

# GÉNIE LOGICIEL

Suite Diagramme de classes  
*Armelle PRIGENT*

## Laboratoire Informatique Image Interaction (L3i)

Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France  
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : <http://l3i.univ-larochelle.fr/>

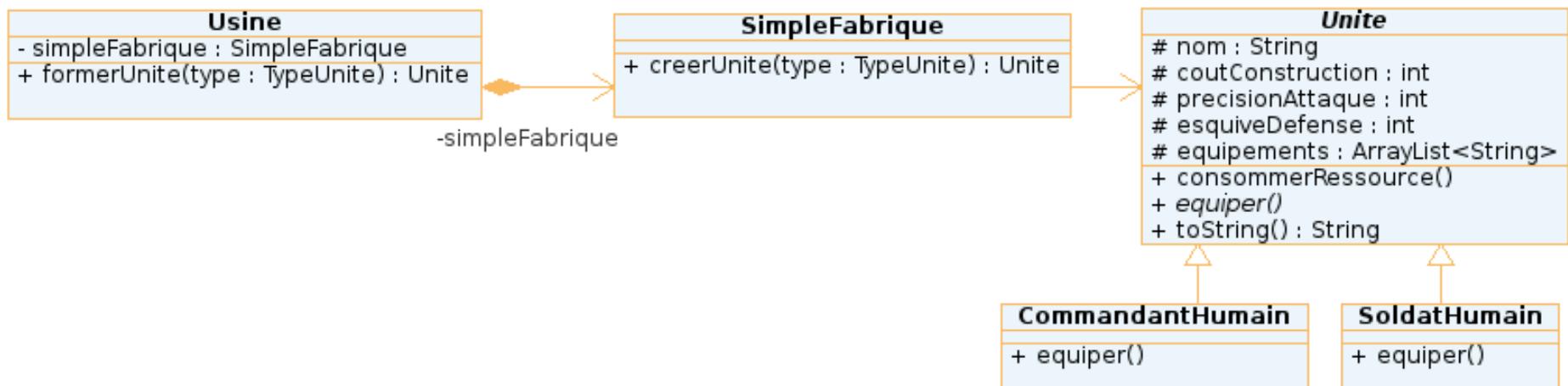
## Un exemple de polymorphisme : la Fabrique

- **Problème** : on souhaite pouvoir construire différents types d'objets en fonction d'un paramètre/contexte donné.
  - Ensemble de conditions et choix de la classe à instancier ? Couplage fort ... Que se passe-t-il si le catalogue d'objets à créer évolue?
- **Solution** :
  - Utilisation d'une fabrique en capsulant la logique de création sur un objet générique
  - Le polymorphisme garantit que les traitements se réalisent sur l'instance réellement créée.
  - Permet de déterminer dynamiquement quel objet d'un ensemble de sous-classes doit être instancié.
- **Utilité**
  - Le client ne peut déterminer le type d'objet à créer qu'à l'exécution
  - Il y a une volonté de centraliser la création des objets

# Une première version simple

- Création des instances au travers d'une simple fabrique

## Implémentation d'un début de solution avec une Simple Fabrique



<http://design-patterns.fr/fabrique-en-java>

# La classe Unité

## Implémentation de la classe abstraite Unité

```
// Classe abstraite dont toutes les unités du jeu hériteront.
public abstract class Unité
{
    protected String nom; // Nom de l'unité.
    protected int coutConstruction; // Coût de construction de l'unité.
    protected int précisionAttaque; // Précision de l'attaque de l'unité.
    protected int esquiveDéfense; // Faculté d'esquiver une attaque de l'unité.
    protected ArrayList équipements; // Tableau des équipements de l'unité.

    // Méthode qui consomme les ressources pour créer une unité.
    public void consommerRessource()
    {
        System.out.println("Consomme "+this.coutConstruction+" ressources pour la création de l'unité.");
    }

    // Méthode abstraite qui permet d'équiper l'unité.
    public abstract void équiper();

    // Méthode générique pour l'affichage de l'unité.
    public String toString()
    {
        String str = "Nom : "+this.nom+"\n";
        str += "Coût de construction : "+this.coutConstruction+"\n";
        str += "Précision d'attaque : "+this.precisionAttaque+"\n";
        str += "Esquive en défense : "+this.esquiveDéfense+"\n";
        str += "Équipements : ";
        for(int i=0; i<this.equipements.size(); i++)
        {
            str += this.equipements.get(i)+" ";
        }
        return str;
    }
}
```

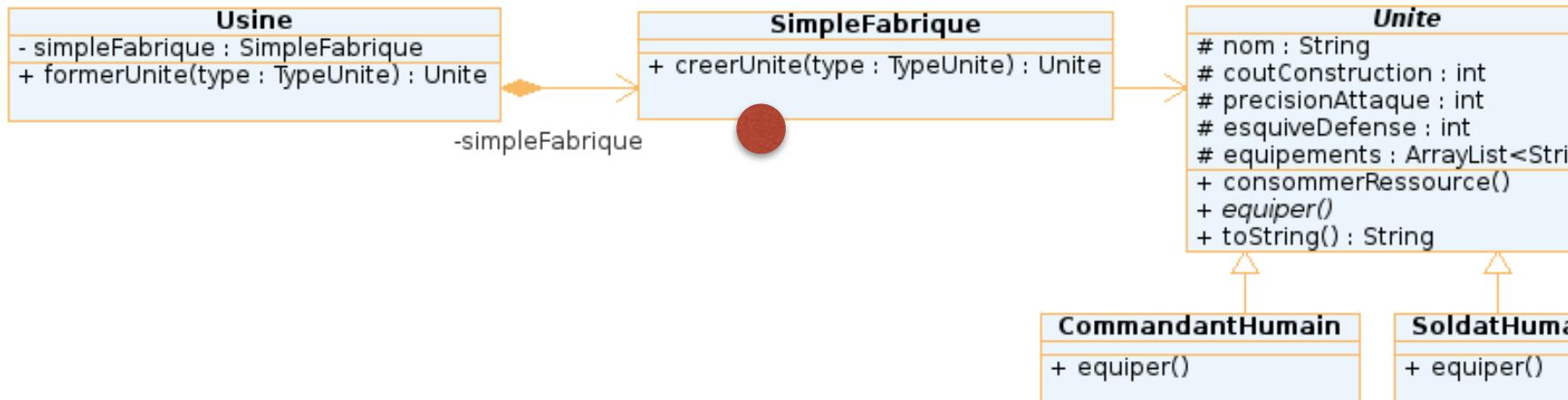
<http://design-patterns.fr/fabrique-en-java>

## Les classes de soldat

```
// Classe représentant un soldat humain.  
public class SoldatHumain extends Unite  
{  
    // Constructeur pour un soldat humain.  
    public SoldatHumain()  
    {  
        this.nom = "Fantassin";  
        this.coutConstruction = 5;  
        this.precisionAttaque = 1;  
        this.esquiveDefense = 2;  
        this.equipements = new ArrayList();  
    }  
  
    // Equiper un soldat humain.  
    public void equiper()  
    {  
        this.equipements.add("Pistolet");  
        this.equipements.add("Bouclier");  
        System.out.println("Equipement d'un soldat humain (Pistolet, Bouclier).");  
    }  
}
```

```
// Classe représentant un commandant humain.  
public class CommandantHumain extends Unite  
{  
    // Constructeur pour un commandant humain.  
    public CommandantHumain()  
    {  
        this.nom = "Lieutenant";  
        this.coutConstruction = 14;  
        this.precisionAttaque = 5;  
        this.esquiveDefense = 2;  
        this.equipements = new ArrayList();  
    }  
  
    // Equiper un commandant humain.  
    public void equiper()  
    {  
        this.equipements.add("Uzi");  
        this.equipements.add("Bouclier");  
        System.out.println("Equipement d'un commandant humain (Uzi, Bouclier).");  
    }  
}
```

