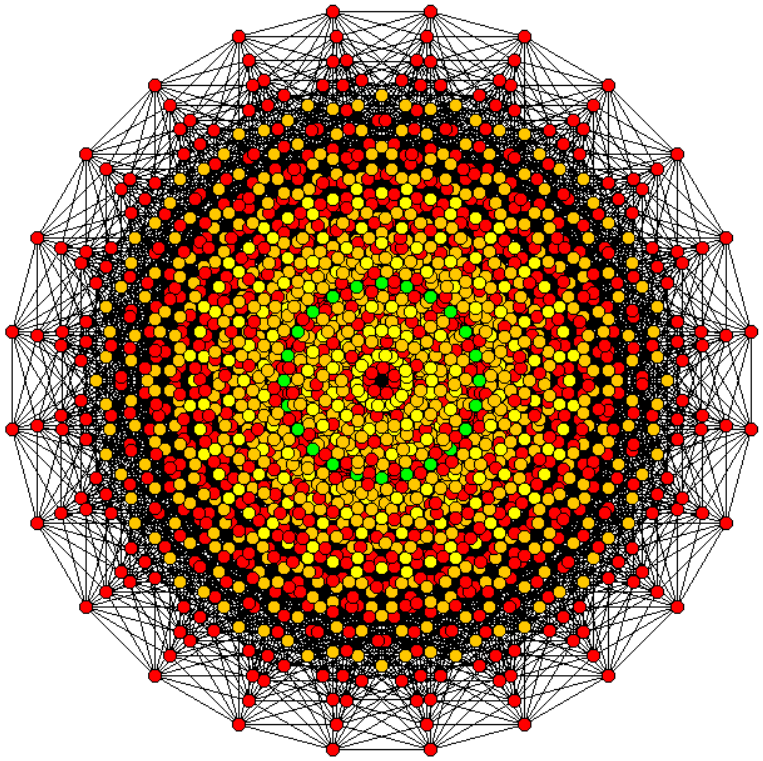

Notion de qualité du logiciel

Ressource libre

Version 1.1, 2021-11-01

Table des matieres

Présentation

En mars 2020, Google publie Building Secure and Reliable Systems.

En préface, le livre démarre par cette question : "*un système peut-il être considéré comme vraiment fiable [reliable] s'il n'est pas fondamentalement sûr [secure] ? Ou peut-il être considéré comme sûr s'il n'est pas fiable ?*".

Cette question nous ramène à la notion de qualité dans le logiciel et ce document présente ici, dans les grandes lignes, les principaux critèques de qualité et nous invitons le lecteur à approfondir ce thème par l'ouvrage de référence, sur lequel cette étude est principalement basée, Conception et Programmation orientées objet (Bertrand Meyer).

Notion de qualité du logiciel

Un bon logiciel métier est un logiciel qui a atteint ses objectifs, à savoir la satisfaction de ses utilisateurs quant à la qualité des services rendus et pour sa facilité d'utilisation, rapidité, robustesse etc.

Selon la catégorie de l'utilisateur, les critères d'appréciation peuvent varier. Par exemple le responsable des achats sera très satisfait si le logiciel de gestion de la chaîne logistique est livré à date prévue, et sans surcoût. Un chef d'atelier sera plus attentif à l'ergonomie du logiciel qu'à son coût, un DPO (pour *Data Protection Officer*) sera particulièrement vigilent à la protection des données personnelles, etc.

La satisfaction est fonction des intérêts de chacun, même si certains, comme la sécurité peuvent être transverses.

La qualité est une affaire de point de vue. Un informaticien chargé de maintenir le code d'une application, sera sensible à son architecture, à la lisibilité du code source.

Les points de vue peuvent se classer en deux catégories, ou facteurs, nommées **externes** et **internes**.

Les **facteurs externes** sont ceux perçus par les utilisateurs au sens large, c'est à dire ceux qui ne sont pas en contact direct avec le code source du logiciel. Les facteurs externes sont les seuls qui comptent en réalité car ils concernent la finalité du produit, sa raison d'être, cependant, le seul moyen d'atteindre des niveaux de qualités visibles satisfaisants pour l'utilisateur (facteurs externes), est de rechercher à atteindre les qualités cachées du logiciel : **facteurs internes**.

