TABLE DES MATIERES

AVERTISSEMENTS 1
TABLE DES MATIERES 2
BIBLIOGRAPHIE 5

INTRODUCTION 6
OBJECTS DU COURS 7
DEFINITION DES CONCEPTS CLES 8

PREMIER CHAPITRE – GENERALITES SUR LE GENIE (L'INGENIERIE) LOGICIEL 9
II.1. DEFINITION ET CONTEXTE D'ETUDES 9
I.2.1. LOIS D'EVOLUTION DU LOGICIEL 12
I.2.2. EVOLUTION DU LOGICIEL E-TYPE 13
I.3. LES PARADIGMES LOGICIELS 14
I.3.1. PARADIGME DE DEVELOPPEMENT LOGICIEL 14
I.3.2. PARADIGME DE CONCEPTION 14
I.3.3. PARADIGME DE PROGRAMMATION 14
I.4. BESOINS DU GENIE LOGICIEL 15
I.5. CARACTERISTIQUES D'UN BON LOGICIEL 16
I.5.1. CRITERES GENERAUX 16
I.5.2. CRITERES EXTERNES 17
I.5.3. CRITERES INTERNES 18
I.6. CATEGORIES DE LOGICIELS 19
I.7. ENJEUX DU GENIE LOGICIEL 20
I.8. DIMENSIONS DU GENIE LOGICIEL 21
I.9. LES PRINCIPES DU GENIE LOGICIEL 21

DEUXIEME CHAPITRE – LE CYCLE DE VIE DU DEVELOPPEMENT LOGICIEL 23
II.1. DEFINITION 23
II.2. ÉTAPES DU CYCLE DE VIE DU DEVELOPPEMENT D'UN LOGICIEL 23
II.1.1. LA COMMUNICATION 24
II.1.2. LA COLLECTION DES EXIGENCES 24
II.1.3. ÉTUDE DU PREALABLE (*ETUDE DE L'OPPORTUNITE*) 24
II.1.4. DEFINITION ET ANALYSE DES BESOINS (*SPECIFICATION*) 25
II.1.5. LA CONCEPTION DU LOGICIEL 26
II.1.5.1. QUALITE D'UNE CONCEPTION LOGICIELLE 27
A. LA COHESION 28
B. LE COUPLAGE 29

C. LA COMPREHENSIBILITE 29
D. L'ADAPTABILITE 29
II.1.6. LE CODAGE 30
II.1.7. TESTS 30
II.1.8. INTEGRATION 30
II.1.9. INSTALLATION 31
II.1.10. MAINTENANCE 31
II.1.11. DISPOSITION 32

TROISIEME CHAPITRE - MODELISATION DE PROCESSUS DE DEVELOPPEMENT LOGICIEL 33
III.1. LE MODELE DU CODE-AND-FIX 33
II.2. LE MODELE DE TRANSFORMATION AUTOMATIQUE 34
II.3. LE MODELE EN CASCADE (WATERFALL MODEL) 35
II.4. LE MODELE EN V 37
III.5. LE CYCLE DE VIE EN SPIRALE 40
III.6. LE MODELE PAR PROTOTYPAGE 44
III.7. LE MODELE PAR INCREMENT 45

QUATRIEME CHAPITRE – PROCESSUS SEQUENTIEL DE DEVELOPPEMENT DU LOGICIEL 46
IV.1. ARCHITECTURE LOGICIELLE 46
IV.1.1. CONCEPTION / ARCHITECTURE 47
IV.1.2. L'ACTIVITE D'ARCHITECTURE 48
IV.1.3. PLACE DANS LE PROCESSUS DE CREATION 48
IV.1.4. REPARTITION DES PROBLEMATIQUES 49
IV.1.5. LA STRUCTURE D'UN LOGICIEL 49
IV.2. PRINCIPES DE CONCEPTION 50
IV.2.1. SEPARATION DES RESPONSABILITES 50
IV.2.2. REUTILISATION 51
IV.2.3. ENCAPSULATION MAXIMALE 52
IV.2.4. COUPLAGE FAIBLE 52
IV.2.5. COHESION FORTE 52
IV.2.6. DRY (DON'T REPEAT YOURSELF) 53
IV.2.7. KISS (*KEEP IT SIMPLE, STUPID*) 54
IV.2.8. YAGNI (*YOU AIN'TGONNA NEED IT*) 54
IV.3. PATRONS LOGICIELS 54
IV.3.1. ARCHITECTURE CLIENT / SERVEUR 55
IV.3.2. ARCHITECTURE EN COUCHES 56
IV.3.3. ARCHITECTURE ORIENTEE SERVICES 57

IV.3.4. ARCHITECTURE MODELE-VUE-CONTROLEUR **58**
IV.3.5. ARCHITECTURE MODELE-VUE-PRESENTATION **59**
IV.4. PRODUCTION DU CODE SOURCE **60**
IV.4.1. CONVENTION DE NOMMAGE **61**
IV.4.2. LANGUE UTILISEE **61**
IV.4.3. FORMATAGE DU CODE **62**
IV.4.4. COMMENTAIRES **62**
IV.5. GESTION DES VERSIONS **63**
IV.5.1. LES LOGICIELS DE GESTION DES VERSIONS **64**
IV.5.2. GESTION CENTRALISEE ET DECENTRALISEE **64**
IV.5.3. PRINCIPAUX LOGICIELS DE GESTION DES VERSIONS **66**
IV.6. TRAVAIL COLLABORATIF DANS LE PROJET LOGICIEL **66**
IV.7. TESTS DU LOGICIEL **67**
IV.8. LA DOCUMENTATION D'UN LOGICIEL **69**
IV.8.1. LA DOCUMENTATION TECHNIQUE **69**
IV.8.2. LA DOCUMENTATION UTILISATEUR **74**
LES FORMES DE LA DOCUMENTATION UTILISATEUR **74**
IV.8.3. LES RECOMMANDATIONS DE REDACTION D'UNE DOCUMENTATION **75**

CINQUIEME CHAPITRE LES METHODES DE DEVELOPPEMENT LOGICIEL **76**
V.1. LES METHODES FONCTIONNELLES **77**
IV.2. LES METHODES OBJET **78**
V.3. LES METHODES ADAPTATIVES **79**
V.3.1. LES METHODES PREDICTIVES **79**
V.3.2. LES METHODES AGILES **79**
V. 3.2.1. LA METHODE RAD **80**
V.3.2.2. LA METHODE DSDM **80**
V.3.2.3. LE MODELE XP **81**
V.3.2.4. LA METHODE SCRUM **82**
V.4. LES METHODES SPECIFIQUES **83**
V.4.1. LE CYCLE ERP **84**
V.4.2. LE MODELE RUP **84**
CONCLUSION **86**

BIBLIOGRAPHIE

- **ACSIOME,** « *Modélisation dans la conception des systèmes d'information* », Masson, 1989

- **Bertrand Meyer**, « *Conception et programmation orientées objet* » Editions Eyrolles, 2000

- **Boehm, B. W.**« *Software engineering economics* », Prentice-Hall, 1981, 0-13-822122-7

- **Booch, Grady,** "*Object-Oriented Analysis and Design, with applications*", 3rd Ed. Addison- Wesley, 2007

- **C. TESSIER,** « *La pratique des méthodes en informatique de gestion,* » Les Editions d'Organisation, 1995

- **GALACSI,** « *Comprendre les systèmes d'information : exercices corrigés d'analyse et de conception,* » Dunod, 1985

- **I. Somerville et Franck Vallée,** « *Software Engineering* » 6th Edition, Addison-Wesley, 2001

- **I. SOMMERVILLE**, « *Le génie logiciel et ses applications,* » Inter-Éditions, 1985

- **Marie-Claude Gaudel**, « Précis de génie logiciel », Editions Dunod, 2012

- **P. ANDRÉ et A. VAILLY,** « *Conception des systèmes d'information : Panorama des méthodes et techniques* », Ellipses, collection TECHNOSUP / Génie Logiciel, 2001

- **P. ANDRÉ et A. VAILLY,** « *Spécification des logiciels – Deux exemples de pratiques récentes : Z et UML* », Ellipses, collection TECHNOSUP / Génie Logiciel, 2001.

INTRODUCTION

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 était perçue à travers ces symptômes :

- *La qualité du logiciel livré était souvent déficiente* : Le produit ne satisfaisait pas les besoins de l'utilisateur, il consommait plus de ressources que prévu et il était 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 ne satisfaisant pas leurs cahiers des charges.
- Les coûts de développement d'un logiciel étaient presque impossible à prévoir et étaient généralement prohibitifs (*excessifs*)
- L'invisibilité du logiciel, ce qui veut dire qu'on s'apercevait souvent que le logiciel développé ne correspondait pas à la demande (*on ne pouvait l'observer qu'en l'utilisant « trop tard »).*
- La maintenance du logiciel était difficile, coûteuse et souvent à l'origine de nouvelles erreurs.

Mais en pratique, il est indispensable d'adapter les logiciels car leurs environnements d'utilisation changent et les besoins des utilisateurs évoluent. Il est rare qu'on puisse réutiliser un logiciel existant ou un de ses composants pour confectionner un nouveau système, même si celui-ci comporte des fonctions similaires. Tous ces problèmes ont alors mené à l'émergence d'une discipline appelée « *le génie logiciel* ». Ainsi, Les outils de génie logiciel et les environnements de programmation pouvaient aider à faire face à ces problèmes à condition qu'ils soient eux-mêmes utilisés dans un cadre méthodologique bien défini.

Nul n'ignore que les plans logiciels sont connus pour *leurs pléthores de budget*, et *leurs cahiers des charges*[1]*non respectées*. De façon générale, le développement de logiciel est une activité complexe, qui est loin de se réduire à la programmation. Le développement de logiciels, en particulier lorsque les programmes ont une certaine taille, nécessite d'adopter une méthode de développement, qui permet d'assister une ou plusieurs étapes du cycle de vie de logiciel.

[1] Le cahier des charges est un document recensant les spécifications/exigences d'un produit

Parmi les méthodes de développement, les approches objet, issues de la programmation objet, sont basées sur une modélisation du domaine d'application[2]. Cette modélisation a pour but de faciliter la communication avec les utilisateurs, de réduire la complexité en proposant des vues à différents niveaux d'abstraction, de guider la construction du logiciel et de documenter les différents choix de conception.

Le génie logiciel (*software engineering*) représente l'application de principes d'ingénierie au domaine de la création de logiciels. Il consiste à identifier et à utiliser des méthodes, des pratiques et des outils permettant de maximiser les chances de réussite d'un projet logiciel. Il s'agit d'une science récente dont l'origine remonte aux années 1970. A cette époque, l'augmentation de la puissance matérielle a permis de réaliser des logiciels plus complexes mais souffrant de nouveaux défauts : délais non respectés, coûts de production et d'entretien élevés, manque de fiabilité et de performances. Cette tendance se poursuit encore aujourd'hui. L'apparition du génie logiciel est une réponse aux défis posés par la complexification des logiciels et de l'activité qui vise à les produire.

OBJECTS DU COURS

Ce cours a pour objectif principal, d'initier les étudiants, à la conception des applications informatiques de façon systématique (*méthodique*) et reproductible (*rééditable*); en les incitant à rechercher et établir les fonctionnalités d'une application, et à les modéliser sous forme de cas d'utilisation et de scénarios ainsi que rechercher les classes et les acteurs nécessaires à la conception de l'application. Et, D'une façon spécifique ce cours vise à :

- Acquérir aux étudiants qui auront suivi ce cours, les bonnes pratiques de conception, comme l'utilisation de patron de conception (*design pattern*), le découpage en couches de l'architecture, la structuration en paquetages et le maquettage ;
- Maîtriser des techniques de génie logiciel, en se focalisant sur les approches par objets et par composants ;
- Présenter un aperçu de l'état de l'art en matière de génie logiciel.
- Exposer les principaux courants de pensées en matière de développement logiciel.
- Proposer un ensemble de pratiques pragmatiques qui permettent de survivre à un projet de développement de logiciel.

[2] C'est par exemple, le langage UML (*Unified Modeling Language*) qui est un langage de modélisation, qui permet d'élaborer des modèles objets indépendamment du langage de programmation, à l'aide de différents diagrammes.

DEFINITION DES CONCEPTS CLES

- ***Le génie logiciel*** est un domaine des sciences de l'ingénieur dont l'objet d'étude est la *conception*, *la fabrication,* et la *maintenance* des *systèmes informatiques complexes*.

- ***Un système*** est un ensemble d'*éléments* interagissant entre eux suivant un certains nombres de principes et de *règles* dans le but de réaliser un *objectif*.

- ***Un logiciel***[3]est un ensemble d'entités nécessaires au fonctionnement d'un processus de traitement automatique de l'information. Parmi ces entités, on trouve par exemple : des programmes (en format *code source* ou exécutables);des documentations d'utilisation ;des informations de configuration.

- **Un modèle** : est une représentation schématique de la réalité.

- **Une base de Données**: ensemble des données (*de l'organisation*) structurées et liées entre elles : stocké sur support à accès direct (*disque magnétique*) ; géré par un SGBD (*Système de Gestion de Bases de Données*), et accessible par un ensemble d'applications.

- **Une analyse** : c'est un processus d'examen de l'existant

- Une **Conception** : est un processus de définition de la future application informatique.

- **Un système d'Information** : ensemble des moyens (*humains et matériels*) et des méthodes se rapportant au traitement de l'information d'une organisation.

YENDE RAPHAEL Grevisse, PhD.
Professeur associé

[3] Un logiciel est en général un sous-système d'un système englobant. Il peut interagir avec des clients, qui peuvent être : des opérateurs humains (*utilisateurs, administrateurs, etc.)* ; d'autres logiciels ; des contrôleurs matériels. Il réalise une spécification c'est-à-dire son comportement vérifie un ensemble de critères qui régissent ses interactions avec son environnement.

PREMIER CHAPITRE – GENERALITES SUR LE GENIE (L'INGENIERIE) LOGICIEL

Le génie logiciel est une branche de l'ingénierie associée au développement de logiciels utilisant des principes, méthodes et procédures scientifiques bien définis. Le résultat de l'ingénierie logicielle est un produit logiciel efficace et fiable. Commençons par comprendre ce que signifie « *le génie logiciel* ». Le terme est composé de deux mots, le logiciel et l'ingénierie :

- *Le logiciel* est plus qu'un code de programme. En faite, un programme est un code exécutable, qui sert à des fins de calcul, par contre, le logiciel, est considéré comme une collection de code de programmation exécutable, des bibliothèques associées et de documentations. Ainsi, lorsque le logiciel, est conçu pour une exigence spécifique, est appelé un « *produit logiciel* ».

- D'autre part, *l'ingénierie* (*génie*) consiste à développer des produits, en utilisant des principes et méthodes scientifiques bien définis.

II.1. DEFINITION ET CONTEXTE D'ETUDES

IEEE définit le génie logiciel comme l'application d'une approche systématique, disciplinée et quantifiable au développement, à l'exploitation et à la maintenance des logiciels; c'est-à-dire l'application de l'ingénierie aux logiciels.

