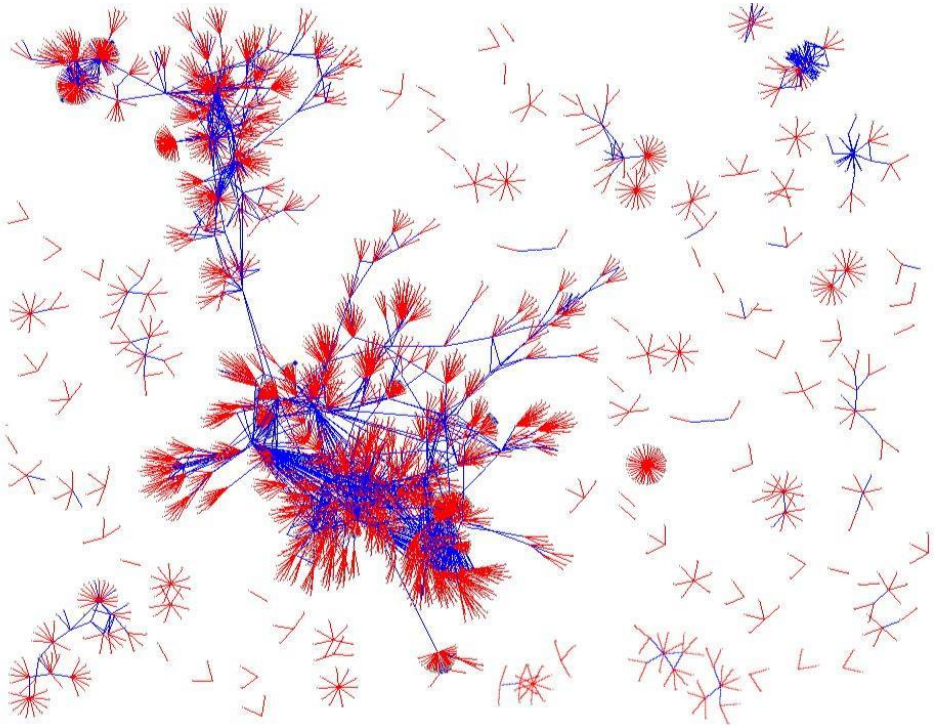


R5.A.04

Qualité Algorithmique

Chapitre2

Qualité et mesure du logiciel

1. Qualité d’une offre logicielle

Offre logicielle

- Offre : ensemble des **produits** livrables (documents et logiciels), des **services** et prestations associées (aide technique, formation, etc.) et des **caractéristiques contractuelles** (délais, coûts, etc.)

