Chapitre 1

INTRODUCTION AU GENIE LOGICIEL

1.1 Analyse de l'existant : Crise du logiciel

Le terme de Génie logiciel a été introduit à la fin des années soixante lors d'une conférence tenue pour discuter de ce que l'on appelait alors " *la crise du logiciel* " (*software crisis*).

Le développement de logiciel était en crise. Les coûts du matériel chutaient alors que ceux du logiciel grimpaient en flèche. Il fallait de nouvelles techniques et de nouvelles méthodes pour contrôler la complexité inhérente aux grands systèmes logiciels.

La crise du logiciel peut tout d'abord se percevoir à travers ces symptomes :
- La qualité du logiciel livré est souvent déficiente. Le produit ne satisfait pas les besoins de l'utilisateur, il consomme plus de ressources que prévu et il est à l'origine de pannes.
- Les performances étaient très souvent médiocres (temps de réponse trop lents).
- Le non respect des délais prévus pour le développement de logiciels satisfaisant leurs cahiers des charges.
- Les coûts de développement d'un logiciel sont presque impossible à prévoir et sont générale- ment prohibitifs (excessifs)
- L'invisibilité du logiciel, ce qui veut dire qu'on s'aperçoit souvent que le logiciel développé ne correspond pas à la demande (on ne peut l'observer qu'en l'utilisant (trop tard !)).
- La maintenance du logiciel est difficile, coûteuse et souvent à l'origine de nouvelles erreurs. Mais en pratique, il est indispensable d'adapter les logiciels car leurs environnements d'utili- sation changent et les besoins des utilisateurs évoluent.
- Il est rare qu'on puisse réutiliser un logiciel existant ou un de ses composants pour confection- ner un nouveau système, même si celui-ci comporte des fonctions similaires.

Tous ces problèmes ont mené à l'émergence d'une discipline appelée "**le génie logiciel**". Les outils de génie logiciel et les environnements de programmation peuvent aider à faire face à ces problèmes à condition qu'ils soient eux-mêmes utilisés dans un cadre méthodologique bien défini.

1.2 Une solution : le Génie Logiciel

1.2.1 Définitions

GENIE LOGICIEL
Le terme **génie logiciel** (en anglais *software engineering*) désigne l'ensemble des méthodes, des techniques et outils concourant à la production d'un logiciel, audelà de la seule activité de programmation.

L'art et la manière de créer un logiciel.

Le génie logiciel est donc l'art de spécifier, de concevoir, de réaliser, et de faire évoluer, avec des moyens et dans des délais raisonnables, des programmes, des documentations et des procédures de qualité en vue d'utiliser un ordinateur pour résoudre certains problèmes.

Le mot **génie**, utilisé en général accompagné d'un adjectif, comme dans génie civil, génie chi- mique ou génie atomique, désigne, d'après le Petit Robert, les connaissances et techniques de l'ingénieur. Ce terme est donc synonyme de science de l'ingénieur (*engineering*).

QU'EST CE QU'UN LOGICIEL ?
Par logiciel on n'entend pas seulement l'ensemble des programmes informatiques (du code) associés à une application ou à un produit, mais également un certain nombre de **documents** se rapportant à ces programmes et nécessaires à leur installation, utilisation, développement et maintenance : spécification, schémas conceptuels, jeux de tests, mode d'emploi, etc.

Pour les grands systèmes, l'effort nécessaire pour écrire cette documentation est souvent aussi grand que l'effort de développement des programmes eux-mêmes.

1.2.2 Qualité exigée d'un logiciel

Si le génie logiciel est l'art de produire de bons logiciels, il est par conséquent nécessaire de fixer les critères de qualité d'un logiciel.

- **La validité** : C'est l'aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications.
- **La fiabilité (ou robustesse)** : C'est l'aptitude d'un produit logiciel à fonctionner dans des conditions anormales (quelque soit l'entrée par exemple).
- **L'extensibilité** : C'est la facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qui lui sont demandées.
- **La réutilisabilité** : C'est l'aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.
- **La compatibilité** : C'est la facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.
- **L'efficacité** : On dit d'un logiciel qu'il est efficace s'il utilise les ressources d'une manière optimale (comme la mémoire et les cycles machine).
- **La portabilité** : C'est la facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels (produit indépendant du genre d'environnement).
- **L'intégrité** : C'est l'aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisé.
- **La facilité d'emploi** : Elle est liée à la facilité d'apprentissage, d'utilisation, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.
- **La maintenabilité** : Elle correspond au degré de facilité de la maintenance d'un produit logiciel.