Fritz Bauer, un informaticien allemand, définit le génie logiciel comme l'établissement et l'utilisation de principes d'ingénierie afin d'obtenir des logiciels économiques, fiables et efficaces sur des machines réelles.

Dans le cadre de ce cours, le génie logiciel[4] sera considéré comme étant un ensemble d'activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser (*normaliser*) la production du logiciel et son suivi. Autrement dit, le génie logiciel est l'art de produire de bons logiciels, au meilleur rapport qualité prix. Remarquons que cette définition fait référence à deux aspects indispensables dans la conception d'un logiciel :

- *Le processus ou procédure de développement des logiciels* - ensemble de formalités, des marches à suivre et des démarches pour obtenir un résultat déterminé et ;
- *La maintenance et le suivi des logiciels* **-** ensemble d'opérations permettant de maintenir le fonctionnement d'un équipement informatique.

Le génie logiciel englobe les tâches suivantes :

- **La Spécification** : capture des besoins, cahier des charges, spécifications fonctionnelles et techniques

[4]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.

- **La Conception :** analyse, choix de la modélisation, définition de l'architecture, définition des modules et interfaces, définition des algorithmes
- **L'Implantation** : choix d'implantations, codage du logiciel dans un langage cible
- **L'Intégration :** assemblage des différentes parties du logiciel
- **La Documentation :** manuel d'utilisation, aide en ligne
- **La vérification** : tests fonctionnels, tests de la fiabilité, tests de la sûreté
- **La Validation** : recette du logiciel, conformité aux exigences du CDC
- **Le Déploiement** : livraison, installation, formation
- **La Maintenance** : corrections, et évolutions.

I.2. ÉVOLUTION DU LOGICIEL

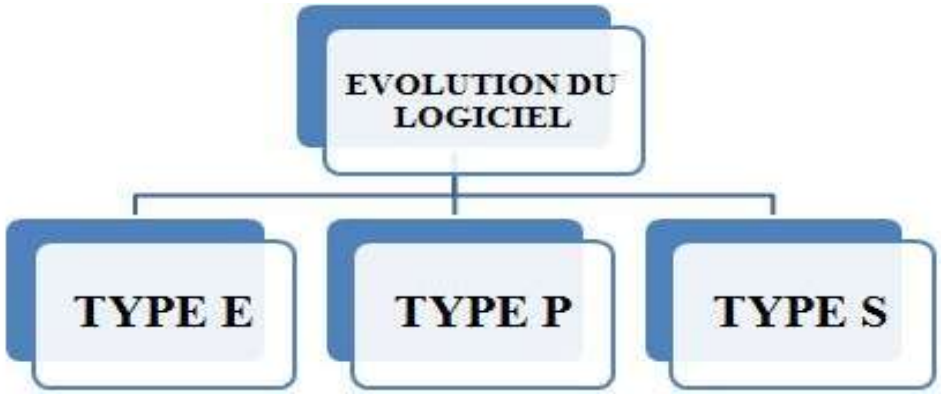
Le processus de développement d'un logiciel à l'aide de principes et de méthodes du génie logiciel est appelé « *évolution logicielle* ». Cela inclut le développement initial du logiciel et sa maintenance et ses mises à jour, jusqu'à ce que le logiciel désiré soit développé, ce qui répond aux exigences attendues.

L'évolution commence par le processus de *collecte des exigences*. Après quoi les développeurs créent un *prototype* du logiciel prévu et le montrent aux utilisateurs pour obtenir leurs commentaires dès les premières étapes du développement de logiciels. Les utilisateurs suggèrent des modifications sur lesquelles plusieurs mises à jour et maintenances consécutives continuent à évoluer. Ce processus passe au logiciel d'origine, jusqu'à ce que le logiciel souhaité soit accompli.

Même lorsque l'utilisateur dispose du logiciel souhaité, la technologie évolutive et les exigences changeantes forcent le logiciel à changer en conséquence. Récréer des logiciels à partir de zéro et aller en tête-à-tête avec des exigences n'est pas réalisable. La seule solution possible et économique consiste à mettre à jour le logiciel existant afin qu'il corresponde aux dernières exigences.

I.2.1. LOIS D'EVOLUTION DU LOGICIEL

Meir Lehman a donné des lois sur l'évolution des logiciels. Il a divisé le logiciel en trois catégories différentes :



- *Type S (type statique)* - Il s'agit d'un logiciel qui fonctionne selon des spécifications et des solutions définies. La solution et la méthode pour y parvenir, les deux sont immédiatement comprises avant le codage. Le logiciel de type s est le moins soumis à des modifications, ce qui en fait le plus simple. Par exemple, programme de calculatrice pour le calcul mathématique.

- *Type P (type pratique)* - Il s'agit d'un logiciel avec un ensemble de procédures. Ceci est défini par exactement ce que les procédures peuvent faire. Dans ce logiciel, les spécifications peuvent être décrites mais la solution n'est pas évidente instantanément. Par exemple, un logiciel de jeu.

- *E-type (type embarqué)* - Ce logiciel fonctionne étroitement comme exigence d'un environnement réel. Ce logiciel a un haut degré d'évolution car il existe divers changements dans les lois, les taxes, etc. dans les situations réelles. Par exemple, logiciel de trafic en ligne.

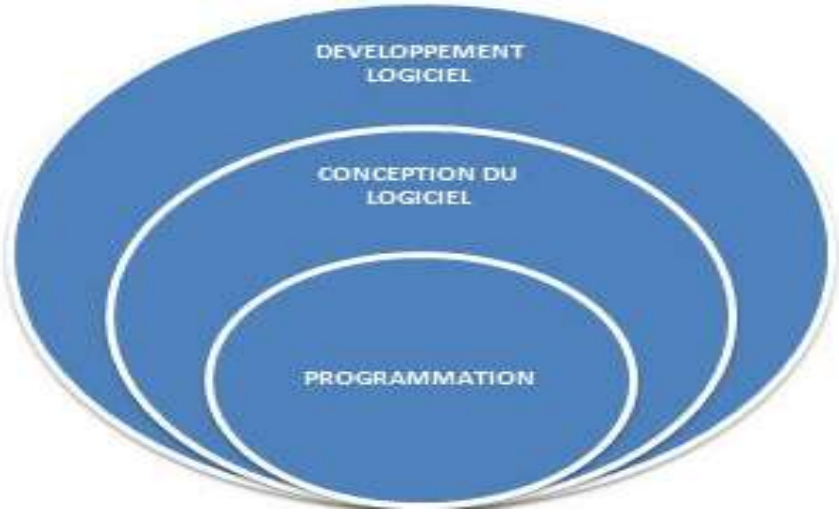
I.2.2. EVOLUTION DU LOGICIEL E-TYPE

Meir Lehman a donné huit lois pour l'évolution des logiciels de type E :

- *Changement continu* - Un logiciel de type E doit continuer à s'adapter aux changements du monde réel, sinon il devient progressivement moins utile.
- *Une complexité croissante* - Au fur et à mesure de l'évolution d'un système logiciel de type E, sa complexité a tendance à augmenter à moins que des travaux ne soient effectués pour le maintenir ou le réduire.
- *Conservation de la familiarité* - La familiarité avec le logiciel ou la connaissance de la façon dont il a été développé, pourquoi il a été développé de cette manière particulière, etc. doit être conservée à tout prix pour mettre en œuvre les modifications du système.
- *Poursuite de la croissance* - Pour qu'un système de type E destiné à résoudre un problème commercial, sa taille de mise en œuvre des modifications augmente en fonction des changements de style de vie de l'entreprise.
- *Réduire la qualité* - Un système logiciel de type E diminue en qualité, sauf si rigoureusement entretenu et adapté à un environnement opérationnel changeant.
- *Systèmes de rétroaction* – C'est un processus du logiciel pouvant se déclencher automatiquement après une opération défaillante, visant à provoquer une action correctrice en sens contraire. Les systèmes logiciels de type E constituent des systèmes de rétroaction multi-boucles à plusieurs niveaux et doivent être traités comme tels pour être modifiés ou améliorés avec succès.
- *Autorégulation* - Les processus d'évolution du système de type E s'autorégulent, la distribution des mesures du produit et du procédé étant presque normale.
- *Stabilité organisationnelle* - Le taux d'activité global effectif moyen dans un système de type E en évolution est invariant pendant la durée de vie du produit.

I.3. LES PARADIGMES LOGICIELS

Les paradigmes logiciels font référence aux méthodes et aux étapes qui sont prises lors de la conception du logiciel. Il existe de nombreuses méthodes proposées et sont en cours de réalisation, mais nous avons besoin de voir où se situent ces paradigmes dans le génie logiciel. Ceux-ci peuvent être combinés en différentes catégories, bien que chacun d'eux soit contenu l'un dans l'autre :



Il est à noter que le paradigme de la programmation est un sous-ensemble du paradigme de conception de logiciels, qui est en outre un sous-ensemble du paradigme de développement logiciel.

I.3.1. PARADIGME DE DEVELOPPEMENT LOGICIEL

Ce paradigme est connu sous le nom de paradigmes d'ingénierie logicielle, où tous les concepts d'ingénierie relatifs au développement de logiciels sont appliqués. Il comprend diverses recherches et la collecte des exigences qui aident le produit logiciel à construire. Ce paradigme fait partie du développement logiciel et inclut : *Le rassemblement des besoins* ; *la Conception des logiciels* et *la planification des tâches*.

I.3.2. PARADIGME DE CONCEPTION

Ce paradigme fait partie du développement de logiciels et comprend : *la conception* ; *la maintenance* et *l'organisation des activités à exécuter.*

I.3.3. PARADIGME DE PROGRAMMATION

Ce paradigme est étroitement lié à l'aspect programmation du développement logiciel. Cela inclut : *Le codage* ; *Le Test* et *L'intégration des besoins.*

I.4. BESOINS DU GENIE LOGICIEL

Les besoins du génie logiciel est dû au taux de changement plus élevé des besoins des utilisateurs et de l'environnement sur lequel le logiciel fonctionne :

- *Gros Logiciels (volumineux)* - Il est plus facile de construire un mur que dans une maison ou un bâtiment, de même que la taille des logiciels devient importante.

- *Évolutivité* - Si le processus logiciel n'était pas basé sur des concepts scientifiques et techniques, il serait plus facile de recréer de nouveaux logiciels que de mettre à niveau un logiciel existant.

- *Prix* – comme L'industrie du matériel a montré ses compétences et une production importante a fait baisser le prix du matériel informatique électronique. Mais le coût du logiciel reste élevé si le processus approprié n'est pas adapté.

- *Nature dynamique* - La nature toujours croissante et adaptative du logiciel dépend énormément de l'environnement dans lequel l'utilisateur travaille. Si la nature du logiciel évolue constamment, de nouvelles améliorations doivent être apportées dans le logiciel existant. C'est là que l'ingénierie logicielle joue un rôle important.

- *Gestion de la qualité* - Un meilleur processus de développement logiciel fournit un produit logiciel de meilleure qualité.

I.5. CARACTERISTIQUES D'UN BON LOGICIEL

La caractéristique d'un logiciel est un ensemble de traits dominants ou l'expression de la correspondance entre une cause (*grandeur d'entrée*) et un effet (*grandeur de sortie*) dans la production ou le processus de développement des logiciels. Un produit logiciel doit évaluer en fonction de ce qu'il offre et de sa facilité d'utilisation. Un bon logiciel doit satisfaire les 3 catégories de critères suivants :

- Critères généraux ;
- Critères externes,
- Critères internes.

I.5.1. CRITERES GENERAUX

Les critères généraux sont en somme, des principes et éléments de référence qui permettent de juger ; d'estimer et de vérifier régulièrement si le processus de développement d'un logiciel possède ou non les différentes propriétés déterminées. Cette catégorie peut se matérialiser selon 3 différents processus (aspects) :

- *Opérationnel* : Cela nous indique dans quelle mesure le logiciel fonctionne bien dans les opérations. Ces opérations peuvent être mesurées entre autre part : *budgétisation, facilité d'utilisation, efficacité, exactitude, fonctionnalité, fiabilité, sécurité.*

- *Transitionnel* : Cet aspect est important lorsque le logiciel est déplacé d'une plate-forme à une autre : *Portabilité, Interopérabilité, Réutilisation et Adaptabilité.*

- *Maintenance* : Cet aspect explique comment un logiciel a la capacité de se maintenir dans une infrastructure et environnement en constante évolution : *maintenabilité, modularité, Flexibilité et Évolutivité*.

En résumé, le génie logiciel est une branche de l'informatique, qui utilise des concepts d'ingénierie bien définis pour produire des *produits logiciels efficaces*, *durables*, *évolutifs*, *économiques* et *ponctuels*.

I.5.2. CRITERES EXTERNES

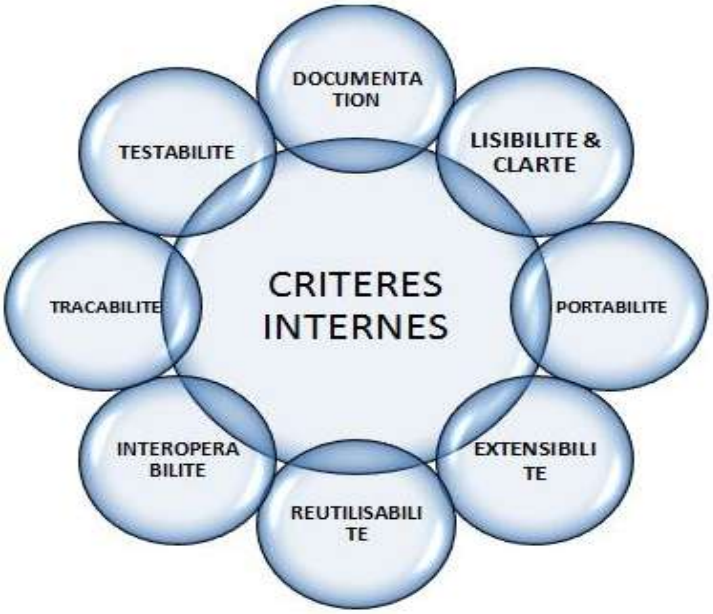
Les critères externes expriment ce qu'est un bon logiciel du point de vue des utilisateurs. Un logiciel de qualité doit être :

- **Fiabilité** : (*correction, justesse et conformité*) : le logiciel est conforme à ses spéciations, les résultats sont ceux attendus.
- **Robustesse et Sureté** :(*dysfonctionnements ou ne plante pas*) : le logiciel fonctionne raisonnablement en toutes circonstances, rien de catastrophique ne peut survenir, même en dehors des conditions d'utilisation prévues
- **Efficacité** : (*Le logiciel fait-il bon usage de ses ressources, en terme d'espace mémoire, et temps d'exécution*),
- **Convivialité et Utilisabilité :** (*Est-il facile et agréable à utiliser*),
- **Documentable** : (*accompagné d'un manuel utilisateur, ou d'un tutoriel).*
- **Ergonomique[5] :** L'architecture du logiciel doit particulièrement être adaptée aux conditions de travail de l'utilisateur
- **Sécurité :** c'est la sûreté (*assurance*) et la garantie offerte par un logiciel, ou l'absence du danger lors de l'exploitation du logiciel.
- **Adéquation et validité :** c'est la conformité au maquettage du logiciel et au but qu'on se propose.
- **Intégrité :** c'est l'état d'un logiciel a conservé sans altération ses qualités et son degré originel. Autrement dit, C'est l'aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisé.