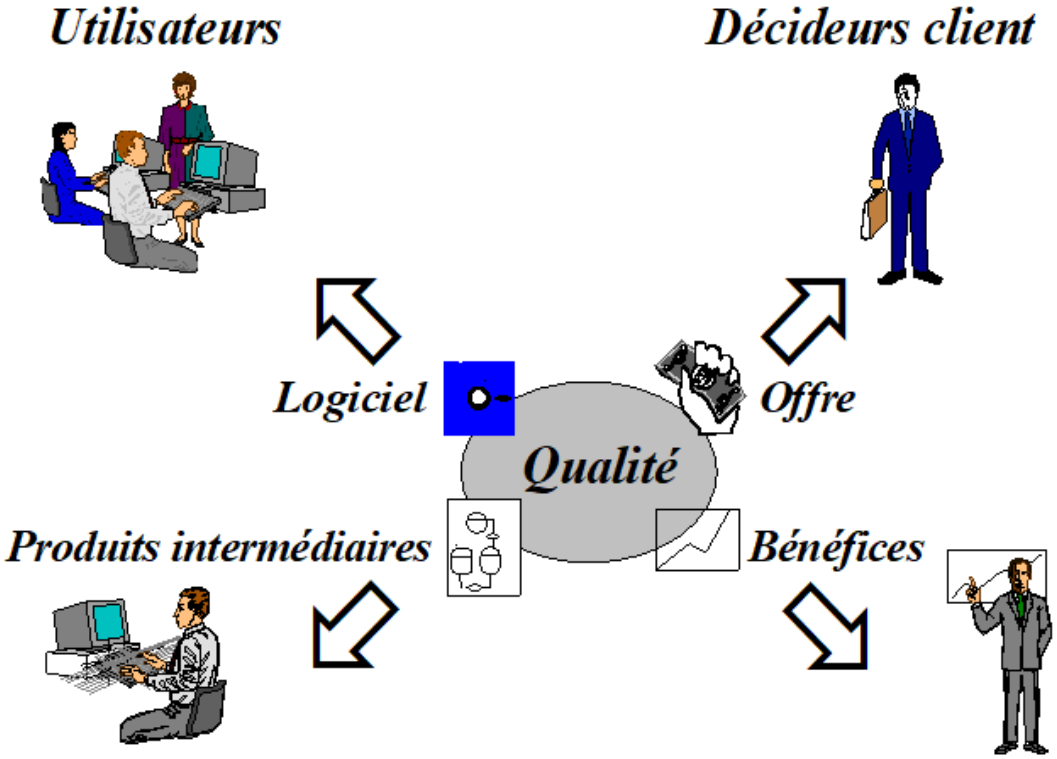
Qualité d'une offre logicielle
Difficulté de poser une définition

- La qualité n'est pas **absolue** : le vieux débat Mini et Rolls-Royce...
- La qualité est sujet à des **contraintes** : coût, ressource (personnel, outils, formation, etc.)
- La qualité est **multi-dimensionnelle** : *fiabilité*, *portabilité*, etc.
- Les critères de qualité ne sont pas **indépendants** : exemple *Maintenabilité* et *Performance*. La qualité est donc une affaire de compromis
- Certains critères de qualité sont **difficiles à mesurer** et ce sont parfois parmi les plus importants : exemple *convivialité.*
- Elle s'exprime et **s'évalue différemment selon le moment** du cycle de vie : documents informelles-semi-formels-formels, code source, exécutable.

En plus, la qualité est une question de point de vue

- Capacité fonctionnelle, fiabilité, rendement, facilité d'utilisation, maintenabilité, portabilité
- Prestations associées

- Coûts, délais, compétences, etc.
- Respect des clauses contractuelles, suivi de projet, satisfaction des utilisateurs, etc.



- Testabilité, lisibilité, etc.
- Maîtrise des coûts et des délais

- Satisfaction du client
- Parts de marché, productivité, etc.

Définition (très-trop) générale

- Les définitions les plus sûres sont les plus générales et les moins utiles…
  - ISO 8402 (Vocabulaire-Qualité): «Aptitude d'une offre à répondre aux besoins explicites ou implicites d'un client»

- **Une autre définition temporalisée**
  - **Avant-projet** : coûts et délais proposés, garanties qualité
  - **En cours de projet** : respect des jalons et des coûts intermédiaires, relations avec le client, etc.
  - **Après projet** : qualité des produits livrés et des prestations associées. Il faut proposer LE bon produit/service et UN bon produit/service

Les normes autour de la qualité

- Vocabulaire sur la qualité : ISO 8402
- **Qualité du produit logiciel**
  - Modèle de définition et d ’évaluation : ISO 25010 (ex ISO 9126)
  - Processus d ’évaluation pour achat ou interne : ISO 25040
- **Cycle de vie** : ISO 12207
- Politique qualité et **certificat** : ISO 9001 (9000-3)
- **Niveau de maturité** : CMMI, ISO 33001 (ex SPICE)

2. Qualité du produit logiciel selon ISO 25010

Principe de ISO/IEC 25010 (ex ISO 9126)