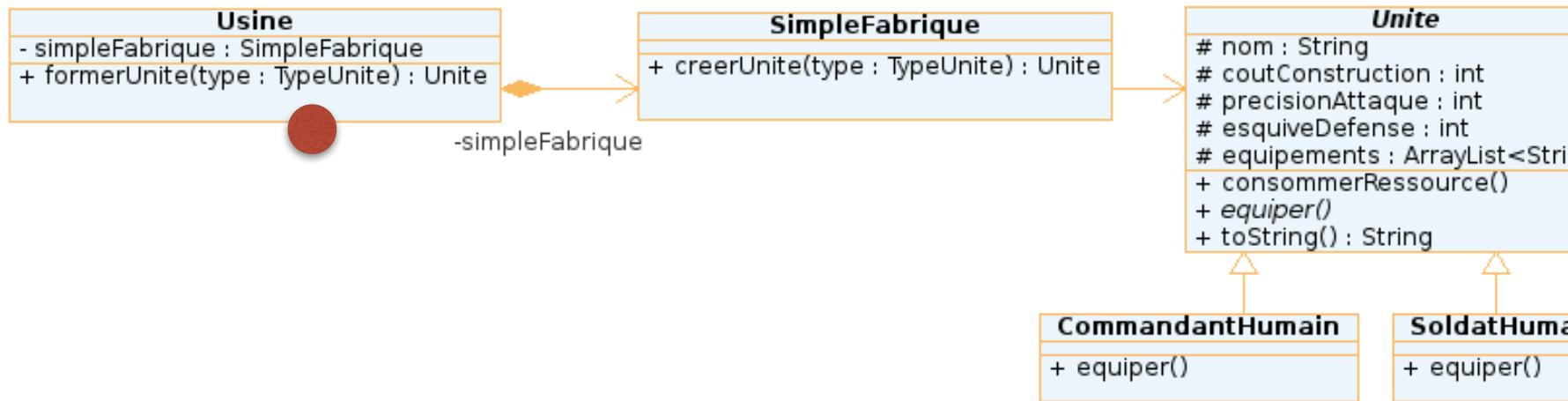
## Implémentation d'un début de solution avec une Simple Fabrique



```
{  
    // La création d'une unité en fonction de son type est encapsulée dans la fabrique.  
    public Unité créerUnité(TypeUnité type)  
    {  
        Unité unite = null;  
        switch(type)  
        {  
            case SOLDAT:unite = new SoldatHumain();break;  
            case COMMANDANT:unite = new CommandantHumain();break;  
        }  
        return unite;  
    }  
}  
  
// Enumération des types d'unités.  
public enum TypeUnité  
{  
    SOLDAT,  
    COMMANDANT  
}
```

<http://design-patterns.fr/fabrique-en-java>

## Implémentation d'un début de solution avec une Simple Fabrique



### Implémentation de la classe Usine

```
// Classe usine qui représente un bâtiment capable de construire des unités.
public class Usine
{
    private SimpleFabrique simpleFabrique;// Attribut contenant la fabrique simple.

    // Le constructeur permet de sélectionner la fabrique à utiliser.
    public Usine()
    {
        this.simpleFabrique = new SimpleFabrique();
    }

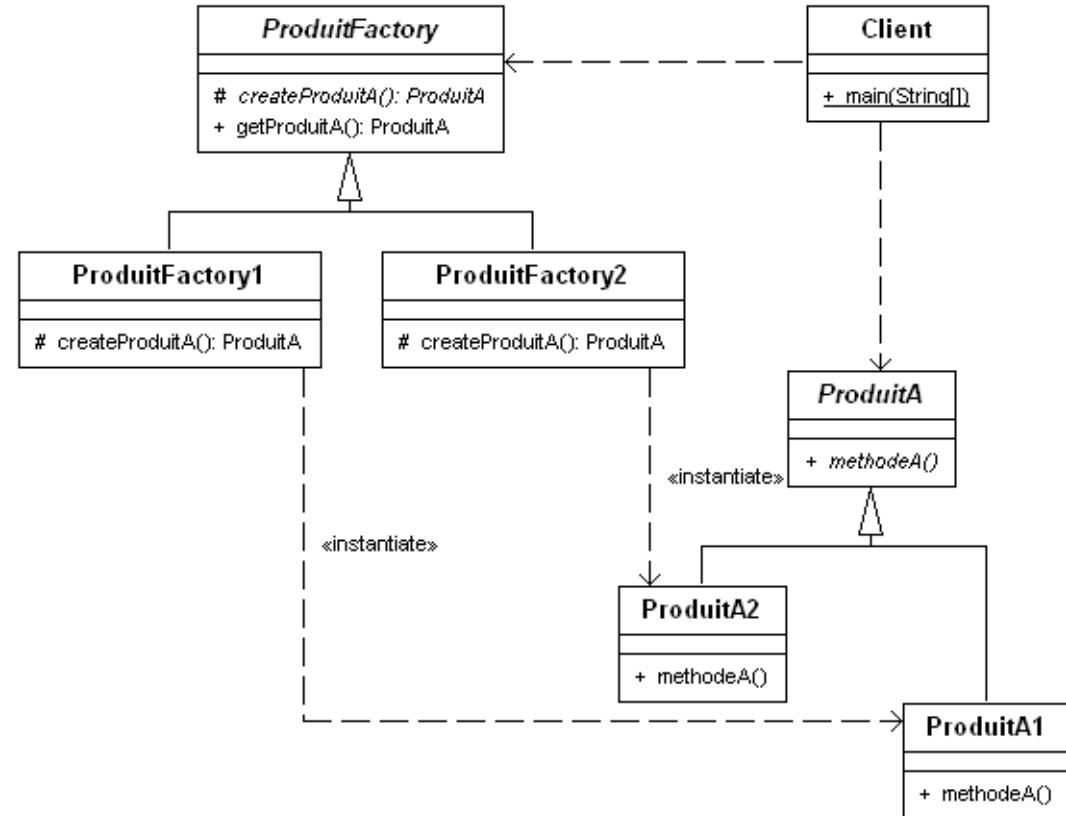
    // Méthode qui permet de construire l'ensemble des unités.
    public Unite formerUnite(TypeUnite type)
    {
        Unite unite = this.simpleFabrique.creerUnite(type);
        unite.consommerRessource();
        unite.equiper();
        return unite;
    }
}
```

<http://design-patterns.fr/fabrique-en-java>

# Fabrique (C) - modèle UML

- Extension à la problématique suivante :

- Que faire lorsque l'on a plusieurs types de produits et qu'on doit gérer des modes de fabrication différents ?



<http://www.jmdoudoux.fr/java/dej/chap-design-patterns.htm>

## Les produits

```
public abstract class ProduitA {  
    public abstract void methodeA();  
}
```

```
public class ProduitA2 extends ProduitA {  
  
    public void methodeA() { System.out.println("ProduitA2.methodeA()"); }  
}
```

```
public class ProduitA1 extends ProduitA {  
  
    public void methodeA() { System.out.println("ProduitA1.methodeA()"); }  
}
```

## Les fabriques

```
public abstract class ProduitFactory {  
  
    public abstract ProduitA createProduitA();  
  
    public ProduitA getProduitA() {  
        return createProduitA();  
    }  
}
```

```
public class ProduitFactory1 extends ProduitFactory {  
  
    public ProduitA createProduitA() {  
  
        return new ProduitA1();  
    }  
}  
  
public class ProduitFactory2 extends ProduitFactory {  
  
    public ProduitA createProduitA() {  
  
        return new ProduitA2();  
    }  
}
```

## Le client

```
public class Client {  
    public static void main(String[] args) {  
        ProduitA produitA = null;  
        produitA = new ProduitFactory1().getProduitA();  
        produitA.methodeA();  
  
        produitA = null;  
        produitA = new ProduitFactory2().getProduitA();  
        produitA.methodeA();  
    }  
}
```

