Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Centre Universitaire de Mila

Département Math & Informatique

Génie logiciel II

CHAPITRE 4
Métriques

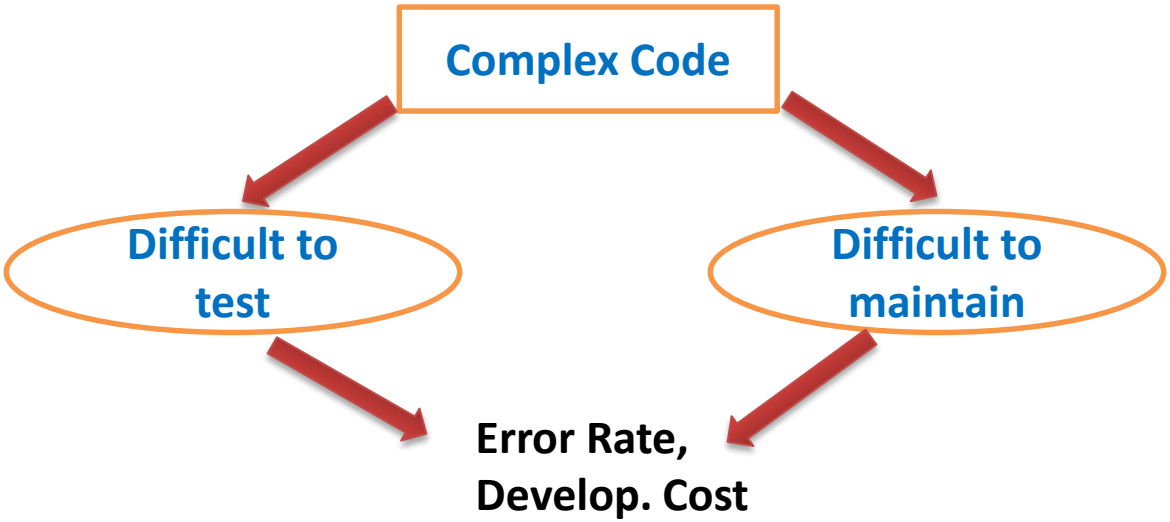
HEDJAZ Sabrine

2019-2020

Métriques

1. Introduction

- La complexité de code a une influence directe sur la qualité et le coût d'un logiciel.
- Elle impacte sur la durée de vie et l'exploitation d'un logiciel, et plus particulièrement sur son taux de défauts, sa testabilité et sa maintenabilité.
- Une bonne compréhension et maîtrise de la complexité d'un code permet de développer un logiciel de meilleure qualité.



Métriques

1. Introduction

- Pour garantir un logiciel de qualité tout en assurant des coûts de test et de maintenance faibles, la complexité d'un logiciel devrait être mesuré dès le début du codage. Ainsi lorsque les valeurs recommandées sont dépassées, le développeur peut intervenir rapidement.
- Pour quantifier la complexité d'un logiciel, on se sert de métriques.
- Les métriques peuvent être classées en trois catégories :
  - ✓ Mesurant le **processus de développement**
  - ✓ Mesurant **des ressources**
  - ✓ Evaluation du **produit logiciel**
- Les métriques de produits mesurent les qualités du logiciel. Parmi ces métriques, on distingue les métriques traditionnelles et les métriques orientées objet.

Métriques

1. Introduction

- Métriques Orienté Objet: prennent en considération les relations entre éléments de programme (classes, méthodes) comme **métriques MOOD, métriques Objet de Chidamber et Kemerer**.
- Métriques traditionnelles: se divisent en deux groupes : les métriques mesurant la **taille et la complexité**, et les métriques mesurant **la structure du logiciel**.
  - ✓ Les métriques mesurant taille et complexité les plus connus sont les métriques **de ligne de code** ainsi que les **métriques de Halstead, …**
  - ✓ Les métriques mesurant la structure d'un logiciel comme la complexité **cyclomatique** de **McCabe** se basent sur des organigrammes de traitement ou des structures de classe.

