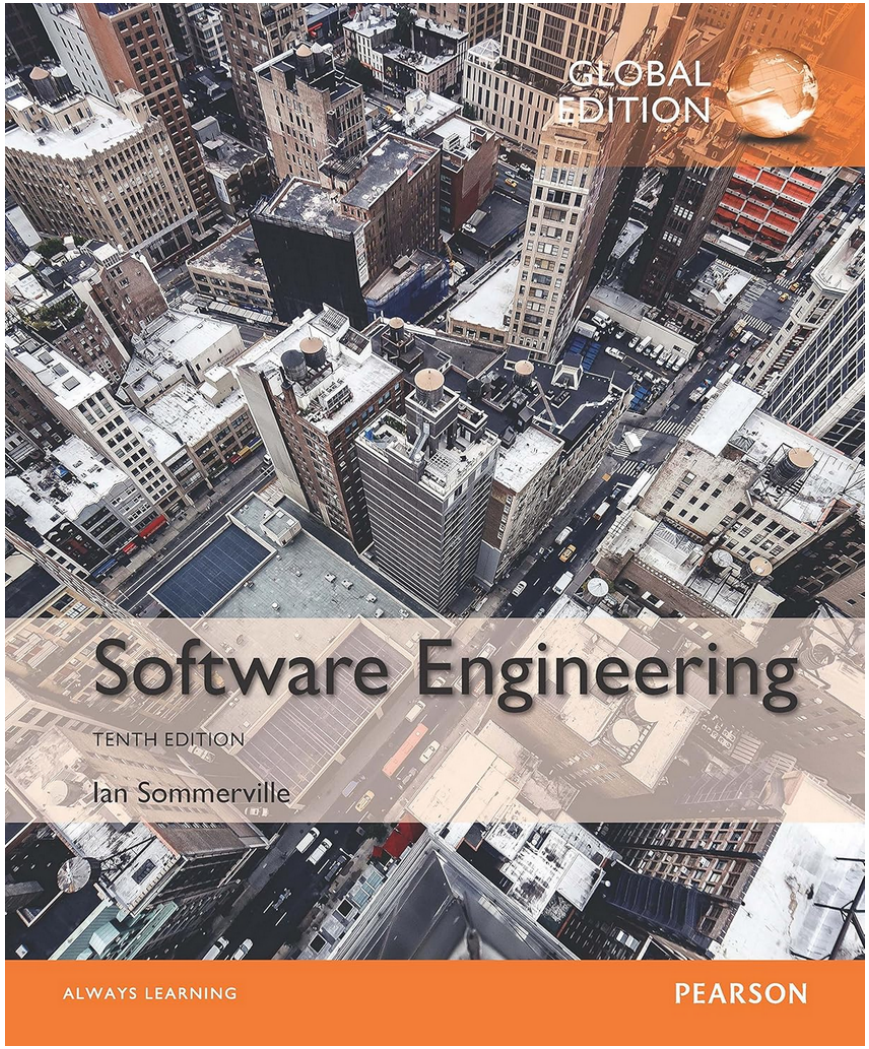
Université Bretagne Sud

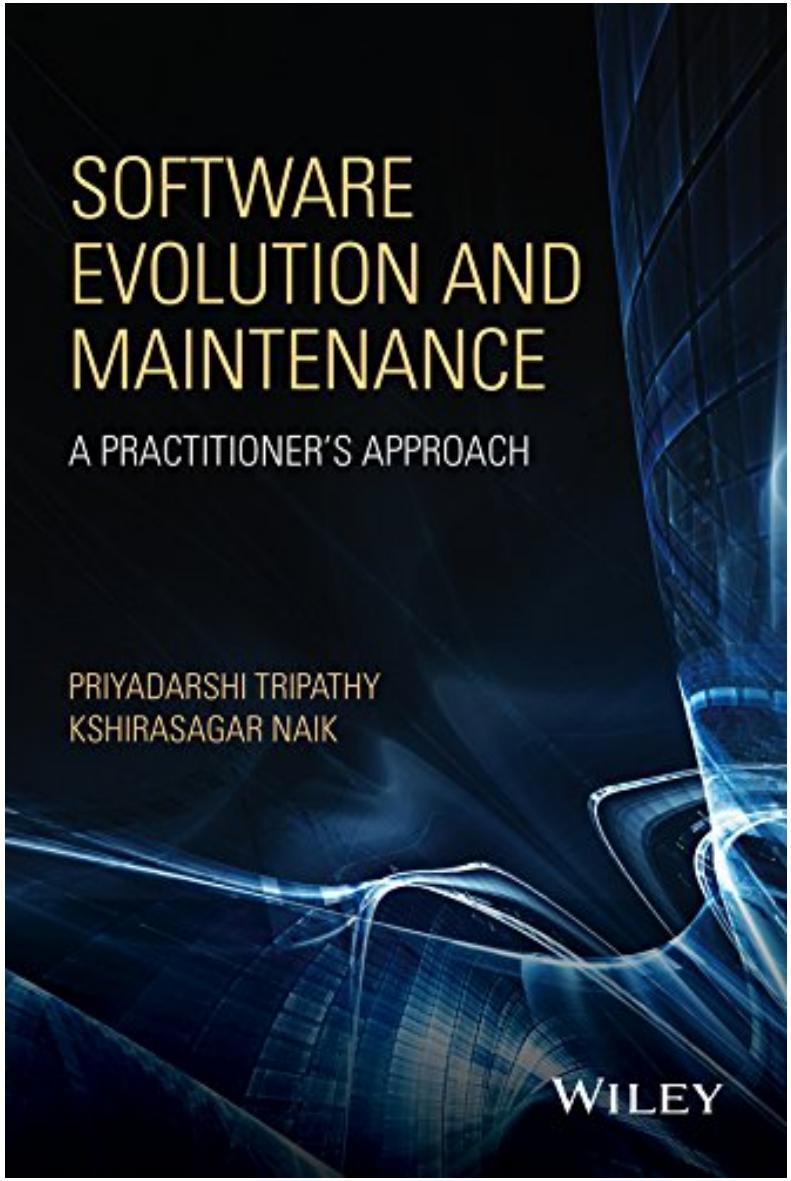
R6.A.06. Maintenance applicative

Chouki Tibermacine
Chouki.Tibermacine@univ-ubs.fr

Vannes
iut :
Université Bretagne Sud

Références bibliographiques





Plan du cours

Contenu du cours

Pendant 3 semaines, chaque semaine : 1,5h CM + 3h TD

- Semaine 1 : Introduction à la maintenance applicative – Définitions, indicateurs et processus
- Semaine 2 : Techniques et outils pour la maintenance – Analyse de bad smells et refactoring
- Semaine 3 : Gestion des évolutions et bonnes pratiques – Documentation, gestion efficace des versions, ...

Modalités d'évaluation

- Un livrable de fin de mini-projet : rapport d'analyse, de correctifs et documentation technique
- Travail par binôme
- Critères d'évaluation
  1. Compréhension des concepts vus en cours magistral ;
  2. Qualité du code (maintenabilité, tests, clarté) ;
  3. Utilisation des outils et méthodes enseignées ;
  4. Rigueur dans la documentation et le respect des bonnes pratiques.

Un peu d'histoire

- En 1965, M. Halpern a introduit le terme "software evolution" ;
- Quelques années plus tard, Belady et Lehman ont publié les lois de l'évolution du logiciel ;
- En 1976, E. Burton Swanson a introduit le terme "software maintenance" ;
- Dans les années 70, IBM appelait les ingénieurs de maintenance tout développeur faisant des modifications intentionnelles sur un code en cours d'exécution qu'ils n'avaient pas développés eux-mêmes ;
- Le rôle était attribué à des personnes différentes des développeurs pour décharger ces derniers des activités de support.

Définitions des concepts de maintenance et d'évolution

Tripathy et Naik, 2014 :

- "Software maintenance means preventing software from failing to deliver the intended functionalities by means of bug fixing" ;
- "Software evolution means a continual change from a lesser, simpler, or worse state to a higher or better state" ;
- Toute activité menée après la livraison d'un logiciel (Bennett & Xu) est considée comme de la maintenance.

Importance de la maintenance