1.2.3 Principes du Génie Logiciel

Principes utilisés dans le Génie Logiciel

- *Généralisation* : regroupement d'un ensemble de fonctionnalités semblables en une fonction- nalité paramétrable (généricité, héritage)
- *Structuration* : façon de décomposer un logiciel (utilisation d'une méthode bottom-up ou

top- down)

- *Abstraction* : mécanisme qui permet de présenter un contexte en exprimant les éléments per- tinents et en omettant ceux qui ne le sont pas
- *Modularité* : décomposition d'un logiciel en composants discrets
- *Documentation* : gestion des documents incluant leur identification, acquisition, production, stockage et distribution
- *Vérification* : détermination du respect des spécifications établies sur la base des besoins identifiés dans la phase précédente du cycle de vie

Chapitre 2 Cycle de vie du logiciel

2.1 Définition

Cycle de vie : ensemble des étapes de la réalisation, de l'énoncé des besoins à la maintenance ou au retrait du logiciel.

L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation. Le cycle de vie permet de détecter les erreurs au plus tôt et ainsi de maîtriser la qualité du produit, les délais de sa réalisation et les coûts associés.

De façon générale, on peut dire que le cycle de vie du logiciel est la période de temps s'étalant du début à la fin du processus du logiciel. Il commence donc avec la proposition ou la décision de développer un logiciel et se termine avec sa mise hors service.

2.2 Étapes du cycle de vie

Il existe de nombreux modèles de cycle de vie, les plus courants comportent les phases suivantes :

- Étude d'opportunité (par des économistes en général)
- Définition et analyse des besoins, spécification (par le commanditaire et des informaticiens) ; élaboration du cahier des charges et des tests de recette/validation
- Conception architecturale et élaboration des tests d'intégration
- Conception détaillée et élaboration des tests unitaires
- Codage (production du code source)
- Tests unitaires et d'intégration
- Implantation chez le commanditaire, essais avec les utilisateurs et validation
- Formation des utilisateurs, utilisation, maintenance, évolution
- Retrait

Ces étapes ne doivent pas être vues comme se succédant les unes aux autres de façon linéaire. Il y a en général (toujours) des retours sur les phases précédentes, en particulier si les tests ne réussissent pas ou si les besoins évoluent.

2.3 Étude d'opportunité ou étude préalable

Le développement est précédé d'une étude d'opportunité ou étude préalable. Cette phase a comme objectif de répondre aux questions suivantes :

- Pourquoi développer le logiciel ?
- Comment procéder pour faire ce développement ?
- Quels moyens faut-il mettre en oeuvre ?

Elle comprend à la fois des aspects techniques et de gestion. Parmi les tâches techniques, groupées sous le terme étude préalable, on peut citer :

- Dresser un état de l'existant et faire une analyse de ses forces et faiblesses ;
- Identifier les idées ou besoins de l'utilisateur ;
- Formuler des solutions potentielles ;
- Faire des études de faisabilité ;
- Planifier la transition entre l'ancien logiciel et le nouveau, s'il y a lieu ;

- Affiner ou finaliser l'énoncé des besoins de l'utilisateur.

2.4 Analyse (Spécification)

Lors de la phase d'analyse, également appelée phase de spécification (*requirements phase, analysis phase, definition phase*), on analyse les besoins de l'utilisateur ou du système englobant et on définit **ce que le logiciel devra faire**. Le résultat de la phase d'analyse est consigné dans un document appelé **cahier des charges du logiciel** ou spécification du logiciel, en anglais : *software requirements, software specification ou requirements specification*.

Il est essentiel qu'une spécification ne définisse que les caractéristiques essentielles du logiciel pour laisser de la place aux décisions de conception (Ne pas faire de choix d'implémentation à ce niveau).

Une spécification comporte les éléments suivants :
- description de l'environnement du logiciel ;
- spécification fonctionnelle (*functional specification*), qui définit toutes les fonctions que le logiciel doit offrir ;
- comportement en cas d'erreurs, c'est-à-dire dans les cas où le logiciel ne peut pas accomplir une fonction ;
- performances requises (*performance requirements*), par exemple : temps de réponse, encombrement en mémoire, sécurité de fonctionnement ;
- interfaces avec l'utilisateur (*user interface*), en particulier le dialogue sur terminal, la présentation des écrans, la disposition des états imprimés, etc.
- interfaces avec d'autres logiciels ;
- interfaces avec le matériel ;
- contraintes de réalisation, telles que l'environnement de développement, le langage de programmation à utiliser, les procédures et normes à suivre, etc.

