

Génie Logiciel Et Programmation Orientée Objet

Partie 1 : Introduction au Génie Logiciel

1. Définition et objectifs du génie logiciel (GL)

1.1. Définition

Domaine des ‘sciences de l’ingénieur’ dont la finalité est la *conception*, la *fabrication* et la *maintenance de systèmes logiciels complexes, sûrs et de qualité* (‘Software Engineering’ en anglais).

Le GL se définit souvent par opposition à la ‘programmation’, c’est à dire la production d’un programme par un individu unique, considérée comme ‘facile’. Dans le cas du GL il s’agit de la fabrication *collective* d’un *système complexe*, concrétisée par un ensemble de documents de conception, de programmes et de jeux de tests avec souvent de *multiples versions*, et considérée comme ‘difficile’.

1.2. Objectifs

La pratique du génie logiciel a pour objectif de diminuer sensiblement les coûts de développement des gros systèmes logiciels en fournissant aux concepteurs et développeurs un ensemble de méthodes, techniques et outils leur permettant de produire des logiciels d’une qualité en constante amélioration et avec une productivité croissante.

2. Les principes du génie logiciel

Les principes fondamentaux du GL sont :

2.1. Rigueur

La production de logiciel est une activité créative, mais qui doit se conduire avec une certaine rigueur.

Le niveau maximum de rigueur est la *formalité*, c’est à dire le cas où les descriptions et les validations s’appuient sur des notations et lois mathématiques. Il n’est pas possible d’être formel tout le temps : il faut bien construire la première description formelle à partir de connaissances non formalisées ! Mais dans certaines circonstances les techniques formelles sont utiles.

2.2. "Séparation des problèmes"

C’est une règle de bons sens qui consiste à considérer séparément différents aspects d’un problème afin d’en maîtriser la complexité.

Elle prend une multitude de formes :

- séparation dans le *temps* (les différents aspects sont abordés successivement), avec la notion de cycle de vie du logiciel que nous étudierons en détail,
- séparation des *qualités* que l’on cherche à optimiser à un stade donné (ex assurer la correction avant de se préoccuper de l’efficacité),
- □□ séparation des ‘*vues*’ que l’on peut avoir d’un système (ex : se concentrer sur l’aspect ‘flots de données’ avant de considérer l’aspect ordonnancement des opérations ou ‘flot de contrôle’),
- □□ séparation du système en *parties* (un noyau, des extensions, ...),
- □□ etc.

2.3. Modularité

Un système est modulaire s'il est composé de *sous-systèmes plus simples*, ou modules. La modularité permet de considérer séparément le *contenu* du module et les *relations entre modules* (ce qui rejoint l'idée de séparation des questions). Elle facilite également la *réutilisation* de composants bien délimités.

Un bon découpage modulaire se caractérise par une *forte cohésion* interne des modules (ex : fonctionnelle, temporelle, logique, ...) et un *faible couplage* entre les modules (relations inter-modulaires en nombre limité et clairement décrites).

Toute l'évolution des langages de programmation vise à rendre plus facile une programmation modulaire, appelée aujourd'hui 'programmation par composants'.

2.4. 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 (c'est encore un exemple de séparation des problèmes).

Une même réalité peut souvent être décrite à différents *niveaux d'abstraction*. Par exemple, un circuit électronique peut être décrit par un modèle mathématique très abstrait (équation logique), ou par un assemblage de composants logiques qui font abstraction des détails de réalisation (circuit logique). L'abstraction permet une meilleure maîtrise de la complexité.

2.5. Anticipation du changement

La caractéristique essentielle du logiciel, par rapport à d'autres produits, est qu'il est presque toujours soumis à des *changements continuels* (corrections d'imperfections et évolutions en fonctions *des besoins qui changent*).

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),
- Être capable de gérer les multiples versions des modules et configurations des versions des modules, constituant des versions du produit complet.

2.6. 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.

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).

2.7. 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.

2.8. Conclusion

Ces principes sont très abstraits et ne sont *pas utilisables directement*. Mais ils font partie du vocabulaire de base du génie logiciel. Ces principes ont un impact réel sur beaucoup d'aspects (on le verra dans la suite du cours) et constituent le type de connaissance le plus stable, dans un domaine où les outils, les méthodes et les techniques évoluent très vite.

3- Le cycle de vie d'un logiciel

Le cycle de vie d'un logiciel, désigne toutes les étapes du développement d'un logiciel, de sa conception à sa disparition. L'objectif d'un tel découpage est de permettre de définir des jalons intermédiaires permettant la validation du développement logiciel, c'est-à-dire la conformité du logiciel avec les besoins exprimés, et la vérification du processus de développement, c'est-à-dire l'adéquation des méthodes mises en œuvre.