- Coût estimé entre **50 % et 90 %** des dépenses totales liées au logiciel sur l'ensemble de son cycle de vie (différentes sources)

Types de maintenance du logiciel

- Swanson a proposé trois types de maintenance selon l'intention (1976) : corrective, adaptative et préventive ;
- Plus tard, on a ajouté la maintenance préventive, qui peut être vue comme faisant partie de la maintenance perfective ;
- Types de maintenance selon les activités menées :
  - Activités pour corriger : Si écarts entre le comportement attendu d'un système et le comportement réel, certaines activités sont alors réalisées pour éliminer ou réduire ces écarts ;
  - Activités pour améliorer : ce qui implique i) des changements dans les besoins (*requirements*), ii) la création de nouveaux besoins, et iii) des changements dans l'implémentation sans impact sur les besoins.

Types de maintenance du logiciel -suite-

Les activités de maintenance peuvent être réparties sur 4 classes (norme ISO/IEC 14764) :

1. **Maintenance corrective**
   - Définition : Corrige les défauts ou les bugs détectés dans le logiciel en production ;
   - Exemples : Correction d'une fonctionnalité défaillante, résolution d'une erreur qui provoque un crash.
2. **Maintenance adaptative**
   - Définition : Ajuste le logiciel pour qu'il continue à fonctionner correctement dans un environnement en évolution (matériel, système d'exploitation, dépendances, etc.) ;
   - Exemples : Migration vers un nouveau serveur ou système d'exploitation, mise à jour des bibliothèques ou des frameworks.

Types de maintenance du logiciel -suite-

3. **Maintenance perfective**
   - Définition : Améliore ou optimise le logiciel pour augmenter ses performances ou enrichir ses fonctionnalités ;
   - Exemples : Ajout d'une nouvelle fonctionnalité demandée par les utilisateurs, optimisation des requêtes pour réduire les temps de réponse.
4. **Maintenance préventive**
   - Définition : Anticipe les problèmes futurs pour réduire les risques d'échec ou d'interruption du logiciel ;
   - Exemples : Refactorisation du code pour réduire la dette technique, mise en place de tests automatisés pour prévenir les régressions.