Voici quelques unes des qualités attendues : correction, facilité d'utilisation, rapidité d'exécution, extensibilité, couverture fonctionnelle, ponctualité, robustesse, réutilisabilité, compatibilité, portabilité, documentation.

La qualité du logiciel dépend de facteurs à la fois externes et internes.

Correction

> La **correction** est la capacité que possède un produit logiciel de mener à bien sa tâche, telle qu'elle a été définie par sa spécification (le logiciel répond-il aux besoins exprimés ?)

C'est la première qualité attendue, si elle n'est pas remplie alors le reste ne compte pas.

Que ferais-je d'une voiture si elle ne peut pas me transporter d'un endroit à un autre ? Sa couleur, l'ergonomie de son tableau de bord sont autant de qualités que je ne saurais que faire.

Les méthodes garantissant la correction sont, en général, conditionnées.

On ne réinvente pas la roue à chaque projet. On réutilise des composants techniques et métiers maisons que l'on garantie mais aussi de tierces parties, parfois imposés par le client.

Les bibliothèques de base sont parfois écrites dans un autre langage que celui utilisé, le compilateur traduit-il correctement le code ? Nous trouvons ici deux classes de correction : une *sémantique* : le sens que le développeur prête au code source est-il conforme à la sémantique du langage ? - par exemple `if ("PARFAIT" == etat)` (etat une variable de type String) peut ne pas donner le résultat escompté - et une autre *technique*, concernant la correction de la traduction automatique du code source en un code exécutable.

La chaîne de responsabilités est organisée en tiers (collaboration horizontale) et en couches verticales :

- Système applicatif
- Objets métier et technique maison
- Autres bibliothèques
- Bibliothèques de base
- Bibliothèques du noyau
- Compilateur
- Système d'exploitation
- Matériel (micro-processeur. [1: https://www.zdnet.fr/actualites/linus-torvalds-parle-en-toute-franchise-des-bugs-de-securite-d-intel-39873065.htm], bus, ..)

La correction est une qualité plus difficile à atteindre qu'il n'y parait. Les tests automatisés, prenant appui sur les spécifications, participent efficacement à la mesure de la correction.

Robustesse

La **rodustesse** est la capacité qu'offre des systèmes à réagir de manière appropriée à la présence de conditions anormales.

La robustesse complète la correction.

Alors que la correction concerne le comportement du système dans des cas conformes à sa spécification, la robustesse caractérise ce qui se passe en dehors de cette spécification.

Un cas est étiqueté d'**anormal** s'il n'est pas couvert pas la spécification.

Éviter les catastrophes, tendre à la mise en place d'un système garantissant une "dégradation harmonieuse".

Le mécanisme de traitement des exceptions favorise la réalisation de logiciel robuste.

Robustesse-Cybersecurité

Ce facteur de qualité est une application de la robustesse, dans le cadre de la cybersécurité.

La qualité **robustesse - cybersécurité** relève de la prise en compte de principes et de bonnes pratiques visant à résister à des usages malveillants du logiciel (cybersécurité).

Exemples de bonnes pratiques

- **Principe de moindre privilège**. Stipule qu'un cas d'utilisation ne doit bénéficier que des privilèges strictement nécessaires à l'exécution de son code.

L'objectif étant de *restreindre les conséquences possibles de comportements inattendus d'un composant, qu'il s'agisse d'un bogue ou de son détournement par un attaquant*. [gansii]

- **Réduction de la surface d'attaque**. *La surface d'attaque ou surface d'exposition est la somme des différents points faibles (les « vecteurs d'attaque ») par lesquels un utilisateur non autorisé (un « pirate ») pourrait potentiellement s'introduire dans un environnement logiciel et en soutirer des données.* [3: Surface d'attaque - wikipédia]

Une approche pour améliorer la sécurité de l'information est de **réduire** la surface d'attaque d'un système, d'un logiciel. Les stratégies de base de réduction d'une surface d'attaque sont les suivants :

- Limitation du nombre de points d'entrée disponibles (par exemple, interdire tout accès à la base de données de l'application en dehors de celle-ci ou d'un compte admin en ssh)
- Limitation des dépendances tiers
- Réduction de la quantité de code en cours d'exécution