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 logiciel, les délais de sa réalisation et les coûts associés.

3-1 Etapes du cycle de vie

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

- Définition des objectifs

Cet étape consiste à définir la finalité du projet et son inscription dans une stratégie globale.

- Analyse des besoins et faisabilité

C'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes, puis l'estimation de la faisabilité de ces besoins.

- Spécifications ou conception générale

Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.

- Conception détaillée

Cette étape consiste à définir précisément chaque sous-ensemble du logiciel.

- Codage (Implémentation ou programmation)

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

- Tests unitaires

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

- Intégration

L'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.

- Qualification (ou recette)

C'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales (Validation).

- Mise en production

C'est le déploiement sur site du logiciel (Exploitation).

- Maintenance

Elle comprend toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

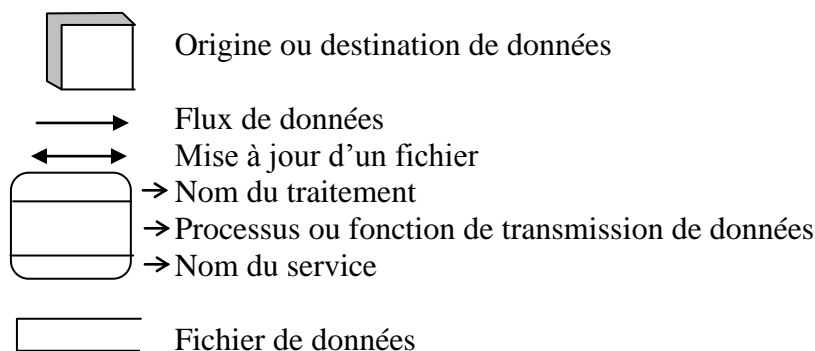
La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement. Le cycle de vie permet de prendre en compte, en plus des aspects techniques, l'organisation et les aspects humains.

3-2 Diagrammes de flux de données (DFD)

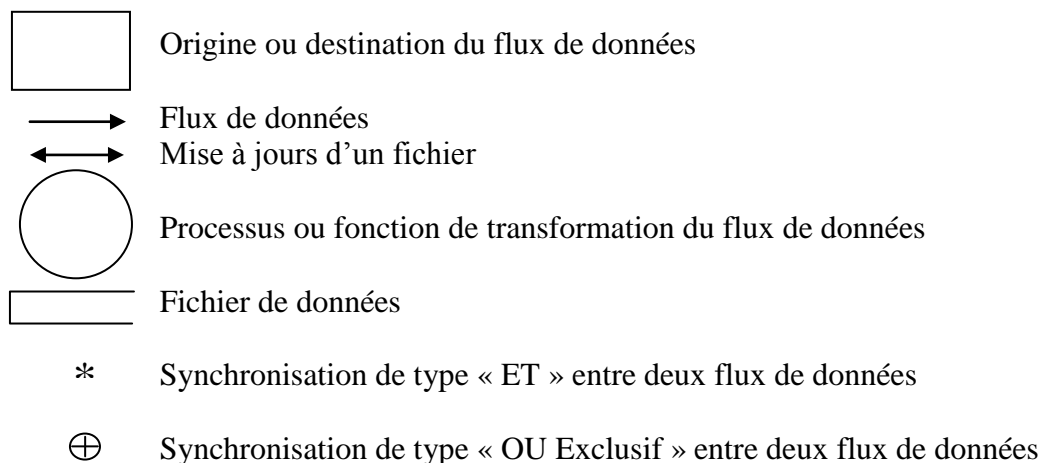
C'est le dossier de spécification le plus important. Les diagrammes de flux de données montrent la manière dont les données d'entrée sont modifiées, via une suite de transformations fonctionnelles pour donner des résultats appelés données de sortie. Ils constituent un moyen intuitif et utile de décrire un système et sont compréhensibles sans formation particulière.

Il y a deux formalismes de présentation des DFD :

a- Formalisme de Gane et Sarson



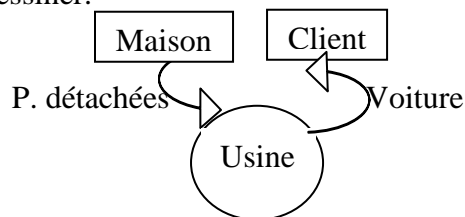
b- Formalisme de Marco et Weinberg



3-2-1 Démarche de construction d'un DFD

La démarche générale de construction d'un DFD est la suivante :

- 1- Identifier les acteurs externes en même temps que les flux d'entrées et de sortie puis les dessiner.