Processus et modèles de cycle de vie

- Maintenance = une phase dans le modèle de la cascade (waterfall : Winston Royce, 1970) ;
- Pendant longtemps, on a considéré la maintenance comme faisant partie du cycle de développement, la dernière étape ;
- Alors que la maintenance doit avoir son propre cycle de vie (Norman Schneidewin, 1987) parce qu'elle est trop coûteuse ;
- Un cycle de vie de la maintenance (SMLC) standard comprend : compréhension du code, modification du code, et revalidation du code (Ned Chapin, 1988).

Processus et modèles de cycle de vie -suite-

- Modèles itératifs :
  - Les logiciels sont construits dans des builds (itérations) ;
  - Ces builds sont des raffinements des précédents builds (et leurs requirements), suite aux feedbacks utilisateurs ... ;
  - La maintenance n'existe pas en tant que phase indépendante.
- Modèles de mini-cycle de changements (Yau et al 1978, Bennet et al 2000 et Tom Mens 2008) : *requête de changement*, *analyse et planification du changement*, *implémentation du changement*, *verification et validatation*, *et documentation du changement*

Activités importantes qui émergent : *program comprehension*, *impact analysis*, *refactoring* et *change propagation*

Standards de maintenance

- Standard ISO : ISO/IEC 14764 publié en 2006 (partie du standard ISO/IEC 1220 sur les cycles de vie du logiciel)
- Standard IEEE : IEEE/EIA 1219 de 1998
- Globalement les mêmes activités listées dans les deux standards : *problem identification*, *analysis*, *design*, *implementation*, *system test*, *acceptance test* et *delivery*
- Chaque standard indique les livrables en entrée et en sortie de chaque activité et les métriques d'évaluation

Software Configuration Management (SCM)

- Gestion de configuration logicielle : ensemble de pratiques, de processus et d'outils utilisés pour gérer les modifications et l'évolution des éléments d'un système logiciel tout au long de son cycle de vie ;
- Objectif : garantir que le logiciel reste cohérent, fonctionnel et traçable malgré les évolutions et les modifications ;
- Il existe un standard pour la SCM : IEEE 1042 de 1988

Objectifs du SCM

1. Contrôle des modifications : Suivre et gérer les modifications apportées aux fichiers source, aux documents, aux configurations et à d'autres éléments du projet ;
2. Traçabilité : Permettre de retracer l'origine des changements et d'identifier les raisons, les responsables et les impacts de chaque modification ;
3. Cohérence : Maintenir une version stable du logiciel malgré des évolutions fréquentes, en évitant les conflits ou les divergences ;
4. Collaboration : Faciliter le travail en équipe en assurant une gestion structurée des contributions des développeurs ;
5. Audit et historique : Fournir un historique détaillé des changements pour des audits ou des résolutions de problèmes.

Activités principales du SCM

1. Gestion des versions : assurer que chaque version du logiciel est identifiable et qu'il est possible de revenir à une version antérieure si nécessaire. Exemples : Git, SVN, Mercurial ;
2. Gestion des configurations : définir et gérer les relations entre les différents éléments du logiciel (code source, bibliothèques, fichiers de configuration) ;
3. Gestion des builds : automatiser et standardiser les processus de compilation et d'intégration pour générer des versions ;
4. Gestion des changements : mettre en place des processus pour évaluer, approuver et implémenter les demandes de modification ;
5. Gestion des livraisons : préparer et distribuer les versions du logiciel, en s'assurant que les configurations sont respectées.

Exemple d'outils de SCM

- Git : Système de contrôle de version distribué largement utilisé.
- Subversion (SVN) : Système de contrôle de version centralisé.
- Jenkins, Github Actions ou Gitlab CI : Outils d'intégration continue pour la gestion des builds.
- Ansible, Puppet, Chef : Pour gérer la configuration des serveurs et des environnements d'exécution.

Reengineering

- Selon Jacobson and Lindstörm (1991) :

  $\textit{Reengineering} = \textit{Reverse engineering} + \Delta + \textit{Forward engineering}$
- *Reverse engineering* : activité consistant à définir une représentation plus abstraite et plus facile à comprendre du système (code source $\longrightarrow$ architecture)
- Δ : capture les alterations faites au système
- *Forward engineering* : processus traditionnel de passage d'une abstraction de haut niveau et d'une conception logique et indépendante de la mise en œuvre à l'implémentation physique du système

Systèmes légataires/hérités (*legacy*)

- Ce sont d'anciens logiciels qui continuent d'être utilisés parce qu'ils répondent toujours aux besoins des utilisateurs, malgré la disponibilité de technologies plus récentes ou de méthodes plus efficaces pour effectuer la tâche ;
- Ils comprennent des procédures ou une terminologie obsolètes, et il est très difficile pour les nouveaux développeurs de comprendre le système ;
- Ils résistent aux modifications et à l'évolution pour répondre aux exigences commerciales nouvelles en constante évolution.

Que faire des systèmes légataires/hérités ?