Il est judicieux de préparer pendant la phase d'analyse les procédures qui seront mises en oeuvre pour vérifier que le logiciel, une fois construit, est conforme à la spécification, que nous l'appel- lerons test de réception (*acceptance test*).

Durant la phase d'analyse, on produit également une version provisoire des manuels d'utilisation et d'exploitation du logiciel.

POINTS CLES
- Pour les gros systèmes, il est difficile de formuler une spécification définitive. C'est pourquoi on supposera que les besoins initiaux du système sont incomplets et inconsistants.
- La définition des besoins et la spécification des besoins constituent des moyens de description à différents niveaux de détails, s'adressant à différents lecteurs.
  - *La définition des besoins* est un énoncé, en langue naturelle, des services que le système est sensé fournir à l'utilisateur. Il doit être écrit de manière à être compréhensible par les décideurs côté client et côté contractant, ainsi que par les utilisateurs et acheteurs potentiels du système.
  - *La spécification des besoins* est un document structuré qui énonce les services de manière plus détaillée. Ce document doit être suffisamment précis pour servir de base contractuelle entre le client et le fournisseur du logiciel. On peut utiliser des techniques de spécification formelle pour rédiger un tel document, mais cela dépendra du bagage technique du client.
- Il est difficile de détecter les inconsistances ou l'incomplétude d'une spécification lorsqu'elle est décrite dans un langage naturel non structuré. On doit toujours imposer une structuration du langage lors de la définition des besoins.

– Les besoins changent inévitablement. Le cahier des charges doit donc être conçu de manière à être facilement modifiable

2.5 La conception du logiciel

La phase d'analyse est suivie de la phase de conception , généralement décomposée en deux phases successives :

1. conception générale, conception globale, conception préliminaire ou conception architec- turale (*preliminary design ou architectural design*)
2. Conception détaillée (*detailed design*)

2.5.1 Conception générale

Si nécessaire, il faut commencer par l'ébauche de plusieurs variantes de solutions et choisir celle qui offre le meilleur rapport entre coûts et avantages.

Il faut ensuite figer la solution retenue, la décrire et la détailler. En particulier, il faut décrire l'architecture de la solution, c'est-à-dire son organisation en entités, les interfaces de ces entités et les interactions entre ces entités. Ce processus de structuration doit être poursuivi jusqu'à ce que tous les éléments du document de spécification ont été pris en compte.

Le résultat de cette démarche est un **document de conception générale**.

Durant la phase de conception générale, il faut également préparer la **phase d'intégration**. A cet effet, il faut élaborer un **plan d'intégration**, y compris un **plan de test d'intégration**.

2.5.2 Conception détaillée

La conception détaillée affine la conception générale. Elle commence par décomposer les entités découvertes lors de la conception générale en entités plus élémentaires. Cette décomposition doit être poursuivie jusqu'au niveau où les entités sont faciles à implémenter et à tester, c'est-à-dire correspondent à des composants logiciels élémentaires. Ce niveau dépend fortement du langage de programmation retenu pour l'implémentation.

Il faut ensuite décrire chaque composant logiciel en détail : son interface, les algorithmes utilisés, le traitement des erreurs, ses performances, etc. L'ensemble de ces descriptions constitue le **document de conception détaillée**.

Pendant la conception détaillée, il faut également préparer la vérification des composants logiciels élémentaires qui fera l'objet de la phase des tests unitaires. Le résultat est consigné dans un document appelé **plan de tests unitaires**. Si nécessaire, il faut de plus compléter le **plan d'intégration**, car de nouvelles entités ont pu être introduites pendant la conception détaillée.

2.5.3 Qualité d'une conception

La composante la plus importante de la qualité d'une conception est la maintenabilité. C'est en maximisant la cohésion à l'intérieur des composants et en minimisant le couplage entre ces composants que l'on parviendra à une conception maintenable.

COHESION

La cohésion d'un composant permet de mesurer la qualité de sa structuration. Un composant devrait implémenter une seule fonction logique ou une seule entité logique. La cohésion est une caractéristique désirable car elle signifie que chaque unité ne représente qu'une partie de la résolution du problème.