# GÉNIE LOGICIEL

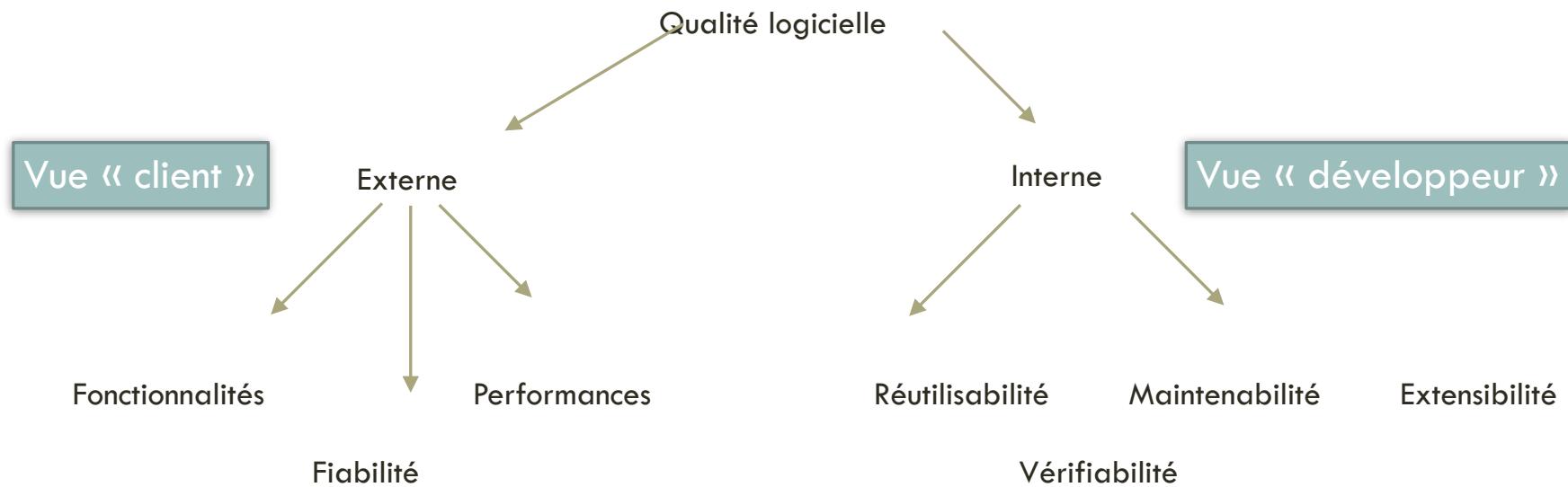
La métrique du logiciel  
*Armelle PRIGENT*

**Laboratoire Informatique Image Interaction (L3I)**

Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France  
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : <http://l3i.univ-larochelle.fr/>

# LA QUALITÉ

*Ensemble des caractères, des propriétés qui font que quelque chose correspond bien ou mal à sa nature, à ce qu'on en attend*



# LA QUALITÉ DU LOGICIEL

**Fiabilité** (ou robustesse) : aptitude d'un produit logiciel à fonctionner dans des conditions anormales

**Compatibilité** : facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.

**Intégrité** : aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés

**Validité** : aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications

**Efficacité** : Utilisation optimale des ressources matérielles.

**Réutilisation** : aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications

**Portabilité** : facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels

**Vérifiabilité** : facilité de préparation des procédures de test

**Extensibilité** : facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qui lui sont demandées

## UNE VISION DE LA DETTE TECHNIQUE

# La dette technique



Côté client



Côté développeur

# LES FACTEURS DE QUALITÉ DU LOGICIEL

## Métrique logicielle : mesure d'une propriété d'un logiciel

- Le nombre de lignes de codes, de commentaires
- Analyse de plus haut niveau sur la qualité de la conception

## Approche quantitative :

- extraire une mesure de la qualité d'un logiciel à partir de l'analyse statistique du code source.

*Avantage* : simplicité de mise en oeuvre.

*Principal problème* : trouver des indicateurs significatifs et les algorithmes correspondants.

## Analyse statique (ex: Sonar Qube)

- Convention de codage
- Détection automatique de comportements à haut risque
- Analyse de la complexité du code selon différentes méthodes

# LES DIFFÉRENTS TYPES DE MÉTRIQUES

## Trois catégories

- Mesurer le développement
- Mesurer les ressources
- Mesurer le logiciel

## Métrique du logiciel

- Métriques traditionnelles
  - Taille et complexité (lignes de code, Halstead)
  - Structure du logiciel (complexité cyclomatique)
- Métriques orientées objets (couplage, abstraction ...)

# METRIQUES TRADITIONNELLES

# UN DOCUMENT À LIRE

28/10/2013

ACTIVE  
UP

MÉTRIQUES ET CRITÈRES D'ÉVALUATION DE LA  
QUALITÉ DU CODE SOURCE D'UN LOGICIEL



Revue d'un professionnel de l'industrie du logiciel | Pierre Mengal

# SOURCE LINES OF CODE (SLOC)

Physique

Logique

Il n'y a actuellement aucun consensus sur la méthode de calcul de cette valeur. Il existe malgré tout une classification proposée par Boehm ([1984](#)) qui est toujours utilisée aujourd'hui mais qui est à interpréter avec beaucoup de précautions.

Classification	Size (KLOC)
Small (S)	2
Intermediate (I)	8
Medium (M)	32
Large (L)	128
Very Large (VL)	512

- Différence entre les frameworks
- Expérience des développeurs
- Pratiques de refactoring
- Pratique de réutilisation / génération de code
- Fiabilité de l'outil de mesure

# DENSITÉ DES COMMENTAIRES

$$DC = CLOC / SLOC$$

En général, entre 20 à 40%

Facteurs :

- Langage/framework choisi
- Qualité du texte rédigé
- Respect des normes sur la rédaction des commentaires
- Pas de commentaire sur le code généré
  - Nécessité du commentaire

# LA DUPLICATION DE CODE

La duplication de code (ou code clone) consiste en la répétition d'instructions similaires ou identiques dans un code source. Toute duplication du code viole le principe « DRY » (Don't Repeat Yourself) formulé pour la première fois par Hunt & Thomas dans l'ouvrage « The Pragmatic Programmer » ([1999](#)). Il est explicité par la phrase suivante : « Dans un système, toute connaissance doit avoir une représentation unique, non-ambiguë, faisant autorité ». Les auteurs suggèrent que la duplication de code tombe généralement dans quatre catégories.

1. **La duplication imposée.** Il arrive parfois que les exigences fonctionnelles imposent la duplication. Par exemple au niveau de la représentation d'une même information qui peut être différente à certains égards dans certaines parties du système. Les limitations de certains langages peuvent également contribuer à la duplication de code (ex : avec CORBA).
2. **La duplication par inadvertance.** Cela peut se produire en faisant des erreurs de design.
3. **La duplication par impatience.** Sous la pression des délais, les développeurs peuvent être tentés de dupliquer le code pour aller plus vite. Certains outils d'insertion de bouts de code préenregistrés favorisent d'ailleurs ce comportement.
4. **La duplication interdéveloppeur.** Des développeurs différents travaillant sur le même code source implémentent des fonctionnalités similaires sans s'en rendre compte.

# LA DUPLICATION DE CODE

Tenter d'atteindre le 0%

## *Actions possibles*

La détection du code dupliqué est donc un bon moyen d'augmenter la qualité du code en permettant d'identifier les parties à retravailler. L'utilisation d'un outil de détection de code dupliqué est donc requise dans tous les projets de développement professionnels.

La duplication du code est souvent due à un manque d'abstraction. Retravailler ces aspects permettra souvent de diminuer la duplication en augmentant la réutilisation.

# LA COMPLEXITÉ CYCLOMATIQUE

Mesure du nombre de chemins dans une fonction/méthode

Outils de calcul automatique

- Au niveau des méthodes
- Pas au niveau global

Une méthode trop complexe ou trop longue poserait les problèmes suivants:

- **Manque de lisibilité du code**: il est plus difficile pour un développeur externe de comprendre le fonctionnement du code. Ce problème pourra avoir un impact considérable sur la maintenabilité (voir [Dette Technique](#)).
- **Perte d'informations pendant le débogage**: moins les méthodes sont décomposées en sous-méthodes, moins riches seront les messages d'erreurs (stack trace moins précis). Cela ralentit le processus d'identification des problèmes.
- **Tests unitaires moins efficaces**: tester une méthode très complexe est souvent inefficace car un bon test unitaire doit pouvoir couvrir la plupart des cas de figure (chemins possibles).

# La complexité cyclomatique

Un code source qui présenterait trop de méthodes avec une complexité cyclomatique importante verrait sa valeur sensiblement réduite étant donné sa maintenabilité moindre.

Une autre façon de mesurer la complexité du code est d'utiliser les métriques d'Halstead ([1977](#)) qui base également ses calculs sur les opérateurs et expressions. Pour une revue en français des deux métriques, voir [Lambertz \(2007\)](#). D'après Watson & McCabe ([1996](#)), la complexité cyclomatique d'une fonction ne devrait pas dépasser 10.

## *Actions possibles*

Utiliser la technique de refactoring appelée **extraction de méthodes** et s'assurer que chaque méthode ne remplit qu'une seule et unique fonction.