Métriques

2. Nombre cyclomatique de Mc Cabe: v(G)

- **La complexité Cyclomatique (complexité de McCabe),** introduite par Thomas **McCabe** en 1976, est le calcul le plus largement répandu des métriques statiques, conçue dans le but d'être indépendante du langage.
- La métrique de McCabe **indique le nombre de chemins linéaires indépendants** dans un module de programme et représente finalement la complexité des flux de donnés. Il correspond au **nombre de branches conditionnelles dans l'organigramme d'un programme.**
- **Le nombre cyclomatique** évalue le nombre de chemins d'exécution dans la fonction et ainsi donne une indication sur **l'effort nécessaire pour les tests du logiciel.**
- Pour un programme qui consiste en seulement des états séquentiels, la valeur pour v(G) est 1.

Métriques

2. Nombre cyclomatique de Mc Cabe: v(G)

- **Mac Cabe** étudier le logiciel en analysant le graphe de contrôle du programme et calcule la complexité structurelle ou nombre cyclomatique de ce graphe.

- Soit:

  n = Nombre de nœuds (blocs d'instructions séquentielles)

  e = Nombre d'arcs (branches suivies par le programme)

  v = nombre cyclomatique

- Calcul du nombre cyclomatique:

  Cas n° 1: 1 point d'entrée; 1 point de sortie

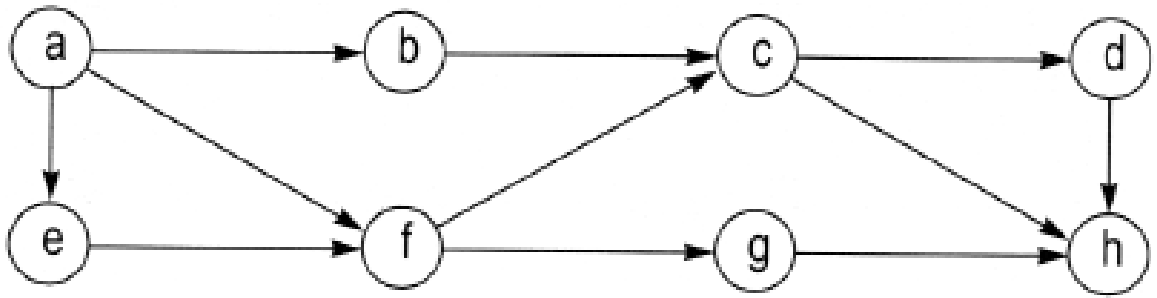
  v= e-n +2

  Cas n° 2 : i points d'entrée; s points de sortie

  v = e - n + i + s

Métriques

2.1. Exemple:



- Calcul du nombre cyclomatique:

  Cas n° 1: 1 point d'entrée={a} 1 point de sortie={h}

  E=11

  N=8

  v= e-n +2 = 11-8+2= 5

Métriques

2.2.Graphe de flot de contrôle (GFC)

- Dans un GFC, **les sommets du graphe** représentent **un bloc de base**, c'est-à-dire un bout de code d'un seul tenant sans sauts ni cibles de sauts. **Les cibles de sauts** marquent **le début d'un bloc** de base, tandis que **les sauts** en marquent **la fin**.

- La plupart des représentations d'un CFG comprennent deux blocs spéciaux : le **bloc d'entrée**, par lequel on entre dans le graphe de flot de contrôle et le **bloc de sortie**, par lequel on le quitte.

- Les arcs représentent les sauts dans le flot de contrôle.