- 2- Nommer les flux de données.
- 3- Nommer les transformations ou fonctions en utilisant des verbes actifs.
- 4- Omettre (au début) les traitements d'erreurs.
- 5- Rejeter les informations de contrôle ou flux de contrôle.

Les DFD ont l'avantage de montrer des transformations sans faire l'hypothèse sur la manière dont ces transformations sont implémentées. Par exemple, un système décrit de cette manière pourrait être réalisé par un seul programme utilisant des unités de programmation qui implémentent chacune une transformation.

3-2-2 Règles de construction d'un DFD

- 1- Pas plus de sept stockages de données par diagramme.
- 2- Pas plus de sept activités par diagramme.
- 3- Les premiers modèles n'ont pas besoin d'être parfait (pas obligés de tous décrire).
- 4- Décomposer à plusieurs niveaux si nécessaire.
- 5- Utiliser des désignations significatives.

3-2-3 Exemple

On considère l'exemple de la gestion simplifiée des inscriptions dans une faculté.

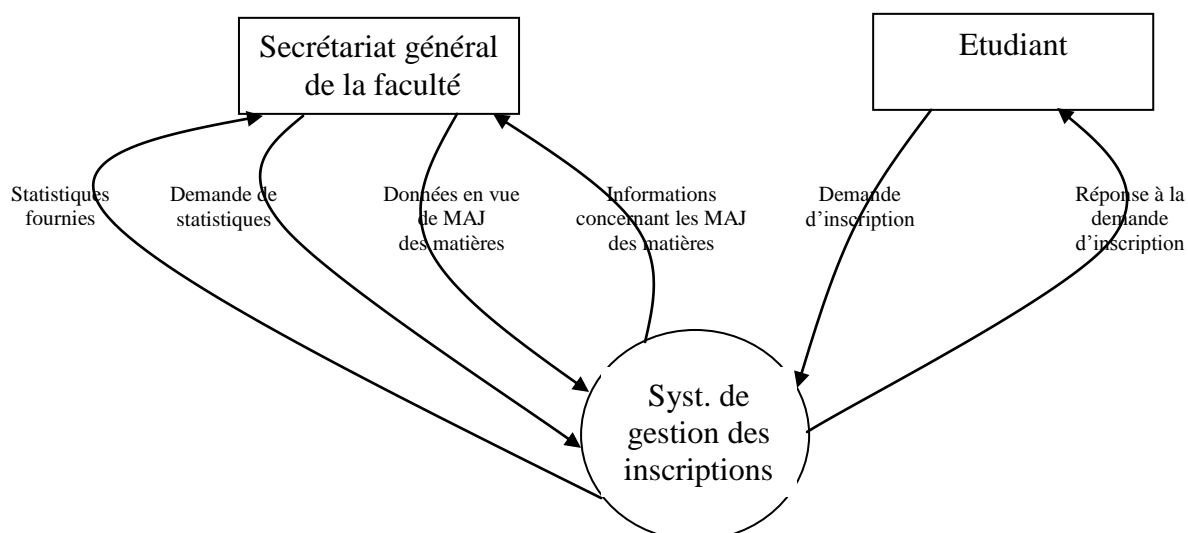


Figure 1.1 : DFD de niveau 1

La figure suivante donne un aperçu de ce qui peut être considéré comme un raffinement ou une décomposition du DFD de la figure précédente.

