**Chapitre 2 –** Les processus de la maintenance

La vision traditionnelle du cycle de vie du logiciel a desservie la maintenance en la décrivant uniquement comme une seule étape à la fin du cycle. Par conséquent, la maintenance des logiciels devrait avoir son propre modèle de cycle de vie. Trois caractéristiques communes des modèles CVML sont trouvées dans la littérature :

* comprendre le code ;
* la modification du code ; et
* revalidation du code.

D'autres modèles considèrent le développement de logiciels en tant que processus itératifs et basés sur l’idée de « changement mini-cycle » (change mini-cycle en anglais).

* Modèles itératifs. Les modèles itératifs partagent l’idée qu’un ensemble complet de
besoins relatifs à un système ne peuvent pas être complètement compris, ou les développeurs ne savent pas comment construire le système complet. Par conséquent, les systèmes sont construits en versions, dont chacune est un raffinement des exigences de la version précédente. Une construction est affinée en tenant compte de la rétroaction des utilisateurs.
* changement mini-cycle. Ces modèles se composent de cinq grandes phases : demande de changement, analyser et planifier le changement, mettre en œuvre le changement, vérifier et valider, et le changement de la documentation. Dans ce modèle de processus, plusieurs activités importantes ont été identifiées, telles que la compréhension du programme, l'analyse d'impact, le refactoring, et la propagation du changement.

Un autre type de modèle d'évolution du logiciel, appelé modèle de maintenance et d’évolution à étapes proposé par Rajlich et Bennett[[1]](#footnote-1). Son objectif principal est d'améliorer la compréhension de l’évolution des logiciels de longue durée de vie. Le modèle considère quatre étapes successives, distinctes de la durée de vie d'un logiciel :

1. Le développement initial. Lorsque la version initiale du système est produite, à ce stade l’architecture du système émerge et se stabilise rapidement.
2. L’Évolution. Après la stabilité initiale, il est facile d'effectuer des modifications simples au système. Des changements importants impliquent un coût plus élevé et un risque plus élevé.
3. L’Entretien. Les développeurs se concentrent principalement sur les tâches de maintenance, telles que la correction de bugs, alors que les modifications architecturales sont rarement effectuées.
4. Retrait. L'organisation décide de remplacer le système pour diverses raisons : (i) il est trop coûteux d’entretenir le système ou (ii) il existe une solution plus récente disponible. Passer d'un système existant difficile à maintenir à un système de solution moderne a ses propres défis impliquant la préparation et la migration des données.
5. **Réingénierie (REENGINEERING) :**

La Réingénierie implique un cycle unique consistant à générer un nouveau système à partir d’un système existant.

Reengineering= Reverse engineering + Δ + Forward engineering.

Le premier élément "reverse engineering" est l'activité de la définition d'une représentation plus abstraite et plus facile à comprendre du système. Par exemple, si l’entrée du processus de reverse engineering est le code source du système, la sortie sera l'architecture du système.

Le troisième élément " Forward engineering " est le processus traditionnel qui permet de passer d'une abstraction de haut niveau, à l’implémentation physique du système. Le deuxième élément représente les modifications effectuées au système d'origine.

1. **LEGACY SYSTEMS**

Un « legacy system » est un ancien logiciel qui continue d'être utilisé parce qu’il répond toujours aux besoins des utilisateurs, en dépit de la disponibilité des technologies plus récentes ou de méthodes plus efficaces d'exécution de la tâche. Plus souvent, un Legacy system comprend des procédures ou une terminologie obsolète, et il est très difficile pour les nouveaux développeurs de comprendre le système.

Pour gérer les Legasy systems, un certain nombre d'options sont disponibles.

* **Gel**. Une organisation décide de ne pas effectuer d'autres travaux sur un Legacy system. Cela implique que soit les services du système ne sont plus nécessaires ou un nouveau système remplace complètement un système existant.

Une organisation peut décider que le support du legacy system ne fait plus partie de sa spécialisation.

* **Effectuer la maintenance**. Dans cette approche, l'organisation continue à maintenir
le système pour une autre période de temps, malgré toutes les difficultés à le faire.
* **Abandonner et redévelopper**. Dans cette approche, le logiciel est redéveloppé de nouveau à partir de zéro, en utilisant de nouvelles plates-formes matérielles et logicielles.
* **Encapsulation**. Dans cette approche, un Legacy system est encapsulé par une nouvelle couche logicielle, cachant ainsi la complexité indésirable des données existantes, des programmes individuels, et des interfaces. L'ancien système exécute les calculs actuels, mais les utilisateurs interagissent avec le système de façon conviviale.
* **Migration**. Dans cette approche, un Legacy system opérationnel est déplacé vers un nouveau matériel et / ou plate-forme logicielle, tout en conservant la fonctionnalité du Legacy system.
1. **ANALYSE D’IMPACT**

L’analyse d'impact est la tâche d'identification des parties du logiciel qui peuvent potentiellement
être affectés si un changement proposé au système est effectué. Le résultat de l'analyse d'impact
peut être utilisé lors de la planification des changements, durant les changements, et le suivi des
effets des changements afin de localiser les sources de nouvelles failles.