- Elle propose de spécifier et d ’évaluer selon des axes qualité appelés des caractéristiques.
- Ces caractéristiques sont éventuellement développées en sous-caractéristiques
- Elles sont scindées en deux catégories :
  - en contexte (« Quality in Use ») => chez les utilisateurs
  - Hors contexte (« External and Internal Quality ») => chez le fournisseur

La qualité en utilisation (chez les utilisateurs)

- 5 caractéristiques qui s'intéressent à des propriétés dont fait preuve le logiciel **dans son environnement réel** avec ses véritables utilisateurs
    1. **Effectiveness** : précision et complétude pour atteindre les buts recherchés
    2. **Efficiency** : gains constaté
    3. **Freedom from risk** : niveau de sécurité des hommes, du matériel, de l'environnement
    4. **Satisfaction** : niveau de satisfaction des utilisateurs (utilité, confiance, plaisir, confort)
    5. **Context coverage** : propriétés précédentes dans le cadre prévu d'utilisation et hors de ce cadre prévu.

La qualité hors contexte (chez le fournisseur)

Ces sont 8 caractéristiques du logiciel lui même

1 sur ce qu'il fait

+

7 sur comment il le fait

Ce qu'il fait

- **Capacité fonctionnelle** : aptitude du logiciel à fournir les fonctions attendues et répondant aux besoins exprimés ou implicites dans les conditions prévues d'utilisation.
  - **Complétude fonctionnelle** : couverture de l'ensemble des besoins des utilisateurs
  - **Exactitude fonctionnelle** : résultats corrects avec le niveau de précision souhaité
  - **Pertinence fonctionnelle** : approche de réalisation des tâches adaptée (pas de bruit, d'étapes inutiles)

Comment il le fait

- **Performance** : aptitude du logiciel le logiciel à procurer les performances appropriées, relativement aux temps et ressources utilisées, dans les conditions prévues d'utilisation.
- **Compatibilité** : aptitude à interopérer avec son envieonnement (d'autres logiciels)
- **Utilisabilité** : aptitude du logiciel à être compris, appris, utilisé et à motiver l'utilisateur quand il est utilisé dans les conditions d 'utilisation prévues.
- **Fiabilité** : aptitude du logiciel à fournir le niveau de performance requis dans les conditions d'utilisation prévues. Tolérance aux fautes, récupération.
- **Sécurité** : aptitude à garantir l'accès aux données et service en fonction de son niveau d'autorisation.
- **Maintenabilité**: aptitude du logiciel à être modifié.
- **Portabilité** : aptitude du logiciel à être porté sur d'autres plateformes.