- **Défense en profondeur**

Terme emprunté à une technique militaire destinée à retarder l'ennemi. Consiste à exploiter plusieurs techniques indépendantes de sécurité, afin de ***réduire le risque*** *lorsqu'un composant particulier de sécurité est compromis ou défaillant.*

En pratique, pour certaines actions aux lourdes conséquences (suppression de données sans possibilité de revenir en arrière, ou à impact métier fort) le développeur pourra appliquer une mesure d'authentification préalable, même si l'utilisateur est considéré comme ayant les droits à ce moment là (dans le cas d'un défaut de sécurité côté utilisateur — une autre personne se faisant passer pour lui suite à un vol de jeton d'authentification, ou à l'emprunt d'une chaise vide dans les locaux de l'entreprise…). Voir ANSSI le modele *zero trust*

Attention cependant de ne pas tomber dans le travers de la programmation défensive. Voir : Défense en profondeur - wikipédia)

- **Valeurs par défaut sécurisées**.

Ce peut être des valeurs initiales d'un formulaire construit dynamiquement ou des valeurs par défaut de connexions à une base de données placées dans un fichier de configuration, etc.

Exemple : 600TB de données MongoDB exposées sur la toile

References

- [gansii] guide ANSSI sur le cloissement système - 2017
- the security development lifecycle - Microsoft

Extensibilité

> L'**extensibilité** est la facilité d'adaptation des produits logiciels aux changements de spécifications.

Un problème d'autant plus pointu que le nombre de lignes de code du logiciel est grand.

Le contexte de forte concurrence, les fusions d'entreprise, l'évolution technologique entraîne souvent des changements majeurs des systèmes d'informations. Plus le logiciel est flexible, plus sa capacité de s'adapter aux changements de spécifications est forte.

Deux principes essentiels à l'amélioration de l'extensibilité :

- Simplicité de conception : Une architecture simple est plus facile à adapter aux changements
- Décentralisation : Une répartition des taches en modules les plus autonomes possibles. Ainsi l'adaptation d'un module n'entraîne pas ou très peu de répercussion sur les autres.

On pourra s'interesser aux principes d'eXtrême Programming suivants :

- YAGNI : https://fr.wikipedia.org/wiki/YAGNI
- KISS : https://fr.wikipedia.org/wiki/Principe_KISS

Réutilisabilité

La **réutilisabilité** est la capacité des éléments logiciels à servir à la construction de plusieurs applications différentes.

Réutilisation de "bout de code", module, composants logiciels spécialisés appelés objets techniques ou objet métiers.

On pourra s'intéresser au principe suivant :

- DRY : don't repeat yourself - wikipedia

Certains modèles de conception sont aussi réutilisés, et même catalogués (design patterns. [5: Design Patterns est un livre de méthodologie appliquée à la conception logicielle écrit par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides et publié en 1994 chez Addison-Wesley. Cet ouvrage aborde le sujet de la programmation orientée objet et introduit le concept des patrons de conception. Wikipédia]) ou méta-catalogués (méta-pattern).

Compatibilité

La **compatibilité** est la faculté avec laquelle des éléments logiciels peuvent être combinés à d'autres.

Exemple : Les formats de fichier d'un logiciel ouverts ou propriétaires.

Adopter des standards. Par exemple XML/JSON comme strutures d'échanges, Web Service, architecture REST, etc.

Efficacité

L'**efficacité** est la capacité d'un système à utiliser le minimum de ressources matérielles.

Les ressources à préserver sont :

- *temps machine* ,
- *espace occupé* en mémoire externe et interne,
- *bande passante* des moyens de communication.

Portabilité

La **portabilité** est la faculté avec laquelle des produits logiciels peuvent être transférés d'un environnement logiciel ou matériel à un autre.

Par exemple la capacité d'un logiciel sous Windows à être porté sous Gnu/Linux.

UX (expérience utilisateur)

L'**expérience utilisateur** (en anglais, user experience, abrégé **UX**) est la qualité du vécu de l'utilisateur dans des environnements numériques ou physiques.

La **facilité d'utilisation** est l'aisance avec laquelle des personnes, de formations et de compétences différentes, peuvent apprendre à utiliser les produits logiciels et à s'en servir pour résoudre des problèmes.

Sont concernés également la facilité d'installation, d'opération et de contrôles.