L’analyse d’impact décrit comment mener, à un coût efficace, une analyse complète d’impact d’un changement dans un logiciel existant. Les mainteneurs doivent posséder une connaissance intime de la structure et du contenu du logiciel. Ils utilisent ce savoir pour faire l’analyse d’impact.

Parmi les principaux objectifs de l’analyse d’impact, on cite :

* Détermination de la portée d’un changement pour établir un plan et implanter le travail ;
* Développement d’estimations justes des ressources nécessaires pour effectuer le travail ;
* L’analyse de coûts/bénéfices du changement demandé.

L’analyse d’impact du changement est effectuée principalement lors de la maintenance des programmes, en vue d’évaluer les effets du changement sur le reste du système et de réduire le risque de s’embarquer dans des dépenses coûteuses. Cette évaluation englobe l’estimation des ressources humaines, financières, temps, efforts et plannings nécessaires pour accomplir le changement.

Les techniques de l’analyse d'impact peuvent être classées en deux catégories comme suit :

* Analyse de traçabilité.
* Analyse de la dépendance.

**4 LE REFACTORING**

Le Refactoring consiste à effectuer des changements à la structure du système logiciel pour le rendre plus facile à comprendre et moins coûteux par rapport aux modifications ultérieures sans modifier le comportement du système. Le Refactoring est réalisé par la suppression du code dupliqué, la simplification du code, et le déplacement du code d’une classe à une autre classe. Il peut être réalisé sans ajouter de nouvelles exigences au système existant.

Dans les méthodologies logicielles agiles, tels que eXtreme Programming (XP), le refactoring est appliqué en continu pour : (i) stabiliser l'architecture du logiciel ; (Ii) rendre le code lisible; et (iii) rendre les tâches d'intégration de nouvelles fonctionnalités dans le système plus flexibles.

Les techniques de refactoring mettent l'accent sur le développement d'une liste de refactoring de base, qui peuvent être combinés pour former un refactoring complexe. La liste originale des refactoring de base contenant des transformations sur le code orienté objet : (i) ajouter
une classe, une méthode ou un attribut ; (ii) renommer une classe, une méthode ou un attribut ; (Iii) déplacer un attribut ou méthode de haut ou en bas de la hiérarchie ; (Iv) supprimer une classe, une méthode ou un attribut ; et (v) extraire des morceaux de code dans des méthodes distinctes.

**5 COMPEHENSION DU PROGRAMME**

Le but de la compréhension du programme est de comprendre un système logiciel existant
pour la planification, la conception, le codage, et les changements de test. T. A. **Corbi** [[2]](#footnote-2) a constaté en 1989 que la compréhension du programme compte pour 50% de l’effort total dépensé tout au long du cycle de vie d'un système logiciel.

Dans le domaine de la compréhension du programme, un modèle mental décrit la représentation mentale du programmeur sur le programme à comprendre.

Une étape clé dans le développement de modèles mentaux est la génération des hypothèses ou des conjectures, et de vérifier leur validité. Les hypothèses sont un moyen pour le programmeur pour comprendre le code d'une manière progressive (incrémentale). Après une certaine compréhension du code, le programmeur fait une hypothèse et la vérifie par lecture de code. La vérification de l'hypothèse consiste soit à accepter l'hypothèse ou à son rejet.

On peut appliquer plusieurs stratégies pour arriver à des hypothèses significatives, telles que
bottom-up, top-down, ou des combinaisons des deux stratégies. Une stratégie bottom-up
fonctionne en commençant par le code, alors qu'une stratégie descendante fonctionne en travaillant à partir d'un objectif de haut niveau.

**6 Réutilisation de logiciel**

Le « software reuse » signifie l’utilisation du savoir logiciel existants ou des artefacts de logiciels existants lors du développement d'un nouveau système. Les parties réutilisables peuvent inclure des artefacts et le savoir logiciel. A noter que la réutilisation ne se limite pas à des fragments de code source. **Capers Jones**[[3]](#footnote-3) a identifié quatre grands types d'artefacts pour la réutilisation :

* la réutilisation des données : Cela implique une normalisation du format de données. Les fonctions réutilisables impliquent un format d'échange de données standard.
* Réutilisation architecturale. Il s'agit de standardiser un ensemble de conventions de conception et de programmation portant sur l'organisation logique du logiciel. L'objectif est de définir un ensemble complet d'éléments fonctionnels qui seront nécessaires pour créer de nouveaux systèmes à partir de composants standard.
* Réutilisation de la conception. Il s'agit de la réutilisation de concepts abstraits qui n'incluent pas les détails de l’implémentation. Ces concepts seront ensuite implémentés selon les besoins de l’application.
* Réutilisation de programmes : concerne la réutilisation du code (COTS – commercial off-the-shelf components).

La réutilisabilité est une propriété des logiciels qui indique le degré avec lequel le logiciel peut être réutilisé.

1. V. T. Rajlich and K. H. Bennett. 2000. A staged model for the software life cycle.IEEE Computer, July, pp. 2–8 [↑](#footnote-ref-1)
2. T. A. Corbi. 1989. Program understanding: challenge for the 1990s.IBM Systems Journal,

28(2), pp. 294–306. [↑](#footnote-ref-2)
3. T. C. Jones. 1984. Reusability in programming: a survey of the state of the art. IEEE

Transactions of Software Engineering, 10(5), 488–494. [↑](#footnote-ref-3)