Génie Logiciel
TD/TP1

Séverine Affeldt

MLDS - LIPADE
UFR Mathématiques et Informatique

Université Paris Descartes

Qualité et maintenance du code

La qualité d'un logiciel dépend notamment de (1) la **qualité perçue** par le client, (2) la **qualité du code** source et de (3) la facilité de **maintenance**.

Il existent différent **indicateurs** possibles pour mesurer la qualité d'un code source

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

Indice de spécialisation

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

- **NORM** nombre de méthodes redéfinies *(Number of Overriden Methods)*
- **DIT** profondeur dans l'arbre d'héritage *(Depth in Inheritance Tree)*
- **NOM** nombre de méthodes de la classe *(Number Of Methods)*

$\rightarrow$ moyenne par paquetages, ensemble de paquetages ou projet

---

Interprétation

Augmentation de l'indice	**Diminution** de l'indice
nbr. méthodes redéfinies augmente la profondeur d'héritage augmente	nbr. méthodes spécifiques à la classe augmente le nombre de méthodes redéfinies diminue

$\rightarrow$ indice de spécialisation $> 1.5 \Leftrightarrow$ mauvaise qualité

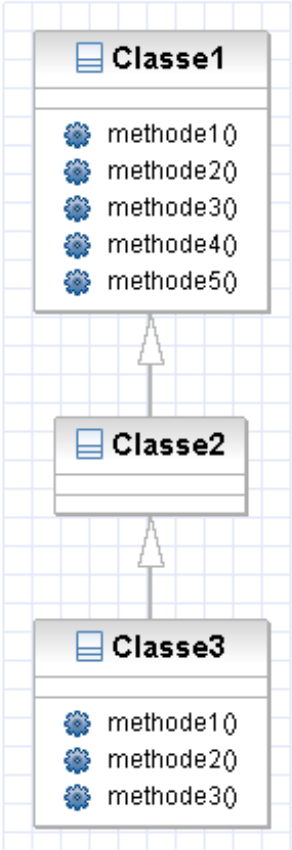
Action possible pour diminuer l'indice

Il faut penser à la **refactorisation** en utilisant des **interfaces**. Un indice de spécialisation est trop élevé lorsqu'un objet hérite d'un autre en redéfinissant beaucoup de méthodes.

Exercice

On considère les trois classes ci-dessous.

1. Donner la valeur de NORM, DIT et NOM[3] pour la classe 3.
2. Donner l'indice de spécialisation[4] de la classe 3.
3. Que peut-on conclure? Proposer une amélioration.



[3] NORM: nombre de méthodes redéfinies; DIT: profondeur dans l'arbre d'héritage; NOM: nombre de méthodes de la classe

[4] $(NORM \times DIT)/(NOM)$

Indice d'instabilité

$$\frac{C_e}{C_a + C_e}$$

- C_a Couplage afférent
  (nbr. classes en dehors du paquetage qui dépendent des classes de ce paquetage)
- C_e Couplage efférent
  (nbr. classes de de paquetage qui dépendent des classes en dehors de ce paquetage)