Constantine et Yourdon [3] identifient, en 1979, sept niveaux de cohésion présentées ci-après, du plus fort au plus faible.

- **La cohésion fonctionnelle** : la meilleure
  - Le module assure une seule fonction
  - Tous les éléments du composant contribuent à atteindre un seul objectif (si un élément est supprimé, l'objectif ne sera pas atteint).
  - Exemple : M1 = Calcul solution ; M2 = imprime solution
- **La cohésion séquentielle**
  - Dans se type de cohésion, la sortie d'un élément est utilisée en entrée d'un autre (dans ce cas, l'ordre des actions est important)
  - Exemple : Saisir, Traiter, Imprimer
- **La cohésion de communication** : bonne
  - Lorsque tous les éléments d'un composant travaillent sur les mêmes données.
  - Exemple : M1 = calculer et imprimer les résultats.
- **La cohésion procédurale** : passable
  Dans ce cas, les éléments d'un composant constituent une seule séquence de contrôle.
- **La cohésion temporelle** : médiocre
  On parle de cohésion temporelle quand dans un même composant sont regroupés tous les éléments qui sont activés au même moment, par exemple, à l'initialisation d'un programme ou encore en fin d'exécution.
- **La cohésion logique** : la moins mauvaise
  - Tous les éléments d'un composant effectuent des opérations semblables comme, par exemple, module qui édite tous les types de transactions
  - difficile à modifier.
- **La cohésion occasionnelle** : la plus mauvaise
  Le découpage en modules conduit à ce qu'une fonction se retrouve assurée par plusieurs modules. Dans ce cas, il n'y a pas de relation entre les éléments du composant.

Couplage

Le couplage est relatif à la cohésion. C'est une indication de la force d'interconnexion des diffé- rents composants d'un système. En règle générale, des modules sont fortement couplés lorsqu'ils utilisent des variables partagées ou lorsqu'ils échangent des informations de contrôle.

La comprehensibilite

Pour modifier un composant dans une conception, il faut que celui qui est responsable de cette modification comprenne l'opération effectuée par ce composant. Cette compréhensibilité dépend d'un certain nombre de caractéristiques

- *La cohésion*. Le composant peut-il être compris sans que l'on fasse référence à d'autres composants ?
- *L'appellation*. Les noms utilisés dans le composant sont-ils significatifs ? Des noms significatifs reflètent les noms des entités du monde réel que l'on modélise.
- *La documentation*. Le composant est-il documenté de manière à ce que l'on puisse établir une correspondance claire entre le monde réel et le composant ? Est-ce que cette correspondance est résumée quelque part.
- *La complexité*. Les algorithmes utilisés pour implémenter le composant sont-ils complexes ?

L'adaptabilite

Si l'on doit maintenir une conception, cette dernière doit être facilement adaptable. Bien sûr, il faut pour cela que les composants soient faiblement couplés. En plus de ça, la conception doit être bien documentée, la documentation des composants doit être facilement compréhensible et consistante avec l'implémentation, cette dernière devant elle aussi être

écrite de manière lisible.

2.6 Implémentation

Après la conception détaillée, on peut passer à la phase d'implémentation, également appelée phase de construction, phase de réalisation ou phase de codage (*implementation phase, construc- tion phase, coding phase*). Lors de cette phase, la conception détaillée est traduite dans un langage de programmation.

2.7 Test unitaire

La phase d'implémentation est suivie de la phase de test (*test phase*). Durant cette phase, les composants du logiciel sont évalués et intégrés, et le logiciel lui-même est évalué pour déterminer s'il satisfait la spécification élaborée lors de la phase d'analyse. Cette phase est en général subdivisée en plusieurs phases.

Lors des **tests unitaires** (*unit test*), on évalue chaque composant individuellement pour s'assurer qu'il est conforme à la conception détaillée. Si ce n'est déjà fait, il faut élaborer pour chaque composant un jeu de données de tests.

Il faut ensuite exécuter le composant avec ce jeu, comparer les résultats obtenus aux résultats attendus, et consigner le tout dans le document des tests unitaires. S'il s'avère qu'un composant comporte des erreurs, il est renvoyé à son auteur, qui devra diagnostiquer la cause de l'erreur puis corriger le composant. Le test unitaire de ce composant est alors à reprendre.