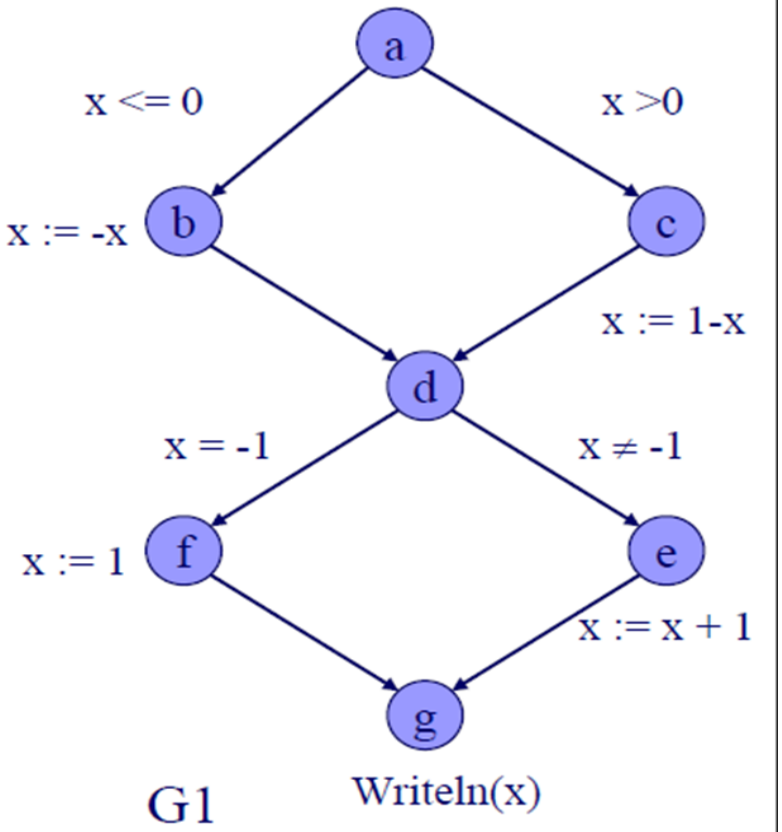
Métriques

2.3. Exemple:

- Soit le programme P1 suivant :

```
if x <= 0 then x := -x
else x := 1 - x;
if x = -1 then x=1
else x := x+1;
writeln(x)
```

Ce programme admet le graphe de contrôle G1.



