

I402A Architecture logicielle

Séance 4

Métrique et évaluation de code

Sébastien Combéfis

2017–2018



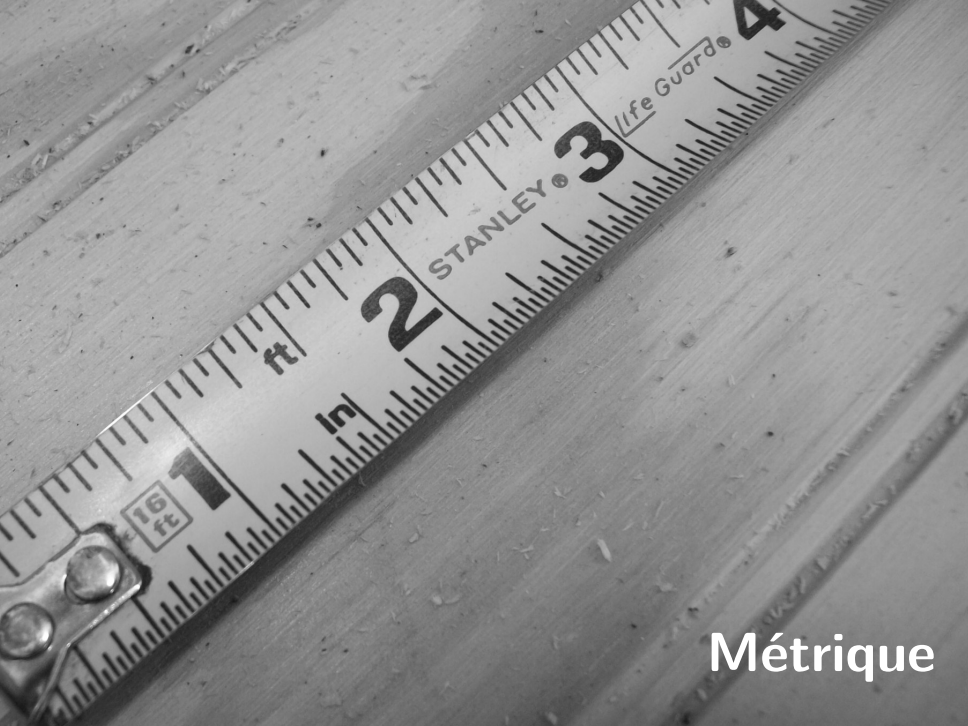
Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Rappels

- Identification et classification des **bad smells**
 - Défaut de design dans un code fonctionnel
 - Techniques de refactoring de code
- Notion d'**anti-pattern**
 - Développement, architecture et gestion de projet
 - Quelques exemples d'anti-patterns et solutions possibles

Objectifs

- Notions de **métrique** et évaluation de propriétés
 - Complexité software de Halstead
 - Complexité cyclomatique de McCabe
 - Complexité Fan-in Fan-out de Henry et Kafura
- **Métriques standards** pour évaluer du code
 - Présentation de plusieurs métriques et propriétés à évaluer
 - Cas particulier des systèmes orientés objets



Métrique

Métrique (1)

- **Mesurer** un critère pour mieux le comprendre



Expression sous la forme d'une valeur, pour évaluer, comparer...

*"When you can **measure** what you are speaking about and express it in **numbers**, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind : it may be the beginnings of knowledge but you have scarcely in your thoughts advanced to the stage of Science." — Lord Kelvin (physicien)*

Métrique (2)

- Mesurer dans le but de pouvoir **contrôler**

Évaluer et améliorer la qualité en fonction de la mesure

*"You cannot **control** what you cannot measure." — Tom DeMarco
(software engineer)*

- Pas évident de bien **déterminer** ce qu'on veut mesurer

Importance du choix des mesures et du critère à évaluer


"In truth, a good case could be made that if your knowledge is meagre and unsatisfactory, the last thing in the world you should do is make measurements; the chance is negligible that you will measure the right things accidentally." — George Miller (psychologist)

- Attribuer nombre à attribut d'une entité du monde réel

Description des entités à l'aide de règles non ambiguës

- Possibilité de mesurer des produits ou processus

Une classe, un module ou la documentation, les tests...

