Contenu de la matière :

Chapitre 1. Introduction

1. Définitions et objectifs
2. Principes du Génie logiciel
3. Qualités attendues d'un logiciel
4. Cycle de vie d'un logiciel
5. Modèles de cycle de vie d'un logiciel

Chapitre 2. Modélisation avec UML

1. Introduction :
   Modélisation, Modèle, Modélisation Orientée Objet, UML en application.
2. Eléments et mécanismes généraux
3. Les diagrammes UML
4. Paquetages

Chapitre 3. Diagramme UML de cas d'utilisation : vue fonctionnelle

Intérêt et définition, Notation

Chapitre 4. Diagramme UML de classes et d'objet : vue statique

1. Diagramme de classes
2. Diagramme d'objets

Chapitre 5. Diagrammes UML : vue dynamique

1. Diagramme d'interaction (séquence t collaboration)
2. Diagramme d'activité
3. Diagramme d'état transition

Chapitre 6. Autres notions et diagrammes UML

1. Composants, déploiement, structures composite.
2. Mécanismes d'extension : langage OCL + les profils.

Chapitre 7. Introduction aux méthodes de développement : (RUP, XP)

Chapitre 8. Patrons de conception et leur place au sein du processus de développement

Référence :

- Bern Bruegge and Allen H. Dutoit, Object-Oriented Software Engineering – using UML, Patterns and Java. Third Edition, Pearson, 2010.
- G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language (UML) reference Guide, Addison-Wesley, 1999.
- G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language (UML) User Guide, Addison-Wesley, 1999.
- G. Booch et al., "Object-Oriented Analysis and Design, with applications. Addison-Wesley, 2007.
- Laurent Audibert. Cours UML 2.0, disponible sur http://www.developpez.com.
- M. Blaha et J. Rumbaugh. Modélisation et conception orientées objet avec UML 2. 2ème édition. Pearson Education, 2005.
- Pierre-Alain Muller. Modélisation objet avec UML. Éditions Eyrolles, 2003.
- Shari Lawrence Pfleeger and Joanne M. Atlee. Software Engineering. Fourth Edition, Pearson, 2010

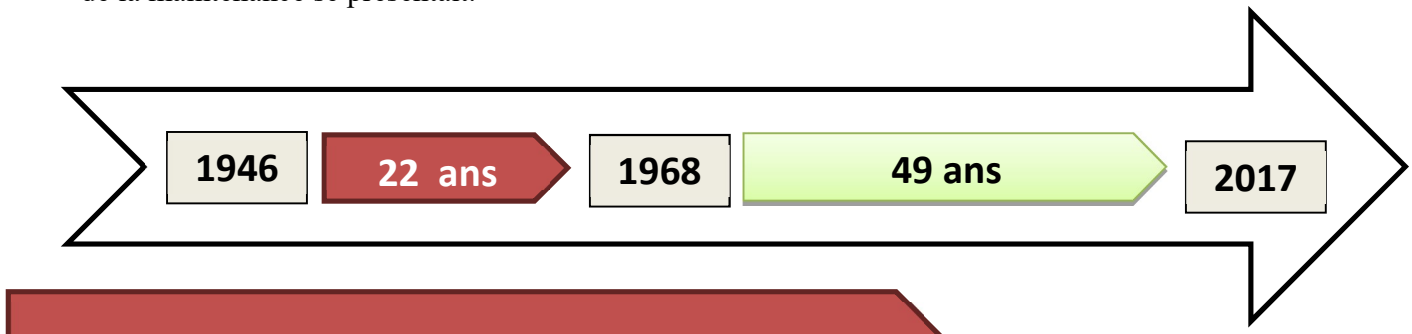
Chapitre 1 : Introduction au Génie logiciel

I. Introduction

Le G.L génie logiciel (*Software engineering*) est une discipline relativement nouvelle. Le terme G.L a été évoqué pour la première fois en 1968 où l'OTAN a coordonné sa mise en place. L'objectif était de promouvoir le développement des logiciels pour répondre aux besoins devenus de plus en plus élevés et complexes et de pallier aux problèmes caractérisant les logiciels de la première ère de l'informatique.

I.1. Contexte général (crise du logiciel)

Le génie logiciel est apparu dans les années 1968 sous la coordination de l'OTAN pour répondre à la crise du logiciel (1968). À cette époque les logiciels n'étaient pas fiables et il était difficile pour les développeurs de fournir dans les délais une application respectant les spécifications du cahier des charges. Même quand le logiciel était bien construit, la difficulté de la maintenance se présentait.