Interprétation

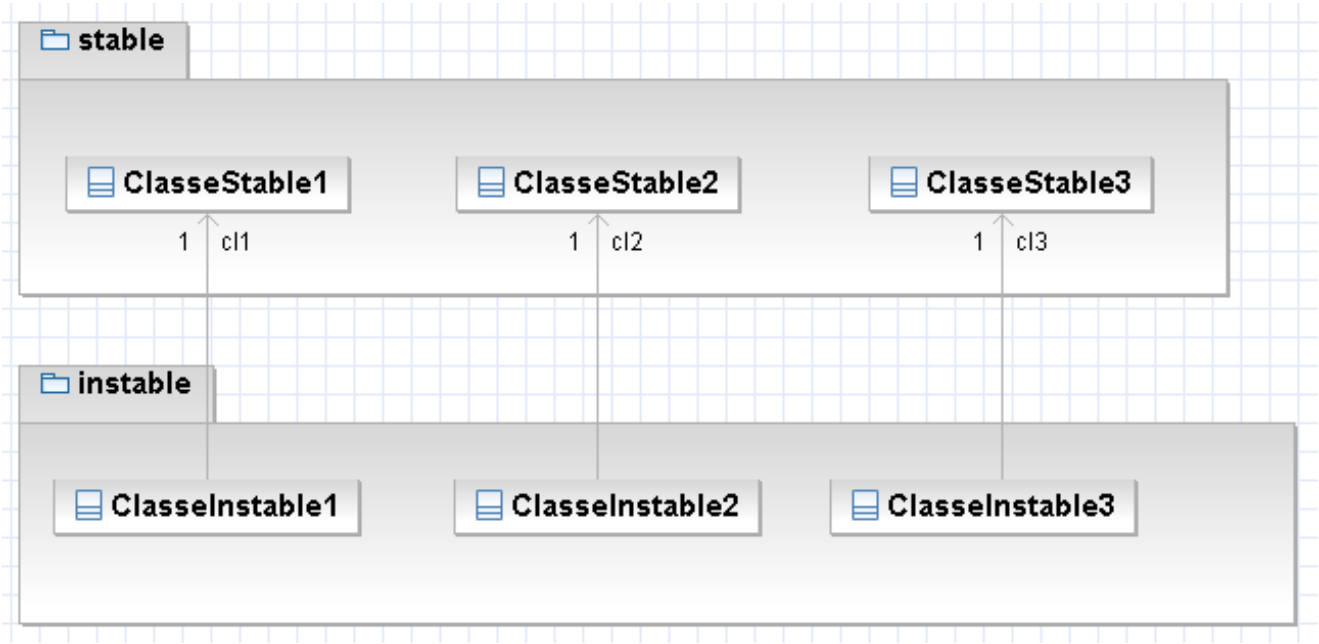
indice proche de 0	indice proche de 1
paquetage stable	paquetage potentiellement à risque

$\rightarrow$ l'indice d'**instabilité** indique les paquetages qui dépendent plus des autres que les autres ne dépendent d'eux. Ces paquetages sont instables car il y un plus grand risque de problèmes en raison de leur **forte dépendence**.

Exercice

On considère l'architecture ci-dessous.

1. Pour le paquetage instable, donner C_a, C_e et l'indice d'instabilité[3,4,5].
2. Pour le paquetage stable, donner C_a, C_e et l'indice d'instabilité.



[3] nbr. classes en dehors du paquetage qui dépendent des classes de ce paquetage

[4] nbr. classes de de paquetage qui dépendent des classes en dehors de ce paquetage

[5] $\frac{C_e}{C_a+C_e}$

Indice d'abstraction

$$\frac{I}{T}$$

- I Nombre d'interfaces et de classes abstraites du paquetage
- T Nombre total de type du paquetage

Interprétation

Cet indice est compris entre 0 et 1. Utilisé seul, il n'apporte pas beaucoup d'information car certains paquetage n'ont pas besoin ou on besoin d'un fort niveau d'abstraction. En revanche, l'indice d'abstraction est intéressant quand il est utilisé conjointement avec l'indice d'instabilité pour le calcul de la **distance from the main sequence**.

Distance from the main sequence

$$|Abstractness + Instability - 1|$$

- *Abstractness* niveau d'abstraction
- *Instability* indice d'instabilité

Interprétation

Cet indice est compris en 0 et 1, et représente l'équilibre entre le niveau d'abstraction et l'intabilité d'un paquetage.