Entité	Exemples d'attribut
Design	Nombre de défauts détectés par une review
Spécification	Nombre de pages 
Code	Nombre de lignes de code, nombre d'opérations
Équipe de développement	Taille de l'équipe, expérience moyenne de l'équipe

Type de métrique (1)

- Mesure **directe** d'une propriété/d'un critère

Nombre de lignes de code, nombre de classes...

- Mesure **indirecte** ou dérivée à partir d'autres mesures

Densité de défauts = nombre de défauts / taille du produit

- **Prédiction** basée sur des mesures

Effort requis pour développer un software

Prédiction

- Utilisation d'un **modèle** de prédiction de variables

Relation entre variables prédites et variables mesurables

- Trois **hypothèses** pour qu'une variable soit prédictible

- 1 On peut mesurer des propriétés d'un logiciel avec exactitude
- 2 Il y a une relation entre ce qu'on veut et ce qu'on sait mesurer
- 3 Relation comprise, validée, exprimable comme modèle/formule

- Peu de métriques sont **prédictibles** en pratique

Difficulté d'établir un modèle précis

Type de métrique (2)

- Plusieurs **types de valeurs** pour une métrique
 - **Nominal** est un label sans ordre
Langage programmation : 3GL, 4GL
 - **Ordinal** avec ordre mais pas comparaison quantitative
Aptitudes programmeur : basse, moyenne, haute
 - **Intervalle** entre des valeurs
Aptitudes programmeur : entre 55^e et 75^e percentiles population
 - **Ratio** de proportionnalité pour comparer
Logiciel : deux fois plus gros que le précédent
 - **Absolu** avec simplement une valeur
Logiciel : 350000 lignes de code

Entité mesurée

- Mesure d'un **produit** concret, typiquement un logiciel

Critères de taille, complexité ou qualité du produit

- **Autres entités** mesurables liées au développement

Critères sur un processus, une ressource ou un projet

Complexité software de Halstead (1)

- Mesure de la complexité software par **Halstead** en 1977

Sur base de l'implémentation effective d'un programme

- Un programme est une séquence d'**opérateurs et opérandes**

- η_1, η_2 nombre d'opérateurs/opérandes uniques



- N_1, N_2 nombre total d'opérateurs/opérandes



"A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands" — Maurice Halstead

Complexité software de Halstead (2)

- Plusieurs **propriétés** calculables sur un logiciel

Basées sur les valeurs mesurées η_1, η_2, N_1 et N_2

- **Volume d'information** du programme mesuré en bits

Taille de n'importe quelle implémentation de l'algorithme

- Plusieurs mesures de la **difficulté et des efforts**

- Difficulté ou propension à faire des erreurs
- Effort pour implémenter ou pour comprendre un algorithme

Complexité software de Halstead (3)

Propriété	Formule
Vocabulaire	$\eta = \eta_1 + \eta_2$
Longueur	$N = N_1 + N_2$
Volume (bits)	$V = N \times \log_2 \eta$
Difficulté	$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
Effort (discrimination mentale élémentaire)	$E = D \times V = \frac{\eta_1 N_2 \log_2 \eta}{2\eta_2}$
Temps d'implémentation (secondes)	$T = \frac{E}{S} = \frac{\eta_1 N_2 N \log_2 \eta}{2\eta_2 S}$
Nombre de bugs	$B = \frac{E^{2/3}}{3000}$ ou $B = \frac{V}{3000}$

- $20 \leq V(\text{fonction}) \leq 1000$ et $100 \leq V(\text{fichier}) \leq 8000$
- Difficulté à cause de nouveau opérateur et d'opérandes répétés
- S est le nombre de Stoud valant 18 pour un informaticien