- *Freeze* : ne plus effectuer de travaux sur le système (services du système ne sont plus utiles) ;
- *Outsource* : décider que la prise en charge de ce système ne constitue pas son activité principale. Sous-traiter le service à une organisation spécialisée ;
- *Poursuivre la maintenance* : continuer à maintenir le système pendant une autre période, malgré toutes les difficultés que cela implique ;
- *Éliminer et redévelopper* : le système est redéveloppé de zéro, en utilisant de nouvelles technos, une nouvelle architecture logicielle, … ;

Que faire des systèmes légataires/hérités ? -suite-

- *Wrap* : envelopper le système d'une nouvelle couche logicielle, masquant la complexité indésirable des données existantes, des programmes et de l'interface ;
  - Le terme "*wrapper*" a été introduit chez IBM par Dietrich et al. à la fin des années 1980 (tirer profit de composants de confiance, bien testés, dans lesquels on a beaucoup investi) ;
- *Migrer* : migrer vers un nouveau système, tout en conservant les fonctionnalités du système existant. L'idée est de minimiser toute perturbation de l'environnement métier existant. Cela implique la restructuration du système existant et l'amélioration des fonctionnalités du système.

Analyse d'impact

- C'est la tâche consistant à identifier les parties du logiciel qui peuvent potentiellement être affectées si une modification proposée au système est effectuée ;
- Le résultat peut être utilisé lors de la planification des modifications, de leur mise en œuvre et du suivi des effets des modifications afin de localiser les sources de nouvelles pannes ;
- Techniques d'analyse d'impact :
  - *Traceability analysis* : identifier les artefacts de haut niveau, tels que les requirements, la conception, le code et les cas de test liés à la fonctionnalité à modifier. Un modèle d'associations entre les artefacts est construit ;
  - *Dependency analysis* : évaluer les effets d'un changement sur les dépendances entre les entités du code source.

Analyse d'impact -suite-

Deux concepts sous-jacents :

- *Ripple effect analysis* (analyse d'effet d'entraînement) :
  - met l'accent sur les répercussions du changement d'un module sur le reste ;
  - impact exprimé en termes de problèmes créés (les quoi et où ?) ;
  - mesurer la complexité avant et après changement (par module).
- *Change propagation* :
  - ensemble d'activité garantissant qu'un changement est répercuté partout où il faut ;
  - la mauvaise compréhension du code, le manque d'expérience et les dépendances inattendues sont quelques-unes des raisons d'échec de la propagation ;
  - si un changement n'est pas propagé correctement, on risque d'introduire de nouveaux bugs.

Refactoring

- Consiste à apporter des modifications à la structure du logiciel pour le rendre plus facile à comprendre et moins coûteux à réaliser des modifications ultérieures, sans modifier le comportement observable du système ;
- Est réalisé par la suppression du code dupliqué, la simplification du code et le déplacement du code vers un module différent, entre autres ;
- Doit être fait de façon continue pour rendre :
  1. l'architecture du logiciel stable ;
  2. le code lisible ;
  3. les tâches d'intégration de nouvelles fonctionnalités dans le système flexibles.
- Préserver le comportement observable : tous les tests réussis avant le refactoring doivent réussir après le refactoring.

Compréhension de code

- L'objectif de la compréhension de programmes est de saisir la structure et les comportements d'un système logiciel existant pour planifier, concevoir, coder et tester les changements ;
- T. A. Corbi a observé en 1989 que la compréhension du programme représente 50 % de l'effort total dépensé tout au long du cycle de vie d'un système logiciel.
- Mesuré récemment jusqu'à 70 % du temps, Minelli et al 2015 ;
- Avec les LLM, cela va s'améliorer (études empiriques en cours) ;
- Cela implique la construction de *modèles mentaux* d'un système sous-jacent à différents niveaux d'abstraction (modèles de bas niveau du code aux modèles de très haut niveau du domaine d'application).

Compréhension de code -suite-

- Une étape clé dans le développement de modèles mentaux consiste à générer des hypothèses, ou des conjectures, et à étudier leur validité ;
- Les hypothèses sont un moyen pour un programmeur de comprendre le code de manière incrémentale (hypothèses validées ou rejetées au fur et à mesure qu'elle/il lise le code) ;
- Le processus d'assimilation (parcourir les pièces d'information dans le code : commentaires, instructions, ...) peut fonctionner de 3 manières : top–down (objectif métier ⟶ instructions dans le code), bottom–up, et opportuniste (hybride)

Plan du cours

Plan du cours

Indicateurs de maintenance

- Ce sont des métriques utilisées pour évaluer la facilité avec laquelle un logiciel peut être modifié, corrigé, ou étendu ;
- Ces indicateurs permettent de suivre la qualité du code, de détecter les zones à risque, et de prioriser les efforts de maintenance pour améliorer la longévité et les qualités (fiabilité, sécurité, ...) du système ;
- Il existe différentes sortes d'indicateurs : indicateurs de base, de qualité de code, de productivité et d'efficacité, ...

Indicateurs de base

- Complexité cyclomatique (vue en cours Qualité Algo) ;
- Taux de duplication de code :
  - pourcentage de lignes de code dupliquées ;
  - une duplication excessive entraîne des difficultés de maintenance : une modif. à répercuter à plusieurs endroits ;
  - seuil recommandé : moins de 5 % de duplication.
- Couverture des tests (Test Coverage) :
  - Pourcentage de code couvert par des tests unitaires ou automatisés ;
  - Une faible couverture augmente les risques de régression lors des modifications.
- Nombre de défauts/bugs :
  - Nombre de bugs détectés et non résolus dans le logiciel ;
  - Les zones avec un nombre élevé de défauts nécessitent souvent des efforts de maintenance importants.