De 1946 à 1968 : Naissance de l'informatique

- Aucune approche méthodologique de développement.
- Programmer et corriger les bugs.
- L'évolution des besoins implique la réécriture des programmes.
- Les logiciels devenaient de plus en plus complexes et de moins en moins fiables.

I.2. Enjeux et risques

Un certain nombre de logiciels fournis (notamment dans la période prématuré de l'informatique) a entraîné plusieurs conséquences matérielles et humaines. La littérature du G.L fait référence à plusieurs dégâts dont on peut citer, à titre indicatif :

- **Le projet TAURUS :** Le projet d'informatisation de la bourse de Londres. Il a été abandonné définitivement après quatre années de labeur et a engendré environ 100 millions de livres de perte.
- **La mission VENUS :** La sonde Mariner qui devait effectuer un passage à 5 000 km de Vénus s'est perdue à 5 000 000 km de ladite planète à cause d'une mauvaise configuration (remplacement d'une virgule par un point : dans le programme Fortran le point avait été remplacé par une virgule (format US des nombres)).
- **Ariane V :** Le 4 juin 1996, lors du premier lancement de la fusée Ariane V, celle-ci explose en vol coûtant ainsi plus de 500 millions de dollar. La cause : un logiciel utilisé sous Ariane IVet intégré sans nouvelle validation dans ArianeV (Ariane V ayant des moteurs plus puissants s'incline plus rapidement qu'Ariane IV, pour récupérer l'accélération due à la rotation de la Terre. Les capteurs ont bien détecté cette inclinaison d'Ariane V, mais le logiciel l'a jugée non conforme à son plan de tir, et a provoqué l'ordre d'autodestruction alors que tout se passait bien).
- **Le Therac-25 :** un appareil d'irradiation thérapeutique a provoqué la mort de 2 personnes, et l'irradiation de 4 autres, à cause d'une erreur logicielle.

Pour mieux comprendre l'enjeu, le *Standish Group*[1] a mené en 1995 une étude sur 8380 projets. Les résultats révèlent le comportement des projets informatiques suivant :

- **16%** des applications sont construites avec succès ;
- **53%** des projets aboutissent mais posent des problèmes tels : la diminution des fonctionnalités fixées de départ, le non-respect des délais spécifiés ou encore l'augmentation des coûts ;

[1] Une société indépendante spécialisée dans l'IT, connue par ses rapports et statistiques sur les projets de mise en œuvre des systèmes d'information dans le secteur public et privé.

- **31%** des projets sont abandonnés.

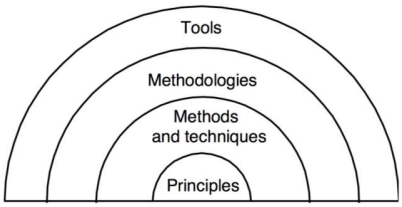
Bien que le G.L a amélioré la situation, il reste moins avancé que les autres domaines de l'ingénierie (tel que le génie civil). Cela peut être dû à plusieurs raisons différentes :

- Le logiciel étant un objet immatériel, il est très facilement modifiable.
- Les défauts d'un logiciel résultent principalement des erreurs de manipulation effectuées par les hommes. Ce qui rend le contrôle très difficile.
- Il est difficile de figer les caractéristiques d'un logiciel au départ...

I.3. Définitions

Le génie logiciel est un domaine de l'ingénierie qui permet la conception, la réalisation et la maintenance des systèmes logiciels de qualité. D'une façon générale, le G.L est le domaine qui couvre les méthodes, la modélisation, les techniques, les outils, les activités, les biens livrables et la gestion de projets relatifs au développement et à la maintenance du logiciel.