Les sous caractéristiques hors contexte

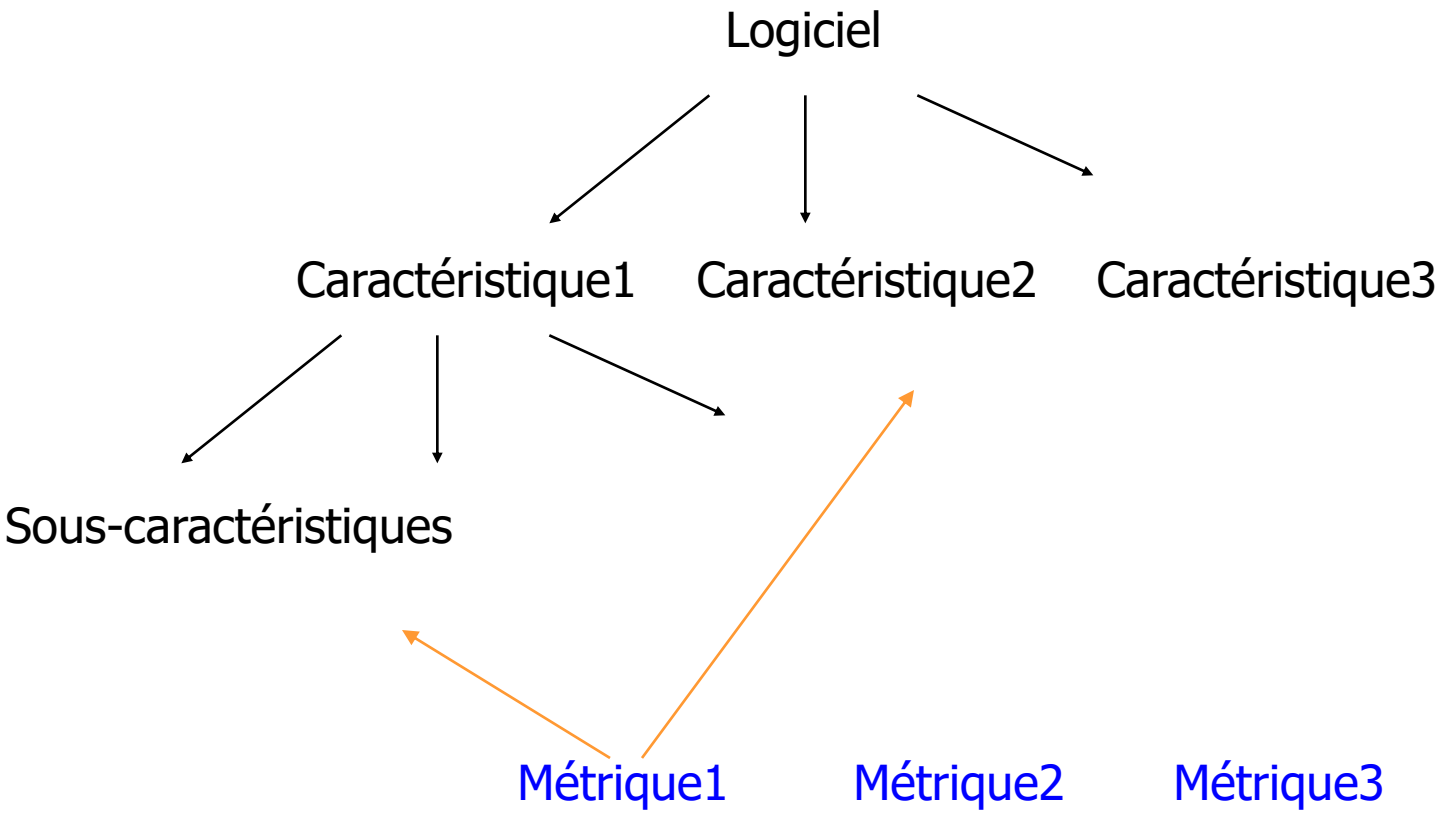
(Sub)Characteristic
Functional suitability
Functional completeness
Functional correctness
Functional appropriateness
Performance efficiency
Time behaviour
Resource utilization
Capacity
Compatibility
Co-existence
Interoperability
Usability
Appropriateness recognizability
Learnability
Operability
User error protection
User interface aesthetics
Accessibility

Reliability
Maturity
Availability
Fault tolerance
Recoverability
Security
Confidentiality
Integrity
Non-repudiation
Accountability
Authenticity
Maintainability
Modularity
Reusability
Analysability
Modifiability
Testability
Portability
Adaptability
Installability
Replaceability

Spécifier et évaluer la qualité

- Chacune des caractéristiques est ensuite spécifiée et évaluée en cours de développement en usant de métriques.
- Une métrique est une mesure quantitative sur un artefact (document, code) renvoyant une valeur.
- En spécification on détermine pour chaque métrique la plage des valeurs acceptables
- En évaluation on calcule la valeur de la métrique et on la compare à la plage souhaitée

Schéma de spécification & évaluation



Types de métriques

- Bien évidemment les métriques se distinguent par leur cible :
  - un artefact non exécutable : Internal Metric
  - le logiciel en exécution simulé : External Metric
  - le logiciel en utilisation réelle : In Use Metric

Assurer la qualité

- Une fois le niveau de qualité attendu spécifié il faut l'obtenir !
  - Intégrer dans le processus de développement des **langages**, guidés par des **méthodes** supportés par des **outils**
    - Faire les choses dans le respect des **bonnes pratiques**.
    - Vérifier tout au long du cycle de vie via des activités de **Vérification et Validation** l'efficacité de ce qui est fait. Lecture croisées, inspection, preuve, simulation, analyse de code et mesure, tests, etc.