[5]Ergonomie est une étude quantitative et qualitative du travail dans une entreprise, visant à améliorer les conditions de travail et à accroitre la productivité.

I.5.3. CRITERES INTERNES

Les critères de qualité internes expriment ce qu'est un bon logiciel du point de vue du développeur. Ces critères sont essentiellement liés à la maintenance d'un logiciel. Un bon logiciel doit être facile à maintenir, et pour cela doit être :



- **Documentable :** (*a-t-il été précédé par un document de conception ou architecture*).
- **Lisibilité et Clarté** : (*est-il écrit proprement, et en respectant les conventions de programmation*),
- **Portabilité***:* Un même logiciel doit pouvoir fonctionner sur plusieurs machines ainsi le rendre indépendant de son environnement d'exécution ;
- **Extensibilité :** *(est-il souple ou flexible ? ou permet-il l'ajout possible de nouvelles fonctionnalités*).
- **Réutilisabilité** : (*des parties peuvent être réutilisées pour développer d'autres logiciels similaires*).
- **Interopérabilité et coulabilité** : Un logiciel doit pouvoir interagir en synergie avec d'autres logiciels.
- **Traçabilité :** c'est la possibilité de suivre un produit aux différents stades de sa production, de sa transformation et de sa commercialisation.
- **Testabilité et vérifiabilité** : c'est la possibilité de soumettre un logiciel à une épreuve de confirmation de la tâche à exécuter.

I.6. CATEGORIES DE LOGICIELS

Il existe 2 catégories des logiciels notamment :

- **LOGICIELS GENERIQUES** : c'est sont des logiciels qui vendus comme les produits courants sur le marché informatique. Dans cette catégorie, on en distingue autant :

  → **Logiciels amateurs** : Il s'agit de logiciels développés par des « *amateurs*[6]» (*par exemple par des gens passionnés ou des étudiants qui apprennent à programmer*). Bref, ce sont des logiciels sans impact économique significatif sur l'ensemble.

  → **Logiciels « jetables » ou « consommables »** : Il s'agit de logiciels comme par exemple les logiciels des traitements de texte ou les tableurs pour les entreprises. Ces logiciels ne coûtent pas très cher, et peuvent être remplacés facilement au sein d'une entreprise sans engendrer des risques majeurs. Ils sont souvent largement diffusés.

- **LOGICIELS SPECIFIQUES** : ce sont des logiciels développés pour une application précise et destinés à un seul client. Dans cette catégorie, on en distingue autant :

  ➘ **Logiciels essentiels au fonctionnement d'une entreprise** : Ce type de logiciel est le fruit d'un investissement non négligeable et doit avoir un comportement fiable, sûr et sécurisé.

  ➘ **Logiciels critiques** : Il s'agit de logiciels dont l'exécution est vitale, pour lesquels une erreur peut coûter très cher ou coûter des vies humaines. Exemple : domaines du transport, de l'aviation, de la médecine, de l'armement, etc.

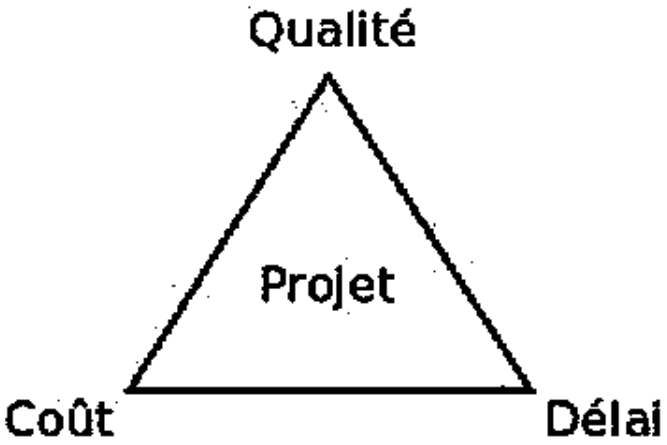
L'objectif de qualité d'un logiciel est différent suivant la catégorie de logiciel. En particulier, les logiciels essentiels au fonctionnement d'une entreprise, et plus encore les logiciels critiques, doivent avoir un haut niveau de qualité.

[6]**Un amateur** peut être considéré comme une personne qui présente une préférence marquée ou exclusive pour une chose ou une activité donnée.

I.7. ENJEUX DU GENIE LOGICIEL

Un enjeu peut être considéré comme un ensemble des risques encourus par un développeur pour en mise en œuvre d'un logiciel. Le génie logiciel vise à rationaliser et à optimiser le processus de production d'un logiciel. Les enjeux associés sont multiples :

- Adéquation aux besoins du client.
- Respect des délais de réalisation prévus.
- Maximisation des performances et de la fiabilité.
- Facilitation de la maintenance et des évolutions ultérieures.



Comme tout projet, la réalisation d'un logiciel est soumise à des exigences contradictoires et difficilement conciliables (*triangle coût-délai-qualité*).La qualité d'un logiciel peut s'évaluer à partir d'un ensemble de facteurs tels que :

- Le logiciel répond-il aux besoins exprimés ?
- Le logiciel demande-t-il peu d'efforts pour évoluer aux regards de nouveaux besoins ?
- Le logiciel peut-il facilement être transféré d'une plate-forme à une autre ? ...

Sans être une solution miracle, le génie logiciel a pour objectif de maximiser la surface du triangle en tenant compte des priorités du client.

I.8. DIMENSIONS DU GENIE LOGICIEL

La dimension peut être envisagée comme étant un ensemble de mesures de chacune des grandeurs nécessaires à l'évaluation du logiciel. Le génie logiciel couvre l'ensemble du cycle de vie d'un logiciel. Il étudie toutes les activités qui mènent d'un besoin à la livraison du logiciel, y compris dans ses versions successives, jusqu'à sa fin de vie. Les dimensions du génie logiciel sont donc multiples :

- Analyse des besoins du client.
- Définition de l'architecture du logiciel.
- Choix de conception.
- Règles et méthodes de production du code source.
- Gestion des versions.
- Test du logiciel.
- Documentation.
- Organisation de l'équipe et interactions avec le client.
- ...

I.9. LES PRINCIPES DU GENIE LOGICIEL

Un certain nombre de grands principes (de bon sens) se retrouvent dans toutes ces méthodes. En voici une liste proposée par Ghezzi[7] :

1. **La rigueur** : Les principales sources de défaillances d'un logiciel sont d'origine humaine. À tout moment, il faut se questionner sur la validité de son action. Des outils de vérification accompagnant le développement peuvent aider à réduire les erreurs. Cette famille d'outils s'appelle (*CASE "Computer Aided Software Engineering")*. C'est par exemple: typeurs, générateurs de code, assistants de preuves, générateurs de tests, outil d'intégration continue, fuzzer, . . .

2. **La Généralisation** : regroupement d'un ensemble de fonctionnalités semblables en une fonctionnalité paramétrable (*généricité*[8] *et héritage*)

3. **La Structuration** : c'est la manière de décomposer un logiciel (*utilisation d'une methode bottom-up ou top-down*). c'est la décomposition des problèmes en sous-problèmes indépendants[9]. Outre, Il s'agit *de la Décorrélation.* les problèmes pour n'en traiter qu'un seul à la fois. *Ou de la Simplification.* les problèmes (*temporairement*) pour aborder leur complexité progressivement. C'est par exemple : le modèle TCP.

[7] C. Ghezzi, "*Fundamentals of Software Engineering*" Prentice Hall, 2nd edition, 2002.
[8]Un logiciel est *générique*lorsqu'il est adaptable.
[9] Edsger W. Dijkstra *On the role of scientific thought*.
http://cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

4. **La modularité**[10] : Il s'agit de partitionner le logiciel *en modules* qui ont *une cohérence interne (des invariants*) ; et possèdent *une interface ne divulguant* sur le contenu du module que ce qui est strictement nécessaire aux modules clients. L'évolution de l'interface est *indépendante* de celle de l'implémentation du module. Les choix d'implémentation sont *indépendants* de l'utilisation du module. Ce mécanisme s'appelle le camouflage de l'information (*information hiding*).

5. **L'abstraction**[11] : Il s'agit de présenter des *concepts généraux* regroupant un certain nombre de cas particuliers et de raisonner sur ces concepts généraux plutôt que sur chacun des cas particuliers. Le fait de fixer *la bonne granularité de détails* permet de raisonner plus efficacement et de factoriser le travail en instanciant le raisonnement général sur chaque cas particulier. C'est par exemple : les classes abstraites dans les langages à objets, le polymorphisme de *Caml* et le *generics* de Java, les foncteurs de *Caml* et le *templates* de C++, . . .

6. **La construction incrémentale**: Un développement logiciel a plus de chances d'aboutir s'il suit un cheminement *incrémental*[12] (*baby-steps*)

7. **La Documentation :** correspond à la gestion des documents incluant leur *identification, acquisition, production, stockage et distribution*. Il est primordial de prévoir les évolutions possibles d'un logiciel pour que la maintenance soit la plus efficace possible. Pour cela, il faut s'assurer que les modifications à effectuer soient le plus locales possibles. Ces modifications ne devraient pas être *intrusives* car les modifications du produit existant remettent en cause ses précédentes validations. Concevoir un système suffisamment riche pour que l'on puisse le modifier incrémentalement est l'idéal.

8. **La Vérification :** c'est la détermination du respect des spéciations établies sur la base des besoins identités dans la phase précédente du cycle de vie.

[10] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", Communications of the ACM Vol. 15 Issue 12, 1972

[11] L'abstraction est un mécanisme qui permet de présenter un contexte en exprimant les éléments pertinents et en omettant ceux qui ne le sont pas

[12] En informatique, c'est une quantité constante ajoutée à la valeur d'une variable à chaque exécution d'une instruction, généralement répétitive, d'un programme

DEUXIEME CHAPITRE – LE CYCLE DE VIE DU DEVELOPPEMENT LOGICIEL

II.1. DEFINITION

De par sa définition, Le Cycle de vie du développement d'un logiciel (SDLC[13]) est un ensemble d'étapes de la réalisation, de l'énoncé des besoins à la maintenance au retrait du logiciel sur le marché informatique. 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 a proposition ou la décision de développer un logiciel et se termine avec sa mise hors service.

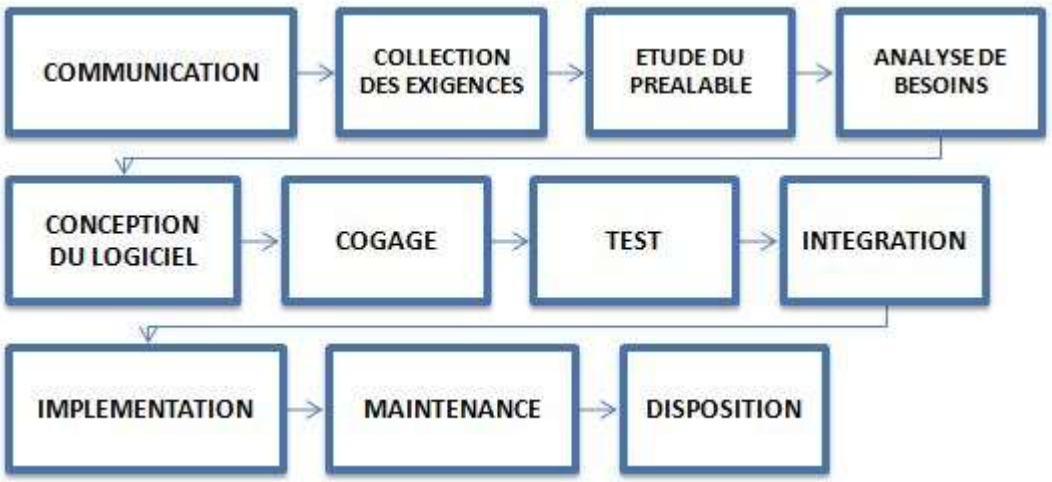
L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation. C'est ainsi que l'objectif principal du cycle de vie du développement d'un logiciel est de permettre la détection des erreurs au plus tôt et par conséquent, maîtriser la qualité du produit, les délais de sa réalisation et les coûts associés.

II.2. ÉTAPES DU CYCLE DE VIE DU DEVELOPPEMENT D'UN LOGICIEL

Le développement de logiciel impose d'effectuer un certain nombre d'étapes. Il existe de nombreux modèles de cycle de vie du développement d'un logiciel, les plus courants comportent les phases suivantes :

- La communication ;
- La collection des exigences ;
- L'étude du préalable (faisabilité ou opportunité) ;
- Définition et analyse de besoins (Spécification) ;
- La conception du logiciel ;
- Le codage ;
- Les tests ;
- L'intégration ;
- L'installation ;
- La maintenance ;
- La disposition.

[13] Software Development Life cycle



II.1.1. LA COMMUNICATION

C'est la première étape du cycle de vie de développement logiciel où l'utilisateur initie la demande du produit logiciel souhaité. Ce dernier contact le service du « *provider* » et essaie de négocier les termes du contrat. Il soumet alors sa demande à l'entreprise du fournisseur des services par écrit.

II.1.2. LA COLLECTION DES EXIGENCES

Cette étape fait avancer le travail de l'équipe du développement du logiciel jusqu'à la visibilité du projet informatique. A ce niveau, l'équipe de développement discute avec certains partenaires des divers problèmes du domaine et essaie de proposer autant d'informations possibles sur les différentes exigences. A ce niveau, les exigences sont contemplées et isolées en fonction des *besoins des utilisateurs* ; les *exigences du système* et *les exigences fonctionnelles*. Les exigences sont collectées en utilisant un nombre donné des pratiques telles que :

- Etude du système ou logiciel existant ou obsolète ;
- Conduite des interviews auprès des utilisateurs et développeurs ;
- Référencer aux différentes bases de données ;
- Et la collection des réponses au moyen de questionnaires.

II.1.3. ÉTUDE DU PREALABLE (*étude de l'opportunité*)

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 œuvre ?

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.

II.1.4. DEFINITION ET ANALYSE DES BESOINS (*spécification*[14])

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 ;
- les 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 et le matériel ;
- contraintes de réalisation, telles que l'environnement de développement, le langage de programmation à utiliser, les procédures et normes à suivre, etc.