Principe de conception d'interface utilisateur : ***Ne prétendez pas connaître l'utilisateur ; vous ne le connaissez pas***.

Conseil : Chercher la simplicité

Couverture

la **couverture fonctionnelle** est l'étendue des possibilités offertes par un système.

Il est conseiller, en début de projet, de définir les limites de l'application, d'éviter la "course aux options".

Quelques Conséquences liées à la course aux options :

- L'utilisateur risque de se perdre dans les options
- Perte de cohérence du produit qui "fait mille choses"
- Risque de dépassement de budget

Introduire de nouvelles fonctions que si les fonctionnalités actuelles offrent entière satisfaction, tout en étant ouvert à la négociation en cas de redéfinition des besoins (agilité)

On pourra s'interesser à la méthode **MoSCoW** dans le but d'affecter des **niveaux de priorités** aux exigences.

Ponctualité

> La **ponctualité** est la capacité d'un système logiciel à être livré au moment désiré par ses utilisateurs, ou avant.

Un super produit qui arrive trop tard peut complètement rater sa cible.

Documentation

La **documentation** est tout ce qui, dans un projet, peut apporter du sens, de la conaissance aux autres, à compter par le code source lui-même.

On distingue :

- Documentation **interne** : orientée développeur. Exemples : Code source, JavaDoc, diagrammes UML, pour une vue architecturale du système permettant d'en apprécier le contenu
- Documentation d'**API** (interface de module) : Permettre aux développeurs de comprendre les fonctionnalités offertes par un module sans en connaître l'implémentation.
- Documentation **externe** : orientée utilisateur, mode opératoire, didacticiel et autres...

Ne documentez pas inutilement votre code, mais faites lui communiquer ce qu'il fait par ses propres mots.

> Pour en finir avec l'**amnésie métier du code source**, le développeur, dans son code, utilise des termes que l'expert métier connait et utilise !
>
> — DDD Eric Evans, ubituitous language

Facteurs contradictoires

Durant la vie d'un projet, il est souvent question de compromis, et les facteurs de qualité n'échappent ps à cette règle.

Difficiles compatibilités et Compromis

Certaines qualités sont difficilement compatibles :

- Efficacité et Portabilité
- Couverture Fonctionnelle et Facilité d'utilisation
- Ponctualité et Couverture Fonctionnelle, Extensibilité
- Robustesse et Facilité d'utilisation

De ce fait, le développeur doit parfois (souvent ?), faire des choix (compromis concertés), mais il y a une qualité pour laquelle le compromis n'est pas négociable : la correction !

Qualités essentielles

Les qualités généralement attendues d'un logiciel est sa *fiabilité* et sa *modularité*.

Fiabilité

La **Correction** et **Robustesse** sont les premiers facteurs de qualité. Ils définissent ensemble le caractère **fiable** du logiciel.

Modularité

L'**Extensibilité** et la **Réutilisabilité** caractérisent la facilité de modification et d'évolution.

Un composant logiciel généralisé pourra être utilisé dans d'autres applications, directement ou par adaptation.

Ces deux qualités sont souvent représentées par le mot : **Modularité**

Autres qualités internes (eXtreme Programming)

Les facteurs de qualité présents dans ce document ne pourrait se passer de quelques recommandations de l'eXtreme Programming.

Une fois et une seule

Kent Beck (eXtreme Programming [xp]) identifie les 2 principes suivants :

1. Le système (code et tests pris dans leur ensemble) doit **communiquer** tout ce que vous avez l'intention de communiquer (comprendre le code **est** la documentation, à rapprocher de #Documentation)
2. Le système ne doit contenir **aucune duplication de code**

Ces deux caractéristiques constituent la règle : **Une Fois et Une Seule**.

Convention de nommage

Puisque tous les développeurs interviennent sur tout le code, il est indispensable d'établir et de respecter des normes de nommage pour les variables, méthodes, objets, classes, fichiers, etc.

Le code est toujours testé

- Présence obligatoire de tests unitaires associés au code.
- Lorsqu'un bogue survient, un test unitaire est créé.

References

- [xp] eXtreme Programming, Kent Beck, CampusPress
- [xp2] L'eXtreme Programming, J.L. Bénard, L. Bossavit, R. Médina, D. Williams, Editions Eyrolles
- eXtreme Programming - wikipedia