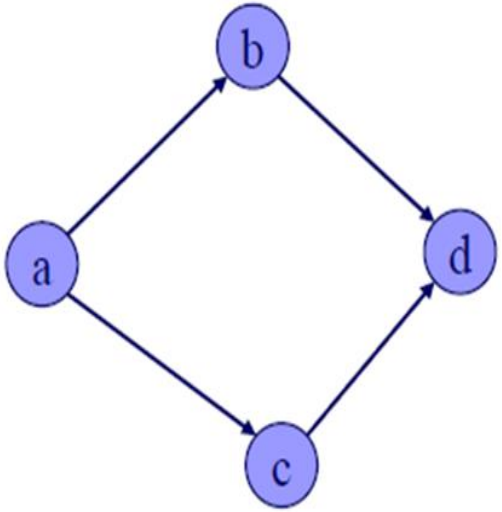
G1

v= e-n +2 = 8-7+2=3

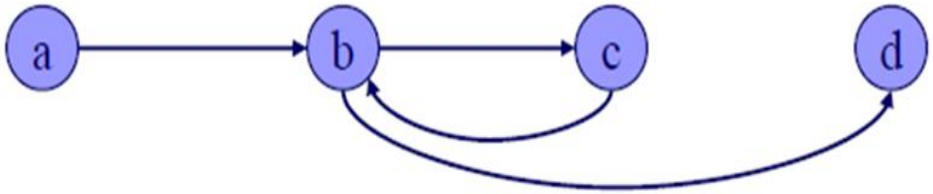
Métriques

2.4. Les différentes formes:

Forme séquentielle



Forme alternative



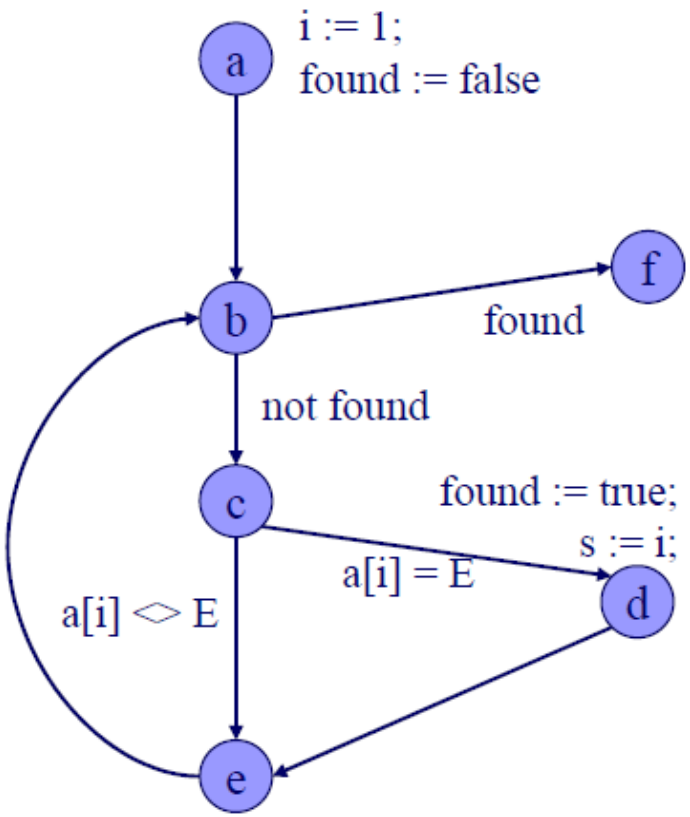
Forme itérative

Métriques

2.4.1. Exemple:

- Soit le programme P2 suivant :

```
i := 1;
found := false;
while (not found) do
      begin
      if (a[i] = E) then
         begin
         found := true;
         s := i;
         end;
      i := i + 1;
      end;
```



G2 = ab [c (1 + d) eb]* f

G2

Métriques

2.5.Graphe de flot de contrôle (GFC)

- Plus le nombre **cyclomatique est grand**, plus il y aura **de chemins d'exécution dans la fonction**, et plus elle sera **difficile à comprendre et à tester**.

- Le principe de base est que chaque fonction devrait avoir un **nombre de cas de tests au moins égal au nombre cyclomatique**, pour que tous les chemins soient **couverts au moins une fois**.

- Les constructions de langage suivants incrémentent le nombre cyclomatique : **if (...), for (...), while (...), case ...:, catch (...) ...etc.** Le calcul commence toujours avec la **valeur 1**. A ceci on ajoute le nombre de nouvelles branches.

Métriques

2.6.Calcule directe du nombre Mc Cabe

- Produire un graphe de contrôle et l'analyser peut s'avérer long dans le cas de programmes complexes.

- Mc Cabe a introduit une nouvelle manière de calculer la complexité structurelle.

$$C = \pi + 1$$

- avec π le nombre de décisions du code
  - ✓ Une instruction **IF** compte pour une décision
  - ✓ Une boucle **FOR** ou **WHILE** compte pour une décision
  - ✓ Une instruction **CASE** ou tout autre embranchement multiple **compte pour une décision de moins** que le nombre d'alternatives.