Complexité software de Halstead (4)

■ Avantages

- Pas besoin d'analyse poussée du flux de contrôle du programme
- Prédications de l'effort, taux d'erreur et temps d'implémentation
- Donne des mesures de qualité globales

■ Inconvénients

- Dépend de l'utilisation d'opérateurs et opérandes dans le code
- Pas de prédiction au niveau du design d'un programme

Exemple

```
1 main()
2 {
3     int a, b, c, avg;
4     scanf("%d %d %d", &a, &b, &c);
5     avg = (a + b + c) / 3;
6     printf("avg = %d", avg);
7 }
```

- **Opérateurs uniques** ($\eta_1 = 10$) : main () {} int scanf & = + / printf
- **Opérandes uniques** ($\eta_2 = 7$) : a b c avg "%d %d %d" 3 "avg = %d"
- **Vocabulaire et longueur** : $\eta = 10 + 7 = 17$ et $N = 16 + 15 = 31$
- **Volume, difficulté et effort** : $V = 126.7$ bits, $D = 10.7$ et $E = 1355.7$
- **Temps d'implémentation** : $T = 75.4$ secondes
- **Nombre de bugs** : $B = 0.04$

Complexité cyclomatique de McCabe (1)

- Mesure du nombre d'instructions de décision

Beaucoup de choix possibles implique une plus grande complexité

- Modèle basé sur un graphe représentant les décisions

Instructions if-else, do-while, repeat-until, switch-case, goto...

- Complexité cyclomatique calculée sur le graphe de flux

$$V(G) = e - n + 2$$

Avec nombre d'arêtes (e) et nombre de nœuds (n)

Complexité cyclomatique de McCabe (2)

- Plusieurs **variantes** possibles selon ce qui est mesuré
 - **Complexité cyclomatique ($V(G)$)**
Nombre de chemins linéaires indépendants
 - **Complexité cyclomatique réelle (ac)**
Nombre de chemins indépendants traversés par les tests
 - **Complexité du design du module ($IV(G)$)**
Pattern d'appels d'un module vers d'autres
- Dans l'idéal, les deux premières doivent **correspondre**
$$V(G) = ac$$

Complexité cyclomatique de McCabe (3)

■ Avantages

- Métrique permettant d'évaluer la facilité de maintenance
- Identifie les meilleures zones où les tests seront importants
- Facile à calculer et mettre en œuvre

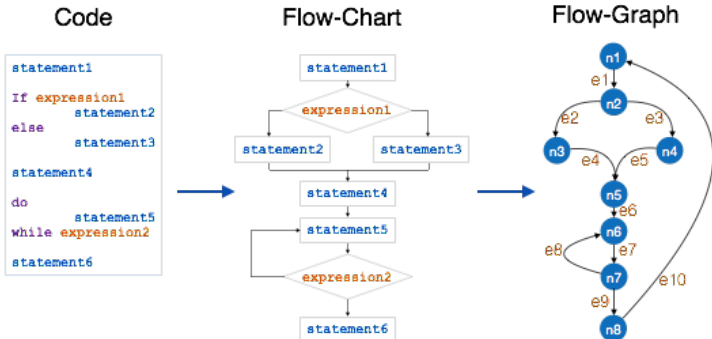
■ Inconvénients

- N'évalue pas la complexité des données, que du contrôle
- Même poids pour les boucles, imbriquées ou non

Exemple

- Identifier blocs délimités par des instructions de décision

Construction du graphe avec nœuds et arêtes



$$V(G) = e - n + 2 = 10 - 8 + 2 = 4$$

Complexité Fan-in Fan-out (1)

- Prise en compte du **flux de données** (Henry et Kafura)