2.8 Intégration et test d'intégration

Après avoir effectué avec succès les tests unitaires de tous les composants, on peut procéder à leur **assemblage**, qui est effectué pendant la **phase d'intégration** (*integration phase*). Pendant cette phase, on vérifie également la bonne facture des composants assemblés, ce qu'on appelle le **test d'intégration** (*integration test*). On peut donc distinguer les actions suivantes :

- construire par assemblage un composant à partir de composants plus petits ;
- exécuter les tests pour le composant assemblé et enregistrer les résultats ;
- comparer les résultats obtenus aux résultats attendus ;

si le composant n'est pas conforme, engager la procédure de modification

- si le composant est conforme, rédiger les comptes-rendus du test d'intégration et archiver sur support informatique les sources, objets compilés, images exécutables, les jeux de tests et leurs résultats.

2.9 Installation

Après avoir intégré le logiciel, on peut l'installer dans son environnement d'exploitation, ou dans un environnement qui simule cet environnement d'exploitation, et le tester pour s'assurer qu'il se comporte comme requis dans la spécification élaborée lors de la phase d'analyse.

Cette phase s'appelle la phase d'installation (*installation phase ou installation and check-out phase*). Les tests effectués durant cette phase prennent des noms variés selon leur nature. On parle parfois de **validation**. Si l'on veut insister sur le fait que ces tests doivent préparer la décision du mandant d'accepter ou non le logiciel, on utilise les termes **test d'acceptance**, **test de recette** ou **test de réception** (*acceptance test*). Enfin, s'il s'agit de montrer le

comporte- ment et les performances du logiciel dans son environnement d'exploitation réel, le terme **test d'exploitation** est d'usage (*operational test*).

2.10 Maintenance

Après l'installation suit la phase d'exploitation et de maintenance (*operation and maintenance phase*). Le logiciel est maintenant employé dans son environnement opérationnel, son comporte- ment est surveillé et, si nécessaire, il est modifié. Cette dernière activité s'appelle la maintenance du logiciel (*software maintenance*).

Il peut être nécessaire de modifier le logiciel pour corriger des défauts, pour améliorer ses perfor- mances ou autres caractéristiques, pour adapter le logiciel à un nouvel environnement ou pour répondre à des nouveaux besoins ou à des besoins modifiés. On peut donc distinguer entre la **maintenance corrective**, la **maintenance perfective** et la **maintenance adaptative**. Sauf pour des corrections mineures, du genre dépannage, la maintenance exige en fait que le cycle de développement soit réappliqué, en général sous une forme simplifiée.

Maintenance corrective

- Corriger les erreurs : défauts d'utilité, d'utilisabilité, de fiabilité…
  - Identifier la défaillance, le fonctionnement
  - Localiser la partie du code responsable
  - Corriger et estimer l'impact d'une modification
- Attention
  - La plupart des corrections introduisent de nouvelles erreurs
  - Les coûts de correction augmentent exponentiellement avec le délai de détection
  - Corriger et estimer l'impact d'une modification
- La maintenance corrective donne lieu à de nouvelles livraisons (release)

Maintenance adaptative

- Ajuster le logiciel pour qu'il continue à remplir son rôle compte tenu du l'évolution des
  - Environnements d'exécution
  - Fonctions à satisfaire
  - Conditions d'utilisation

Ex : changement de SGBD, de machine, de taux de TVA

Maintenance perfective, d'extension

- Accroître/améliorer les possibilités du logiciel
- Ex : les services offerts, l'interface utilisateur, les performances…
- Donne lieu à de nouvelles versions

Une fois qu'une version modifiée du logiciel a été développée, il faut bien entendu la **distribuer**. De plus, il est en général nécessaire de fournir à l'exploitant du logiciel une **assistance technique** et un **support de consultation**.

En résumé, on peut dire que la maintenance et le support du logiciel comprennent les tâches suivantes :

- effectuer des dépannages pour des corrections mineures ;
- réappliquer le cycle de développement pour des modifications plus importantes ;
- distribuer les mises à jour ;
- fournir l'assistance technique et un support de consultation ;
- maintenir un journal des demandes d'assistance et de support.

A un moment donné, on décide de **mettre le logiciel hors service**. Les tâches correspondantes sont accomplies durant la **phase de retrait** (*retirement phase*) et

comprennent :

- avertir les utilisateurs ;
- effectuer une exploitation en parallèle du logiciel à retirer et de son successeur ;

arrêter le support du logiciel