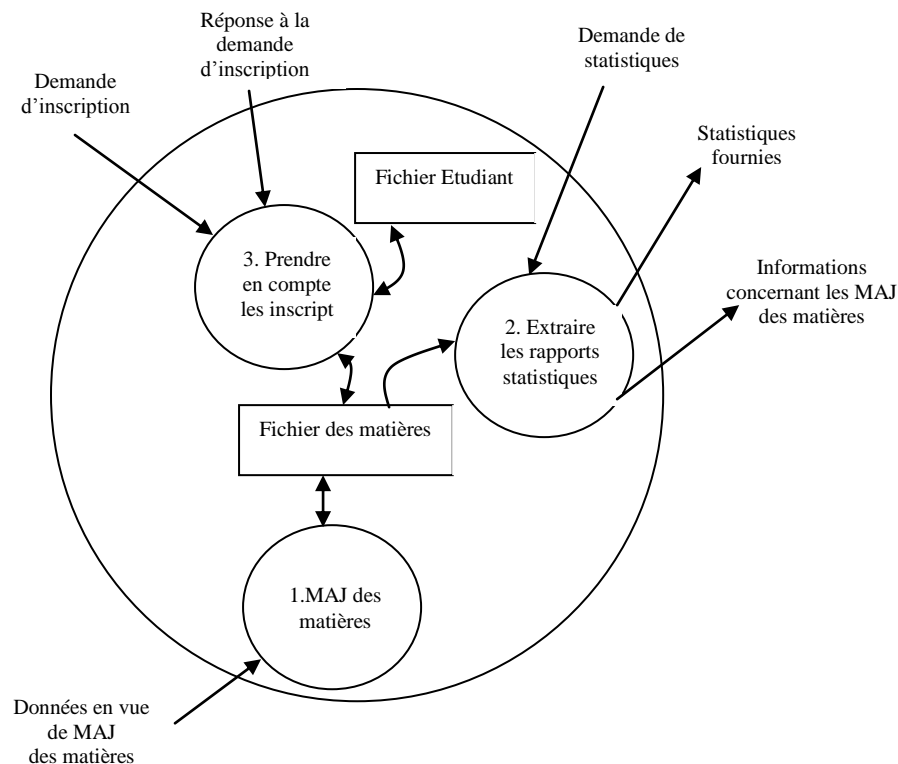


Figure 1.2 : Explosion du DFD (niveau2)

On remarque que la figure précédente comporte toujours autant de flux d'entrée et de sortie que le diagramme de la figure 1.1.

4 Modèles de cycles de vie d'un logiciel

4.1 Modèle de cycle de vie en cascade

Le modèle de cycle de vie en cascade (figure 1.3) a été mis au point dès 1966, puis formalisé aux alentours de 1970.

Dans ce modèle, le principe est très simple : chaque phase se termine à une date précise par la production de certains documents ou logiciels. Les résultats sont définis sur la base des interactions entre étapes, ils sont soumis à une revue approfondie et on ne passe à la phase suivante que s'ils sont jugés satisfaisants.

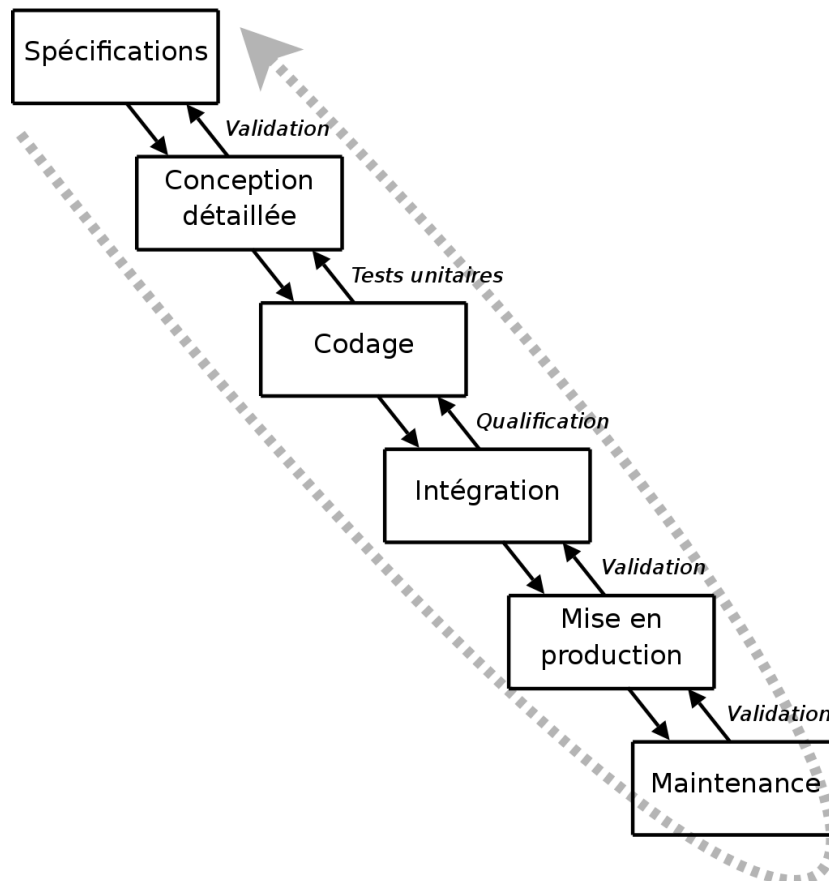


Figure 1.3: Modèle du cycle de vie en cascade

Le modèle original ne comportait pas de possibilité de retour en arrière. Celle-ci a été rajoutée ultérieurement sur la base qu'une étape ne remet en cause que l'étape précédente, ce qui, dans la pratique, s'avère insuffisant.