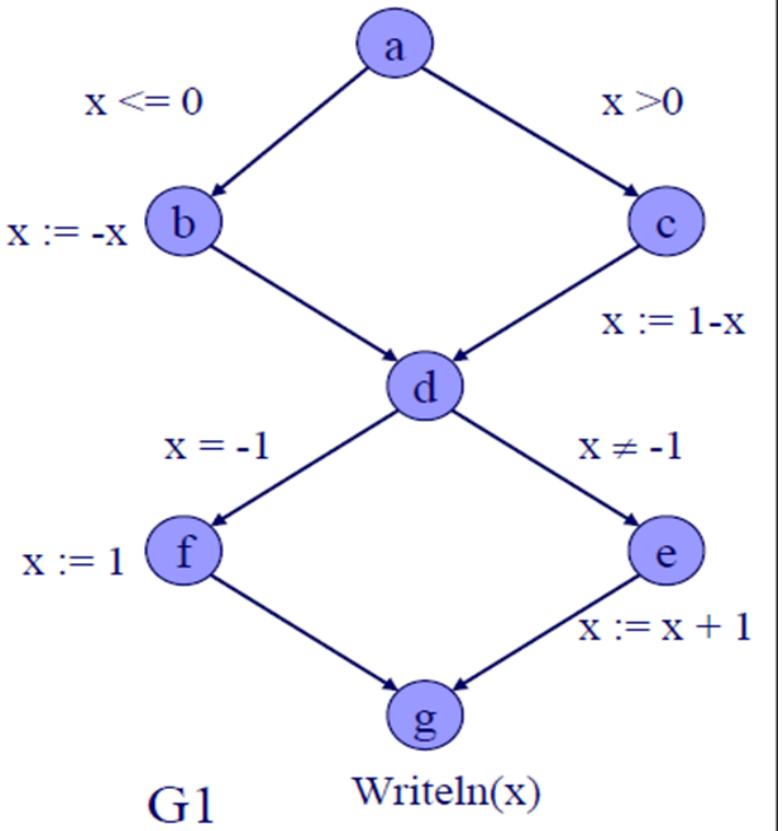
Métriques

2.7. Exemple:

- Soit le programme P1 suivant :

```
if x <= 0 then x := -x
else x := 1 - x;
if x = -1 then x=1
else x := x+1;
writeln(x)
```

Ce programme admet le graphe de contrôle G1.



G1

C = π + 1= 2+1=3

Métriques

2.8.Limites acceptable pour le nombre cyclomatique

- Une fonction devrait avoir une nombre cyclomatique inférieur à 15. Si une fonction a plus que **15** chemins d'exécution il est difficile à comprendre et à tester.

- Pour un fichier le nombre cyclomatique ne devrait pas dépasser **100**.

- La valeur maximum du nombre cyclomatique peut être définie comme un critère de qualité dans le plan qualité.

Métriques

3. Les Métriques de Halstead

- Les métriques de complexité de **Halstead** qui procurent une mesure quantitative de complexité ont été introduit par l'américain Maurice **Halstead**. Ils sont basées sur l'interprétation du code comme une séquence de marqueurs, classifiés comme un opérateur ou une opérande.

- Toutes les métriques de **Halstead** sont dérivées du nombre d'opérateurs et d'opérandes :
  - ✓ Nombre des opérandes distincts (n2) **(jetons qui contiennent une valeur)**
  - ✓ Nombre total des opérandes (N2)
  - ✓ Nombre des opérateurs distincts (n1) **(virgules, parenthèses, opérateurs arithmétiques…)**
  - ✓ Nombre total des opérateurs (N1)

Métriques

3. Les Métriques de Halstead

- Sur la base de ces chiffres on calcule :

  La Longueur du programme (N) : N = N1 + N2

  La Taille du vocabulaire (n) : n = n1 + n2

- A partir de là, on obtient le **Volume du Programme (V):**

  V = N * log2(n)

- **Le volume d'une fonction** devrait être compris entre 20 et 1000. Le volume d'une fonction, d'une ligne et sans paramètre, qui n'est pas vide est d'environ 20. Une fonction avec un volume supérieur à 1000 comporte probablement trop de choses. **Le volume d'un fichier** devrait être au minimum à 100 et au maximum à 8000.

Métriques

3. Les Métriques de Halstead

- **Le Niveau de difficulté (D)** ou **propension d'erreurs** du programme est proportionnel au **nombre d'opérateurs distincts (n1)** dans le programme et dépend également du **nombre total d'opérandes (N2)** et du **nombre d'opérandes distincts (n2)**. Si les mêmes opérandes sont utilisés plusieurs fois dans le programme, il est plus enclin aux erreurs.