[14] *Élaboration du cahier des charges et des tests de recette/validation*

Il est judicieux de préparer pendant la phase d'analyse les procédures qui seront mises en œuvre pour vérifier que le logiciel, une fois construit, est conforme à la spécification, que nous l'appellerons ***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. Les Points clés :

- 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

II.1.5. LA CONCEPTION DU LOGICIEL

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

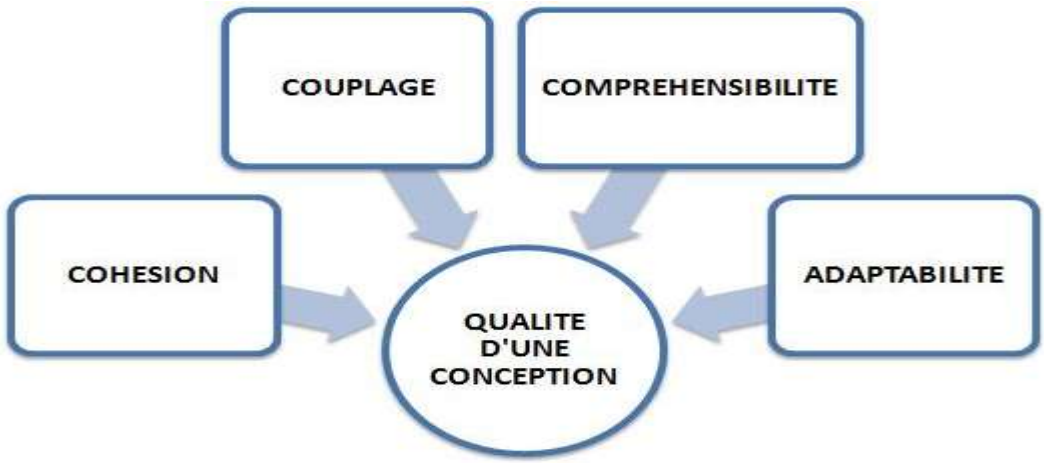
- **Conception générale ou conception architecturale (*preliminary design ou architectural design*) :** Si nécessaire, il faut commencer par l'ébauche de plusieurs variantes de solutions et choisir celle qui offre le meilleur rapport entre coûts et avantages. Il faut ensuite figer la solution retenue, 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 aient 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***.

- **Conception détaillée (*detailed design*)** : 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.

II.1.5.1. QUALITE D'UNE CONCEPTION LOGICIELLE

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 :



A. La Cohésion

La cohésion d'un composant permet de mesurer la qualité de sa structuration, autrement dit, c'est un ensemble des forces solidaires développées à l'intérieur d'un logiciel. 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*** identifient, en 1979, sept niveaux de cohésion présentées ci-après, du plus fort au plus faible :

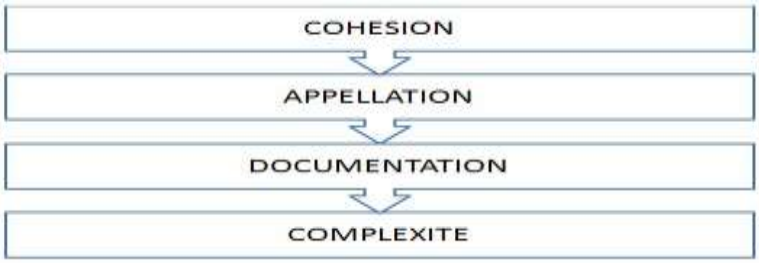
- **La cohésion fonctionnelle** : Ici, Le module assure une seule fonction et Tous les éléments du composant contribuent à atteindre un seul objectif (si *un élément est supprimé, l'objectif ne sera pas atteint)*.
- **La cohésion séquentielle** : Dans ce type de cohésion, la sortie d'un élément est utilisée en entrée d'un autre (*dans ce cas, l'ordre des actions est important*)
- **La cohésion de communication** : Lorsque tous les éléments d'un composant travaillent sur les mêmes données.
- **La cohésion procédurale** : Dans ce cas, les éléments d'un composant constituent une seule séquence de contrôle.
- **La cohésion temporelle** : 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** : Tous les éléments d'un composant effectuent des opérations semblables comme, par exemple, module qui édite tous les types de transactions et difficile à modifier.
- **La cohésion occasionnelle** : 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.

B. Le Couplage

C'est la situation de deux ou plusieurs états dont les impératifs de défense sont assurés par l'intégration de leurs forces respectives dans le cadre d'une stratégie commune. Le couplage est relatif à la cohésion. C'est une indication de la force d'interconnexion des différents 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.

C. La compréhensibilité

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 des 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 ?

D. L'adaptabilité

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.

II.1.6. LE CODAGE

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

II.1.7. TESTS

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.

II.1.8. INTEGRATION

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

II.1.9. 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.

II.1.10. 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 comportement 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 performances 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. Voici les différentes formes de maintenance qui peuvent exister en génie logiciel :

- ***La Maintenance corrective*** : c'est une maintenance qui Corrige les erreurs et les défauts d'utilité, d'utilisabilité, de fiabilité... cette maintenance Identifie également les défaillances, et les dysfonctionnements en localisant la partie du code responsable. Elle Corrige et estime l'impact d'une modification. Attention, La plupart des corrections introduisent de nouvelles erreurs et Les coûts de correction augmentent exponentiellement avec le délai de détection. Bref, la maintenance corrective donne lieu à de nouvelles livraisons (*release*).

- ***La Maintenance adaptative :*** c'est une maintenance qui ajuste le logiciel pour qu'il continue à remplir son rôle compte tenu du l'évolution des Environnements d'exécution, des Fonctions à satisfaire et des Conditions d'utilisation. C'est par exemple le changement de SGBD, de machine, de taux de TVA, an 2000 ...

- ***La Maintenance perfective et d'extension*** : c'est une maintenance qui sert a accroître et améliorer les possibilités du logiciel afin de donne lieu à de nouvelles versions. C'est par exemple ; les services offerts, l'interface utilisateur, les performances...

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.

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

II.1.11. DISPOSITION

Comme en informatique, le temps est un facteur indispensable qui doit être surveillé quant à tout ce qui concerne le développement des produits informatiques. C'est ainsi qu'un logiciel peut subir une détérioration proportionnellement à sa performance. Il peut être complètement obsolète ou exigera une intense mise à jour. Cette phase inclut l'archivage des données et les exigences de composants logiciels ; la fermeture du système ; la disposition planifiée des activités et de la terminaison du système quant à la période appropriée de la terminaison du logiciel. Bref, cette phase concerne la gestion de versions.

TROISIEME CHAPITRE - MODELISATION DE PROCESSUS DE DEVELOPPEMENT LOGICIEL

La modélisation de développement logiciel aide les développeurs à sélectionner la stratégie pour développer le produit logiciel. Ainsi, chaque modélisation de développement possède ses outils propres, ses méthodes et ses procédures qui permettent d'exprimer clairement et définissent le cycle de vie de développement du logiciel.

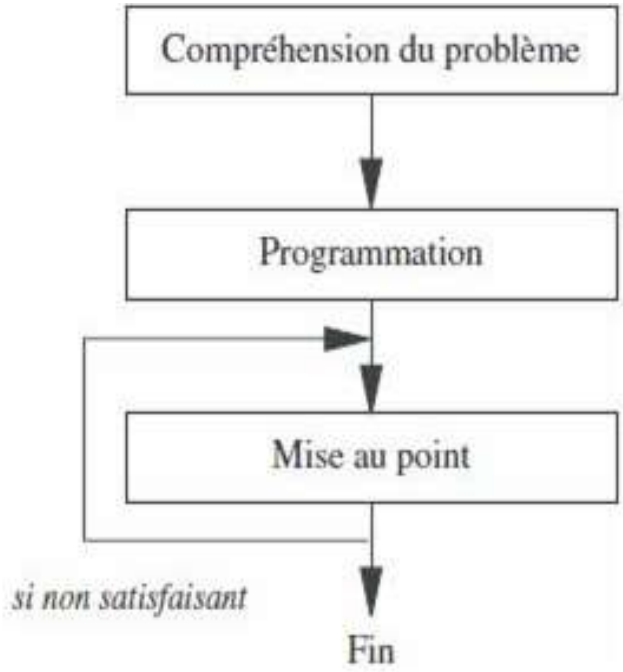
Les modèles du cycle de vie du logiciel sont des « *plans de travail* » qui permettent de planifier le développement. Plus le logiciel à développer est complexe (*taille, algorithmes*) et critique, plus il est important de bien contrôler le processus de développement et plus les documents qui accompagnent le logiciel doivent être précis et détaillés.

Il s'avère de nos jours, que nous ne pouvons plus avoir une démarche unique dans le développement de projets informatiques, mais qu'il faut construire le découpage temporel en fonction des caractéristiques de l'entreprise et du projet. On s'appuie pour cela sur des découpages temporels génériques, appelés *modèles de développement* (*process models*) ou modèles de cycle de vie d'un projet informatique. Les principaux modèles sont :

- le modèle du *code-and-fix* ;
- le modèle de la transformation automatique ;
- le modèle de la cascade ;
- le modèle en V ;
- le modèle par incrément;
- le modèle par prototypage ;
- le modèle de la spirale.

III.1. LE MODELE DU CODE-AND-FIX

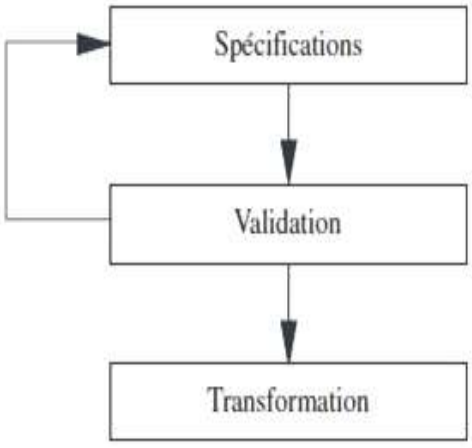
C'est un modèle qui repose sur la possibilité d'une détermination facile des besoins : une première phase de Compréhension du problème est suivie d'une phase de Programmation ; puis une phase de Mise au point, parfois en collaboration avec l'utilisateur du futur système, est répétée jusqu'à l'atteinte du résultat visé.



Le modèle du code-and-fix.

II.2. LE MODELE DE TRANSFORMATION AUTOMATIQUE

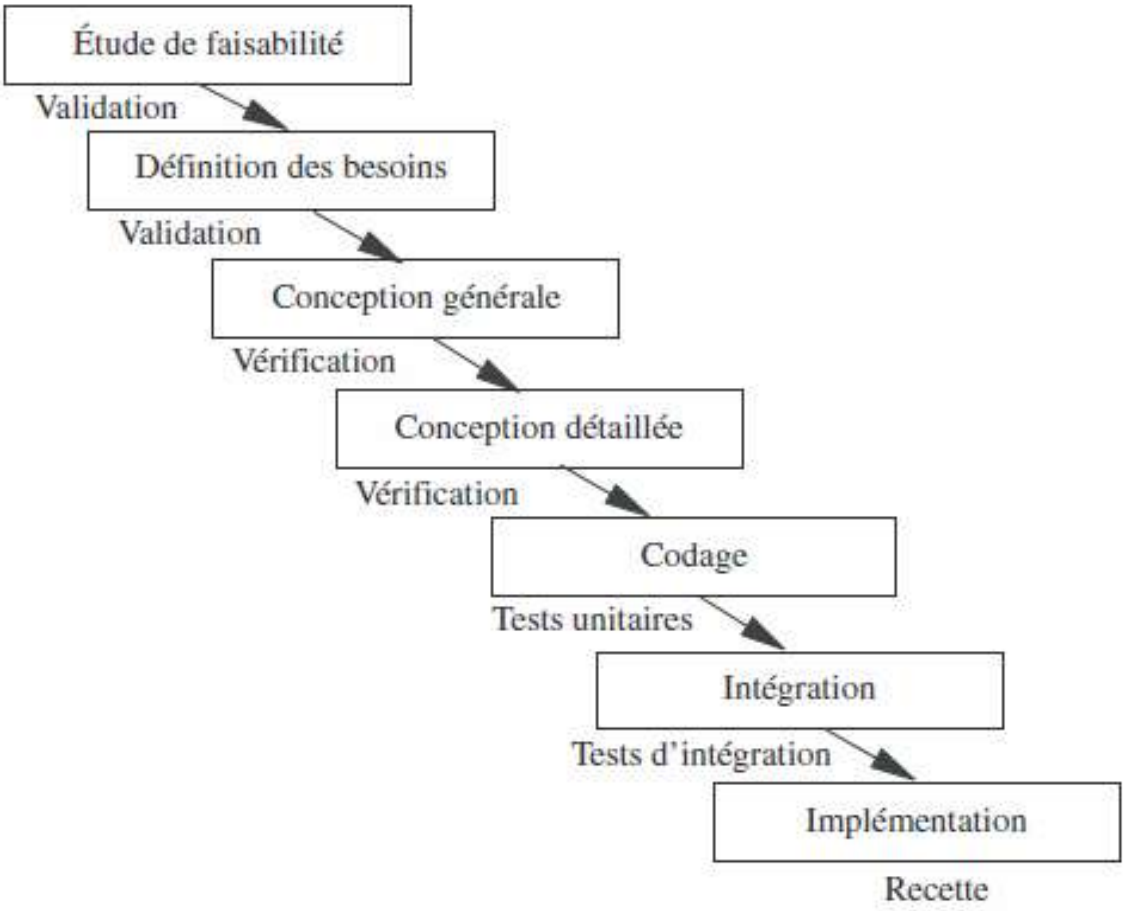
C'est un modèle basé sur la possibilité de transformer automatiquement des spécifications en programmes. L'essentiel de l'effort va donc porter sur une description exhaustive des spécifications qui devront être complètement validées. Une succession de cycles *Phase de Spécifications – Phase de Validation* s'achève par la *phase de Transformation*, où s'effectue la génération du code.



Le modèle de la transformation automatique

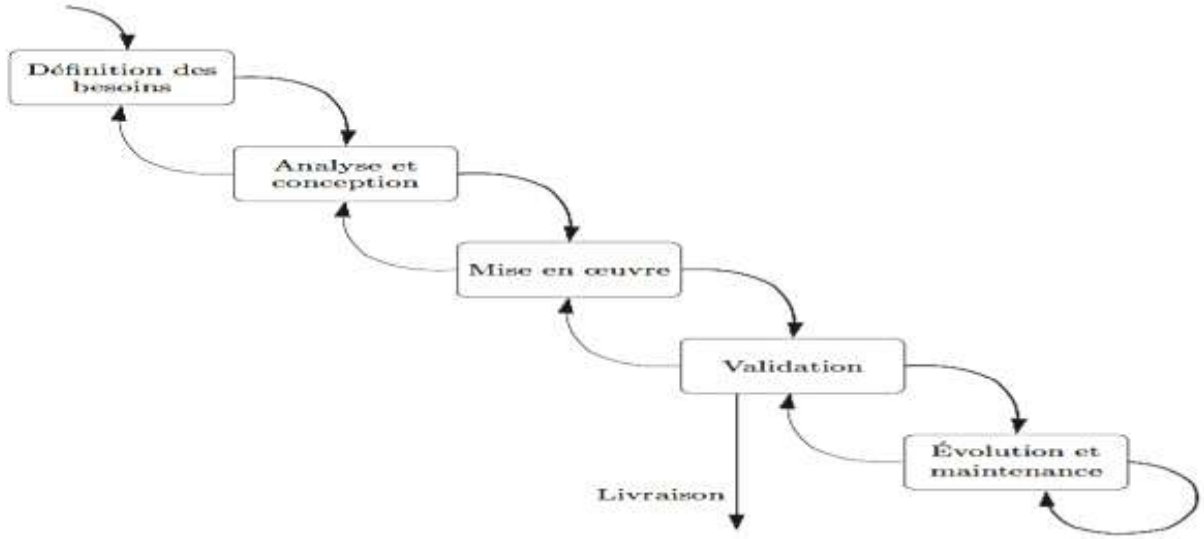
II.3. LE MODELE EN CASCADE (WATERFALL MODEL)

C'est un modèle qui a comme objectif majeur de jalonner (*présenter sobrement*) rigoureusement le processus de développement et de définir de façon précise les rôles respectifs du fournisseur qui produit un livrable et du client qui accepte ou refuse le résultat. Le découpage temporel se présente comme une succession de phases affinant celles du découpage classique : Étude de faisabilité, Définition des besoins, Conception générale, Conception détaillée, Programmation, Intégration, Mise en œuvre. Chaque phase donne lieu à une validation officielle. Si le résultat du contrôle n'est pas satisfaisant, on modifie le livrable. En revanche, il n'y a pas de retour possible sur les options validées à l'issue de phases antérieures.