L'inconvénient majeur du modèle de cycle de vie en cascade est que la vérification du bon fonctionnement du système est réalisée trop tardivement: lors de la phase d'intégration, ou pire, lors de la mise en production.

4.2 Modèle de cycle de vie en V

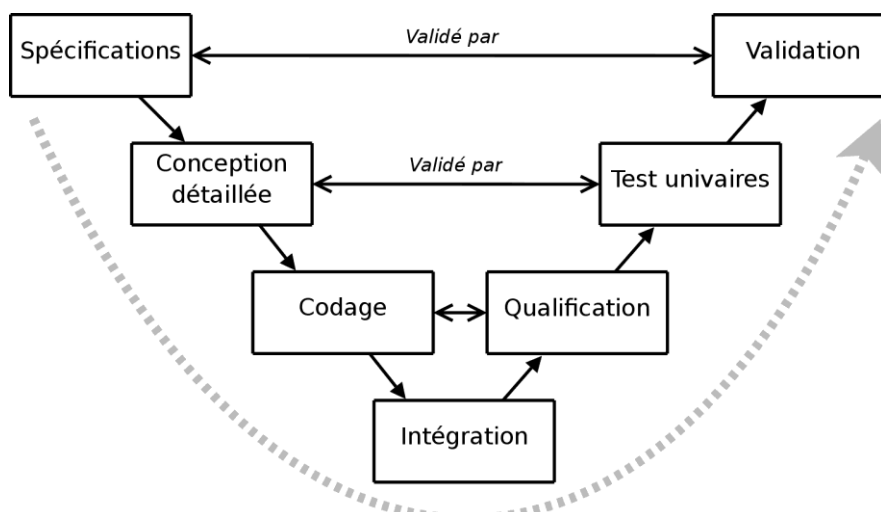


Figure 1.4: Modèle du cycle de vie en V

Le modèle en V (figure 1.4) demeure actuellement le cycle de vie le plus connu et certainement le plus utilisé. Il s'agit d'un modèle en cascade dans lequel le développement des tests et du logiciel sont effectués de manière synchrone.

Le principe de ce modèle est qu'avec toute décomposition doit être décrite la recombinaison et que toute description d'un composant est accompagnée de tests qui permettront de s'assurer qu'il correspond à sa description.

Ceci rend explicite la préparation des dernières phases (validation-vérification) par les premières (construction du logiciel), et permet ainsi d'éviter un écueil bien connu de la spécification du logiciel : énoncer une propriété qu'il est impossible de vérifier objectivement après la réalisation.

Cependant, ce modèle souffre toujours du problème de la vérification trop tardive du bon fonctionnement du système.

4.3 Modèle de cycle de vie en spirale

Proposé par B. Boehm en 1988, ce modèle est beaucoup plus général que le précédent. Il met l'accent sur l'activité d'analyse des risques : chaque cycle de la spirale se déroule en quatre phases :

- 1- détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
- 2- analyse des risques, évaluation des alternatives ;
- 3- développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
- 4- revue des résultats et vérification du cycle suivant.

L'analyse préliminaire est affinée au cours des premiers cycles. Le modèle utilise des maquettes exploratoires pour guider la phase de conception du cycle suivant. Le dernier cycle se termine par un processus de développement classique.

4-4 Modèle par incrément

Dans les modèles précédents, 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 avantages de ce type de modèle sont les suivants :

- chaque développement est moins complexe ;
- les intégrations sont progressives ;
- il est ainsi possible de livrer et de mettre en service chaque incrément ;
- il permet un meilleur lissage du temps et de l'effort de développement grâce à la possibilité de recouvrement (parallélisation) des différentes phases.

Les risques de ce type de modèle sont les suivants :

- remettre en cause les incréments précédents ou pire le noyau ;
- ne pas pouvoir intégrer de nouveaux incréments.

Les noyaux, les incréments ainsi que leurs interactions doivent donc être spécifiés globalement, au début du projet. Les incréments doivent être aussi indépendants que possibles, fonctionnellement mais aussi sur le plan du calendrier du développement.

5- Les bases de la qualité du logiciel

Pour évaluer la qualité d'un système, identifions les facteurs que l'on voudrait retrouver dans tous les bons logiciels. En plus de devoir fournir les fonctionnalités requises, un bon logiciel doit aussi posséder les facteurs clés suivants :

(1) Fiabilité : C'est la confiance que l'on peut attribuer à un logiciel. Cela signifie que le logiciel doit faire ce que les utilisateurs attendent de lui et ne doit pas tomber en panne plus souvent que sa spécification ne l'autorise.