Nombre de flux de données et structures de données globales

- Utilise une **longueur** comme SLOC ou CC de McCabe

$$HK = Length \times (Fan_{in} \times Fan_{out})^2$$

Avec information locale entrant (Fan_{in}) et sortant (Fan_{out})

- Variante par **Shepperd** sans multiplier par une longueur

$$S = (Fan_{in} \times Fan_{out})^2$$

Complexité Fan-in Fan-out (2)

- **Flux d'information** de la procédure A vers B
 - A appelle B
 - B appelle A et utilise sa valeur de retour
 - A et B appelés par C , qui passe la valeur de retour de A à B
- Définition des **flux de données** entrant et sortant
 - Fan_{in} = procédures appelées par celle-ci + paramètres lus + variables globales accédées
 - Fan_{out} = procédures appelant celle-ci + paramètres de sortie + variables globales écrites

Complexité Fan-in Fan-out (3)

■ Avantages

- Peut être évaluée avant même d'avoir l'implémentation
- Prend en compte les programmes contrôlés par les données

■ Inconvénients

- Complexité nulle pour procédure sans interaction externe

Exemple

```
1 char * strncat(char *ret, const char *s2, size_t n) {
2     char *s1 = ret;
3     if (n > 0) {
4         while (*s1)
5             s1++;
6         while (*s1++ = *s2++) {
7             if (--n == 0) {
8                 *s1 = '\\0';
9                 break;
10            }
11        }
12    }
13    return ret;
14 }
```

- Entrées ($fan_{in} = 3$)
- Sorties ($fan_{out} = 1$)
- Complexité Fan-in Fan-out non pondérée : $S = 3^2 = 9$
- Complexité Fan-in Fan-out pondérée : $HK = 10 \times 9 = 90$

Mesure de la modularité

- Évaluation du **couplage et de la cohésion** de modules
 - Fan_{in} de M compte modules qui appellent fonctions de M
 - Fan_{out} de M compte modules appelés par M
- Modules avec un **Fan_{in} nul** suspects
 - Code mort
 - En dehors des frontières du système
 - Approximations de la notion d'appel n'est pas assez précise

Métrique de complexité

- Trois principales **métriques de complexité**
 - Mesure de l'effort avec Halstead
 - Mesure de la structure avec McCabe
 - Mesure du flux d'information avec Henry et Kafura/Shepperd
- Métriques développées pour **langages impératifs**
 - Peut néanmoins être utilisé en orienté objet*



METRE

Métrique standard de code

Complexité cyclomatique

- **Complexité cyclomatique** d'une fonction ou méthode

Nombre de chemins linéaires dans une fonction

- Pas toujours mesurable par **outils d'analyse statique**

Estimation avec nombre d'instructions if, while, for...

- Ne devrait pas dépasser une **valeur de 10** en moyenne

- Diminution de la lisibilité et compréhension par d'autres
- Moins d'information de debugging, stack trace moins précise
- Tests unitaires plus complexes et moins efficaces

Ligne de code source

- Nombre de **lignes de code source** (SLOC)
 - Lignes « physiques » présentes directement dans le fichier
 - Lignes « logiques » effectivement exécutées
- **Classification** selon Boehm à interpréter avec précaution

Small (S) : 2 KLOC, Intermediate (I) : 8, Medium (M) : 32
Large (L) : 128, Very Large (VL) : 512
- Métrique à utiliser avec **grande précaution**

Ne mesure pas l'effort de production, ni la valeur d'un logiciel

Densité des commentaires

- **Densité des commentaires** (DC) par rapport aux lignes de code

$$DC = SLOC / CLOC \text{ (comment line of code)}$$

- Nombre de lignes de commentaire ne définit pas **leur qualité**

Entre 20% et 40% semble normal, en dehors suspect

- **Précautions** similaires qu'avec SLOC à prendre

En plus, code bien rédigé est sa propre documentation

Couverture de code

- Proportion de code source **couverte par des tests**

Nombre d'instructions, méthodes, classes... parcourues

- Généralement des **tests automatisés**, mais aussi des manuels

Couvre les tests unitaires, fonctionnels et de validation

- Diminution de la probabilité d'**erreurs d'exécution** ou bugs

Plus facile à évoluer, maintenir, refactorer grâce aux tests

Duplication de code

- Répétition de **code similaire** ou identique dans un code source

Violation claire du principe DRY (don't repeat yourself)