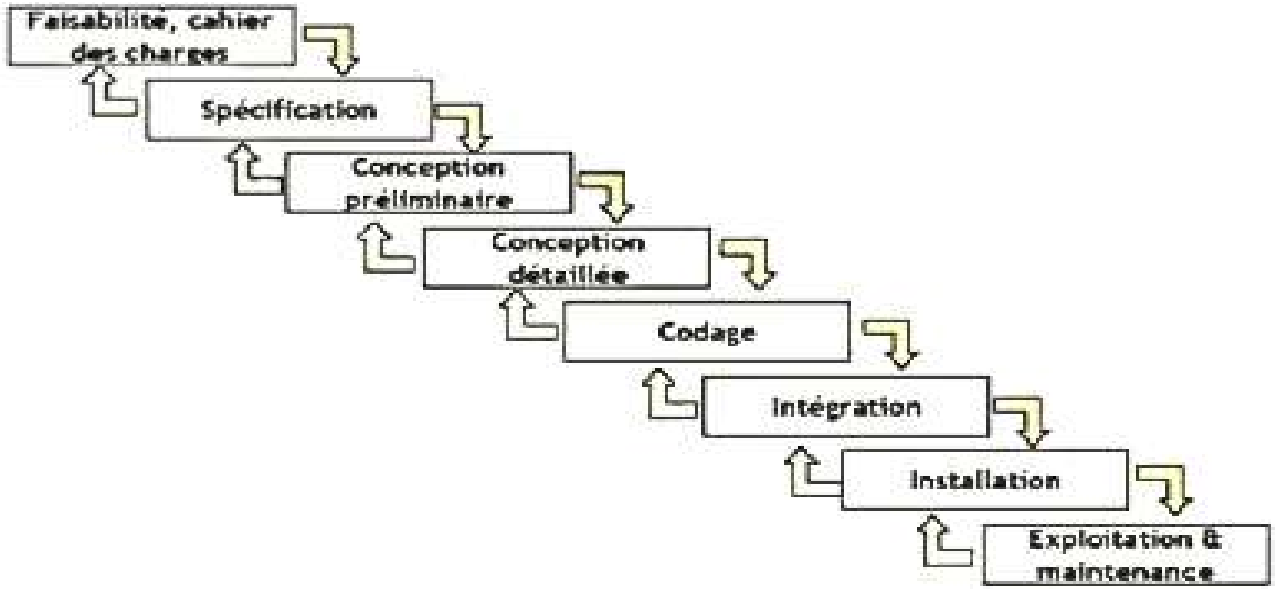


Le cycle de vie dit de la « ***cascade*** » date de ***1970***, il est l'œuvre de ***Royce***. Ce Cycle de vie est linéaire (*présentoir*) sans aucune évaluation entre le début du projet et la validation. Ici, Le projet est découpé en phases successives dans le temps et à chaque phase correspond une activité principale bien précise produisant un certain nombre de livrables et On ne passe à l'étape suivante que si les résultats de l'étape précédente sont jugés satisfaisants.

L'activité d'une étape se réalise avec les résultats fournis par l'étape précédente ; ainsi, chaque étape sert de contrôle du travail effectué lors de l'étape précédente et Chaque phase ne peut remettre en cause que la phase précédente ce qui, dans la pratique, s'avère insuffisant. L'élaboration des spécifications est une phase particulièrement critique : les erreurs de spécifications sont généralement détectées au moment des tests, voire au moment de la livraison du logiciel à l'utilisateur. Leur correction nécessite alors de reprendre toutes les phases du processus. Ce modèle est mieux adapté aux petits projets ou à ceux dont les spécifications sont bien connues et fixes. Dans le modèle en cascade, on effectue les différentes étapes du logiciel de façon séquentielle. Les interactions ont lieu uniquement entre étapes successives : on s'autorise des retours en arrière uniquement sur l'´étape précédente. Par exemple, un test ne doit pas remettre en cause la conception architecturale.

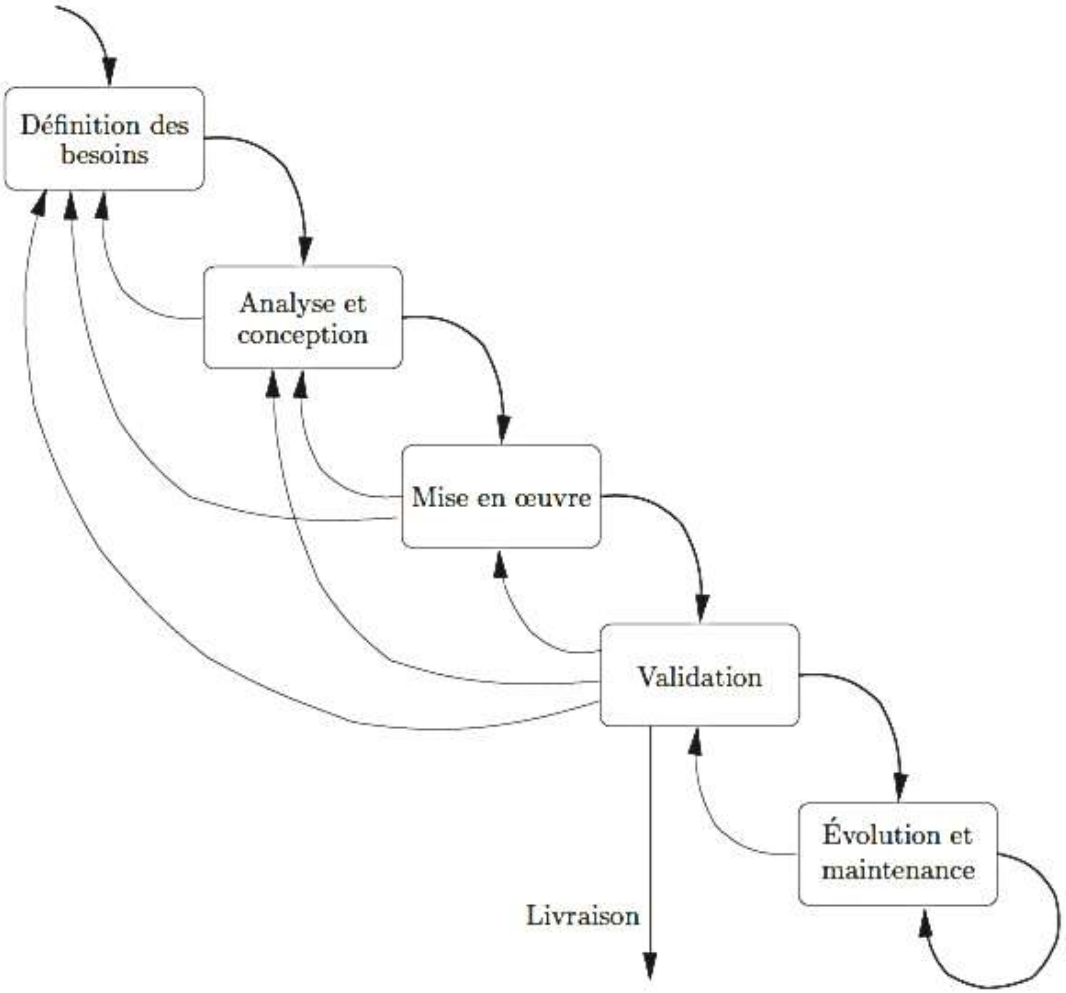


Modèle du cycle de vie en cascade N° 1



Modèle du cycle de vie en cascade N°2

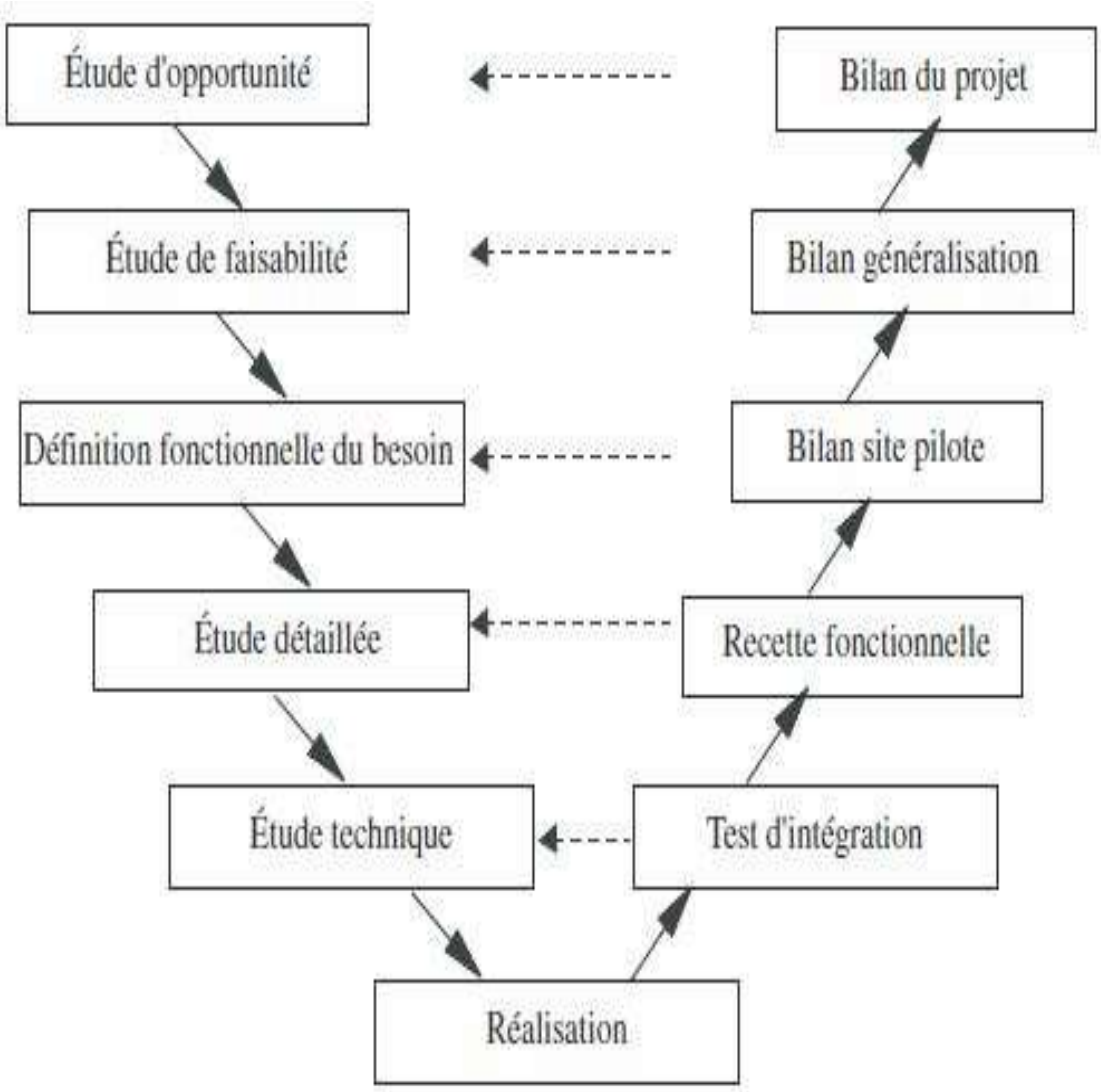
Lorsque qu'un test remet en cause la conception détaillé ou architecturale ou, pire, les spécifications ou la définition des besoins, le modèle devient incontrôlable :



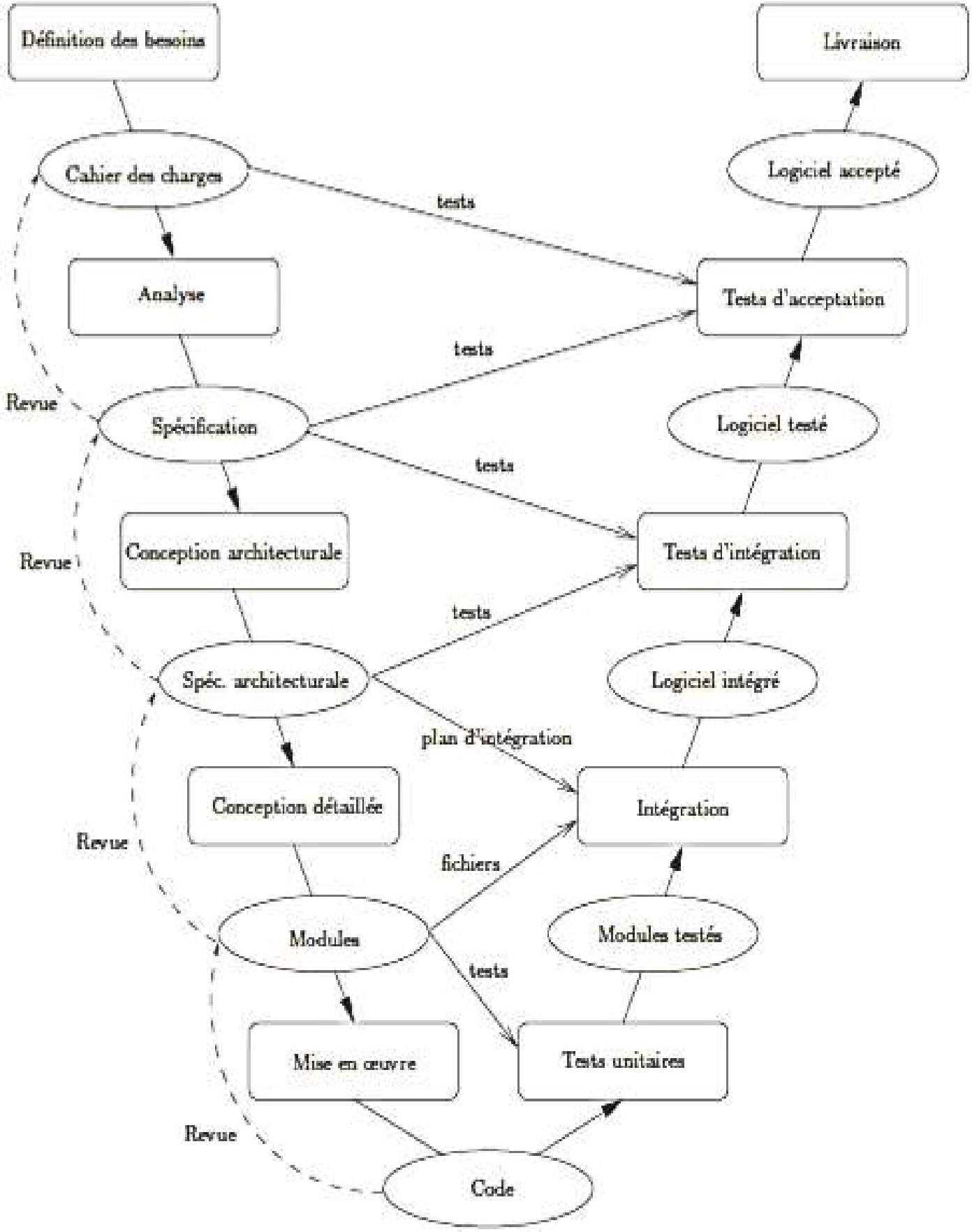
Modèle du cycle de vie en cascade Incontrôlable

II.4. LE MODELE EN V.

