de spécialisation
 - Indice d'instabilité
 - complexité cyclomatique
 - Indice d'abstraction
 - ...
- Graphe de dependances entre paquetages



Conventions de code : pourquoi faire ?

- **Facilité la relecture et la compréhension du code**
 - Développement en équipe
 - Homogénéité du code
 - Relecture partagée
 - Maintenance (souvent réalisée par d'autres développeurs)
- **=> définition de conventions strictes pour l'écriture du code**
 - Respect des conventions : en partie outillé
 - auto-indentation, remise en forme, etc.

Checkstyle

= Vérifie le formatage et la présentation d'un code source Java

= respect de conventions de code

- Open-source (version 5.7)
 - <http://checkstyle.sourceforge.net>
- Plugins pour intégration dans les différents IDEs (Eclipse, Netbeans, Jedit, ...)
- Tâche ANT
- Vérifications réalisées :
 - présence de commentaires Javadoc,
 - conventions pour le hommage des attributs et des méthodes,
 - longueur des lignes de code,
 - bonne utilisation des import,
 - espacements entre les caractères,
 - détection de code dupliqué
 - ...

PMD

Analyse du code source Java

- Erreurs "classiques" ou Portions de code "potentiellement" source de problèmes = **Anti-patterns** de développement
 - Code mort : variables, méthodes inutilisées, ...
 - Expressions trop complexes
 - **if** non-nécessaires
 - Mauvais "design"
 - classes hors d'un paquetage
 - ...
 - Pas de redéfinition de equals() et hashCode()
 -
 - Objets instanciés inutilement
 - Code dupliqué / redondant
 - Mauvaises syntaxes
 - underscore
 - if sans { ... }
 - ...
- Logiciel open-source (version 5.0.5)
 - <http://pmd.sourceforge.net>
- Usage en ligne de commande ou via des plugins (Eclipse, Netbeans, ...)
 - tâches ANT / Maven

Findbugs



Analyse du Bytecode Java (.class)

= Détection de bugs "potentiels" basée sur des patterns = bugs classiques

- transtypage impossible
- comparaison avec ==
- equals() implémentée mais pas hashCode()
- Exceptions capturées, mais jamais levées,
- attributs non-initialisés dans le(s) constructeur(s)
- ...

- Open-source (version 2.03)
 - <http://findbugs.sourceforge.net>
- Intégration dans Maven
- Plugins pour Eclipse, Netbeans,
- ...



Pour aller plus loin

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