- **Quatre types** principaux de duplication de code

Imposée, par inadvertance, par impatience et interdéveloppeur

- Diminution de la **maintenabilité** d'un code

Augmente le risque d'introduction de bugs

Couplage

- Le **couplage** mesure les liens qui existent entre des classes
 - **Afferent** (Ca) nombre de références vers classe mesurée
Uniquement les références externes
 - **Efferent** (Ce) nombre de types que la classe connaît
Héritage, implémentation, paramètre, variable, exception...
- Classe **beaucoup référencée** (afferent) est importante
Impact grand d'une modification, mais bonne réutilisation
- Non respect de **responsabilité unique** si grand efferent
Instabilité potentiellement élevée avec augmentation dépendances

Instabilité

- **Instabilité** d'un module mesure sa résistance au changement

Un module stable est difficile à changer

- Se mesure par le **rapport** couplage efferent et couplage total

$$\text{Instabilité} = \frac{C_e}{C_e + C_a}$$

- Code de qualité **facile à modifier** grâce à instabilité

Très stable de 0,0 à 0,3 ou très instable de 0,7 à 1,0

Abstractness

- Niveau d'abstraction par rapport aux autres classes

Rapport entre types abstraits internes et autres types internes

- Abstraction élevée recommandée dans classe très stable

Module complètement concret (1,0) ou abstrait (1,0)

- Métrique généralement pas utilisée seule

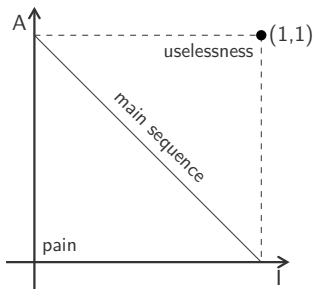
Est souvent combinée avec instabilité, par exemple

Distance from main sequence (1)

- Équilibre d'un module entre abstractness et instabilité

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

- Mesure **non normalisée** en divisant le résultat par $\sqrt{2}$



Distance from main sequence (2)

- Droite oblique **main sequence** donne les bonnes situations
 - Classe très stable, mais abstraite
 - Classe très instable, mais contrainte
- Le plus **proche possible** de la main sequence lorsque $D \rightarrow 0$
Valeur supérieure à 0,7 peut indiquer un problème

Lack of Cohesion of Methods (1)

- **Manque de cohésion** des méthodes (LCOM)

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

*Avec nombre de méthodes (M), nombre de champs d'instance (F),
nombre de méthodes appelant un champ donné (MF)*

- Une classe de devrait avoir qu'**une seule raison de changer**
 - Permet d'évaluer le principe de responsabilité unique
 - Diminution cohésion si peu en commun dans fonctionnalités
- **Cohésion forte** méthodes et champs d'instance se référencent
LCOM > 0,8 et F, M > 10 suspect

Lack of Cohesion of Methods (2)

- Autre mesure du manque de cohésion des méthodes

$$LCOM\ HS = \left(M - \frac{\sum_F MF}{F} \right) \times (M - 1)$$

*Avec nombre de méthodes (M), nombre de champs d'instance (F),
nombre de méthodes appelant un champ donné (MF)*

- Variante de Henderson-Sellers simplifiée et normalisée

Valeur supérieure à 1 est suspecte

Relational cohesion

- Nombre moyen de **relations internes** à un module

$$H = \frac{R + 1}{N}$$

*Avec nombre de références internes au module (R),
nombre total de types contenu dans le module (NF)*