Le modèle en V du cycle de vie du logiciel précise la conception des tests : (*les tests système sont préparés à partir de la spécification; les tests d'intégration sont préparés à partir de la conception architecturale ; les tests unitaires sont préparés à partir de la conception détaillée des composants*). Dérivé du modèle de la cascade, le modèle en V du cycle de développement montre non seulement l'enchaînement des phases successives, mais aussi les relations logiques entre phases plus éloignées. Ce modèle fait apparaître le fait que le début du processus de développement conditionne ses dernières étapes. Le modèle du cycle de vie en V est souvent adapté aux projets de taille et de complexité moyenne.



La première branche correspond à un modèle en cascade classique. Toute description d'un composant est accompagnée de définitions de tests. Avec les jeux de tests préparés dans la première branche, les étapes de la deuxième branche peuvent être mieux préparées et planifiées. La seconde branche correspond à des tests effectifs effectués sur des composants réalisés. L'intégration est ensuite réalisée jusqu'à l'obtention du système logiciel final. L'avantage d'un tel modèle est d'éviter d'énoncer une propriété qu'il est impossible de vérifier objectivement une fois le logiciel réalisé. Le cycle en V est le cycle qui a été normalisé, il est largement utilisé, notamment en informatique industrielle et en télécommunication. Ce modèle fait également apparaitre les documents qui sont produits à chaque étape, et les «*revues*» qui permettent de valider les différents produits.

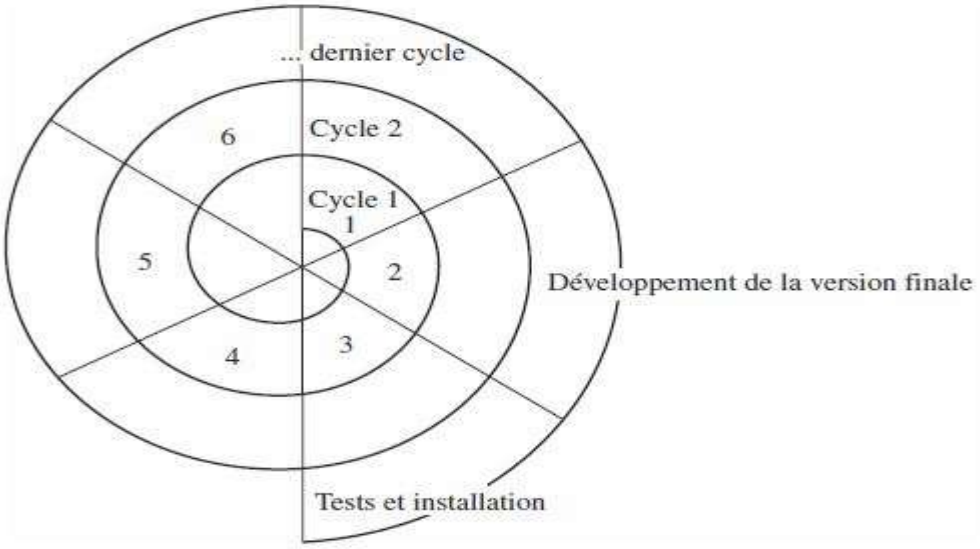


Modèle du cycle de vie en « V » N° 2

III.5. LE CYCLE DE VIE EN SPIRALE

Ce modèle repose sur le même principe que le modèle évolutif, mais il s'inscrit dans une relation contractuelle entre le client et le fournisseur. De ce fait les engagements et validations présentent un caractère formalisé. Chaque cycle donne lieu à une contractualisation préalable, s'appuyant sur les besoins exprimés lors du cycle précédent. Un cycle comporte six phases :

- Analyse du risque ;
- Développement d'un prototype (*modèle, archétype…)* ;
- Simulation et essais du prototype ;
- Détermination des besoins à partir des résultats des essais ;
- Validation des besoins par un comité de pilotage ;
- Planification du cycle suivant.



Le modèle en spirale

Le dernier cycle permet de développer la version finale et d'implémenter le logiciel.

Pour corriger les travers de la démarche linéaire (*en Cascade*) sont apparus des modèles dits en spirales, proposé par ***B. Boehm en 1988***, où les risques, quels qu'ils soient, sont constamment traités au travers de bouclages successifs :

- chaque spire confirme et affine les spires précédentes en menant des activités de même nature successivement ;
- L'analyse ou la conception ne sont plus effectuées dans une seule phase ou étape mais sont conduites en tant qu'activités qui se déroulent sur de multiples phases ;
- A chaque étape, après avoir défini les objectifs et les alternatives, celles-ci sont évaluées par différentes techniques (*prototypage, simulation, ...),* l'étape est réalisée et la suite est planifiée.

- Le nombre de cycles est variable selon que le développement est classique ou incrémental ;
- Ce modèle met l'accent sur l'analyse des risques tels que les exigences démesurées par rapport à la technologie, la réutilisation de composants, calendriers et budgets irréalistes, la défaillance du personnel, etc.
- Ce modèle est utilisé pour des projets dont les enjeux (*risques*) sont importants

Chaque cycle de la spirale se déroule en quatre phases :

1. Un cycle de la spirale commence par l'élaboration d'objectifs tels que la performance, la fonctionnalité, etc. on énumère ensuite les différentes manières de parvenir à ces objectifs, ainsi que les contraintes. On évalue ensuite chaque alternative en fonction de l'objectif.

2. L'étape suivante consiste à évaluer les risques pour chaque activité, comme l'analyse détaillée, le prototypage, la simulation, etc.

3. Après avoir évalué le risque, on choisit un modèle de développement pour le système. Par exemple, si les principaux risques concernent l'interface utilisateur, le prototypage évolutif pourrait s'avérer un modèle de développement approprié. Le modèle de la cascade peut être le plus approprié si le principal risque identifié concerne l'intégration des sous-systèmes. Il n'est pas nécessaire d'adopter un seul modèle à chaque cycle de la spirale ou même pour l'ensemble d'un système. Le modèle de la spirale englobe tous les autres modèles.

4. La situation est ensuite réévaluée pour déterminer si un développement supplémentaire est nécessaire, auquel cas il faudrait planifier la prochaine étape. (*on estime au cours d'une procédure de revue, si on doit passer au prochain cycle de la spirale ou non*).

Les principaux risques et leurs remèdes, tels que définis par Boehm, sont les suivants :

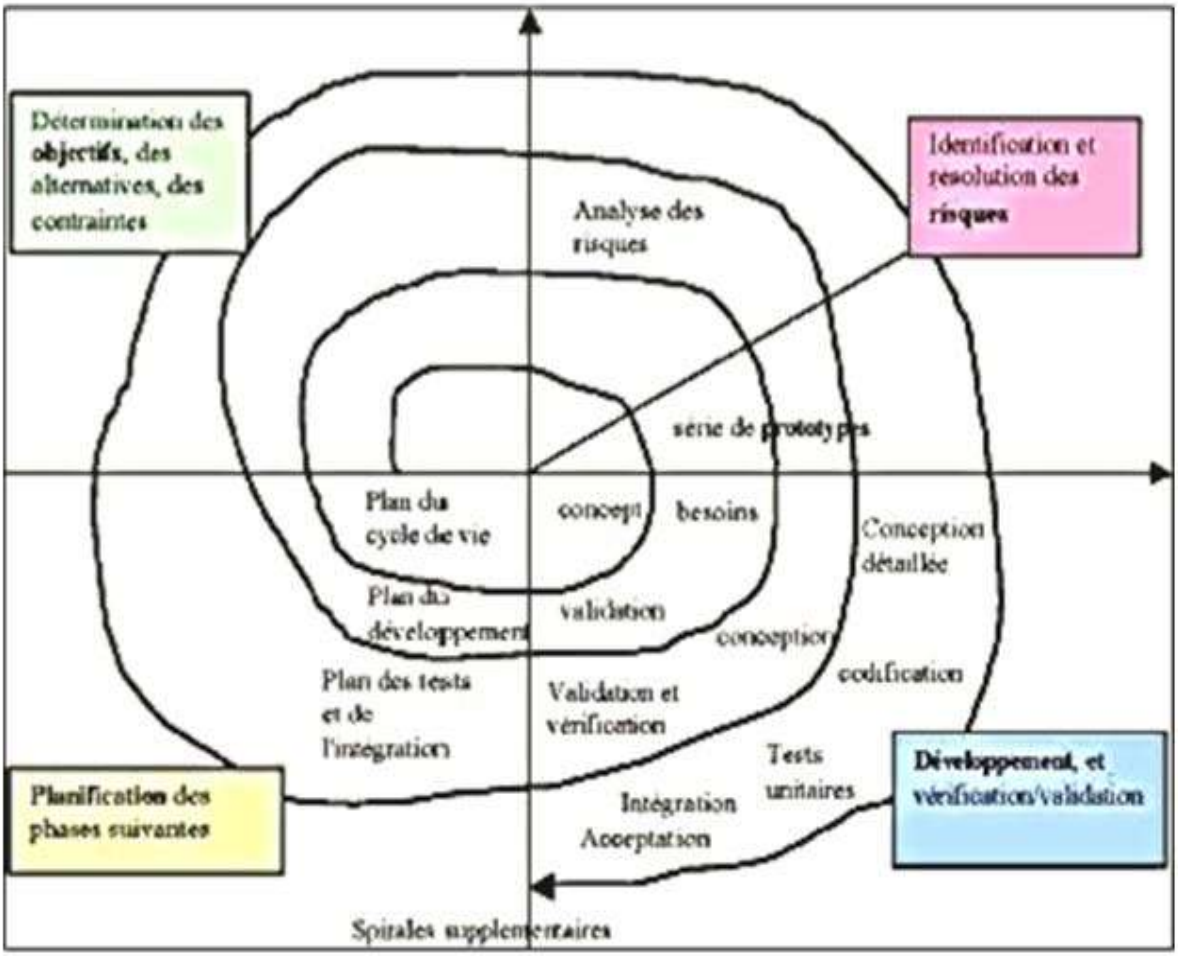
1. *La défaillance de personnel* : embauche de haut niveau, formation mutuelle, leaders, adéquation profil/fonction, ...
2. *Le calendrier et budgets irréalistes* : estimation détaillée, développement incrémental, réutilisation, élagage des besoins, ...
3. *Le développement de fonctions inappropriées* : revues d'utilisateurs, manuel d'utilisation précoce,...

4. *Le développement d'interfaces utilisateurs inappropriées* : maquettage, analyse des tâches,
5. *Le produit « plaqué or »* : analyse des coûts/bénéfices, conception tenant compte des coûts, ...
6. La volatilité des besoins : développement incrémental de la partie la plus stable d'abord, masquage d'information, ...
7. Les problèmes de performances : simulations, modélisations, essais et mesures, maquettage,
8. les exigences démesurées par rapport à la technologie : analyses techniques de faisabilité, maquettage,...
9. les tâches ou composants externes défaillants : audit des sous-traitants, contrats, revues, analyse de compatibilité, essais et mesures, ...

Il n'est pas nécessaire d'adopter un seul modèle à chaque cycle de la spirale ou même pour l'ensemble d'un système. Le modèle de la spirale englobe tous les autres modèles. Le prototypage peut être utilisé dans une spirale pour résoudre le problème de la spécification des besoins, puis il peut être suivi d'un développement basé sur le modèle conventionnel de la cascade. On peut utiliser la transformation formelle pour une partie du système à haute sécurité, et une approche basée sur la réutilisation pour l'interface utilisateur.

Le modèle du cycle de vie en spiral est un modèle itératif (*répété*), où la planification de la version se fait selon une analyse de risques. L'idée est de s'attaquer aux risques les plus importants assez tôt, afin que ceux-ci diminuent rapidement. De façon générale, les risques liés au développement de logiciels peuvent être répartis en quatre catégories :

- les risques commerciaux (*placement du produit sur le marché, concurrence*);
- les risques financiers (*capacités financières suffisantes pour réaliser le produit*);
- les risques techniques (*la technologie employée est-elle ´éprouvée ?)* ;
- les risques de développement (*l'équipe est-elle suffisamment expérimentée ?).*



Modèle du cycle de vie en spirale

Supposons qu'une équipe doive développer un logiciel comportant une interface graphique. Un exemple de risque pourrait être un manque de compétence de l'équipe de développement dans l'écriture d'interfaces. Dans ce cas, il faut tenir compte d'un temps de formation, et d'un retard possible dans le développement de l'interface. Dans processus de développement en spiral, on s'attaque au développement de l'interface assez tôt, afin de détecter le plus rapidement possible les problèmes ´éventuels, et d'avoir le temps d'y remédier.