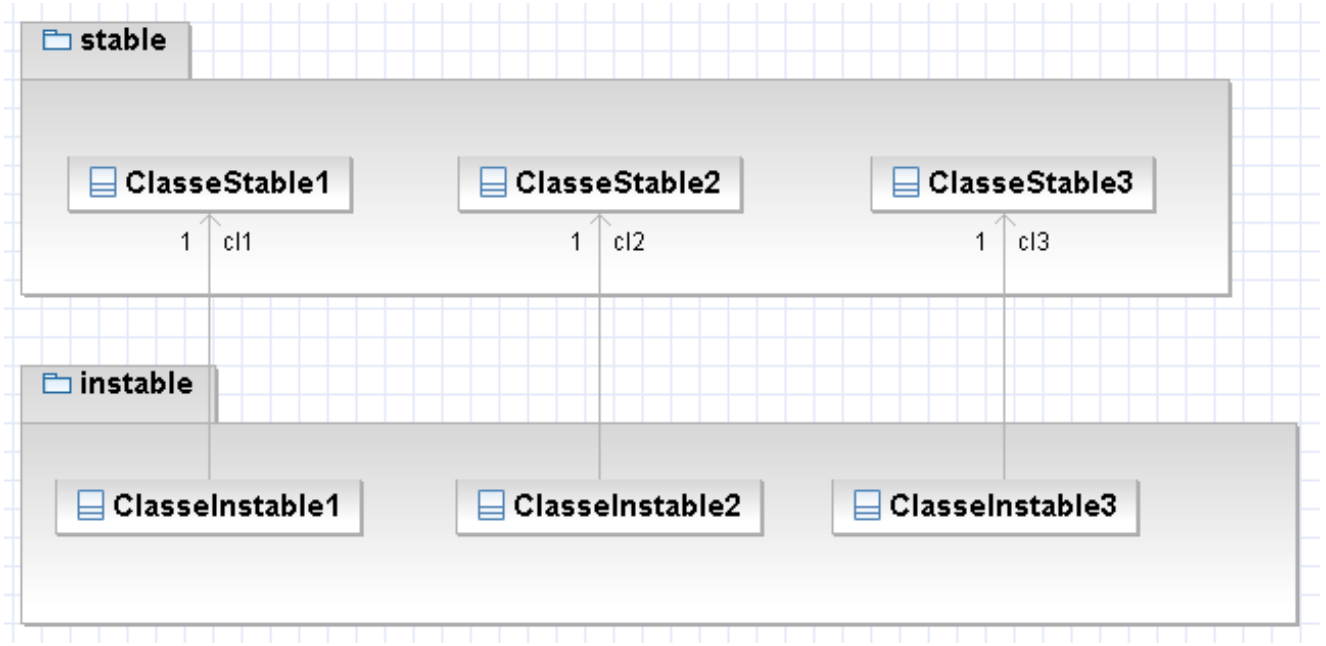
Quand cet indice vaut 1, le design du paquetage est considéré comme bon. La distance from the main sequence tend vers 0 lorsque (1) le paquetage est instable mais possède peu d'interfaces ou (2) beaucoup d'autres paquetages dépendent de ce paquetage mais celui-ci possède beaucoup d'interfaces.

$$\rightarrow \text{distance from the main sequence} > 0.5 \Rightarrow \text{refactorisation}$$

Exercice

On considère l'architecture ci-dessous.

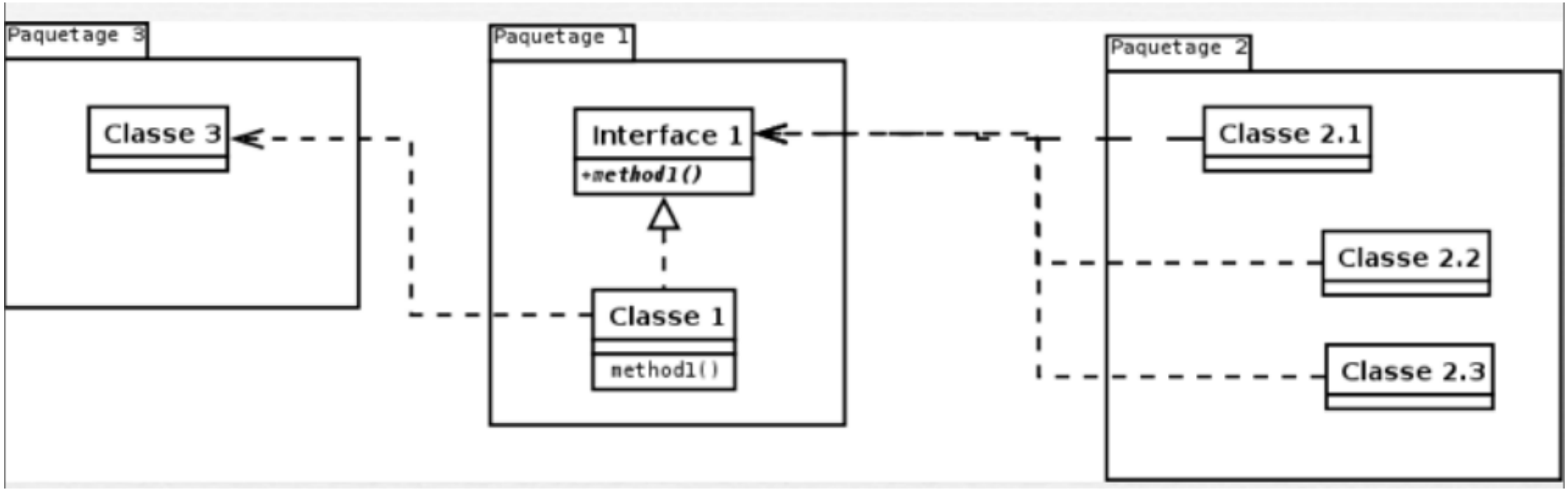
1. Pour le paquetage instable, donner *Instability*, *Abstracness* et la distance from the main sequence, $|Abstractness + Instability - 1|$.
2. Pour le paquetage stable, donner *Instability*, *Abstracness* et la distance from the main sequence, $|Abstractness + Instability - 1|$.
3. Est-ce cohérent?