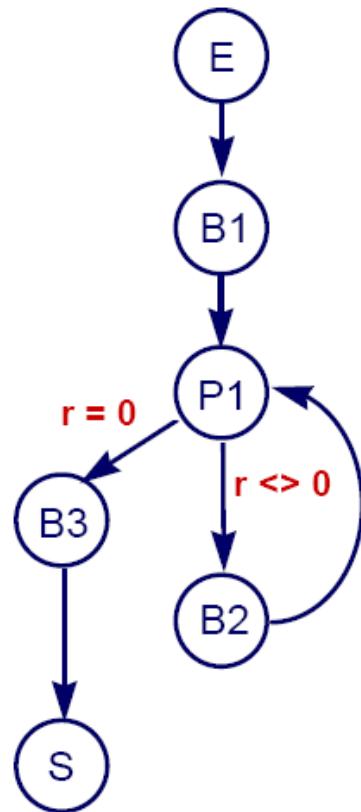
Faire un contrôle du niveau de complexité cyclomatique en continu pendant le développement permet d'obtenir une meilleure conception du code.

# CALCUL DE LA COMPLEXITÉ CYCLOMATIQUE : A - N + 2

**Précondition :**  $p_0 \geq q_0 > 0$

**Effet :**  $q = \text{pgcd}(p_0, q_0)$

```
read(p, q);           B1
r := p - p // q * q;
while r <> 0 do    P1
begin
  p := q;
  q := r;
  r := p - p // q * q
end;
write(q);            B3
```



# CORRECTION

Nb Mac Cabe = 2

Chemins :

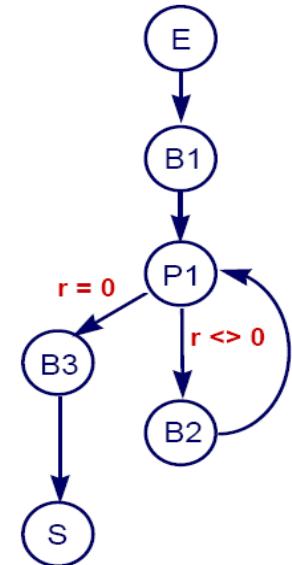
- 1 : E - B1 - P1 - B2 - P1 - B3 - S
- 2 : E - B1 - P1 - B3 - S

Tests sensibilisant les chemins

- 1 : 25 - 10
- 2 : 12 - 4

**Précondition** :  $p_0 \geq q_0 > 0$   
**Effet** :  $q = \text{pgcd}(p_0, q_0)$

```
read(p, q) ;          B1
r := p - p // q * q ;
while r <> 0 do      P1
begin
  p := q;
  q := r;
  r := p - p // q * q
end ;
write(q) ;           B3
```



## EXEMPLE

```
1 public class MonProg8a{
2
3     public static void main(String[] args){
4
5         Tortue rosalie = new Tortue();
6         int y = 100;
7         int x;
8
9         while (y<=550){                      // plusieurs créneaux
10            rosalie.saute(50,y);
11            x = rosalie.position().x();
12            while(x<550){                  // pour dessiner un créneau
13                for (int i=0; i<2; i++){    // premier demi-créneau
14                    rosalie.avance(50);
15                    rosalie.tourneDroite(90);
16                }
17                for (int i=0; i<2; i++){    // deuxième demi-créneau
18                    rosalie.avance(50);
19                    rosalie.tourneGauche(90);
20                }
21                x = rosalie.position().x();
22            }
23            y+=100;
24        }
25    }
26 }
```

# LES MÉTRIQUES DE HALSTEAD

## Bibliographie

- <http://www.virtualmachinery.com/sidebar2.htm>
- [http://www.verifysoft.com/fr\\_cmtpp\\_mscoder.pdf](http://www.verifysoft.com/fr_cmtpp_mscoder.pdf)

## MÉTRIQUES DE HALSTEAD - VOLUME

Les métriques de Halstead se basent sur les opérateurs et les opérandes

- Nombre total d'opérateurs uniques ( $n_1$ )
- Nombre total des opérateurs ( $N_1$ )
- Nombre total des opérandes uniques ( $n_2$ )
- Nombre total des opérandes ( $N_2$ )

Longueur du programme :  $N = N_1 + N_2$

Taille du vocabulaire  $n = n_1 + n_2$

VOLUME DU PROGRAMME :

- $V = N * \log_2(n)$

# MODE DE CALCUL

Int x = x + 1

**Operateurs uniques - n1 : 2**

- =
- +

**Operandes uniques - n2: 3**

- Int
- x
- 1

**Operateurs : N1 = 2**

**Operandes : N2 = 4 ( x apparait deux fois)**

Longueur :  $N1+N2 = 6$

Vocabulaire :  $n1 + n2 = 5$

Volume :  $N \log_2(n) = 6 \log_2(5) = 13,93$

# MÉTRIQUES DE HALSTEAD - DIFFICULTÉ ET NIVEAU DE PROGRAMME

Le niveau de difficulté d'un programme : risque d'erreurs

Dépendant

- du nombre d'opérateurs uniques ( $n_1$ )
- du nombre d'opérandes ( $N_2$ )
- Du nombre d'opérandes uniques ( $n_2$ )

DIFFICULTE DU PROGRAMME :

- $D = (n_1/2) * (N_2/n_2)$
- Exemple : int  $x=x+1$ 
  - $D = 2/2*4/3 = 1,333$

**Operateurs uniques** -  $n_1 : 2$

**Operandes uniques** -  $n_2 : 3$

**Operateurs** :  $N_1 = 2$

**Operandes** :  $N_2 = 4$

**L'utilisation récurrente des mêmes opérandes dans un programme est source d'erreurs**

# MODE DE CALCUL

Int x = x + 1

**Operateurs uniques - n1 : 2**

- =
- +

**Operandes uniques - n2: 3**

- Int
- x
- 1

Longueur :  $N1+N2 = 6$   
Vocabulaire :  $n1 + n2 = 5$   
Volume :  $N \log_2(n) = 6 \log_2(5) = 13,93$   
 $D = (n1/2) * (N2/n2) = 2/2 * 4/3 = 1,33$

**Operateurs : N1 = 2**

**Operandes : N2 = 4 ( x apparait deux fois)**

# MODE DE CALCUL

Int  $y = x + 1$

**Operateurs uniques - n1 : 2**

- =
- +

**Operandes uniques - n2: 4**

- Int
- x
- Y
- 1

**Operateurs : N1 = 2**

**Operandes : N2 = 4**

Longueur :  $N1+N2 = 6$

Vocabulaire :  $n1 + n2 = 6$

Volume :  $N \log_2(n) = 6 \log_2(6) = 15,509778$

$D = (n1/2) * (N2/n2) = 2/2 * 4/4 = 1$

## MÉTRIQUES DE HALSTEAD - DIFFICULTÉ ET NIVEAU DE PROGRAMME

Le niveau de programme est l'inverse du niveau de difficulté:

- $L = 1/D$

Un programme de bas niveau est plus enclin aux erreurs qu'un programme de haut niveau.

Ex 1 : 3/4

Ex2 : 1

# MÉTRIQUES DE HALSTEAD - EFFORT À L'IMPLÉMENTATION

L'effort à l'implémentation est proportionnel au volume (V) et au niveau de difficulté (D)

- $E = V*D$

Proposition d'estimation du temps d'implémentation de Halstead

- $T = E/18$
- Temps d'implémentation en secondes

Les bugs fournis : estimation du nombre d'erreurs dans le programme  
**Temps plus court pour le second programme**