I.4. Principes du G.L



Le génie logiciel est une discipline très vaste, liée à tous les domaines de l'informatique. Il définit des principes de développement, établit des règles à suivre, invente des techniques, développe des méthodes et propose des outils pour garantir la satisfaction de critères ou principes établis. Ces principes sont très abstraits et ne sont pas utilisables directement, mais ils font partie du vocabulaire de base du génie logiciel. Ces principes sont :

1. Rigueur

Bien que la production de logiciel est une activité créative, les activités logicielles doivent être réalisées rigoureusement : suivi des processus adaptés, utilisation correcte des techniques adaptées, fourniture des livrables prévus (documents, modèles, code), validation de toutes les livraisons,…

2. Séparation des problèmes

Afin de maitriser la complexité d'un P.D.L, le G.L tente de se concentrer sur un seul aspect du problème à la fois et le traiter de façon indépendante que ce soit dans le *temps* (ex :

séparation des phases de développement : notion de cycle de vie) ou la séparation des *qualités* que l'on cherche à optimiser à un stade donné (ex : se concentrer uniquement sur la sécurité),…etc. Il faut choisir des aspects suffisamment indépendants (les aspects trop liés doivent être traités ensemble).

3. Modularité

Consiste à diviser le projet global en un ensemble de modules de moindre complexité. Chaque module traite une partie du problème. L'ensemble de ces modules doivent être compréhensibles, homogènes, indépendants. Un bon découpage modulaire se caractérise par une *forte cohésion* interne des modules et un *faible couplage* entre les modules (relations inter modulaires en nombre limité et clairement décrites).

4. Abstraction

L'abstraction consiste à ne considérer que les *aspects jugés importants* d'un système à un moment donné, en faisant abstraction des autres aspects.

5. Anticipation du changement

Un logiciel est très souvent soumis à des *changements* (corrections, évolutions). Ceci requiert des efforts particuliers pour *prévoir*, faciliter et gérer ces évolutions inévitables. Il faut par exemple :

- Faire en sorte que les changements soient les plus localisés possibles (bonne modularité),
- Etre capable de gérer les multiples versions des modules et configurations des versions des modules, constituant des versions du produit complet.

6. Généricité

Il est parfois avantageux de remplacer la résolution d'un problème spécifique par la résolution d'un problème plus général. Cette solution générique (paramétrable ou adaptable) pourra être *réutilisée* plus facilement. Par exemple plutôt que d'écrire une identification spécifique à un écran particulier, écrire (ou réutiliser) un module générique d'authentification (saisie d'une identification - éventuellement dans une liste - et éventuellement d'un mot de passe).

7. Construction incrémentale

Un procédé incrémental atteint son but par étapes en s'en approchant de plus en plus ; chaque résultat est construit en étendant le précédent. On peut par exemple réaliser d'abord un noyau des fonctions essentielles et ajouter progressivement les aspects plus secondaires. Ou encore, construire une série de prototypes simulant plus ou moins complètement le système envisagé.

I.5. Qualités d'un logiciel :

Le G.L procède en plusieurs étapes dans la fabrication d'un programme informatique. Ceci permet de garantir la **qualité** du produit et le **respect des besoins** de l'utilisateur tout en respectant **les délais et les coûts fixés** au départ. Après plusieurs travaux, les chercheurs ont défini la qualité d'un logiciel à partir de plusieurs facteurs, notamment :

1. **Correction / Utilité / conformité / Validité :** Adéquation entre le besoin effectif des utilisateurs et les fonctions offertes par le logiciel. Autrement dit : « la capacité que possède un logiciel de mener à bien sa tâche telle qu'elle a été définie par sa spécification ou dans le cahier des charges ».
2. **Fiabilité :** Aptitude à assurer de manière continue le service attendu (pas de panne)
3. **Robustesse :** La capacité de réagir de manière appropriée à la présence de conditions anormales
4. **Extensibilité / adaptabilité :** La capacité d'adapter des produits logiciels aux changements de spécification.
5. **Réutilisabilité :** La capacité des éléments logiciels à servir à la construction de plusieurs applications différentes.
6. **Compatibilité :** La facilité avec laquelle des éléments logiciels peuvent combinées à d'autres.
7. **Efficacité :** La capacité d'un logiciel à utiliser le minimum de ressources matérielles (temps CPU, mémoire, espace, bande passante,…).
8. **Facilité d'utilisation et d'apprentissage :** La facilité avec laquelle des personnes (avec des formations et des compétences différentes) peuvent apprendre à utiliser le logiciel et s'en servir pour résoudre des problèmes.

En plus de ces facteurs, d'autres peuvent être mentionnés en l'occurrence la **ponctualité**, **traçabilités, vérifiabilité, intégrité,….**

☞ Il faut noter que certains des facteurs précédents sont contradictoires lors de leur réalisation effective. On doit faire un choix en fonction du contexte.

