

Séance 13

Évaluation de la qualité logicielle



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

- Séparation du **cœur** des fonctionnalités étendues
 - Définition et limites du cœur de l'application
 - Interface plugins pour l'extension
- Développement de **plugins** et librairie dynamique
 - Techniques de connexions d'un modules/plugins
 - Définition des contrats entre les plugins et le cœur

Objectifs

- Grands principes de la **qualité logicielle**
 - Organisation et décomposition en sous-systèmes
 - Principes « slogans » à respecter
 - Sept principes clés du développement logiciel
- Deux exemples concrets d'**évaluation de la qualité**
 - Évaluation de la complexité avec les function point
 - Mesure de la complexité POO avec les métriques MOOD

TODAY

ANGER NOT

WORRY NOT

GRATEFUL BE

**DILIGENT,
IN YOUR ENDEAVOURS, BE**

**TO OTHERS,
KINDNESS SHOW**



Grands principles

Don't Repeat Yourself (DRY)

- Réduction de la **répétition d'information**

De code, de données, de documentation, de configuration...

- Facilite la **modification** d'un élément unique

Changement localisé sans impact sur éléments non en lien

- **Augmente** généralement la qualité du code

Maintenabilité, lisibilité, réutilisabilité, cout et test

Décomposition en sous-systèmes (1)

- Diviser un système en composants de tailles gérables

Composants, sous-systèmes, modules, fonctions, classes...

- Réduire complexité jusque responsabilité unique (SRP)

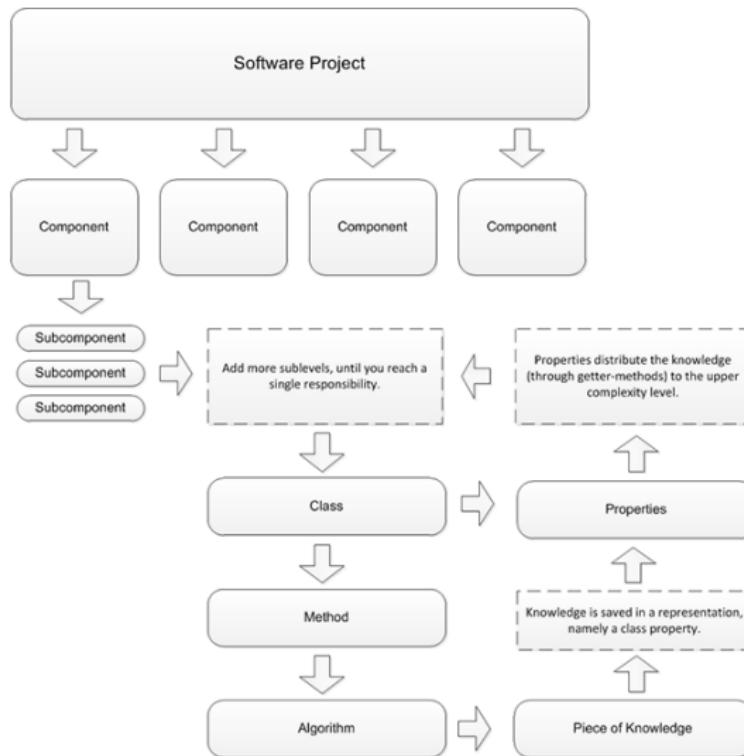
Par exemple jusque classe pour programmation orientée objet

- Différence entre élément de connaissance et sa représentation

Connexion unique base de données, partage variable connexion

- “*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*”

Décomposition en sous-systèmes (2)



Planification

- Division d'une application à penser dès le départ

Hiérarchie visuelle de composants avec du DRYception

- Mauvais code pas souvent produit par mauvais codeur
 - Souvent conséquences de décisions de manager
 - Code pas DRY symptôme de mauvaise qualité, pas raison
 - Hacks ajoutés près des deadlines et milestones
 - Grande compagnie IT fondée par personne avec skill technique

Write Everything Twice (WET)

- Écrire plusieurs fois **la même chose**

Duplication de code, de donnée, de structure...

- Formulaire HTML avec **même étiquette** pour id, label

Et nom de fonction et de variable script, nom du champ DB...

- **Combattre** à l'aide de générateur de code, de framework...

Tout outil automatisé qui assure la cohérence des noms

KISS

Keep It Simple Stupid (KISS)

- Un système fonctionne mieux s'il reste simple

Simplicité doit être un objectif clé dans tout design

- HTTP très simple, mais très puissant et largement utilisé

Méthode de requête, code statut, argument plain text

- Ne pas hésiter à "think out-of-the-box"

Réexprimer les exigences d'un client, aller vers solutions simples

Dead On Arrival (DOA)

- Pleins de **solutions complexes** et half-baked

Multitude de bloatware se complexifiant avec le temps

- Difficulté de faire une bonne **analyse cout-bénéfice**

Surtout sans avoir une certaine expertise technique

- Combattre par cahier des charges et **planification** précis

Ne pas succomber à des changements d'exigences hors contrôle

You Ain't Gonna Need It (YAGNI)

- N'ajouter une fonctionnalité que lorsque vraiment nécessaire
Ne pas vouloir prévoir trop le futur ou être trop générique
 - Prise en compte du contexte et de l'environnement
Pas besoin d'abstraire le driver DB si sur plateforme LAMP
- “80% of the time spent on a software project is invested in 20% of the functionality.”