Indicateurs de qualité de code

- Taille des méthodes/classes :
  - Méthodes ou classes trop longues sont plus difficiles à comprendre et à maintenir ;
  - Bonne pratique : taille méthode < 20 à 30 lignes.
- Dette/Endettement technique (Technical Debt) :
  - Temps estimé nécessaire pour corriger les problèmes identitiés dans le code (détaillée plus loin) ;
- Nombre de violations des bonnes pratiques :
  - Nombre de violations des standards de codage ou des bonnes pratiques (par exemple, nommage, indentation).
  - Une accumulation de mauvaises pratiques rend le code difficilement maintenable.
- Stabilité du code :
  - Mesure la fréquence des changements dans le code source ;
  - Des fichiers souvent modifiés peuvent indiquer une instabilité ou une mauvaise conception.

Indicateurs de productivité et d'efficacité

- Temps moyen de correction des défauts :
  - Durée moyenne entre la détection d'un bug et sa résolution ;
  - Un temps élevé peut indiquer des problèmes de complexité ou de documentation.
- Temps moyen de livraison :
  - Temps nécessaire pour livrer une nouvelle fonctionnalité ou une correction ;
  - Cela reflète l'efficacité de l'équipe et la qualité des processus.
- Fréquence de livraison :
  - Nombre de livraisons (mises à jour ou correctifs) réalisées dans une période donnée ;
  - Une fréquence élevée, sans dégradation de la qualité, est un signe positif.

Outils : Jira, Trello, GitHub Insights, GitLab Analytics

Indicateurs d'usage et de comportement

- MTBF (Mean Time Between Failures) :
  - Temps moyen entre deux défaillances du logiciel ;
  - Indique la fiabilité du système en production.
- MTTR (Mean Time To Repair) :
  - Temps moyen pour réparer une défaillance ;
  - Un MTTR élevé peut indiquer un manque de documentation ou une conception complexe.
- Taux d'incident répétitif :
  - Proportion de bugs ou incidents récurrents dans le système ;
  - Une forte proportion indique des problèmes de maintenance sous-jacents.

Indicateurs spécifiques à la maintenance adaptative

- Effort d'adaptation :
  - Temps nécessaire pour modifier le logiciel afin de répondre à de nouvelles exigences (par exemple, compatibilité avec une nouvelle plateforme).
- Nombre de dépendances externes :
  - Les dépendances à des bibliothèques tierces ou des technologies spécifiques peuvent rendre les adaptations plus complexes.

Synthèse : les 5 indicateurs clés

- Complexité cyclomatique : pour évaluer la complexité du code ;
- Taux de couverture des tests : pour garantir la robustesse et réduire les régressions ;
- MTTR et MTBF : pour mesurer la fiabilité et la rapidité des corrections ;
- Duplication de code : pour éviter la multiplication des points de maintenance ;
- Endettement/Dette technique : pour prioriser les efforts de maintenance.

Comment utiliser ces indicateurs ?

- Analyse régulière avec des outils (SonarQube, CodeClimate) ;
- Tableaux de bord pour suivre les indicateurs clés au fil du temps ;
- Priorisation des actions en fonction de l'impact des problèmes sur les utilisateurs ou l'équipe.

Outils pour la mesure des indicateurs

Imaginons une équipe de dév. qui souhaite mesurer son temps moyen de livraison (lead time) et son temps moyen de correction des bugs (MTTR) :

1. Utiliser Jira pour suivre le temps entre la création d'une tâche et sa clôture ;
2. Utiliser GitHub Insights pour mesurer le temps nécessaire pour revoir et fusionner le code ;
3. Combiner les données dans Power BI pour obtenir un graphique comparatif.

Plan du cours

Qu'est-ce que la dette technique ?

- La dette technique désigne les compromis réalisés lors du développement, où des solutions rapides ou non optimales sont adoptées pour répondre à des contraintes de temps, de ressources ou d'autres priorités ;
- Ces choix peuvent accélérer la livraison initiale d'un produit mais entraînent des coûts supplémentaires à long terme, comme une complexité accrue, des défauts ou des retards dans les évolutions futures ;
- C'est une métaphore qui compare les efforts supplémentaires nécessaires pour améliorer un logiciel mal conçu ou mal entretenu à des intérêts accumulés sur une dette financière.

Exemples courants de dette technique

1. Code mal organisé ou non documenté ;
2. Absence ou insuffisance de tests unitaires ou automatisés ;
3. Utilisation de technologies obsolètes ;
4. Manque de normalisation dans l'architecture ou les pratiques de codage ;
5. Solutions provisoires non corrigées (ex. "quick fixes").

Comment mesurer la dette technique ?

Difficile à mesurer de manière absolue, mais plusieurs approches permettent d'évaluer son impact.

- Outils et métriques automatisés pour l'estimer en identifiant les problèmes dans le code source
  - SonarQube : évalue la dette technique en termes de "temps nécessaire pour corriger les problèmes" (Technical Debt Time) ;
  - CodeClimate, Lint, etc., pour identifier les duplications de code, violations de règles, complexité cyclomatique, etc.
- Temps de maintenance : le temps passé à corriger des bugs, adapter le code existant à de nouvelles versions de bibliothèques, … par rapport à celui consacré à écrire du nouveau code

Comment mesurer la dette technique ? -suite-

- Indicateurs subjectifs :
  - Feedback des développeurs : demander aux équipes de pointer les parties du code difficilement modifiables ou sources fréquentes de problèmes (code reviews) ;
  - Vélocité décroissante : Si la capacité d'une équipe à livrer des fonctionnalités diminue avec le temps, cela peut être un signe de dette technique.
- Coût du refactoring : ratio entre le temps passé sur des améliorations de qualité et celui consacré à des évolutions directes est un indicateur de dette technique ;
- Indicateurs de performance : temps d'exécution des tests unitaires, ou temps de déploiement ou de build.