I.6. Cycles de vie d'un logiciel

Pour livrer un logiciel de qualité, il faut en maitriser le processus de son élaboration. Ce processus est composé de différentes étapes dont la succession forme le cycle de vie du logiciel (C.V.L).

La plupart des modèles des processus reprennent les activités fondamentales mais les organisent différemment

I.6.1. Composantes du C.V.L

Le cycle de vie du logiciel comprend au minimum les étapes suivantes :

I.6.1.1. Spécification

Consiste à définir les fonctionnalités que doit offrir le logiciel en satisfaisant les exigences des utilisateurs (collecte des exigences) et l'ensemble des contraintes ou les taches qui se répètent dans le domaine du logiciel (analyse du domaine).

I.6.1.2. Conception

Déterminer la façon dont le logiciel fournit les différentes fonctionnalités recherchées ; En définissant d'abord la structure du système (conception architecturale) ensuite déterminer la façon dont les différentes parties du système agissent entre elles (conception des interfaces) et finalement les algorithmes pour ces différentes parties (conception détaillée).

I.6.1.3. Codage (implémentation ou programmation)

C'est la traduction dans un langage de programmation des fonctionnalités définies.

I.6.1.4. Tests

Tests unitaires : Ils permettent de vérifier individuellement que chaque sous-ensemble du logiciel est implémenté conformément aux spécifications.

Intégration : S'assurer de l'interfaçage des différents éléments (modules) du logiciel.

Validation (qualification ou recette) : La vérification de la conformité du logiciel aux spécifications initiales.

I.6.1.5. Déploiement (Mise en production / Livraison)

Rendre le logiciel opérationnel sur le site du client (installation), enseigner aux utilisateurs à se servir du logiciel (formation) et répondre aux questions des utilisateurs (assistance)

I.6.1.6. Maintenance

Elle comprend toutes les actions correctives et évolutives sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de C.V.L entre le client et l'équipe de développement. Le C.V.L permet de prendre en compte, en plus des aspects techniques, l'organisation et les aspects humains.

I.6.2.Modèle de cycle de vie

Le C.V.L introduit une vision temporelle du processus dont tous logiciel suit depuis sa naissance jusqu'au son obsolescence. Ce cycle de vie se manifeste selon plusieurs manière ou modèle dont voici quelques exemples :

I.6.2.1.Modèle de cycle de vie en cascade

Le modèle en cascade est modèle très ancien (les années 70). Son principe est basé sur use séquence de phase où chaque phase se termine à une date précise par la production de certains documents. Les résultats sont soumis à une revue approfondie et on ne passe à la phase suivante que s'ils sont jugés satisfaisants (chaque étape est validée).

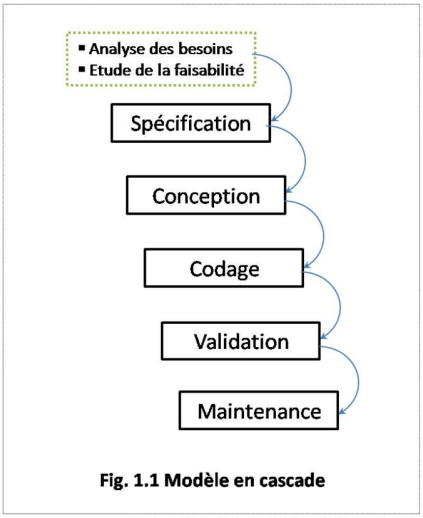
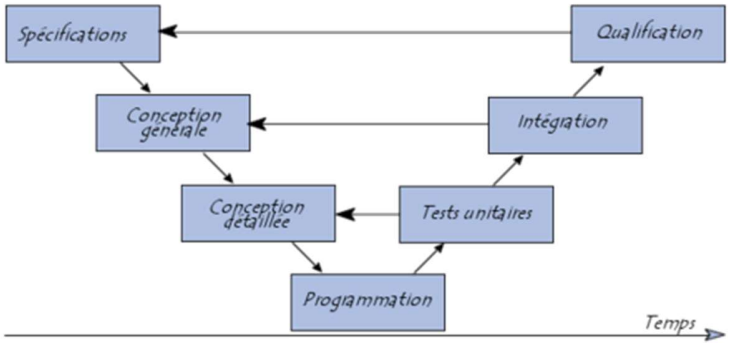


Fig. 1.1 Modèle en cascade

NB : Un autre modèle appelé modèle en cascade avec retour est dérivé du modèle original où il introduit la possibilité de retour en arrière.