DRY vs YAGNI

- DRY et WET réduisent la complexité d'un projet

Garder en tête le temps et donc le budget qui est alloué

- DRY décompose en composants de taille gérable

Simplifier au maximum pour faciliter l'implémentation

- YAGNI réduit le nombre de composants

N'implémenter que ce qui est nécessaire

LBYL versus EAFP

- Easier to Ask Forgiveness than Permission (EAFP)

Gestion des erreurs à l'aide du mécanisme d'exception

- Look Before You Leap (LBYL)

Vérification préalable à l'aide d'instructions conditionnelles

```
1 try:  
2     x = my_dict["key"]  
3 except KeyError:  
4     # handle missing key  
5  
6 if "key" in my_dict:  
7     x = my_dict["key"]  
8 else:  
9     # handle missing key
```

DAMP

Descriptive and Meaningful Phrases (DAMP)

- Importance de la **lisibilité** d'un code

Doit pouvoir être compris par le programmeur et les autres

- Contraire au principe **DRY** d'une certaine manière

Ajout d'une certaine redondance pour la lisibilité

- Tolérable dans du **code de test** qui doit être très lisible

Structure très similaire et répétée entre tests

Sept principes du développement logiciel (1)

- “*Boy, I was young once. What I've learned!*”

— DavidHooker, 11/11/11.

Propose sept grands principes à garder en tête et appliquer

1 The reason it all exists

Software existe pour fournir valeur à ses utilisateurs

2 Keep It Simple Stupid (KISS)

Tout design doit être le plus simple possible, mais pas trop

3 Maintain the vision

Garder une intégrité conceptuelle est très importante

Sept principes du développement logiciel (2)

4 What you produce, others will consume

Toujours spécifier, designer et implémenter pour les autres aussi

5 Be open to the future

Plus longue durée de vie, plus grande valeur

6 Plan ahead for reuse



Économiser ressources et efforts en réutilisant des composants

7 Think !

Bien prendre le temps de penser avant de passer à l'action



Quality
and Value

Critère de qualité

Architecture

- Importance d'avoir des logiciels avec **architecture de qualité**

Permet de satisfaire toute une série de propriétés

- Plusieurs **facteurs de qualité** possibles

- Maintenabilité d'un système
- Possibilité de faire facilement évoluer un système
- Performance d'exécution du système et de ses composants
- Pertinence du choix qui colle à la réalité



Maintenabilité

- Maintenabilité d'un système simple et efficace

Maintenance en cas de panne, de migration, de bugs, d'attaque...

- Plusieurs éléments à prendre en compte

- L'architecture doit être adaptée et pas trop lourde, ni complexe
- Pas introduire de nouveaux problèmes lors d'une maintenance
- Ne doit pas nécessiter la présence de l'architecte

Évolutivité

- Architecture doit pouvoir **évoluer** au cours de la vie logicielle

Plusieurs types de changements suite à facteurs externes

- Plusieurs **raisons et types** d'évolutivité

- Adaptation à une plus grosse charge de trafic
- Nouvelles règles/normes de sécurité, de protocole...
- Suivre l'évolution du marché et s'adapter aux utilisateurs

- Ne pas investir trop d'efforts dans un **marché incertain**

Cout important et situation éventuellement non productive

Performance

- Compromis à trouver entre **deux situations** souhaitables
 - Architecture très performante, mais complexe
 - Architecture très simple, mais moins performante
- Multitude de possibilités d'**implémenter la même fonctionnalité**
Identification d'un développeur avec solution hyper performante
- Plusieurs outils de mesure outre des **tests fonctionnels**
Profiler, load/stress test, performance counter

Pertinence

- Plusieurs choix de **design et d'implémentation**

Choix technologique ou types d'implémentation choisis

- **Pertinence** de ces choix par rapport à la réalité

- Code produit pour répondre à des besoins non existants
- Frameworks/plateformes beaucoup trop lourd pour les besoins
- Attention au “*CV driven development*” et choix personnels
- Prise en charge difficile des nouveaux développeurs



- Il faut suivre une pensée **pragmatique**

Et se tourner vers les standards de l'industrie

Style et lisibilité

- **Style** d'un code similaire à structuration d'un texte

Importance d'écrire code avec mise en page structurée et lisible

- Plusieurs **points d'attention** à garder en tête

- Ne pas se limiter au code compilé fonctionnel
- Va jouer un rôle très important pour la maintenabilité
- Outil d'analyse statique ou évaluation humaine