3. Analyse de code et métriques

Fonctionnalités des outils d'analayse

- Les principaux outils d'analyse de code (généralement multi-langages vont permettre de calculer automatiquement des indicateurs quantitatifs pour vérifier le respect des objectifs de qualité pour le projet/l'entreprise.
  - Calcul de **métriques**
  - Détection de bad smells
  - Détection de points de vulnérabilité et d'attention sécurité
  - Détection d'erreurs de codage classiques
  - Vérification du respect de conventions de codage
  - Requêtage de code et écriture de règles de codage propriétaires
  - Etc.

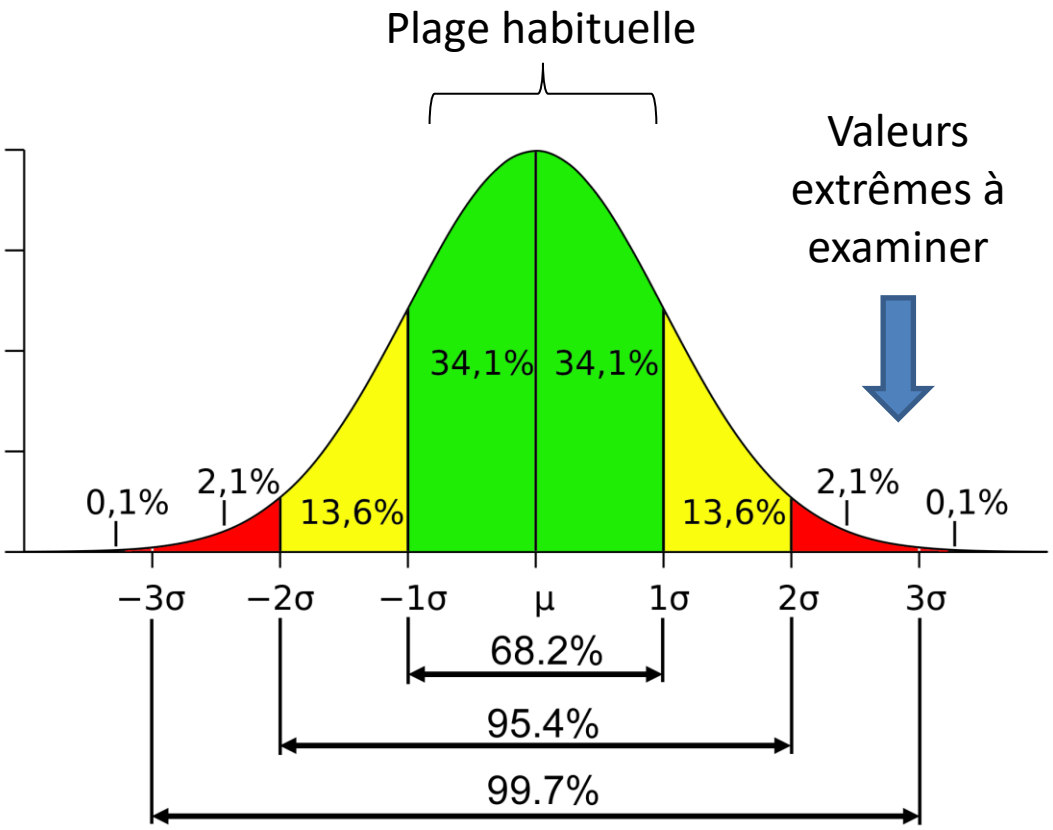
Cas particulier d'indicateurs : les métriques de code