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

- **Le Niveau de programme (L)** est l'inverse du Niveau de difficulté. Un programme de bas niveau est plus enclin aux erreurs qu'un programme de haut niveau.

$$L = 1 / D$$

Métriques

3. Les Métriques de Halstead

- **L'Effort à l'implémentation (E)** est proportionnel au volume (V) et au niveau de difficulté (D). Cette métrique est obtenu par la formule suivante en emd (elementary mental discrimination):

$$E = V * D$$

- Halstead a découvert que diviser l'effort par **18** donne une approximation pour **le Temps pour implémenter** (T) un programme **en secondes**.

$$T = E / 18$$

- Il est même possible d'obtenir le **« nombre de bugs fournis » (B)** qui une estimation du nombre d'erreurs dans le programme. Cette valeur donne une indication pour le nombre d'erreurs qui devrait être trouver lors du test de logiciel. Le « nombre de bugs fournis » est calculé selon la formule suivante:

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

Métriques

3.1. Exemple1:

```
z = 0;
while x > 0
    z = z + y;
    x = x - 1;
end-while
print (z);
```

- Opérateurs :

  = ; while/end-while > + -

  print ()

  $\eta_1 = 8$

- Opérandes : = z 0 x y 1

  $\eta_2 = 5$

Métriques

3.2. Exemple2:

```
z = 0;
while x > 0
    z = z + y;
    x = x - 1;
end-while
print (z);
```

Opérateurs		Opérands	
=	3	z	4
;	4	0	2
w/ew	1	x	3
>	1	y	1
+	1	1	1
-	1		
print	1		
()	1		

N1= 13; N2= 11; N= 24

Métriques

4. Flux d'information d'henry kifura

- Cette métrique introduite par **Henry et Kafura** en 1981, prend en compte les liaisons **inter modules**. La relation de base est qu'un module appelle un autre module, ou est appelé par un autre module. Elle propose de calculer pour chaque module sa complexité à partir de sa taille « length », son **« fan in »** et son **« fan out ».**

- ✓ **« fan in » d'un module:** nombre de modules qui appellent un module
- ✓ **« fan out » d'un module:** nombre de modules appelé par un module

$$\textbf{Complexité du module} = \textbf{Length} * (\textbf{fan in} * \textbf{fan out})^2$$

Où **length** est la longueur du module en terme de milliers de lignes de codes KLOC.

Métriques

5. Métriques Objet de Chidamber et Kemerer

- ✓ Ensemble de métriques (Metric Suite for Object Oriented Design)
  - Evaluation des classes d'un système
  - La plupart des métriques sont calculées classe par classe
  - Le passage au global n'est pas clair, une moyenne n‘étant pas très satisfaisante

Métriques

5. 1. M1. Méthodes pondérées par classe

- WMC : Weighted Methods per Class

$$WMC = \frac{1}{n} \times \sum_{i=0}^{n} c_i \ (Mi)$$

avec C un ensemble de n classes comportant chacune M_i méthodes dont la complexité (le poids) est noté c_i

5. 1. M2. Profondeur de l'arbre d'héritage

- DIC : Depth of Inheritance Tree
  - Distance maximale entre un nœud et la racine de l'arbre d'héritage de la classe concernée
  - Calculée pour chaque classe

Métriques

5. 3. M3. Nombre d'enfants

- NOC : Number Of Children
  - Nombre de sous-classes dépendant immédiatement d'une classe donnée, par une relation d'héritage
  - Calculée pour chaque classe

5. 4. M4. Couplage entre classe

- Dans un contexte OO, le couplage est l'utilisation de méthodes ou d'attibuts d'une autre classe.
  - Deux classes sont couplées si les méthode déclarées dans l'une utilisent des méthodes ou instancie des variables définies dans l'autre
  - Le relation est symétrique : si la classe A est couplée à B, alors B l'est à A
- CBO : Coupling Between Object classes
  - Pour chaque classe, nombre de classes couplées
  - Calculée pour chaque classe

Métriques

5. 5. M5. Références à une classe

- {RS} : ensemble des méthodes qui peuvent être exécutées en réponse à un message reçu par un objet de la classe considérée
  - Réunion de toutes les méthodes de la classe avec toutes les méthodes appelées directement par celles-ci
  - Calculée pour chaque classe

$$RFG = |RS|$$

Où « **RS** » représente l'ensemble des méthodes impliquées en réponse à un message reçu par la classe; il est défini par la formule suivante:

$$RS = \{M\} \cup all \ \{R_i\}$$

Tels que M représente l'ensemble de toutes les méthodes invoquées dans la classe **C**, R_i est l'ensemble des méthodes dans les autres classes qui sont appelées par une méthode « **i** » de la classe **C**.