- Grande valeur indique classes d'un module en **forte relation**

Valeur typiquement entre 1.5 et 4.0

Taille de l'instance

- Mesure la **quantité de mémoire** utilisée pour une instance

Nombre d'octets mémoire pour stocker un objet instancié

- **Somme des tailles** des champs de la classe et des hérités

Peut être calculé de manière statique

- Gros objets peuvent **dégrader les performances**

Valeur maximale conseillée de 64

Indice de spécialisation

- Degré de spécialisation d'une classe

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

*Avec nombre de méthodes redéfinies (NORM),
nombre total de méthode (NOM),
profondeur de l'arbre d'héritage (DIT)*

- Augmentation de l'indice avec **profondeur et redéfinitions**

Valeur supérieure à 1.5 suspecte

Nombre d'éléments

- Compte d'un **nombre** d'éléments dans une classe
 - **Paramètres** d'une méthode à limiter à 5
 - **Variables** déclarées dans une méthode à limiter à 8
 - **Surcharges** de méthodes à limiter à 6
- Simplification en utilisant des **données structurées**
Classe, structure, tuples...

Violation de règles de codage

- Compter nombre de **règles de codage** violées

Règles souvent spécifiques à un langage de programmation

- Règles couvrant **plusieurs catégories** de critères évalués

Maintenabilité, fiabilité, efficacité, portabilité, utilisabilité

- Certaines **alertes** ne représentent pas forcément un bug

Pas forcément nécessaire de vouloir toutes les corriger

Code de qualité (1)

■ Valeurs optimales à atteindre pour les métriques standards

Il s'agit de valeurs moyennes en dehors desquelles c'est suspect

Métrique	Valeur optimale
Complexité cyclomatique (CC)	10
Ligne de code source (SLOC)	> 20 difficile à comprendre, > 40 complexe
Densité des commentaires	entre 20%–40%
Couverture de code	100%
Duplication de code	0%
Distance from main sequence (D)	< 0.7
Lack of cohesion of methods (LCOM)	< 0.8 (avec $F, M < 10$)
Lack of cohesion of methods (LCOM HM)	< 1

Code de qualité (2)

- **Valeurs optimales** à atteindre pour les métriques standards

Il s'agit de valeurs moyennes en dehors desquelles c'est suspect

Métrique	Valeur optimale
Relational cohesion (H)	entre 1.5–4.0
Taille de l'instance (octets)	< 64
Indice de spécialisation	< 1.5
Paramètres d'une méthode	≤ 5
Variables locales à une méthode	≤ 8
Versions surchargées d'une méthode	≤ 6

Voir <http://www.ndepend.com/docs/code-metrics>

Analyse statique de code

- **Analyse statique** de code à l'aide d'un analyseur syntaxique

Facile à réaliser car tout langage possède un tel analyseur

- Plusieurs **problèmes** par rapport à la mesure de la qualité
 - Pas souvent d'intuition réelle pour la plupart des métriques
 - Ignore environnement, domaine d'application, algorithmes particuliers, utilisateurs...
 - Facile à contourner par un code obscur



Systeme orienté objet

Système Orienté Objet (1)

- Méthodes pondérées par classe

$$WMC = \sum_{i=1}^n c_i$$

Avec méthodes M_1, \dots, M_n de complexités c_1, \dots, c_n

- Profondeur de l'arbre d'héritage d'une classe
 - *DIT* la maximale en cas d'héritage multiple
 - Réutilisabilité et maintenabilité compliquées si grand *DIT*

Système Orienté Objet (2)

- **Nombre d'enfants** (directs) d'une classe

Doit être minimisé sans quoi le design est considéré mauvais

- **Couplage entre classes** lors d'appels de méthodes/variables

- Un design encapsulé donnera un petit *CBO*
- Une classe indépendante est facile à tester et réutiliser

- **Réponse d'une classe** à des sollicitations externes

Nb de méthodes pouvant être exécutées en réponse à un message

Système Orienté Objet (3)