Exercice

On considère l'architecture ci-dessous.

1. Pour les trois paquetages, donner *Instability*, *Abstracness* et la distance from the main sequence, $|Abstractness + Instability - 1|$.
2. Pour le paquetage 1, donner *Instability*, *Abstracness* et la distance from the main sequence, $|Abstractness + Instability - 1|$.
3. Donner la distance from the main sequence pour l'ensemble.



Complexité cyclomatique

Définition Nombre de **chemins** linéairement **indépendants** qu'il est possible d'emprunter dans cette méthode. Cette complexité correspond au nombre de points de décision de la méthode (`if, case, while...`) +1 (chemin principal).

Interprétation

Une méthode avec une forte complexité cyclomatique est plus difficile à comprendre et à maintenir.

- $\geq 30 \Rightarrow$ refactoriser la méthode
- $\leq 30 \Rightarrow$ acceptable si suffisamment de tests

$\rightarrow$ cette complexité est liée à la notion de couverture de code: un chemin $\Rightarrow$ un test unitaire

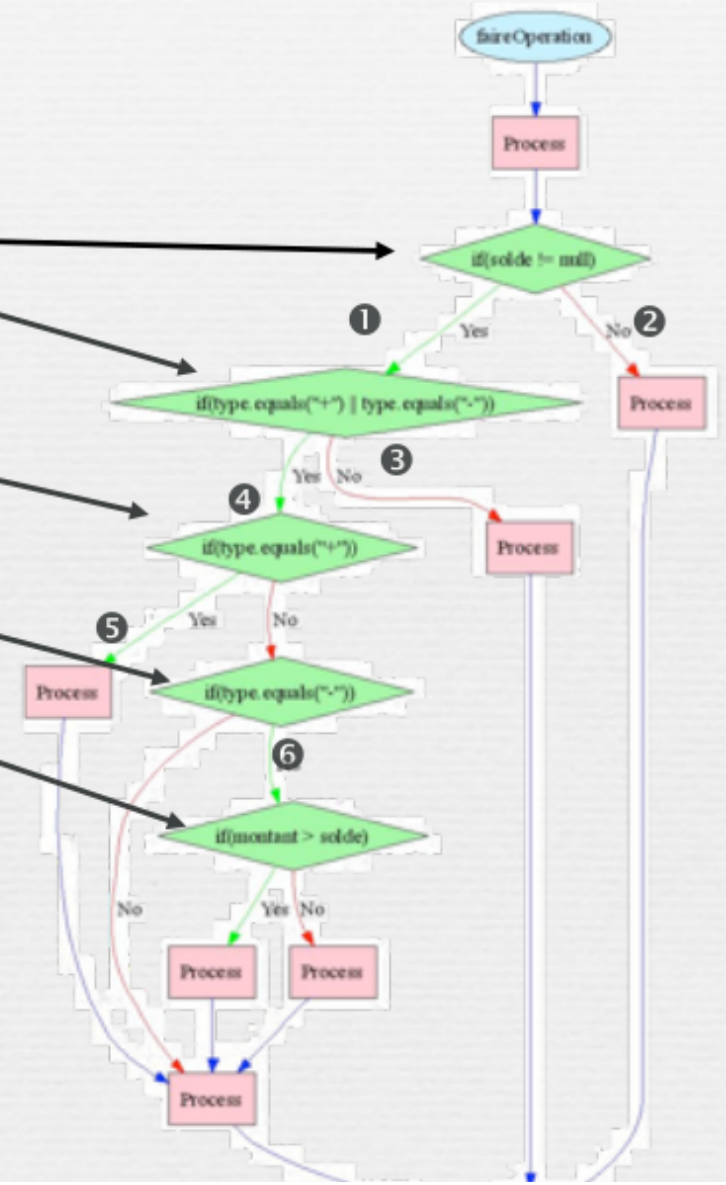
Exercice

Combien de tests unitaires faut-il pour la méthode ci-dessous?

```
package banque;


public class Banque {
   private Double solde;

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



Quelques outils de détection des risques

Il existe de nombreux outils permettant de calculer des statistiques et de détecter des comportements à risque au niveau du code source.

C'est particulièrement le cas en `Java` où on peut compter sur de très nombreux plugins `eclipse`: Cobertura, Crap4j, PMD, FindBugs, Eclipse Metrics, JDepend.

- **Crap4j** permet de détecter les classes et les méthodes à refactoriser rapidement en appliquant sur le code source un algorithme basé sur la complexité cyclomatique et la couverture par les tests.
- **PMD** analyse le code source à la recherche de section de codes connus pour poser problèmes. Ce sont des "anti-pattern", à l'inverse des *Design Patterns*. Les anti-pattern à rechercher sont configurables et il est possible de n'inclure que ceux qui sont intéressants pour un projet donné.
- **FindBugs** analyse le code source à la recherche de schémas de codes problématiques. La différence entre FindBugs et PMD tient essentiellement en la nature des problèmes détectés.
- **Eclipse Metrics** plugin eclipse qui analyse le code source pour calculer un certain nombre de métriques (dont celles vu aujourd'hui), pour un projet, un paquetage ou une classe. Il est également capable de présenter un graphe des dépendances entre paquetages en 3D.