- $B = (E^{(2/3)})/3000$

Int x = x+1

Volume :  $N \log_2(n) = 6 \log_2(5) = 13,93$

$D = (n_1/2) * (N_2/n_2) = 2/2 * 4/3 = 1,33$

$E = 13,93 * 1,33 = 18,52$

$T = 18,52/18$

Int y = x + 1

Volume :  $N \log_2(n) = 6 \log_2(6) = 15,509778$

$D = (n_1/2) * (N_2/n_2) = 2/2 * 4/4 = 1$

$E = 15,509 * 1$

$T = 15,509 / 18$

## L'INDEX DE MAINTENABILITÉ

Calculé à partir des mesures de Halstead et du nombre cyclomatique

AveV = moyenne du volume de Halstead par module

AveG = moyenne de la compléxité cyclomatique par module

AveLoc = moyenne du nombre de ligne de code (physique) par module

**Mlwoc = 171 - 5.2 \* ln(aveV) -0.23 \* aveG -16.2 \* ln(aveLOC)**

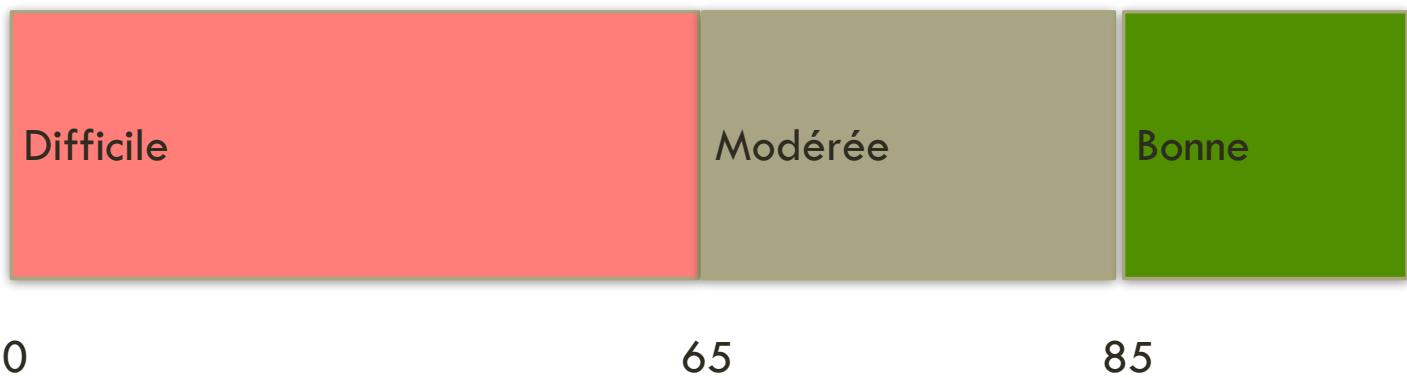
# L'INDEX DE MAINTENABILITÉ

Pour les lignes de commentaire

- $Mlcw = 50 * \sin(\sqrt{2.4 * perCM})$
- perCM

Index de maintenabilité :

- $MI = MIwoc + Mlcw$



Metrics		Value
Cyclomatic number	(v(G))	2
Number of physical lines	(LOCphy)	17
Number of program lines	(LOCpro)	14
Number of blank lines	(LOCbl)	2
Number of commented lines	(LOCcom)	1
Program length	(N)	55
Number of operators	(N1)	29
Number of operands	(N2)	26
Vocabulary size	(n)	21
Number of unique operators	(n1)	10
Number of unique operands	(n2)	11
Program volume	(V)	241.577
Number of delivered bugs	(B)	0.067
Difficulty level	(D)	11.818
Effort to implement	(E)	2855.006
Program level	(L)	0.085
Implementation time	(T)	158.611
Max nesting depth	(MaxND)	2
Maintainability Index w.o.c	(MIwoc)	96.109
MI comment weight	(MICw)	18.348
MI with comments	(MI)	114.456

**Longueur : N1+N2 =**  
**Vocabulaire : n1 + n2 =**  
**Volume : Nlog2(n) =**  
**D=(n1/2)\*(N2/n2) =**  
**E = V\*D =**  
**T = E/18**

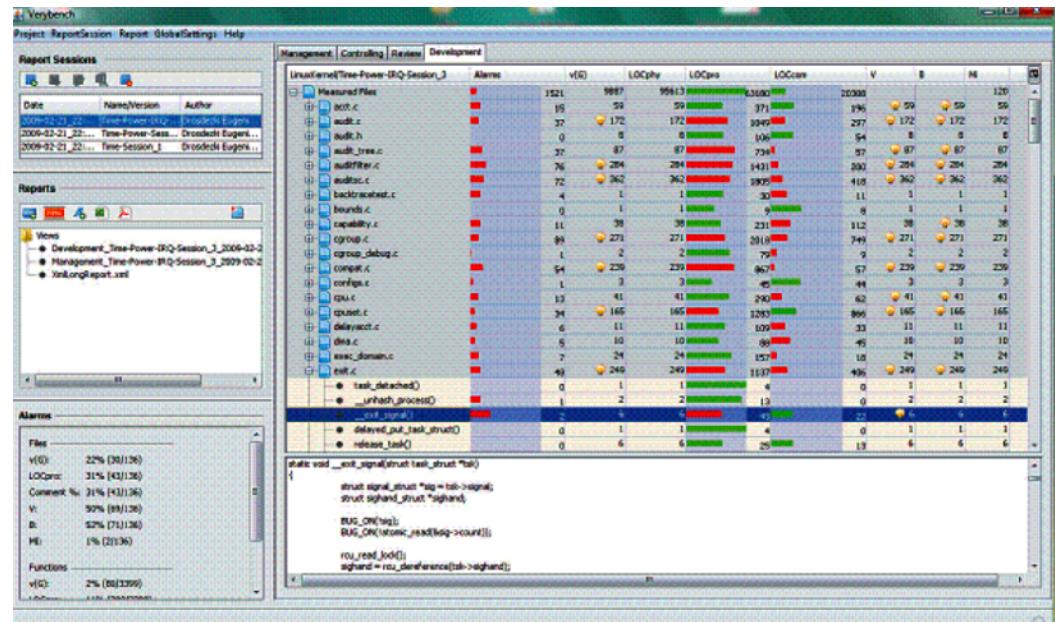
$$\begin{aligned}
 \text{MIwoc} &= 171 - 5.2 * \ln(\text{aveV}) \\
 &- 0.23 * \text{aveG} - 16.2 * \ln(\text{aveLOC})
 \end{aligned}$$

# LES OUTILS

Peu (ou pas) de logiciels libres pour calculer ces valeurs

## Logiciels commerciaux

- TestWell CMT ++ / TestWell CMT Java
- Sonar code



# METRIQUES ORIENTÉES OBJET

# AFFERENT ET EFFERENT COUPLING

L'Afferent coupling représente le nombre de références vers la classe mesurée. Ces références doivent être externes à cette classe. Les références internes ne comptent pas. Cette métrique donne une bonne indication de l'importance de la classe dans le code.

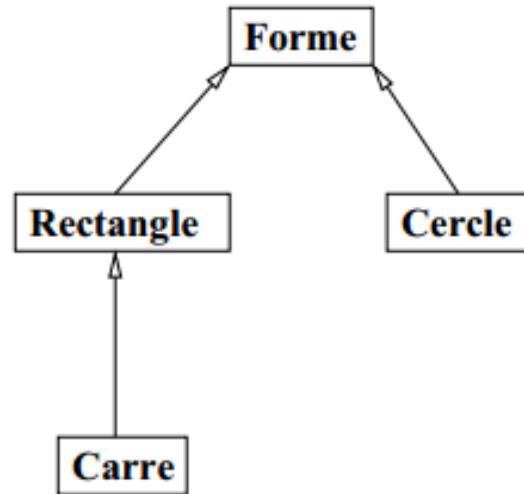
L'Efferent coupling est la mesure du nombre de types que la classe « connaît ». Cela comprend: l'héritage, l'implémentation d'interfaces, les types des paramètres, les types de variable, les exceptions levées et capturées. En bref, tous les types mentionnés dans le code source de la classe mesurée.

## *Actions possibles*

Vérifier que le code source est bien utilisé dans son entièreté et éliminer tout code superflu.

Séparer les responsabilités dans des classes différentes. Respecter la loi de Demeter ou « Law of Demeter » ([Lieberherr & Holland 1989](#)) peut également contribuer à réduire un trop fort couplage. Une étude de Basili & al. ([1996](#)) suggère que le respect de cette loi permet de réduire les bugs de manière significative. Il faut cependant vérifier que cette action ne soit pas exagérément appliquée car elle pourrait contribuer à réduire la lisibilité du code en le rendant plus complexe inutilement.

# Afferent et efferent coupling



Prenons un exemple avec le principe de l'héritage en orienté objet. La classe **Carre** référence la classe **Rectangle** qui elle référence la classe **Forme**. La classe **Cercle** quant à elle référence la classe **Forme** également.

Dans cet exemple, l'Afferent coupling de **Forme** est de 2 puisque 2 autres classes la référencent directement. Il est de 1 pour **Rectangle**, mais de 0 pour **Carre** et **Cercle**. L'Efferent coupling est de 0 pour **Forme** car il ne référence aucune autre classe. Il est donc de 1 pour toutes les autres classes du schéma. Le schéma ci-dessous résume le principe du couplage.