- De nombreux travaux montrent que les attributs internes du produit logiciel ont un impact sur ses attributs externes.
  - Exemple : taille => nombre de bugs ou Complexité => maintenabilité
- Une « bonne » structure interne donne une bonne qualité du produit
- Agir sur la structure interne permet de contrôler la qualité externe du produit
- Des *attributs internes* peuvent être mesurés sur du code source.
- Grâce à la prédiction, des actions correctives peuvent être entreprises, réduisant ainsi les coûts de maintenance.
- Pour cette mesure au niveau du code il a été proposé plusieurs dizaines de « métriques ».
- Certains outils permettent de calculer de nombreuses métriques et d'afficher des tableaux de bord.
  - Exemple typique **JArchitect** (85 métriques) : https://www.jarchitect.com/

Difficultés pour utiliser les métriques de code

- Certaines métriques ont fait l'objet d'études empiriques (qualité de la valeur prédictive d'attributs autres ou externes) mais leur exploitation reste malgré toit délicate:
  - Uniformité parfois relative des formules de calcul entre outils.
  - Comment fixer une plage de tolérance pour une métrique dans des domaines applicatifs différents, des langages différents, des entreprises et équipes de développement différentes ?
  - Comment exploiter un usage hiérarchique d'une même métrique (exemple complexité méthode => classes => Packages => Application?).
  - Comment synthétiser dans un tableau de bord des métriques différentes (agréger par exemple la taille – complexité + cohésion + couplage?).
  - Quel niveau hiérarchique d'affichage et métriques de synthèse pour un tableau de bord « utile » au chef de projet, développeur, etc.?
  - Chaque entreprise développe son usage des métriques…

A minima : les métriques pour la détection des « anomalies »



Typologie des métriques

- Taille
- Complexité
- Couplage
- Cohésion
- *Profondeur d'héritage :* **Depth of Inheritance Tree (DIT), Number of Children (NOC)**
- *Niveau d'abstraction :* **Abstractness**

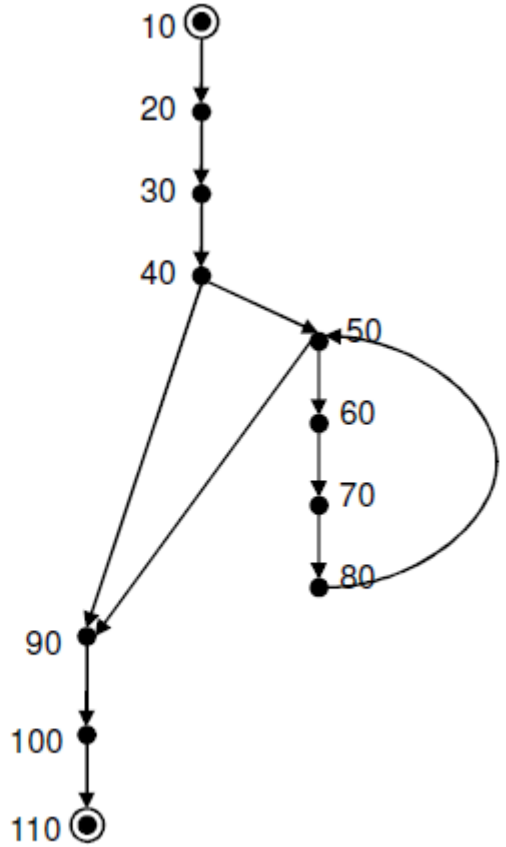
…

Mesure de la taille : la reine LOC

- Elle peut être mesurée aussi bien au niveau du code (nombre de lignes de code) qu'au niveau des documents de spécification (nombre de points de fonctions) ou de la conception (nombre de classes UML).
- Au niveau du code : la mesure la plus utilisée est le ***Nombre de Lignes de Code***
  - Mesure ambiguë : différentes façons de la mesurer : inclusion ou non des commentaires et des déclarations, quid des instructions sur plusieurs lignes ? Dépendant du langage…
    - Concept de LOC logique (cf outil JArchitect)
  - On sépare généralement le code pur => NCLOC du code des commentaires => CLOC.
    - Taille totale LOC = NCLOC + CLOC
  - On peut ainsi mesurer la densité de commentaires par CLOC/LOC
  - Métrique calculable hiérarchiquement
    - Application, Package, Classe, Méthode
    - facilement agrégable…
- D'autres mesures de taille complémentaire : nombre de package, nombre des types (classe, interface, enumération) d'un package, nombre de méthodes ou d'attributs d'une classe