(2) Ouverture : Le logiciel doit être maintenable. Comme les logiciels ayant une longue durée de vie sont sujets à des changements réguliers, ils devront être écrits de manière à ce que ces changements ne s'avèrent pas trop coûteux. {Préparer la maintenance dès les premières étapes}

(3) Coût : L'un des buts du génie logiciel est de réduire le coût du logiciel dans sa globalité. Une réduction du coût total d'un développement demande de fournir un effort lors de la définition des besoins, lors de la conception, puis lors de la vérification et de la validation.

(4) Efficacité : Maximiser l'efficacité peut rendre le logiciel plus difficile à maintenir. L'efficacité signifie qu'un système ne doit pas gaspiller ses ressources, comme la mémoire ou les cycles machines.

(5) Interface : Le logiciel doit offrir une interface utilisateur conviviale et appropriée. Beaucoup de logiciels ne sont pas utilisés au maximum de leurs possibilités parce que leur interface les rend difficiles à utiliser. La conception de l'interface doit être adaptée aux capacités et au bagage des utilisateurs.

Remarque : Il est difficile d'optimiser tous ces facteurs car certains s'excluent mutuellement (par exemple, offrir une bonne interface à l'utilisateur peut réduire l'efficacité). La figure suivante montre que les coûts nécessaires pour améliorer l'efficacité d'un logiciel risquent de croître de manière exponentielle. De petites améliorations peuvent revenir très cher.

6- Des méthodes fonctionnelles aux méthodes Objet

6-1 Méthodes fonctionnelles ou structurées

Les méthodes fonctionnelles (également qualifiées de méthodes structurées) trouvent leur origine dans les langages procéduraux. Elles mettent en évidence les fonctions à assurer et proposent une approche hiérarchique descendante et modulaire.

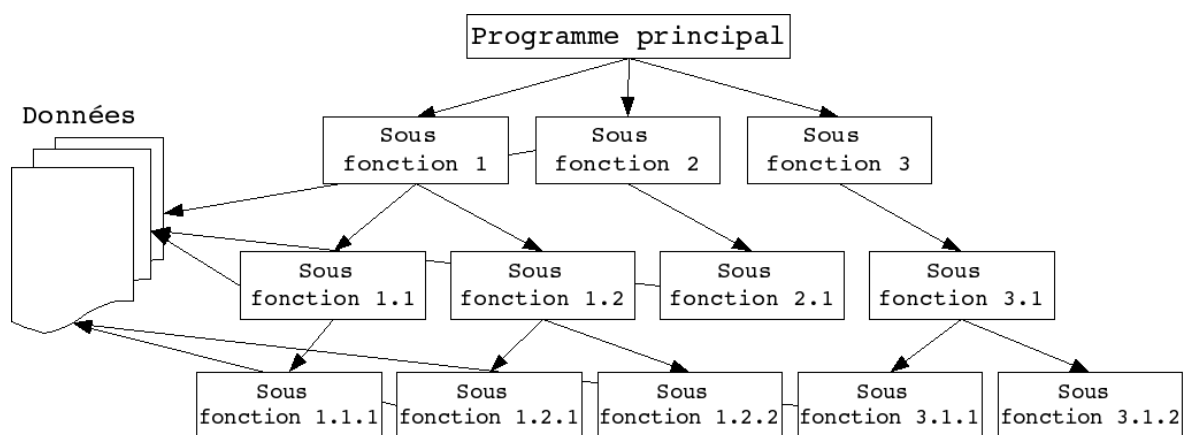


Figure 1.5: Représentation graphique d'une approche fonctionnelle

Ces méthodes utilisent intensivement les raffinements successifs pour produire des spécifications dont l'essentielle est sous forme de notation graphique en diagrammes de flots de données. Le plus haut niveau représente l'ensemble du problème (sous forme d'activité, de données ou de processus, selon la méthode). Chaque niveau est ensuite décomposé en respectant les entrées/sorties du niveau supérieur. La décomposition se poursuit jusqu'à arriver à des composants maîtrisables (figure 1.5).

L'approche fonctionnelle dissocie le problème de la représentation des données, du problème du traitement de ces données. Sur la figure 1.5, les données du problème sont représentées sur la gauche. Des flèches transversales matérialisent la manipulation de ces données par des sous-fonctions. Cet accès peut-être direct (c'est parfois le cas quand les données sont regroupées dans une base de données), ou peut être réalisé par le passage de paramètre depuis le programme principal.