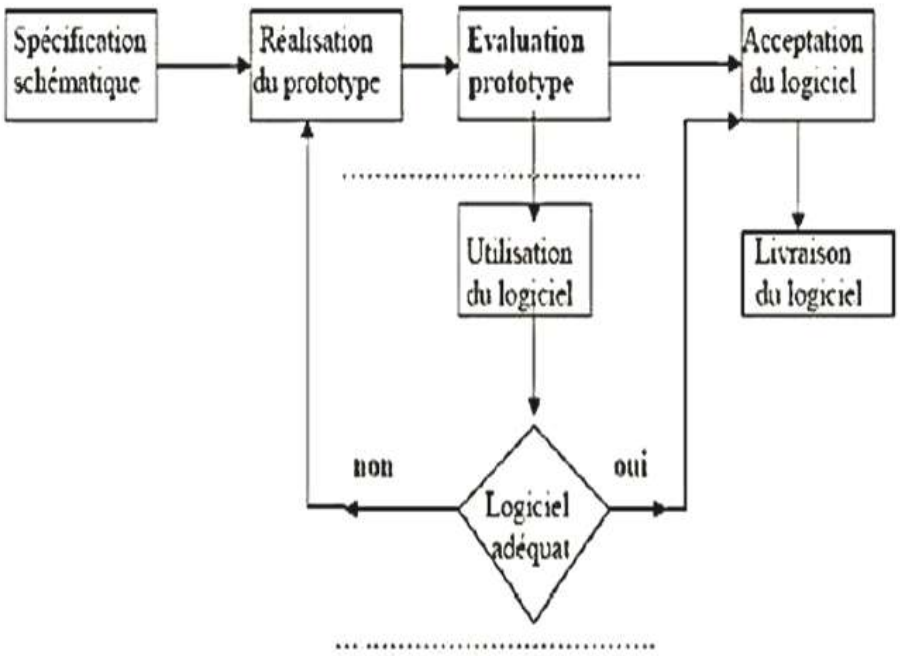
Dans un cycle de vie en cascade, les activités qui comportent le plus de risques ont lieu à la fin, lors de la mise en œuvre et de la validation. En s'attaquant d`es le début aux activités les plus risquées, le modèle en spirale permet d'accélérer la réduction du risque au cours du développement du logiciel : les risques sont importants lors des premières itérations, et diminuent lors des dernières itérations.

III.6. LE MODELE PAR PROTOTYPAGE

Il est quelquefois difficile de formuler une esquisse des besoins, surtout lorsque l'on connaît peule domaine. Dans ces cas-là, on ne peut pas espérer de manière réaliste définir les besoins de manière définitive avant le début du développement du logiciel. Un modèle de processus basé sur le prototypage se révèle alors plus approprié que le modèle classique de la cascade. Le prototypage permet de contourner la difficulté de la validation liée à l'imprécision des besoins et caractéristiques du système à développer. Cela veut dire que lorsqu'il est difficile d'établir une spécification détaillée, on a recours au prototypage qui est considéré, dans ce cas, comme un modèle de développement de logiciels.

Il s'agit d'écrire une première spécification et de réaliser un sous-ensemble du produit logiciel final. Ce sous ensemble est alors raffiné incrémentalement et évalué jusqu'à obtenir le produit final. On en distinguera deux types de prototypage :

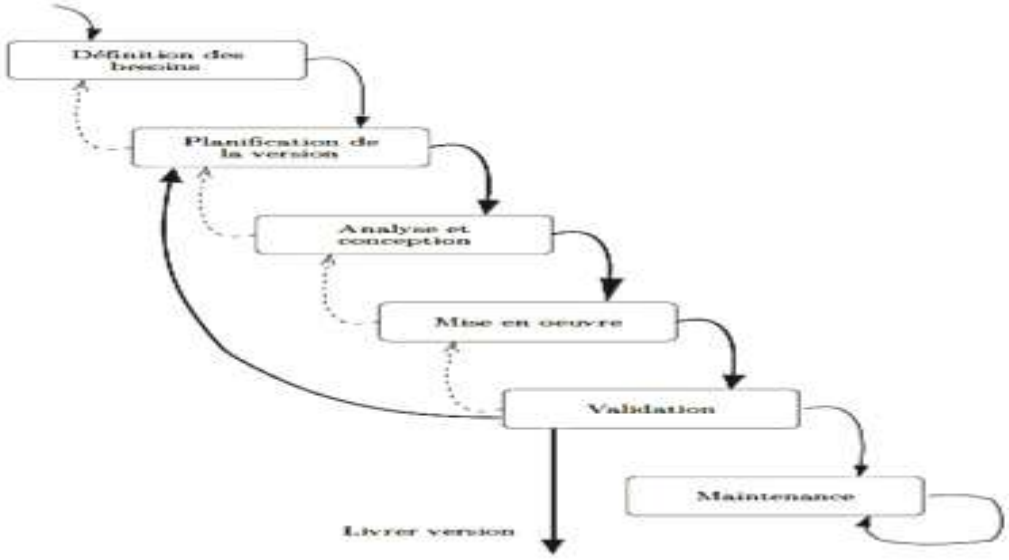
- ***Le prototypage Jetable*** : ici, le squelette du logiciel n'est créé que dans un but et dans une phase particulière du développement.
- ***Le prototypage Evolutif*** : ici, on conserve tout, au long du cycle de développement. Il est amélioré et complété pour obtenir le logiciel final



Modèle du cycle de vie par prototypage

III.7. LE MODELE PAR INCREMENT

Le modèle incrémental est un modèle itératif, qui consiste à sélectionner successivement plusieurs incréments. Un incrément du logiciel est un sous-ensemble du logiciel complet, qui consiste en un petit nombre de fonctionnalités.



Modèle du cycle de vie par incrément

Dans les modèles spirales, en V ou en cascade, un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus. Dans le modèle par incréments, seul un sous ensemble est développé à la fois. Dans un premier temps un logiciel noyau est développés, puis successivement, les incréments sont développés et intégrés. Chaque incrément est développé selon l'un des modèles précédents. Dans ce modèle, les intégrations sont progressives et il peut y avoir des livraisons et des mises en service après chaque intégration d'incrément. Le modèle incrémental présente comme avantages :

- chaque développement est moins complexe ;
- les intégrations sont progressives ;
- possibilité de livraisons et de mises en service après chaque incrément ;
- meilleur lissage du temps et de l'effort de développement à cause de la possibilité de recouvrement des différentes phases. l'effort est constant dans le temps par opposition au pic pour spécifications détaillées pour les modèles en cascade ou en V.

Le modèle incrémental présente comme Risques

- la remise en cause du noyau de départ ;
- la remise en cause des incréments précédents;
- ou encore l'impossibilité d'intégrer un nouvel incrément

QUATRIEME CHAPITRE – PROCESSUS SEQUENTIEL DE DEVELOPPEMENT DU LOGICIEL

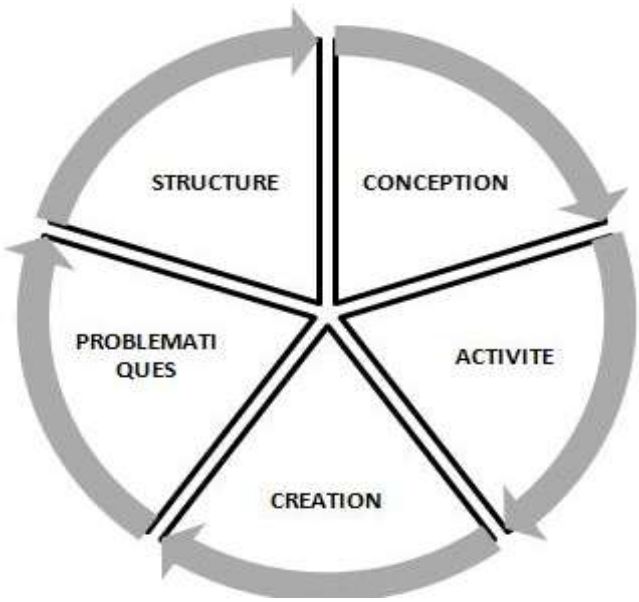
Un processus séquentiel de développement permet de décrire l'enchainement des différentes étapes du développement. L'objectif est de proposer un processus qui permet de contrôler le développement, afin que le logiciel soit :

- Soit livrer dans les délais ;
- Respecte le budget ;
- Soit de bonne qualité.

IV.1. ARCHITECTURE LOGICIELLE

Le Petit Robert définit l'architecture comme étant « *l'art de construire les édifices* ». Ce mot est avant tout lié au domaine du génie civil : on pense à l'architecture d'un monument ou encore d'un pont. Par analogie, l'architecture logicielle peut être définie comme étant « *l'art de construire les logiciels* ». Selon le contexte, l'architecture logicielle peut désigner :

- **L'activité d'architecture**, c'est-à-dire une phase au cours de laquelle on effectue les grands choix qui vont structurer une application : langages et technologies utilisés, découpage en sous-parties, méthodologies mises en œuvre...
- **Le résultat de cette activité**, c'est-à-dire la structure d'une l'application, son squelette. Dans le domaine du génie civil, on n'imagine pas se lancer dans la construction d'un bâtiment sans avoir prévu son apparence, étudié ses fondations et son équilibre, choisi les matériaux utilisés, etc. Dans le cas contraire, on va au-devant de graves désillusions...



Cette problématique se retrouve dans le domaine informatique. Comme un bâtiment, un logiciel est fait pour durer dans le temps. Il est presque systématique que des projets informatiques aient une durée de vie de plusieurs années. Plus encore qu'un bâtiment, un logiciel va, tout au long de son cycle de vie, connaître de nombreuses modifications qui aboutiront à la livraison de nouvelles versions, majeures ou mineures. Les évolutions par rapport au produit initialement créé sont souvent nombreuses et très difficiles à prévoir au début du projet. Exemple : « ***le logiciel VLC*** » n'était à l'origine qu'un projet étudiant destiné à diffuser des vidéos sur le campus de l'Ecole Centrale de Paris. Sa première version remonte à l'année 2001.

Dans le domaine du génie civil, les objectifs de l'architecture sont que le bâtiment construit réponde aux besoins qu'il remplit, soit robuste dans le temps et (*notion plus subjective*) agréable à l'œil. L'architecture logicielle poursuit les mêmes objectifs. Le logiciel créé doit répondre aux besoins et résister aux nombreuses modifications qu'il subira au cours de son cycle de vie. Contrairement à un bâtiment, un logiciel mal pensé ne risque pas de s'effondrer. En revanche, une mauvaise architecture peut faire exploser le temps nécessaire pour réaliser les modifications, et donc leur coût. Les deux objectifs principaux de toute architecture logicielle sont la réduction des coûts (*création et maintenance*) et l'augmentation de la **qualité** du logiciel. La qualité du code source d'un logiciel peut être évaluée par un certain nombre de mesures appelées **métriques de code** : indice de maintenabilité, complexité cyclomatique, etc.

IV.1.1. CONCEPTION / ARCHITECTURE

Il n'existe pas de vrai consensus concernant le sens des mots « ***architecture*** » et « ***conceptio***n » dans le domaine du développement logiciel. Ces deux termes sont souvent employés de manière interchangeable. Il arrive aussi que l'architecture soit appelée « *conception préliminaire* » et la conception proprement dite « *conception détaillée* ». Certaines méthodologies de développement incluent la définition de l'architecture dans une phase plus globale appelée « conception ». Cela dite, la distinction suivante est généralement admise et sera utilisée dans le cadre de ce cours :

- **L'architecture logicielle (*software architecture*)** considère le logiciel de manière globale. Il s'agit d'une vue de haut niveau qui définit le logiciel dans ses grandes lignes : que fait-il ? Quelles sont les sous-parties qui le composent ? Interagissent-elles ? Sous quelle forme sont stockées ses données ? etc.

- **La conception logicielle (*software design*)** intervient à un niveau de granularité plusfin et permet de préciser comment fonctionne chaque sous-partie de l'application. Quel logiciel est utilisé pour stocker les données ? Comment est organisé le code ? Comment une sous-partie expose-t-elle ses fonctionnalités au reste du système ? Etc.

La perspective change selon la taille du logiciel et le niveau auquel on s'intéresse à lui :

- Sur un projet de taille modeste, architecture et conception peuvent se confondre.
- A l'inverse, certaines sous-parties d'un projet de taille conséquente peuvent nécessiter en elles-mêmes un travail d'architecture qui, du point de vue de l'application globale, relève plutôt de la conception...

IV.1.2. L'ACTIVITE D'ARCHITECTURE

Tout logiciel, au-delà d'un niveau minimal de complexité, est un édifice qui mérite une phase de réflexion initiale pour l'imaginer dans ses grandes lignes. Au cours de cette phase, on effectue les grands choix structurant le futur logiciel : langages, technologies, outils... Elle consiste notamment à identifier les différents éléments qui vont composer le logiciel et à organiser les interactions entre ces éléments. Selon le niveau de complexité du logiciel, l'activité d'architecture peut être une simple formalité ou bien un travail de longue haleine. L'activité d'architecture peut donner lieu à la production *de diagrammes* représentant les éléments et leurs interactions selon différents formalismes, par exemple UML.

IV.1.3. PLACE DANS LE PROCESSUS DE CREATION

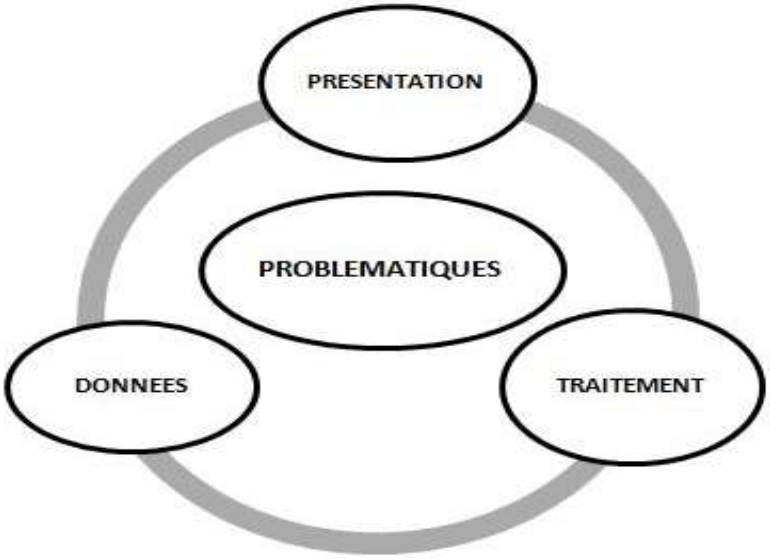
L'activité d'architecture intervient traditionnellement vers le début d'un projet logiciel, dès le moment où les besoins auxquels le logiciel doit répondre sont suffisamment identifiés. Elle est presque toujours suivie par une phase de conception. Les évolutions d'un projet logiciel peuvent nécessiter de nouvelles phases d'architecture tout au long de sa vie. C'est notamment le cas avec certaines méthodologies de développement itératif ou agile, où des phases d'architecture souvent brèves alternent avec des phases de production, de test et de livraison.

IV.1.4. REPARTITION DES PROBLEMATIQUES

De manière très générale, un logiciel sert à automatiser des traitements sur des données. Toute application informatique est donc confrontée à trois problématiques[15] :

- **la problématique de présentation** : consiste à gérer les interactions avec l'extérieur, en particulier l'utilisateur : saisie et contrôle de données, affichage.
- **la problématique des traitements :** consiste à effectuer sur les données des opérations (*calculs*) en rapport avec les règles métier (« *business logic* »).
- **la problématique des données :** consiste à accéder et stocker les informations qu'il manipule, notamment entre deux utilisations.

La phase d'architecture d'une application consiste aussi à choisir comment sont gérées ces trois problématiques, autrement dit à les répartir dans l'application créée.