Mesure de la complexité : la reine *complexité cyclomatique de Mc Cabe*

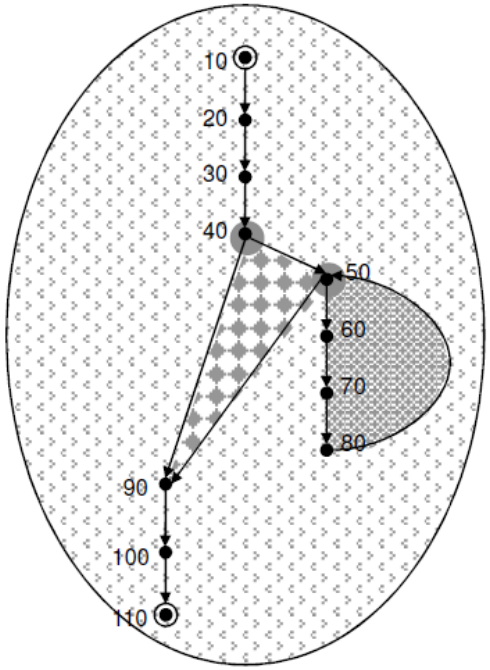
```
10 lire p
20 lire e
30 Cal := 1
40 Si e ~= 0 alors
50 Tant que e > 0 faire
60 cal := cal * p
70 e := e - 1
80 Fin_faire
90 Fin_si
100 écrire cal
110 fin
```



Calcul de l'indice McCabe

- Pour un graphe de flot de contrôle F comportant n nœuds (dont d nœuds prédicats) et e arcs, il existe trois façons de la mesurer :
  - *V(F) = e – n + 2*
  - *V(F) = 1 + d*
  - *V(F) = r* (r est le nombre de régions de F)

v(F) = 12 – 11 + 2
= 1 + 2
= 3



Utilisation pratique de cette mesure

- Calculable au niveau de chaque méthode mais comment l'agréger pour définir la complexité d'une classe, d'un package, d'une application ?
- Au niveau d'une fonction/méthode :
  - Valeur seuil de 10?
  - A mixer avec le nombre de ligne de code
    - *The Software Assurance Technology Center (SATC) at NASA has found the most effective evaluation is a combination of size and (Cyclomatic) complexity. The modules with both a high complexity and a large size tend to have the lowest reliability. Modules with low size and high complexity are also a reliability risk because they tend to be very terse code, which is difficult to change or modify*.

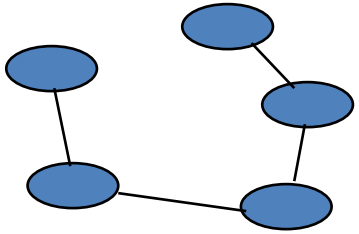
Mesure de la qualité d'une architecture : couplage et cohésion

- Il est admis qu'une « bonne » architecture logicielle vérifie :
  - Une structuration en « blocs emboités » de taille et de complexité raisonnable.
  - Chaque bloc affiche une forte **cohésion** et un faible **couplage** avec les autres blocs.
- Le bon programmeur maîtrise l'art du « découpage » et du « découplage » en intégrant les bonnes idées portées par :
  - les concepts des langages de programmation OO (encapsulation, héritage, interface, classes et méthodes abstraites, redéfinition, polymorphisme)
  - les méthodes de génie logiciel
  - ainsi que les **patrons de conception**.