La SADT (Structured Analysis Design Technique) est probablement la méthode d'analyse fonctionnelle et de gestion de projets la plus connue. Elle permet non seulement de décrire les tâches du projet et leurs interactions, mais aussi de décrire le système que le projet vise à étudier, créer ou modifier, en mettant notamment en évidence les parties qui constituent le système, la finalité et le fonctionnement de chacune, ainsi que les interfaces entre ces diverses parties. Le système ainsi modélisé n'est pas une simple collection d'éléments indépendants, mais une organisation structurée de ceux-ci dans une finalité précise.

En résumé, l'architecture du système est dictée par la réponse au problème (i.e. la fonction du système).

6-2 L'approche orientée objet

L'approche orientée objet considère le logiciel comme une collection d'objets dissociés, identifiés et possédant des caractéristiques. Une caractéristique est soit un attribut (i.e. une donnée caractérisant l'état de l'objet), soit une entité comportementale de l'objet (i.e. une fonction). La fonctionnalité du logiciel émerge alors de l'interaction entre les différents objets qui le constituent. L'une des particularités de cette approche est qu'elle rapproche les données et leurs traitements associés au sein d'un unique objet.

Comme nous venons de le dire, un objet est caractérisé par plusieurs notions :

- L'identité

L'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, etc.)

- Les attributs

Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations sur l'état de l'objet.

- Les méthodes

Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

La difficulté de cette modélisation consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une existence matérielle (chien, voiture, ampoule, personne, ...) ou bien virtuelle (client, temps, ...).

La Conception Orientée Objet (COO) est la méthode qui conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur la fonction qu'il est censé réaliser.

En résumé, l'architecture du système est dictée par la structure du problème.

6-3 Approche fonctionnelle vs. approche objet

Tout programme qu'il est possible d'écrire dans un langage pourrait également être écrit dans n'importe quel autre langage. Ainsi, tout ce que l'on fait avec un langage de programmation par objets pourrait être fait en programmation impérative. La différence entre une approche fonctionnelle et une approche objet n'est donc pas d'ordre logique, mais pratique.

L'approche structurée privilégie la fonction comme moyen d'organisation du logiciel. Ce n'est pas pour cette raison que l'approche objet est une approche non fonctionnelle. En effet, les méthodes d'un objet sont des fonctions. Ce qui différencie sur le fond l'approche objet de l'approche fonctionnelle, c'est que les fonctions obtenues à l'issue de la mise en œuvre de l'une ou l'autre méthode sont distinctes. L'approche objet est une approche orientée donnée. Dans cette approche, les fonctions se déduisent d'un regroupement de champs de données formant une entité cohérente, logique, tangible et surtout stable quant au problème traité. L'approche structurée classique privilégie une organisation des données postérieure à la découverte des grandes, puis petites fonctions qui les décomposent, l'ensemble constituant les services qui répondent aux besoins.

En approche objet, l'évolution des besoins aura le plus souvent tendance à se présenter comme un changement de l'interaction des objets. S'il faut apporter une modification aux données, seul l'objet incriminé (encapsulant cette donnée) sera modifié. Toutes les fonctions à modifier sont bien identifiées : elles se trouvent dans ce même objet : ce sont ses méthodes. Dans une approche structurée, l'évolution des besoins entraîne souvent une dégénérescence, ou une profonde remise en question, de la topologie typique de la figure 1.5 car la décomposition des unités de traitement (du programme principal aux sous-fonctions) est directement dictée par ces besoins. D'autre part, une modification des données entraîne

généralement une modification d'un nombre important de fonctions éparpillées et difficiles à identifier dans la hiérarchie de cette décomposition.

En fait, la modularité n'est pas antinomique de l'approche structurée. Les modules résultant de la décomposition objet sont tout simplement différents de ceux émanant de l'approche structurée. Les unités de traitement, et surtout leur dépendance dans la topologie de la figure 1.5 sont initialement bons. C'est leur résistance au temps, contrairement aux modules objet, qui est source de problème. La structure d'un logiciel issue d'une approche structurée est beaucoup moins malléable, adaptable, que celle issue d'une approche objet.

Ainsi la technologie objet est la conséquence ultime de la modularisation du logiciel, démarche qui vise à maîtriser sa production et son évolution. Mais malgré cette continuité logique les langages objet ont apporté en pratique un profond changement dans l'art de la programmation : ils impliquent en effet un changement de l'attitude mentale du programmeur.

7- Test et maintenance du logiciel

7-1 Test du logiciel

Le test est l'exécution ou l'évaluation d'un programme et de ses données (avec des jeux d'essai), par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou pour identifier les différences entre les résultats attendus et les résultats obtenus.

L'objectif du test est, donc, d'exécuter un programme avec l'intention de trouver des erreurs dans le but d'apporter une valeur ajoutée en matière de qualité et de fiabilité. Le coût de ce processus (qui représente 50% du coût total de développement) doit être compenser par l'amélioration de la valeur du produit logiciel testé (en terme de fiabilité, donc de qualité).