# STABILITÉ / INSTABILITÉ CE/(CE+CA)

L'instabilité d'un module est définie par son niveau de résistance au changement. Plus un module est stable, moins il est facile de le changer. Elle est obtenue en calculant le rapport entre l'efferent coupling au couplage total. Cette valeur est donc obtenue en divisant l'efferent coupling à la somme de l'efferent coupling et l'afferent coupling.

Exemple de calcul de l'instabilité avec l'exemple cité dans la section de l'Afferent & efferent coupling :

Classe	Afferent Coupling (Ca)	Efferent Coupling (Ce)	Instability (Ce / (Ce + Ca))
Forme	2	0	$0 / (0 + 2) = 0$
Rectangle	1	1	$1 / (1 + 1) = 0.5$
Cercle	0	1	$1 / (1 + 0) = 1$
Carre	0	1	$1 / (1 + 0) = 1$

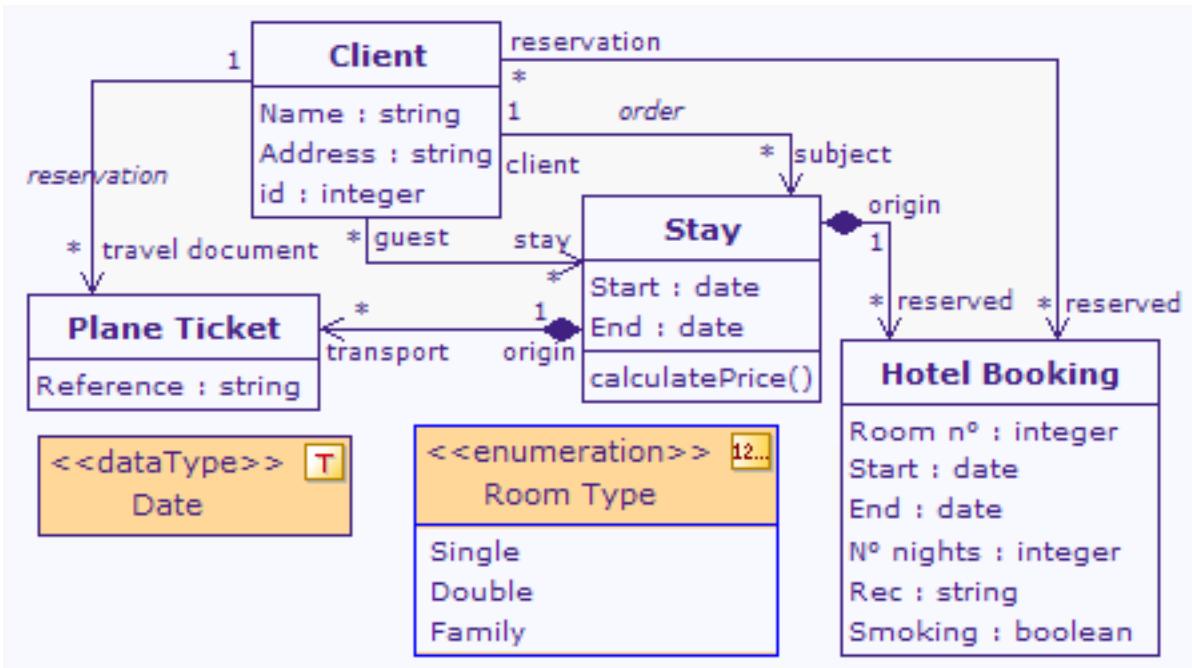
Ainsi Forme est très stable car le rapport entre ses dépendances efferantes sont nulles par rapport à

# STABILITÉ

L'une des particularités d'un code source de qualité est qu'il est très aisément modifiable ([Martin 2000](#)). Dans un contexte très compétitif où l'entreprise doit sans cesse intégrer de nouvelles fonctionnalités, c'est un atout majeur. Il est donc intéressant d'avoir un certain niveau d'instabilité dans le code pour permettre ces changements. Les classes doivent être soit très stables (entre 0,0 à 0,3) soit très instables (0,7 à 1,0). Dans le cas des classes très stables, il est recommandé d'avoir un niveau d'abstraction élevé (voir Abstractness).

## *Actions possibles*

Séparer les responsabilités dans des classes différentes. Il est également possible d'avoir un nombre important de modules stables s'ils sont conçus pour être facilement extensibles ([Martin 2000](#), page 25).



# ABSTRACTNESS

L'abstractness d'un module indique son niveau d'abstraction par rapport aux autres classes présentes dans le code. On obtient cette valeur en calculant le rapport entre les types abstraits internes (classes abstraites & interfaces) et les autres types internes. La valeur se situe entre 0 et 1, 0 indiquant un module complètement concret et 1 un module complètement abstrait.

Par exemple une classe abstraite en Java ou en C# est une classe comme les autres à une différence près : elle ne peut pas être instanciée. Elle est donc utilisée principalement comme classe de base dans l'héritage. Par exemple la classe « Chien » hérite de la classe « Animal ». On peut instancier un « Chien », mais on veut empêcher d'instancier un objet « Animal ». Pour ce faire, on déclare « Animal » comme classe abstraite. Une interface peut être comparée à une classe mais dont on a gardé que les signatures de ses membres (méthodes & propriétés par exemple). On n'hérite pas d'une interface, mais on l'implémente. C'est-à-dire que la classe qui utilise une interface donnée doit fournir le code pour les membres présents dans cette dernière.

Cette métrique, à elle seule, n'est pas utile. Combinée avec la métrique **Instability**, elle permet de calculer la **Distance from main sequence** (section suivante). Elle dépend de la mesure d'instabilité.

# DISTANCE FROM MAIN SÉQUENCE

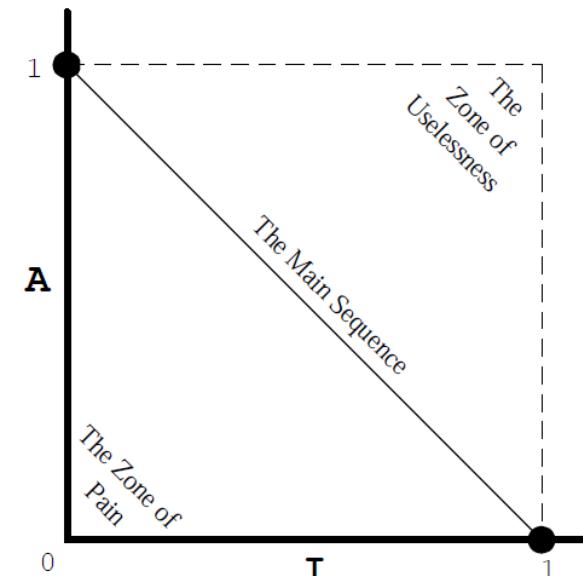
Relation entre l'abstraction et l'instabilité

$$D' = | A+I-1 |$$

Une classe très stable mais très abstraite sera donc considérée comme très bonne. Une classe très instable mais très concrète également. En réalité, plus on se trouve proche de la ligne qui relie ces deux points, mieux c'est.

Tout comme les autres métriques en relation avec le coupling, elle peut apporter des informations intéressantes qui peuvent être utilisées pour améliorer l'architecture et la qualité du logiciel. Il faut cependant tenir compte du fait qu'elles ne sont pas parfaites et l'utilisation de ces dernières comme seul indicateur est téméraire ([Martin 2000](#), page 27).