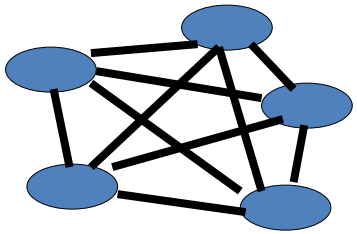
Mesure du couplage et de la cohésion

- **Le couplage** évalue les liens qu'entretiennent les différents « blocs » (classe/Package) d'une application.
  - Quantitativement : le nombre des liens
  - Qualitativement : la bande passante de chacun des liens (existence, une méthode, toutes les méthodes, etc.)
- **La cohésion** quantifie la force de solidarité et d'harmonisation qui lie les données et les opérations d'un « bloc » (classe/Package).
- Couplage et cohésion sont deux faces de la même pièce. Ils sont contra-variants.
  - faible couplage <=> forte cohésion
  - fort couplage <=> faible cohésion

Couplage



Faible couplage
très peu de lien, faible bande passante



Couplage très fort
Nombreux liens,
forte bande
passante

Règle *: S'il y a N sous-systèmes* ***le* nombre de liens** *doit être plus proche de N-1 que de N(N-1)/2 et la* **bande passante** *entre les sous-systèmes doit être la plus faible possible*

Couplage : cas OO

Certaines métriques distinguent le couplage sortant (de qui je dépends = **Efferent Coupling**) du couplage entrant (qui dépend de moi = **Afferent Coupling**)

D'autres examinent les deux sens comme la métrique CBO.

Coupling Between Objects (CBO)

- Définition:
  - Nombre d'autres classes auxquelles une classe est couplée. Deux classes sont dites couplées si une méthode de l'une utilise une méthode ou un attribut de l'autre

$$CBO(c) = |\{d \in C - \{c\} \mid uses(c,d) \vee uses(d,c)\}|$$

$$uses(c,d) \Leftrightarrow \left(\exists m \in M_I(c), \exists m' \in M_I(d), m' \in PIM(m)\right) \vee \left(\exists m \in M_I(c), \exists a \in A_I(d), a \in AR(m)\right)$$

Couplage fort/faible

- Un couplage excessif entre classes se fait au détriment de la modularité et empêche la réutilisation.
- Pour promouvoir l'encapsulation, le couplage entre classes devrait être minimal.
- Plus une classe est couplée à d'autres classes, plus une modification de cette classe peut influencer de classes.
- Une mesure du couplage est importante pour déterminer la complexité du test des diverses parties d'un logiciel.
- L'effort de test d'une classe devrait être d'autant plus élevé que le CBO de cette classe est élevé.

Cohésion : cas OO

- Lack of COhesion in Methods (LCOM)
  - Définition
    - LCOM1 compte le nombre de paires de méthodes qui n'accèdent pas aux mêmes attributs

$$LCOM1(c) = |P|$$

avec

$$P = \{\{m_1, m_2\} \,|m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \emptyset\}$$

    - LCOM2 compte le nombre de paires de méthodes qui n'accèdent pas aux mêmes attributs diminué du nombre de paires de méthodes qui accèdent aux mêmes attributs

$$LCOM2(c) = \begin{cases} |P| - |Q| \; si \; |P| > |Q| \\ 0 \qquad\qquad\quad sinon \end{cases}$$

avec

$$Q = \{\{m_1, m_2\} \,|m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \emptyset\}$$

Cohésion

- Une forte cohésion est signe d'une bonne encapsulation
- Une faible cohésion suggère qu'une classe doit être décomposée en 2 ou plusieurs classes
- Toute mesure de disparité entre les méthodes aide à trouver les fautes de conception
- Une faible cohésion augmente la complexité et par conséquent la probabilité d'existence d'erreurs dans le processus de développement

Exemple d’outil de calcul de métriques de code source (pour Java)

https://www.jarchitect.com/