Le processus de test confronte deux problèmes essentiels :

1- Identifier des ensembles de données similaires aux données réelles (appelées cas de test), qui seront manipulées par le système.

2- Comment aborder le test ?

7-1-1 Cas de test

Il est généralement impossible de tester toutes les données d'entrée possibles. Il faut, donc, sélectionner des données d'entrée : C'est les cas de test.

Les cas de test doivent faire fonctionner l'ensemble du système, et permettre de vérifier que ce dernier se comporte de la façon prévue en présence de données incorrectes.

Il existe deux approches de construction de cas de test :

- **L'approche fonctionnelle** (test de boîte fermée ou boîte noire): Les cas de test sont construits à partir des spécifications (on s'intéresse aux fonctionnalités du logiciel).
- **Approche structurelle** (test de boîte ouverte ou boîte blanche): Les cas de test sont construits à partir de la structure interne du programme. On doit suivre les chemins des données à travers les structures de contrôle du programme et parfois des données.

Le deuxième problème posé (comment aborder le test) est dû au fait qu'il n'est pas réaliste de vouloir tester un logiciel comme une seule unité. De la même façon que pour le processus de conception, les opérations de test se feront par étapes, chaque étape constituant la suite logique de l'étape précédente.

7-1-2 Les étapes de test

On peut distinguer plusieurs étapes dans le processus de test :

- Les tests unitaires :

Les tests unitaires sont un processus ayant pour but de tester chaque composant du logiciel pris isolément. Dans un système bien conçu, chaque fonction doit avoir une spécification clairement établie. Les tests unitaires présentent plusieurs avantages, parmi lesquels on peut citer :

- Concentration de l'effort de test sur chaque composant.
- En cas de comportement anormal du composant, il est plus facile de découvrir l'erreur, de la localiser, et enfin de la corriger.
- Les tests unitaires introduisent la notion de parallélisme durant la campagne de test, dans le sens où plusieurs composants du logiciel peuvent être testés indépendamment des autres et surtout simultanément, ce qui permet de gagner du temps.

- Les tests d'intégration

A ce stade, les différents composants du logiciel qui ont été testés individuellement, devront être progressivement assemblés, pour construire un programme exécutable qui va être testé. Ces tests permettent de découvrir des erreurs de conception ou de codage, et de vérifier la bonne utilisation des interfaces entre modules. Ils permettent également de vérifier que le système, dans son ensemble, réalise bien les fonctions spécifiées lors de la définition des besoins.

- Les tests de validation

Les tests de validation interviennent à la fin du processus d'intégration. Ils permettent de tester le système avec des données réelles. Ils mettent en évidence des erreurs dans un logiciel ne satisfaisant pas les niveaux de fonctionnalité ou de performance attendus par l'utilisateur.

N.B : Les différentes étapes de test doivent être planifiées à l'avance, relativement tôt dans le cycle de développement du logiciel. Cependant, ces différentes étapes peuvent être enchaînées différemment selon la stratégie de test adoptée.

7-2 Maintenance du logiciel

Le terme « Maintenance » est appliqué au processus de modification d'un programme après sa livraison et pendant son exploitation. Ces modifications peuvent demander de simples changements pour corriger des erreurs de codification, des changements plus coûteux pour corriger des erreurs de conception ou des réécritures consistantes pour corriger des erreurs de spécifications ou répondre à de nouveaux besoins.

Le terme « Maintenance » signifie généralement les changements apportés à un programme afin de corriger les erreurs ou pour fournir de nouvelles facilités.

Il existe trois types de maintenance d'un logiciel :

- 1- Maintenance perfective (évolutive)
- 2- Maintenance adaptative
- 3- Maintenance corrective

7-2-1 Maintenance perfective

Elle intervient suite à des changements dans les spécifications d'un point de vue fonctionnel ou de performance. Elle concerne donc les modifications demandées par l'utilisateur ou les

programmeurs du système. Cela demande d'installer dans le programme de nouvelles fonctionnalités ou d'améliorer les performances.

7-2-2 Maintenance adaptative

Elle est due aux changements de l'environnement du programme (environnement logiciel ou matériel). Elle concerne donc l'adaptation du programme à ce nouvel environnement.

7-2-3 Maintenance corrective

C'est la maintenance qui permet de corriger les erreurs jusqu'à présent ignoré dans le système (correction des défauts résiduels).

Remarque

Des études statistiques ont montré que presque 65% de la maintenance est perfective, 18% adaptative et 17% corrective.