- **Éditeurs de code** intègrent outil d'assistance au style

Contrôle avec CheckStyle ou StyleCop, par exemple

Documentation

- Plusieurs **artefacts** à produire en plus du code source

En plus des commentaires de documentation intégré dans le code
- Deux principaux **types de documentation** selon le destinataire
 - Pour l'utilisateur du software (end application, framework...) 
 - Pour l'équipe de développement (documentation technique)
- Plusieurs autres **informations additionnelles**
 - Liste de SDK supportés, outils d'une version bien déterminée
 - Historique du code, release notes
 - Instructions de configuration de l'environnement 

Fiabilité

- **Fiabilité** mesure nombre de défaillances sur un intervalle

Une valeur élevée indique un logiciel peu fiable

- Possibilité de tester la fiabilité avec **des tests**

Tests manuels, tests de montée en charge...

- **Tolérance aux pannes** et réaction suite à une occurrence

Gestion des erreurs et problèmes enregistrés dans journaux

Portabilité

- Logiciel **portable** si fonctionne sur plusieurs systèmes
Plateforme autant hardware que software
- Utilisation de langages de programmation de **haut niveau**
Exécution sur une machine virtuelle comme Java, C#...
- Forte **dépendance** à une plateforme peut être négative
Sauf pour systèmes très spécifiques

Sécurité

- Un code de qualité doit prendre en compte sa **sécurité**

De plus en plus important, surtout avec app en ligne

- **Exigences de sécurité** très variables d'un projet à l'autre

- Très élevée pour application bancaire, centrale nucléaire...
- Peu élevée pour app locale sur données publiques...

- Possibilité d'**audit de sécurité** semi-automatisé

Injection SQL... par exemple Wapiti (<http://wapiti.sourceforge.net/>)

Nombre de bugs

- Nombre de bugs réel est difficilement connu

Difficile à identifier par analyse statique ou lecture du code

- Liste des bugs connus dans un système de bug tracking

Permet d'identifier si une version d'un code est fiable ou non



FUNCTION :

Function point

Function point (1)

- Mesure de la **complexité d'un software** pas sur le code source
Utilisation des exigences fonctionnelles
- Deux principaux buts des **function points**
 - Quantifier les fonctionnalités d'un système
 - Mesurer le développement et le maintenance
Indépendamment implémentation, projet et organisation
- **Plusieurs standards** ont été développés
IFPUG, NESMA, COSMIC, MkII...

International Function Point User Group (1)

- Utilisation de **cinq paramètres** (simple, moyen ou complexe)
 - **Input externe** unique vers le système
 - **Output externe** unique produit par le système
 - **Fichier logique interne** maintenant données et contrôle
 - **Fichier d'interface externe** partagé à l'extérieur
 - **Requêtes externe** comme input+output
- Pondération par **paramètre et complexité** selon une table

Paramètre	Simple	Moyen	Complex
Input externe	3	4	6
Output externe	4	5	7
Fichier logique interne	3	4	6
Fichier d'interface externe	7	10	15
Requête externe	5	7	10

International Function Point User Group (2)

- Ajustement en évaluant la **complexité de l'environnement**

Description d'un système par quatorze caractéristiques générales

-
- | | |
|--------------------------------|-----------------------|
| 1. Data communication | 8. On-line update |
| 2. Distributed data processing | 9. Complex processing |
| 3. Performance | 10. Reusability |
| 4. Heavily used configuration | 11. Installation ease |
| 5. Transaction rate | 12. Operation ease |
| 6. Online data entry | 13. Multiple sites |
| 7. End-user efficiency | 14. Facilitate change |
-

- Évaluation de chaque caractéristique **de 0 à 5**

No influence, incidental, moderate, average, significant, essential

International Function Point User Group (3)

- Évaluation de **chaque composant** avec les cinq paramètres

$$UFP = \sum FP = \sum occurrence \times complexity$$

- **Degré total d'influence** des caractéristiques du système

$$TDI = \sum characteristic \times value$$

- Calcul du **facteur d'ajustement de valeur**

$$VAF = 0.01 \times TDI + 0.65$$

- Enfin, on termine en obtenant les **points de fonction ajustés**

$$AFP = VAF \times UFP$$

Utilisation des function points

- Complexité d'un software par rapport à ses fonctionnalités
Il s'agit d'un modèle d'évaluation parmi d'autres
- Function points peuvent être utilisées dans d'autres métriques
 - **Cout** : €/ AFP
 - **Qualité** : erreurs / AFP
 - **Productivité** : FP / personnes-mois

Systèmes 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%

Héritage



- 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

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

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/rlei_ki/8587714166
- https://cdn.tutsplus.com/net/uploads/legacy/2047_softwarePrinciples/dry.png
- <https://www.flickr.com/photos/wetwebwork/2402908982>
- <https://www.flickr.com/photos/bobafred/10759141>
- <https://www.flickr.com/photos/joseluisbriz/16794007759>