Comment gérer et réduire la dette technique ?

1. Évaluation continue : intégrer des outils comme SonarQube dans les chaînes CI/CD pour surveiller la qualité en permanence.
2. Refactoring régulier : consacrer du temps spécifiquement pour réduire la dette technique, par exemple dans chaque sprint.
3. Code review : mettre en place des revues de code rigoureuses pour identifier les problèmes en amont.
4. Automatisation des tests : réduire la dette liée au manque de couverture de tests.
5. Documentation : améliorer la documentation pour réduire la dette cognitive associée à un code peu compréhensible.
6. Priorisation : évaluer l'impact business de la dette technique pour savoir quelles parties doivent être corrigées en priorité.

Dette technique dans SonarQube

- Elle est mesurée en termes de temps estimé nécessaire pour résoudre les problèmes identifiés dans le code ;
- Cette estimation est exprimée en heures, jours ou semaines et permet aux équipes de prioriser et de planifier le travail de refactoring ;
- Elle est calculée à partir de plusieurs types d'anomalies détectées par SonarQube :
    1. Bugs : défauts dans le code qui peuvent entraîner des comportements inattendus ou des pannes ;
    2. Vulnérabilités : problèmes de sécurité détectés dans le code ;
    3. Code Smells : mauvaises pratiques de développement (duplications, complexité élevée, etc.) ;
    4. Test Coverage : manque de tests pour certaines parties critiques du code.

Estimation de la dette dans SonarQube

- La dette technique totale est donnée par :

$$\textit{Technical Debt} = \Sigma(\textit{Cout de reparation des problemes}) \quad (1)$$

- Chaque type de problème a un coût de réparation estimé (par exemple, en minutes ou en heures), défini par les règles de qualité configurées dans SonarQube ;
- Exemple d'estimation pour un Code Smell :
  - Problème : une méthode dépasse 100 lignes ;
  - Coût estimé par SonarQube pour refactoriser cette méthode : 30 minutes ;
  - Si plusieurs méthodes sont concernées, la dette associée s'accumule.

Exemple de mesure

Supposons un projet analysé par SonarQube avec les résultats :

Type de problème	Nombre d'occurrences	Temps par occ.	Dette totale
Bugs	10	15 min	2.5 heures
Vulnérabilités	5	30 min	2.5 heures
Code Smells	50	10 min	8.3 heures
Duplications de Code	20	20 min	6.7 heures

- Dette technique totale : $2.5 + 2.5 + 8.3 + 6.7 = 20$ *heures*.

SonarQube affiche une estimation en jours-hommes, selon une conversion (par défaut 8 heures par jour).

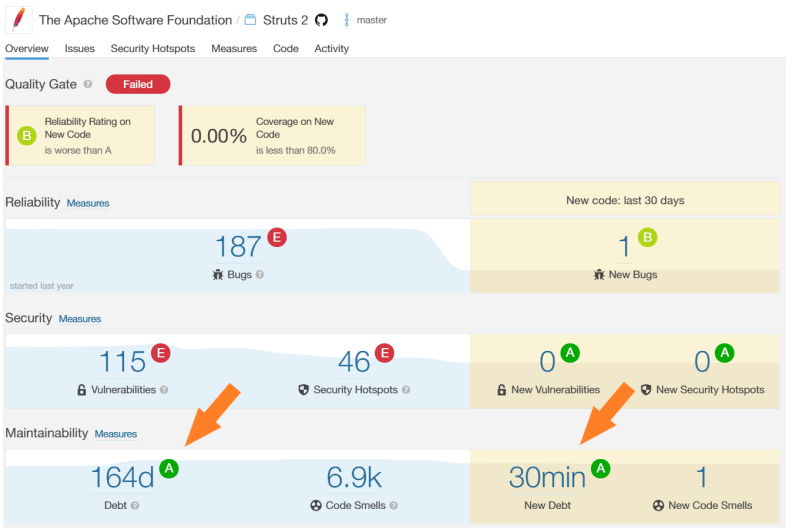
Ratio de dette technique

- SonarQube calcule également un ratio de dette technique, appelé *Maintainability Rating*, qui donne une perspective sur la qualité du projet
- Formule :
  $$\textit{Ratio de dette technique} = \frac{\text{Technical debt}}{\text{Effort total de developpement}}$$
- Si un projet a nécessité 100 jours de développement et présente 5 jours de dette technique :
  $$\textit{Ratio de dette technique} = \frac{5}{100} = 5\%$$
- SonarQube classe ensuite la maintainability sur une échelle de A (meilleure, ratio < 5%) à E (pire, ratio > 50%).

Visualisation dans SonarQube

Dans le tableau de bord de SonarQube :

1. il y a une section dédiée à la Maintainability avec :
   - Le temps total de la dette technique ;
   - Le ratio par rapport au projet.
2. Les problèmes spécifiques sont listés avec leur gravité (Blocker, Critical, Major, Minor, Info) et leur coût de réparation estimé.