Exercice

On considère le code source ci-dessous.

1. Analyser ce code. Que fait-il?
2. Quelles modifications peut-on apporter pour améliorer sa lisibilité?

```
#include<stdio.h>
#define TM 100
int main(void){
float x,v,c[TM];
int i,n;
float x0,s;
scanf("%d",&n);
for(i=0;i<=n;i++){scanf("%f",&c[i]);}
scanf("%f",&x);
v= c[0];x0=x;s=v;
for(i=1;i<=n;i++){s=s+c[i]*x0;x0=x0*x;}
v=s;printf("%f",v);
return(0);}
```

Exercice

```
#include<stdio.h>

compare(char a[TM],char b[TM]){
        int i;
        while((a[i]==b[i]) && ((a[i]!='\0') && (b[i]!='\0')))
        { I++;}

        if (a[i]==b[i]) return(0);
        if (a[i]=='\0' || b[i]=='\O') return(-2);
        if (a[i]>b[i]) return(1);

        return(-1);
}

int main(void){
        String a[TM],b[TM];
        int comp;
        printf("Saisissez un mot a et un mot b:");
        scanf("%s %s",&a,&b);
        comp=compare(a,b);

        switch (comp)
        {
                case 0: printf("%s et %s sont égaux! \n", a, b);
                case 1: printf("%s est lexico-graphiquement inferieure a %s\n",a,b);
                case (-1): printf("%s est lexico-graphiquement supérieure a %s\n",b,a);
                default: printf("Il y a inclusion!\n");
        }
}
```

Ce code source correspond à un programme qui utilise la fonction `compare(a,b)` permettant la comparaison de deux chaînes de caractères `a` et `b`. La fonction retourne une valeur négative, nulle ou positive selon que `a` est lexico-graphiquement inférieure, égale ou supérieure à `b`.

1. Analyser et corriger les erreurs qui se trouvent dans ce code.
2. Quelles modifications peut-on apporter pour améliorer sa lisibilité?

Exercice

Un client aimera disposer d'un programme qui lui permet de calculer le nombre de combinaisons possible de *p* objets parmi *n*. La fonction est donnée par la formule suivante :

$$C_n^p = \frac{n!}{p!\,(n-p)!}$$

Le client propose le test suivant (méthode main) pour vérifier le programme :

```
int main(void){
        int n,p;
        double c;
        printf("Calculons C(n,p)\n");
        printf("Entrez la valeur de n ");
        scanf("%d",&n);
        printf("Entrez la valeur de p ");
        scanf("%d",&p);
        c=factoriel(n)/(factoriel(p)*factoriel(n-p));
        printf(" C(%d,%d) vaut %lf", n,p,c);
        printf("\n");
        return(0);
}
```

Proposez une solution pour ce client.
Peut-on proposer une solution optimisée ?
Est-ce qu'on peut avoir une solution sans utiliser l'opération de multiplication* ?