Plus la valeur est proche de zéro, mieux c'est. Une valeur supérieure à 0.7 peut être problématique mais il est difficile d'éviter de tels résultats dans certains cas ([Smacch](#))



# LACK OF COHESION OF METHODS

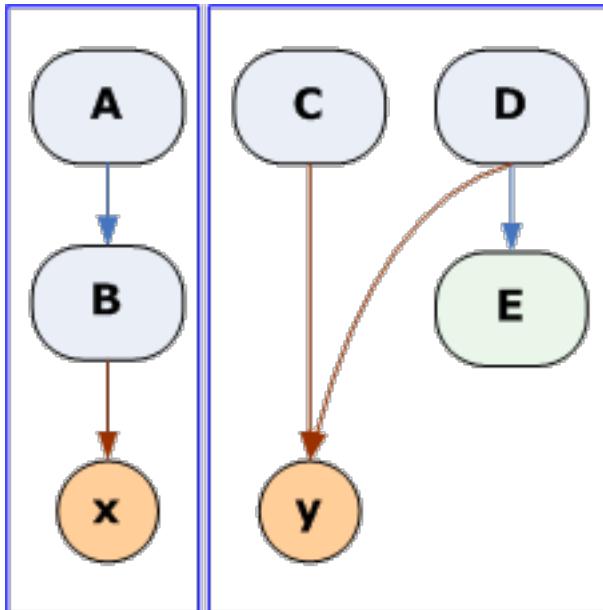
$$LCC = 1 - \frac{\sum MF}{M \times F}$$

- M est le nombre de méthodes dans la classe (inclus les méthodes statiques et d'instance, les constructeurs, les propriétés et les events).
- F est le nombre de champs d'instance.
- MF est le nombre de méthodes de la classe appelant un champ donné.
- Sum(MF) est donc la somme totale de la valeur obtenue par MF sur l'ensemble des champs de la classe.

Une classe qui contient des méthodes et des champs d'instance qui se réfèrent beaucoup entre elles aura une très bonne cohésion. Cela signifie en clair que ces méthodes et champs répondent à un même besoin. Si la cohésion est faible, cela pourrait indiquer que la classe ne respecte pas ces principes et qu'il faut la diviser.

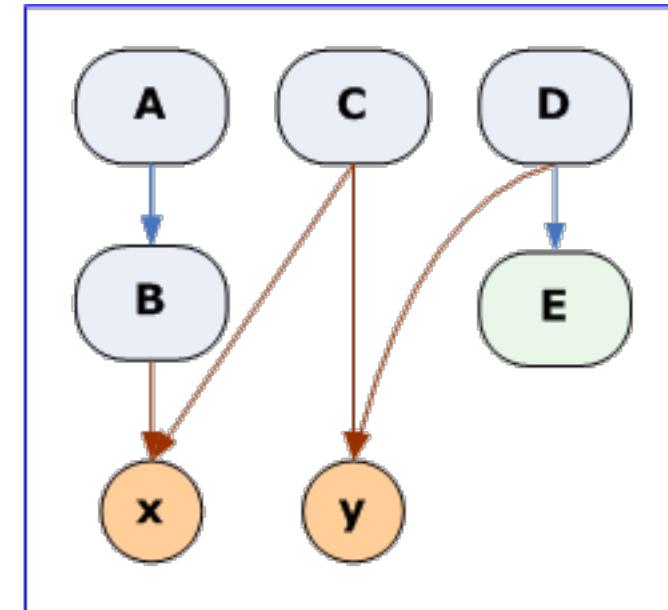
$$LCOM = 1 - \frac{\sum MF}{M \times F}$$

5 méthodes  
2 champs d'instance



$$1 - 3/10 = 0,7$$

Manque de cohésion



$$1 - 4/10 = 0,6$$

# RELATIONAL COHÉSION

La relational cohesion est le nombre moyen de relations internes à un module ([Smacchia 2013](#)). Si R représente le nombre de références qui sont internes au module et N le nombre total de types qu'il contient, alors la relational cohesion sera calculée à l'aide de la formule  $H = (R + 1) / N$ .

Si un module contient 50 classes et que ces 50 classes réfèrent au moins 2 autres classes du même module, le calcul donnera:  $(100 + 1 / 50)$  soit 2.02. Une module ou ensemble de classes devrait avoir une valeur élevée. Cela signifie que les différentes classes qui le composent sont fortement en relation.

Il s'agit donc là d'une métrique intéressante pour évaluer la qualité du code. Une valeur se situant entre 1.5 & 4.0 est conseillée ([Smacchia 2013](#)). Une valeur trop élevée pourrait néanmoins témoigner d'une complexité trop grande.

# TAUX DE SPÉCIALISATION

L'index de spécialisation (specialization index) donne comme indication le degré de spécialisation d'une classe. Il est calculé avec la formule suivante :

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

- NORM (Number of Overriden Methods) est le nombre de méthodes redéfinies
- DIT (Depth in Inheritance Tree) est la profondeur de l'arbre d'héritage (0 = pas d'héritage)
- NOM (Number of Methods) est le nombre de méthodes de la classe

Plus la classe redéfinit des méthodes qu'elle hérite d'autres classes et plus la profondeur de l'héritage est grand, plus cet indice augmente. Il diminue quand le nombre de méthodes spécifiques augmente et quand les méthodes redéfinies diminuent. Une valeur de 1.5 est considérée comme trop élevée ([Larive 2013](#)).

## EXEMPLE

NOM : 3

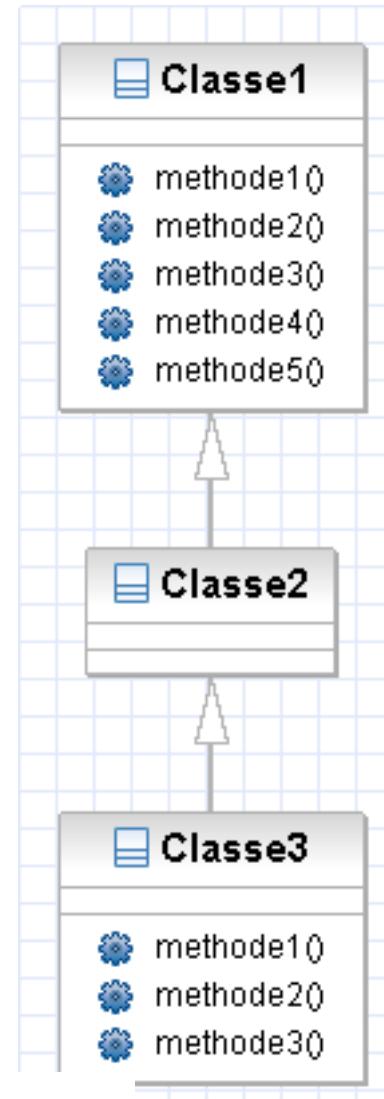
DIT : 3

NORM : 3

$$IS = (3 * 3 / 3) = 3$$

Refactoring nécessaire

Interface



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

Métrique	Méthode	Langage	Localisation	Valeurs possibles	Valeur repère	Intérêt
Complexité cyclomatique	AS	Tous	A-MO-C-ME	1 à $\infty$	Max 10	Elevé
SLOC	AS	Tous	A-MO-C-ME	1 à $\infty$	-	Faible
Densité des commentaires	AS	Tous	A-MO-C-ME	0 à 100%	Entre 20% & 40%	Modéré
Couverture de code	AS*	Tous	A-MO-C-ME	0 à 100%	80%	Elevé
Duplication de code	AS	Tous	A-MO-C-ME	0 à 100%	0	Faible
Afferent coupling	AS	Tous	MO-C	0 à $\infty$	-	Elevé
Efferent coupling	AS	Tous	MO-C	0 à $\infty$	Max 20	Elevé
Instability	AS	Tous	MO	0 à 1	Entre 0.0 & 0.3 et entre 0.7 et 1.0	Elevé
Abstractness	AS	Tous	MO	0 à 1	-	Elevé
Distance from main sequence	AS	OO	MO	0 à 1	Max 0.7	Elevé
LCOM	AS	OO	ME	0 à $\infty$		Elevé
Relational Cohesion	AS	OO	MO	0 à $\infty$	Entre 1.5 & 4.0	Elevé
Specialization Index	AS	OO	C	0 à $\infty$	Maximum 1.5	Elevé
Size of instance	AS	OO	C	0 à $\infty$	Max 64	Faible

Légende : AS = Analyse statique | EX = Expert | AU = Audit | A = Application | MO = Module | C = Classe | ME = Méthode | OO = Orienté Objet