Plan du cours

Phases d'un processus de maintenance

Les processus de maintenance suivent généralement les étapes :

1. Identification du besoin de maintenance :
   - Collecte des retours d'utilisateurs, des logs système, ou des résultats de tests ;
   - Identification des bugs, des inefficacités ou des besoins d'amélioration.
2. Analyse de la demande :
   - Évaluation de la demande pour déterminer sa faisabilité et son impact ;
   - Classification selon les types de maintenance (corrective, adaptative, etc.).
3. Planification :
   - Définition des priorités en fonction de la criticité des problèmes ;
   - Estimation des ressources nécessaires (temps, budget, équipe).

Phases d'un processus de maintenance

4. Conception et implémentation :
   - Rédaction de spécifications pour les changements ;
   - Développement et test des correctifs ou améliorations.
5. Validation et déploiement :
   - Tests de validation pour s'assurer que les modifications fonctionnent comme prévu ;
   - Déploiement dans l'environnement de production avec suivi des performances.
6. Documentation :
   - Mise à jour de la documentation technique et utilisateur pour refléter les modifications apportées.
7. Suivi et surveillance :
   - Surveillance continue pour détecter de nouveaux problèmes ou vérifier la performance des changements.

Méthodes et pratiques associées

- Gestion de configuration (vue précédemment) :
  - Permet de suivre les versions et les modifications du logiciel pour éviter les conflits ou pertes de données ;
  - Outils : Git, entre autres.
- Tests de maintenance : deux types de test
  - Tests de régression : Vérifient que les nouvelles modifications n'ont pas introduit de nouveaux bugs ;
  - Tests de performance : Évaluent l'impact des modifications sur les performances globales.

Méthodes et pratiques associées -suite-

- Suivi et automatisation :
  - Utilisation d'outils pour surveiller le comportement en production, comme Prometheus/Grafana ;
  - Automatisation des déploiements et des tests, avec des outils comme GitHub Actions.
- Gestion des incidents :
  - Réponse rapide aux incidents critiques via des processus clairs ;
  - Utilisation d'outils comme Datadog pour alerter les équipes.

Outils pour la maintenance

- Outils de gestion des demandes et des correctifs : Jira, Bugzilla, Trello ;
- Outils de supervision : Datadog, Promotheus/Grafana, New Relic ;
- Outils de tests : Cypress ou Selenium (tests fonctionnels), JUnit (tests unitaires) ;
- Outils de documentation : Confluence, Notion.

Exemple de processus

Une application mobile rencontre des crashs sur certains appareils après une mise à jour.

1. Identification :
   - Les utilisateurs signalent le problème via un système de tickets (ex. Jira) ;
   - Les logs montrent des erreurs liées à une bibliothèque obsolète.
2. Analyse : l'équipe technique analyse l'erreur et découvre que la bibliothèque utilisée est incompatible avec certains systèmes ;
3. Planification : la mise à jour de la bibliothèque est planifiée comme une tâche prioritaire ;

Exemple de processus -suite-

4. Implémentation :
   - La bibliothèque est mise à jour et le code ajusté ;
   - Les modifications sont testées localement.
5. Validation : les tests sont effectués sur différents appareils pour vérifier la correction ;
6. Déploiement : une nouvelle version de l'application est publiée ;
7. Documentation : les détails de la mise à jour et des tests sont documentés ;
8. Suivi : l'application est surveillée pour s'assurer que le problème est bien résolu.

Demande de correctif (Bug Report) – Structure type

1. Résumé : une phrase concise décrivant le problème.
   - Exemples : “Erreur 500 lors de la soumission d’un formulaire”, “Crash de l’application sur Android 11 lors du lancement”
2. Description détaillée :
   - Contexte : Où et comment le problème est survenu. Exemple : "Le problème se produit dans la page de paiement lorsque l’utilisateur clique sur ’Valider’."
   - Étapes pour reproduire : un guide clair permettant de reproduire le problème. Exemple : "Étape 1 : aller sur l’écran X, ..."
   - Résultat actuel : Ce qui se passe réellement. Exemple : "Le formulaire génère une erreur 500 et la soumission échoue."
   - Résultat attendu : Ce qui aurait dû se produire. Exemple : "Le formulaire devrait être soumis avec succès."
   - Fréquence : Indiquer si le problème est constant ou intermittent. Exemple : "Le problème survient 8 fois sur 10."

Demande de correctif (Bug Report) – Structure type

3. Pièces jointes :
   - Captures d'écran ou vidéos : Illustrent le problème ;
   - Logs ou traces : Fournissent des détails techniques sur l'erreur ;
   - Fichiers spécifiques : S'ils sont pertinents (exemple : fichier de configuration).
4. Détails techniques :
   - Plateforme : Système d'exploitation et version. Ex : "Android 11, Samsung Galaxy S21" ou "Chrome 114 sur Windows 10." ;
   - Version du logiciel : Numéro de version de l'application ou du composant concerné. Ex : "Version 1.3.5." ;
   - Environnement : Indiquer si le problème survient en production, en test, ou en développement. Ex : "Environnement : Production.".