I.6.2.2.Modèle de cycle de vie en V

Ce modèle présente une structuration de la phase de validation où les tests sont définis à la fin de chaque phase (i.e. le développement des tests et du logiciel sont effectués de manière synchrone).



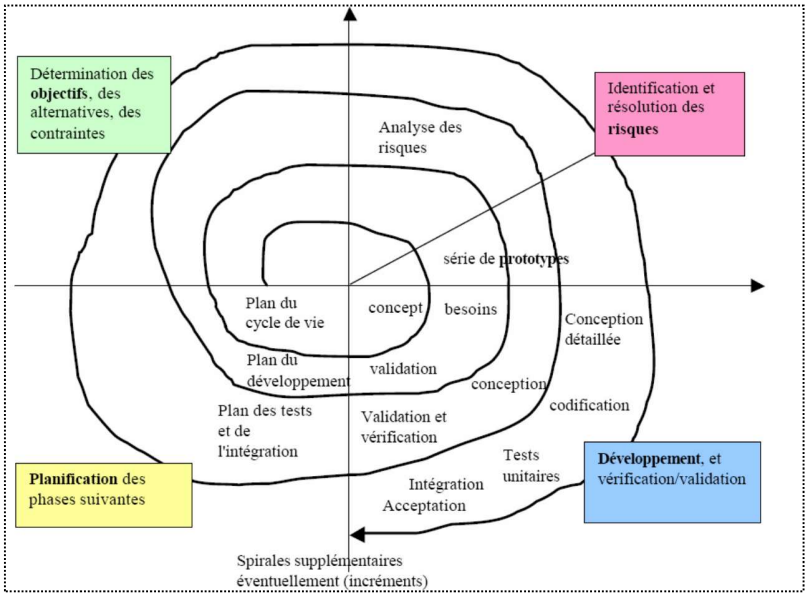
Ceci rend explicite la préparation des dernières phases (vérification-validation) par les premières constructions du logiciel, Cependant, ce modèle souffre toujours du problème de la vérification tardive du bon fonctionnement du système.

☞ **TD :** Dressez un tableau qui résume les avantages et les inconvénients des modèles classiques.

I.6.2.3. Modèle de cycle de vie en spirale (Boehm 1988)

Le cycle de vie est représenté à l'aide d'une spirale dont le nombre de cycles est variable. Chaque boucle représente une phase du développement : La boucle la plus interne traite la première phase (faisabilité) et la plus externe traite de la livraison. Chaque cycle de cette spirale se déroule en quatre phases :

- Définition des objectifs de la phase
- étude des risques, évaluation des solutions de remplacement et éventuellement conception
- Développement et vérification de la solution (un modèle classique peut être utilisé)
- Examen du résultat et planification de la phase suivante.



I.6.2.4. Modèle par incrément (D.L Parnas)

Dans les modèles classiques, un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus. Dans les modèles par incrément un seul ensemble de composants est développé à la fois : des incréments viennent s'intégrer à un noyau de logiciel développé au préalable (chaque incrément est développé selon l'un des modèles précédents). Les incréments doivent être indépendants que possibles (fonctionnellement mais aussi sur le plan du calendrier du développement). Les noyaux, les incréments ainsi que leurs interactions doivent donc être spécifiés globalement, au début du projet.

Les avantages de ce type de modèle sont les suivants :

- Une première version du système est fournie rapidement.
- Management plus aisée : chaque développement est moins complexe ; Les intégrations sont progressives ; et Il est ainsi possible de livrer et de mettre en service chaque incrément.
- Les risques d'échec sont diminués : Les problèmes sont identifiés assez tôt, les parties importantes sont fournies en premier et seront donc testées plus longuement et particulièrement la prise en charge des exigences du client qui peuvent évolués ou changés dans le temps.

Cependant ces modèles comportent certains risques ou limites comme par exemple :

- La remise en cause des incréments précédents ou même le noyau et des fois on ne peut pas intégrer de nouveaux incréments.
- Les incréments difficiles à définir : mapper des exigences sur des incréments est complexe.
- Difficile de concevoir une architecture stable ou facilement évolutive.

NB : D'autres modèles existent : modèle en **Y,** ….etc.