<b>Number of variables</b>	AS	Tous	ME	0 à ∞	Max 5	Faible
<b>Number of methods</b>	AS	OO	C	0 à ∞	Max 8	Faible
<b>Number of overloads</b>	AS	OO	C	0 à ∞	Max 6	Faible
<b>Number of coding rules violations</b>	AS	Tous	A-MO-C-ME	0 à ∞	-	Elevé
<b>Maintenabilité</b>	EX	Tous	A-MO-C-ME	-	-	Elevé
<b>Evolutivité</b>	EX	Tous	A-MO-C-ME	-	-	Elevé
<b>Performance</b>	EX + AS + AU	Tous	A-MO-C-ME	-	-	Modéré
<b>Pertinence</b>	EX	Tous	A-MO-C-ME	-	-	Modéré
<b>Style &amp; lisibilité</b>	EX + AS	Tous	A-MO-C-ME	-	-	Modéré
<b>Documentation</b>	EX	Tous	A-MO-C-ME	-	-	Modéré
<b>Fiabilité</b>	EX	Tous	A-MO-C-ME	-	-	Elevé
<b>Portabilité</b>	EX	Tous	A-MO-C-ME	-	-	Modéré
<b>Sécurité</b>	EX	Tous	A-MO-C-ME	-	-	Modéré
<b>Nombre de bugs</b>	EX	Tous	A-MO-C-ME	0 à ∞	-	Modéré

# ECLIPSE METRICS

## Installation plugin

- Install new software
- <http://metrics.sourceforge.net/>

## Activation sur chaque package

The screenshot shows the Eclipse Properties dialog for a Java project. The left pane lists various configuration sections: Resource Builders, Checkstyle, Java Build Path, Java Code Style, Java Compiler, Java Editor, Javadoc Location, Metrics (which is highlighted with an orange bar), PMD, Project References, Refactoring History, Run/Debug Settings, Task Repository, Task Tags, Validation, and WikiText. The right pane contains a single checkbox labeled "Enable Metrics".

type filter text

Metrics

Enable Metrics

- Resource Builders
- Checkstyle
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Metrics**
- PMD
- PMD
- Project References
- Refactoring History
- Run/Debug Settings
- Task Repository
- Task Tags
- Validation
- WikiText



# INSTALLATION METRICS

Problems Javadoc Declaration Console djUnit Coverage Report Metrics - CoffeeMaker - Number of Static Methods (avg/max per type)   

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
+ Number of Static Methods (avg/max per type)	10	0.769	2.665	10	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Total Lines of Code	861					
+ Afferent Coupling (avg/max per packageFragment)		0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Normalized Distance (avg/max per packageFragmen		0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Number of Classes (avg/max per packageFragment)	13	4.333	1.247	6	/CoffeeMaker/acctest/fixtures/edu/ncsu/...	
+ Specialization Index (avg/max per type)		0.128	0.397	1.5	/CoffeeMaker/unittests/edu/ncsu/csc326...	
+ Instability (avg/max per packageFragment)		1	0	1	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Number of Attributes (avg/max per type)	39	3	1.961	6	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Number of Packages	3					
+ Method Lines of Code (avg/max per method)	520	4.228	8.917	64	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	editRecipe
+ Weighted methods per Class (avg/max per type)	203	15.615	13.006	50	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Number of Overridden Methods (avg/max per type)	3	0.231	0.421	1	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Number of Static Attributes (avg/max per type)	5	0.385	1.077	4	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Nested Block Depth (avg/max per method)		1.301	0.698	5	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	editRecipe
+ Number of Methods (avg/max per type)	113	8.692	4.794	16	/CoffeeMaker/acctest/fixtures/edu/ncsu/...	
+ Lack of Cohesion of Methods (avg/max per type)		0.475	0.28	0.833	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ McCabe Cyclomatic Complexity (avg/max per metho		1.65	1.687	12	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	editRecipe
+ Number of Parameters (avg/max per method)		0.472	0.616	4	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	addInventory
+ Abstractness (avg/max per packageFragment)		0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Number of Interfaces (avg/max per packageFragme	0	0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Efferent Coupling (avg/max per packageFragment)		3	2.449	6	/CoffeeMaker/acctest/fixtures/edu/ncsu/...	
+ Number of Children (avg/max per type)	0	0	0	0	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	
+ Depth of Inheritance Tree (avg/max per type)		2.385	0.923	3	/CoffeeMaker/unittests/edu/ncsu/csc326...	

Voilà ce à quoi correspondent les différentes colonnes des résultats :

- **Metric** : La métrique concernée
- **Total** : Indique le résultat total pour cette métrique, par exemple le nombre total de lignes de code
- **Mean** : Indique la moyenne. Vous pouvez voir dans le nom de la métrique sur quels éléments est calculée la moyenne, par exemple pour "Methods Lines of Code" : "avg/max per method" veut dire que la moyenne sera la moyenne de lignes de code par méthode.
- **Std. Dev** : Indique l'écart type entre la moyenne et le maximum
- **Maximum** : Indique la valeur maximale atteinte par notre projet pour cette métrique
- **Ressource causing Maximum** : Indique la classe ou le package qui cause le maximum pour cette métrique
- **Method** : Indique la méthode qui cause le maximum pour cette métrique. Elle est souvent pas employée

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Static Methods (avg/max per type)	10	0.769	2.665	10	/CoffeeMaker/src/edu/ncsu/csc326/coffe...	

# LES DIFFÉRENTES MÉTRIQUES

<http://baptiste-wicht.developpez.com/tutoriels>

- **Lines of Code (LOC)**: Le nombre total de lignes de code. Les lignes blanches et les commentaires ne sont pas comptabilisés
- **Number of Static Methods (NSM)**: Le nombre de méthodes statiques dans l'élément sélectionné.
- **Afferent Coupling (CA)**: Le nombre de classes hors d'un package qui dépendent d'une classe dans le package
- **Normalized Distance (RMD)**: RMA + RMI - 1: Ce nombre devrait être petit, proche de zéro pour indiquer une bonne conception.
- **Number of Classes (NOC)**: Le nombre de classes dans l'élément sélectionné.
- **Specialization Index (SIX)**: NORM \* DIT / NOM: Moyenne de l'index de spécialisation.
- **Instability (RMI)**: CE / (CA + CE) : Ce nombre vous donnera l'instabilité de votre projet. C'est-à-dire les dépendances entre les paquets.
- **Number of Attributes (NOF)**: Le nombre de variables dans l'élément sélectionné.
- **Number of Packages (NOP)**: Le nombre de packages dans l'élément sélectionné.
- **Method Lines of Code (MLOC)**: Le nombre total de lignes de codes dans les méthodes. Les lignes blanches et les commentaires ne sont pas comptabilisés
- **Weighted Methods per Class (WMC)**: La somme de la complexité cyclomatique de McCabe pour toutes les méthodes de la classe.
- **Number of Overridden Methods (NORM)**: Le nombre de méthodes redéfinies dans l'élément sélectionné.
- **Number of Static Attributes (NSF)**: Le nombre de variables statiques dans l'élément sélectionné.
- **Nested Block Depth (NBD)**: La profondeur du code
- **Number of Methods (NOM)**: Le nombre de méthodes dans l'élément sélectionné.
- **Lack of Cohesion of Methods (LCOM)**: Une mesure de la cohésion d'une classe. Plus le nombre est petit est plus la classe est cohérente, un nombre proche de un indique que la classe pourrait être découpée en sous-classe. Néanmoins, dans le cas de Javabean, cette métrique n'est pas très correcte, car les getteurs et les setteurs sont utilisés comme seules méthodes d'accès aux attributs. Le résultat est calculé avec la méthode d' Henderson-Sellers : on prend m(A), le nombre de méthodes accédant à un attribut A, on calcule la moyenne de m(A) pour tous les attributs, on soustrait le nombre de méthodes m et on divise par (1-m).
- **McCabe Cyclomatic Complexity (VG)**: La complexité cyclomatique d'une méthode. C'est-à-dire le nombre de chemins possibles à l'intérieur d'une méthode, le nombre de chemin est incrémenté par chaque boucle, condition, opérateur ternaire, ... Il ne faut pas que ce nombre soit trop grand pour ne pas compliquer les tests et la compréhensibilité de la méthode.
- **Number of Parameters (PAR)**: Le nombre de paramètres dans l'élément sélectionné.
- **Abstractness (RMA)**: Le nombre de classes abstraites et d'interfaces divisés par le nombre total de classes dans un package. Cela vous donne donc le pourcentage de classes abstraites par package
- **Number of Interfaces (NOI)**: Le nombre d'interfaces dans l'élément sélectionné.
- **Efferent Coupling (CE)**: Le nombre de classes dans un packages qui dépendent d'une classe d'un autre package.
- **Number of Children (NSC)**: Le nombre total de sous-classes directes d'une classe
- **Depth of Inheritance Tree (DIT)**: Distance jusqu'à la classe Object dans la hiérarchie d'héritage.