Métriques

5. 6. M6. Manque de cohésion des méthodes

- Un module (ou une classe) est « cohésif » losque tous ses éléments sont étroitement liés
- LCOM (Lack of COhesion in Methods) tente de mesurer l'absence de ce facteur
- Posons
  - I_i l'ensemble des variabes d'instance utilisées par la méthode i
  - P l'ensemble les paires de (I_i, I_j) ayant une intersection vide
  - Q l'ensemble des paires de (I_i, I_j) ayant une intersection non vide

$$LCOM = \max(|P| - |Q|, 0)$$

Métriques

5. 6. M6. Manque de cohésion des méthodes

$$LCOM = \max(|P| - |Q|, 0)$$

- LCOM peut être visualisé comme un graphe bi-partite
  - Le premier ensemble de nœuds corespond aux n différents attributs et le second aux m différentes méthodes
  - Un attribut est lié à une fonction si elle y accède ou modifie sa valeur
  - L'ensemble des arcs est Q
  - Il y a $n \times m$ arcs possibles, et $|P| = n \times m - |Q|$

Métriques

6. Métriques MOOD

- Ensemble de métriques pour mesuer les attributs des propriétés suivantes :
  - Encapsulation
  - Héritage
  - Couplage
  - Polymorphisme

Métriques

6.1. Encapsulation

- MHF : Method Hiding Factor (10-30%)

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

avec

- $M_d(C_i)$ le nombre de méthodes déclarées dans une classe C_i
- $M_h(C_i)$ le nombre de méthodes cachées
- TC le nombre total de classes.

Métriques

6.1. Encapsulation

- AHF : Attribute Hiding Factor (70-100%)

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

avec

- $A_d(C_i)$ le nombre d'attributs déclarés dans une classe C_i
- $A_h(C_i)$ le nombre d'atributs cachés

Métriques

6.2. Facteurs d'héritage

- MIF : Method Inheritance Factor (65-80%)

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

avec

- $M_i(C_i)$ le nombre de méthodes héritées (et non surchargées) de C_i
- $M_a(C_i)$ le nombre de méthodes qui peuvent être appelées depuis la classe i

Métriques

6.2. Facteurs d'héritage

- AIF : Attribute Inheritance Factor (50-60%)

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

avec

- $A_i(C_i)$ le nombre d'attributs hérités de C_i
- $A_a(C_i)$ le nombre d'attributs auxquels C_i peut accéder

Métriques

6.3. Facteur de couplage

- CF : Coupling Factor (5-30%)
  - Mesure le couplage entre les classes sans prendre en compte celui dû à l'héritage

$$CF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} client(C_i, C_j)}{TC^2 - TC}$$

avec

- $client(C_i, C_j) = 1$ si la classe i a une relation avec la classe j, et 0 sinon
- Les relations d'héritage ne sont pas prises en compte dans les relations

Métriques

6.4. Facteur de polymorphisme

- PF : Polymorphism Factor (3-10%)
  - Mesure le potentiel de polymorphisme d'un système

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} M_n(C_i) \times DC(C_i)}$$

avec

- $M_o(C_i)$ le nombre de méthodes surchargées dans la classe i
- $M_n(C_i)$ le nombre de nouvelles méthodes dans la classe i
- $DC(C_i)$ le nombre de descendants de la classe i

Conduite de projet (Gestion de projet)

Tests logiciels

Chapitre 5