Demande de correctif (Bug Report) – Structure type

5. Gravité et priorité
   - Gravité (Severity) : Impact du problème sur les utilisateurs. Exemple : Mineur, Majeur, Bloquant.
   - Priorité (Priority) : Urgence de la résolution selon les objectifs du projet. Exemple : Faible, Normale, Élevée.
6. Catégorie ou composant concerné : Identifier le module, la page, ou la fonctionnalité impactée. Exemple : "Module de gestion des utilisateurs."
7. Liens et historique
   - Tickets liés : Référencer d'autres tickets associés au problème. Ex : "Ce problème pourrait être lié au ticket [ABC-123]."
   - Changements récents : Indiquer si des modifications récentes pourraient être à l'origine. Ex : "Le problème est apparu après la mise à jour 1.3.0."

Exemple de demande de correctif (bug report)

- Résumé : Erreur 500 lors de la soumission du formulaire de paiement.
- Description :
  - Contexte : Le problème survient lorsque l'utilisateur tente de soumettre un formulaire dans la section "Paiement".
  - Étapes pour reproduire :
    1. Se connecter avec un compte utilisateur valide ;
    2. Naviguer vers la section "Paiement" ;
    3. Remplir le formulaire et cliquer sur "Valider".
  - Résultat actuel : Le serveur retourne une erreur 500.
  - Résultat attendu : La soumission devrait réussir et afficher une confirmation de paiement.
  - Fréquence : Le problème se produit systématiquement.

Exemple de demande de correctif -suite-

- Pièces Jointes : Screenshot_500_ Error.png, Logs_2024-12-27.txt ;
- Plateforme : Chrome 114 sur Windows 10 ;
- Version du logiciel : Version 1.3.5 ;
- Gravité : Bloquant ;
- Priorité : Élevée ;
- Liens : Potentiellement lié au ticket [BUG-456].

Que faire de cette demande de correctif ?

Étape 1 : Analyse initiale du bug

1. Validation des informations :
   - Vérifiez que le rapport est complet (description, étapes pour reproduire, gravité, etc.) ;
   - Assurez-vous que les pièces jointes (captures d'écran, logs) sont suffisantes pour analyser le problème.
2. Reproduction du problème :
   - Confirmez l'existence du bug en suivant les étapes décrites dans le rapport ;
   - Si le bug n'est pas reproductible, demandez des clarifications ou des données supplémentaires à la personne qui l'a signalé.

Que faire de cette demande de correctif ?

Étape 2 : Analyse d'impact

1. Évaluer la portée :
   - Nombre d'utilisateurs affectés : Tous, un segment spécifique ?
   - Environnement impacté : Production (critique), Test (- urgent) ;
   - Modules ou fonctionnalités liés : Identifiez les parties du code ou modules affectés.
2. Impact sur le business :
   - Ce bug bloque-t-il des transactions ou des fonctionnalités critiques ?
   - Y a-t-il une perte financière, des plaintes clients ou des risques pour la réputation de l'entreprise ?
3. Complexité de la résolution :
   - Analysez les logs et identifiez la cause potentielle du problème (e.g., serveur, base de données, logique métier).
   - Temps estimé pour corriger : S'agit-il d'une correction rapide ou nécessitant une refactorisation importante ?

Que faire de cette demande de correctif ?

Étape 3 : Priorisation

- Évaluer la gravité et la priorité :
  - Gravité : Le problème est bloquant (erreur 500 en production) ;
  - Priorité : Probablement élevée ou critique dans cet exemple.
- Placer dans la roadmap :
  - Si le problème est bloquant, il doit être traité immédiatement en dehors de la planification normale ;
  - Si non bloquant, le planifier dans un sprint futur selon la priorité.

Que faire de cette demande de correctif ?

Étape 4 : Planification de la correction

- Assigner un développeur :
  - Désigner un développeur compétent pour résoudre le problème ;
  - Ajouter des détails ou des tâches spécifiques au ticket pour guider la correction.
- Définir un délai : Estimer le temps nécessaire pour la correction et communiquer une date limite.

Que faire de cette demande de correctif ?

Étape 5 : Correction et validation

- Corriger le bug
  - Le développeur applique les modifications nécessaires au code ;
  - Inclure des tests unitaires ou d'intégration pour valider la correction.
- Tester la correction :
  - Tester en environnement de développement ou de test ;
  - Vérifier que le problème est résolu sans introduire de régressions.
- Feedback : ajoutez des commentaires ou des notes une fois le problème corrigé.

Que faire de cette demande de correctif ?

Étape 6 : Déploiement

- Planifier la mise en production
  - Déployer la correction en production lors d'une prochaine release ou via un hotfix ;
  - Informer les parties prenantes que le problème est résolu.
- Surveillance post-déploiement : s'assurer que le bug ne réapparaît pas et qu'il n'y a pas d'effets secondaires.

Sur l'exemple précédent

1. Impact :
   - Le bug bloque le paiement, ce qui peut entraîner une perte de revenus ;
   - Gravité élevée, priorité critique.
2. Correction immédiate :
   - Un développeur est assigné pour résoudre rapidement l'erreur 500 ;
   - Le problème semble lié à une mauvaise validation côté serveur, nécessitant une correction dans l'API.
3. Validation :
   - Tester les paiements dans des environnements de test avec différents scénarios ;
   - Ajouter des tests pour éviter que ce problème ne réapparaisse.