- **Nombre de variables** par classe (NVC)

Nombre moyen de variables publiques et privées par classe

- **Nombre de paramètres** par méthode (APM)

Nombre de paramètres divisé par nombre de méthodes (< 0.7)

- **Nombre d'objets** (NOO)

Nombre d'objets extraits du code source

Métriques de MOOD

- Métriques proposées par l'équipe du **projet MOOD** en 1994

Sous la direction de Abreau

- Niveau du **système complet** pour mesurer plusieurs aspects
 - **Encapsulation** avec facteur de dissimulation méthode, attribut
 - **Héritage** avec facteur d'héritage méthode, attribut
 - **Polymorphisme** avec facteur polymorphisme
 - **Couplage** avec facteur couplage

Encapsulation (1)

- Mesure de l'**encapsulation** des variables et des méthodes

$$MHF = \frac{\sum_{i=1}^M (1 - V(M_i))}{M}$$

Avec M méthodes de visibilité $V(M_i)$ chacune

- Mesure de la **visibilité** d'une méthode

$$V(M_i) = \frac{\#\{C_j \mid \text{classe } C_j \text{ peut appeler } M_i \text{ et } M_i \text{ pas dans } C_j\}}{C - 1}$$

Avec C classes dans tout le système

Encapsulation (2)

- MHF de 100% si toutes les méthodes sont **privées**

Tend vers 0% lorsque le nombre de méthodes publiques augmente

- **Cacher les méthodes** est une bonne pratique

- Augmentation réutilisabilité et diminution complexité
- Un MHF bas indique implémentation pas assez abstraite
- Un MHF haut reflète un faible nombre de fonctionnalités

- **Augmenter MHF** diminue densité de bugs et augmente qualité

Valeurs acceptables situées entre 8% et 25%

- Facteur d'héritage des méthodes d'une classe

$$MIF = \frac{\sum_{i=1}^C Mi(C_i)}{\sum_{i=1}^C Ma(C_i)}$$

Avec $Mi(C_i)$ nombre de méthodes héritées par C_i et non redéfinies,

Avec $Ma(C_i)$ nombre total de méthodes dans C_i

- Valeurs acceptables pour facteurs d'héritage méthode/attribut
 - Entre 20% et 80% pour MIF
 - AIF devrait être entre 0% et 48%

Polymorphisme

- **Facteur de polymorphisme** basé sur la redéfinition

$$PF = \frac{\sum_{i=1}^C Mo(C_i)}{\sum_{i=1}^C (Mn(C_i) \times DC(C_i))}$$

*Avec $Mo(C_i)$ nombre de méthodes redéfinies dans C_i ,
 $Mn(C_i)$ nombre de nouvelles méthodes définies dans C_i ,
et nombre de descendants DC de la classe C_i*

- Mesure indirecte de la **liaison dynamique** dans un système

Opportunités pour la redéfinition $Mn(C_i) \times DC(C_i)$

Couplage

- A est couplé à B si A appelle méthode/variable dans B

Ne prend pas en compte le couplage par héritage

- **Couplage** d'une classe avec une autre

$$CF = \frac{\sum_{i=1}^C \sum_{j=1}^C is_client(C_i, C_j)}{C(C-1)}$$

$$\text{Avec } is_client(A, B) = \begin{cases} 1 & \text{si } A \neq B \text{ et } A \text{ est couplé à } B \\ 0 & \text{sinon} \end{cases}$$

- Augmenter CF augmente **densité de défauts**

Effort de rework pour trouver et corriger défaut augmente

Crédits

- https://www.flickr.com/photos/mad_house_photography/4311409835
- https://www.tutorialspoint.com/software_engineering/images/cyclomatic_complexity.png
- <https://www.flickr.com/photos/alainbachellier/125739388>
- <https://www.flickr.com/photos/nomadcrocheter/5334037661>