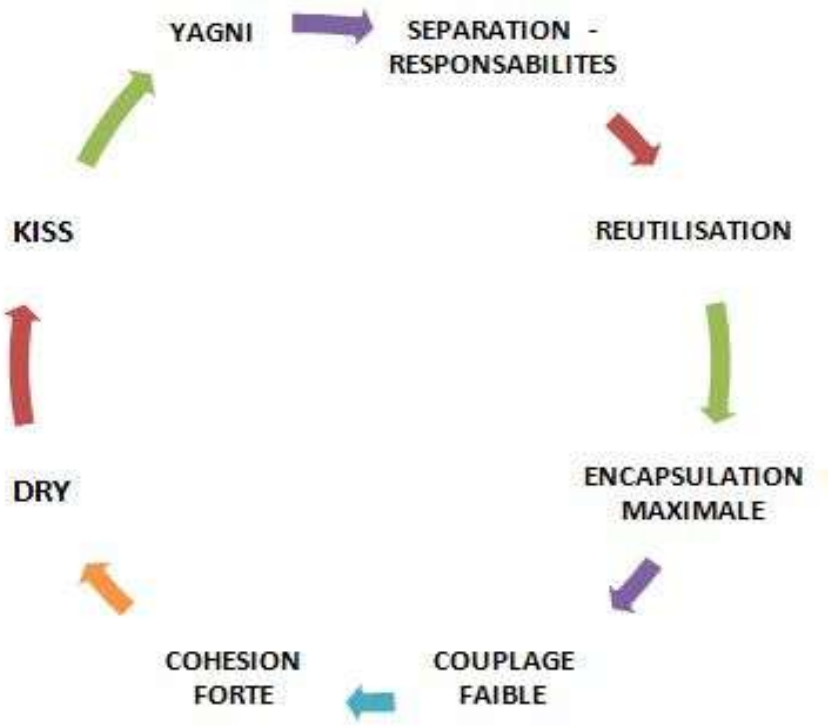
IV.1.5. LA STRUCTURE D'UN LOGICIEL

Le résultat de l'activité du même nom, l'architecture d'un logiciel décrit sa structure globale, son squelette. Elle décrit les principaux éléments qui composent le logiciel, ainsi que les flux d'échanges entre ces éléments. Elle permet à l'équipe de développement d'avoir une vue d'ensemble de l'organisation du logiciel, et constitue donc en elle-même une forme de documentation. On peut décrire la structure d'un logiciel selon différents points de vue. Entre autres, une vue **logique** mettre l'accent sur le rôle et les responsabilités de chaque partie du logiciel. Une vue **physique** présentera les processus, les machines et les liens réseau nécessaires.

[15] Dans certains cas de figure, l'une ou l'autre problématique seront très réduites (logiciel sans utilisateur, pas de stockage des données, etc.).

IV.2. PRINCIPES DE CONCEPTION

Cette partie présente les grands principes[16] qui doivent guider la création d'un logiciel, et plus généralement le travail du développeur au quotidien :



IV.2.1. SEPARATION DES RESPONSABILITES

Le **principe de séparation des responsabilités** (*separation of concerns*) vise à organiser un logiciel en plusieurs sous-parties, chacune ayant une responsabilité bien définie. Ce principe est sans doute le principe de conception le plus essentiel. Ainsi construite de manière modulaire, l'application sera plus facile à comprendre et à faire évoluer. Au moment où un nouveau besoin se fera sentir, il suffira d'intervenir sur la ou les sous-parties concernées. Le reste de l'application sera inchangée : cela limite les tests à effectuer et le risque d'erreur. Une construction modulaire encourage également la réutilisation de certaines parties de l'application.

Le **principe de responsabilité unique** (*single responsibility principle*) stipule quant à lui que chaque sous-partie atomique d'un logiciel (exemple : une classe) doit avoir une unique responsabilité (*une raison de changer*) ou bien être elle-même décomposée en sous-parties. Exemples d'applications de ces deux principes :

[16] Certains étant potentiellement contradictoires entre eux, il faudra nécessairement procéder à des compromis ou des arbitrages en fonction du contexte du projet.

- Une sous-partie qui s'occupe des affichages à l'écran ne devrait pas comporter de traitements métier, ni de code en rapport avec l'accès aux données.
- Un composant de traitements métier (*calcul scientifique ou financier, etc.*) ne doit pas s'intéresser ni à l'affichage des données qu'il manipule, ni à leur stockage.
- Une classe d'accès à une base de données (*connexion, exécution de requêtes*) ne devrait faire ni traitements métier, ni affichage des informations.
- Une classe qui aurait deux raisons de changer devrait être scindée en deux classes distinctes.

IV.2.2. REUTILISATION

Un bâtiment s'édifie à partir de morceaux de bases, par exemple des briques ou des moellons. De la même manière, une carte mère est conçue par assemblage de composants électroniques. Longtemps, l'informatique a gardé un côté artisanal : chaque programmeur recréait la roue dans son coin pour les besoins de son projet. Mais nous sommes passés depuis plusieurs années à une ère industrielle. Des logiciels de plus en plus complexes doivent être réalisés dans des délais de plus en plus courts, tout en maintenant le meilleur niveau de qualité possible. Une réponse à ces exigences contradictoires passe par la réutilisation de briques logicielles de base appelées librairies, modules ou plus généralement **composants**. En particulier, la mise à disposition de milliers de projets *open source* via des plates-formes comme « ***GitHub*** » ou des outils comme « ***NuGet, Composer*** ou ***npm*** » ont permis aux équipes de développement de faire des gains de productivité remarquables en intégrant ces composants lors de la conception de leurs applications.

A l'heure actuelle, il n'est pas de logiciel de taille significative qui n'intègre plusieurs dizaines, voire des centaines de composants externes. Déjà testé et éprouvé, un composant logiciel fait simultanément baisser le temps et augmenter la qualité du développement. Il permet de limiter les efforts nécessaires pour traiter les problématiques *techniques* afin de se concentrer sur les problématiques *métier*, celles qui sont en lien direct avec ses fonctionnalités essentielles. Voici parmi bien d'autres quelques exemples de problématiques techniques adressables par des composants logiciels :

- Accès à une base de données (*connexion, exécution de requêtes*).
- Calculs scientifiques.
- Gestion de l'affichage (*moteur 3D*).
- Journalisation des évènements dans des fichiers.

IV.2.3. ENCAPSULATION MAXIMALE

Ce principe de conception recommande de n'exposer au reste de l'application que le strict nécessaire pour que la sous-partie joue son rôle. Au niveau d'une classe, cela consiste à ne donner le niveau *d'accessibilité publique* qu'à un nombre minimal de membres, qui seront le plus souvent des méthodes. Au niveau d'une sous-partie d'application composée de plusieurs classes, cela consiste à rendre certaines *classes privées* afin d'interdire leur utilisation par le reste de l'application.

IV.2.4. COUPLAGE FAIBLE

La définition du couplage est la suivante : « une entité (*sous-partie, composant, classe, méthode*) est **couplée** à une autre si elle dépend d'elle », autrement dit, si elle a besoin d'elle pour fonctionner. Plus une classe ou une méthode utilise d'autres classes comme classes de base, attributs, paramètres ou variables locales, plus son couplage avec ces classes augmente. Au sein d'une application, un couplage fort tisse entre ses éléments des liens puissants qui la rend plus rigide à toute modification (*on parle de « code spaghetti »*). A l'inverse, un couplage faible permet une grande souplesse de mise à jour. Un élément peut être modifié (*exemple : changement de la signature d'une méthode publique*) en limitant ses impacts.

Le couplage peut également désigner une dépendance envers une technologie ou un élément extérieur spécifique : un SGBD, un composant logiciel, etc. Le principe de responsabilité unique permet de limiter le couplage au sein de l'application : chaque sous-partie a un rôle précis et n'a que des interactions limitées avec les autres sous parties. Pour aller plus loin, il faut limiter le nombre de paramètres des méthodes et utiliser des **classes abstraites** ou des **interfaces** plutôt que des implémentations spécifiques (« *Program to interfaces, not to implémentations* »).

IV.2.5. COHESION FORTE

Ce principe recommande de placer ensemble des éléments (*composants, classes, méthodes*) ayant des rôles similaires ou dédiés à une même problématique. Inversement, il déconseille de rassembler des éléments ayant des rôles différents. Exemple : ajouter une méthode de calcul métier au sein d'un composant lié aux données (*ou à la présentation*) est contraire au principe de cohésion forte.

IV.2.6. DRY (DON'T REPEAT YOURSELF)

Ce principe vise à éviter la redondance au travers de l'ensemble de l'application. Cette redondance est en effet l'un des principaux ennemis du développeur. Elle a les conséquences néfastes suivantes :

- Augmentation du volume de code ;
- Diminution de sa lisibilité ;
- Risque d'apparition de bogues dû à des modifications incomplètes.
- La redondance peut se présenter à plusieurs endroits d'une application, parfois de manière inévitable (*réutilisation d'un existant*). Elle prend souvent la forme d'un ensemble de lignes de code dupliquées à plusieurs endroits pour répondre au même besoin, comme dans l'exemple suivant.

```
function A() {
// ...
// Code dupliqué
// ...
}
functionB() {
// ...
// Code dupliqué
// ...
}
```

La solution classique consiste à factoriser les lignes auparavant dupliquées.

```
function C() {
// Code auparavant dupliqué
}
function A() {
// ...
// Appel à C()
// ...
}
function B() {
// ...
// Appel à C()
// ...
      }
```

Le principe DRY est important mais ne doit pas être appliqué de manière trop scrupuleuse. Vouloir absolument éliminer toute forme de redondance conduit parfois à créer des applications inutilement génériques et complexes. C'est l'objet des deux prochains principes.

IV.2.7. KISS (*KEEP IT SIMPLE, STUPID*)

KISS est un autre acronyme signifiant ***Keep It Simple, Stupid*** et qu'on peut traduire par « *Ne complique pas les choses* ». Ce principe vise à privilégier autant que possible la simplicité lors de la construction d'une application. Il part du constat que la complexité entraîne des surcoûts de développement puis de maintenance, pour des gains parfois discutables. La complexité peut prendre la forme d'une architecture surdimensionnée par rapports aux besoins (*over engineering*), ou de l'ajout de fonctionnalités secondaires ou non prioritaires. Une autre manière d'exprimer ce principe consiste à affirmer qu'une application doit être créée selon l'ordre de priorité ci-dessous :

- *Make it work*.
- *Make it right*.
- *Make it fast*.

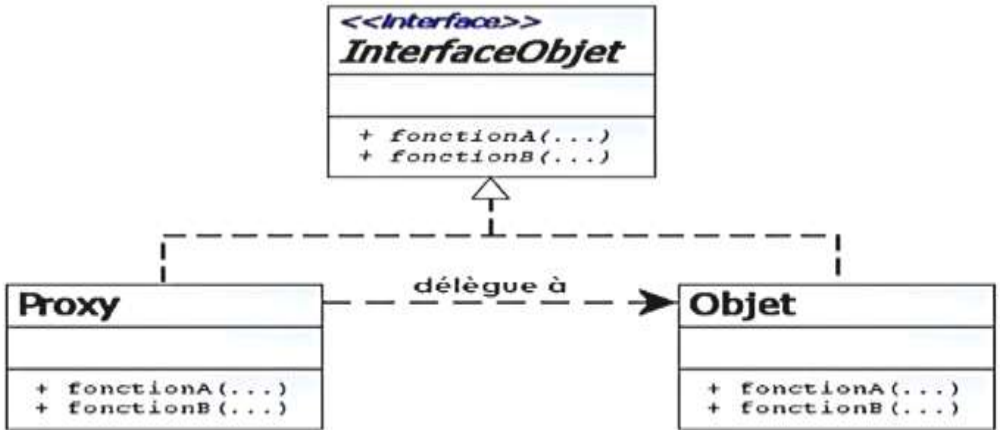
IV.2.8. YAGNI (*YOU AIN'TGONNA NEED IT*)

Ce troisième acronyme signifie « ***You Ain't Gonna Need It*** ». Corollaire du précédent, il consiste à ne pas se baser sur d'hypothétiques évolutions futures pour faire les choix du présent, au risque d'une complexification inutile (principe KISS). Il faut réaliser l'application au plus simple et en fonction des besoins actuels. Le moment venu, il sera toujours temps de procéder à des changements (refactorisation ou *refactoring*) pour que l'application réponde aux nouvelles exigences.

IV.3. PATRONS LOGICIELS

Un **patron de conception** (*design pattern*) aussi appelé « ***Motif de Conception*** » est une solution standard à un problème de conception. L'ensemble des patrons de conception constitue un catalogue de bonnes pratiques issu de l'expérience de la communauté. Leurs noms forment un vocabulaire commun qui permet d'identifier immédiatement la solution associée.

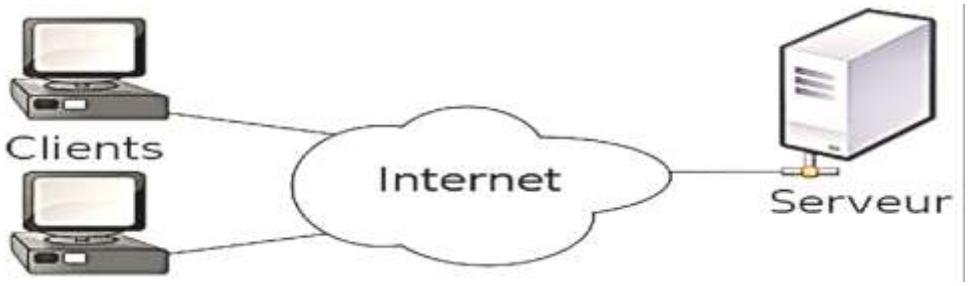
Les patrons de conception ont été popularisés par le livre *Design Patterns – Elements ofReusable Object-Oriented Software* sorti en 1995 et coécrit par quatre auteurs (le *Gang OfFour*, ou *GoF*). Ce livre décrit 23 patrons, auxquels d'autres se sont rajoutés. Chaque patron décrit un problème à résoudre puis les éléments de sa solution, ainsi que leurs relations. Le formalisme graphique utilisé est souvent un diagramme de classes UML.L'exemple ci-dessous décrit le patron de conception **Proxy**, dont l'objectif est de substituer un objet à un autre afin de contrôler l'utilisation de ce dernier.



Cette partie du cours présente les patrons logiciels (*software patterns*), qui sont des modèles de solutions à des problématiques fréquentes d'architecture ou de conception. Il n'existe pas une architecture logicielle parfaite qui s'adapterait à toutes les exigences. Au fil du temps et des projets, plusieurs architectures-types se sont dégagées. Elles constituent des patrons d'architecture (*architecture patterns*) qui ont fait leurs preuves et peuvent servir d'inspiration pour un nouveau projet. A l'occurrence :

IV.3.1. ARCHITECTURE CLIENT / SERVEUR

L'architecture client/serveur caractérise un système basé sur des échanges réseau entre des clients et un serveur centralisé, lieu de stockage des données de l'application.



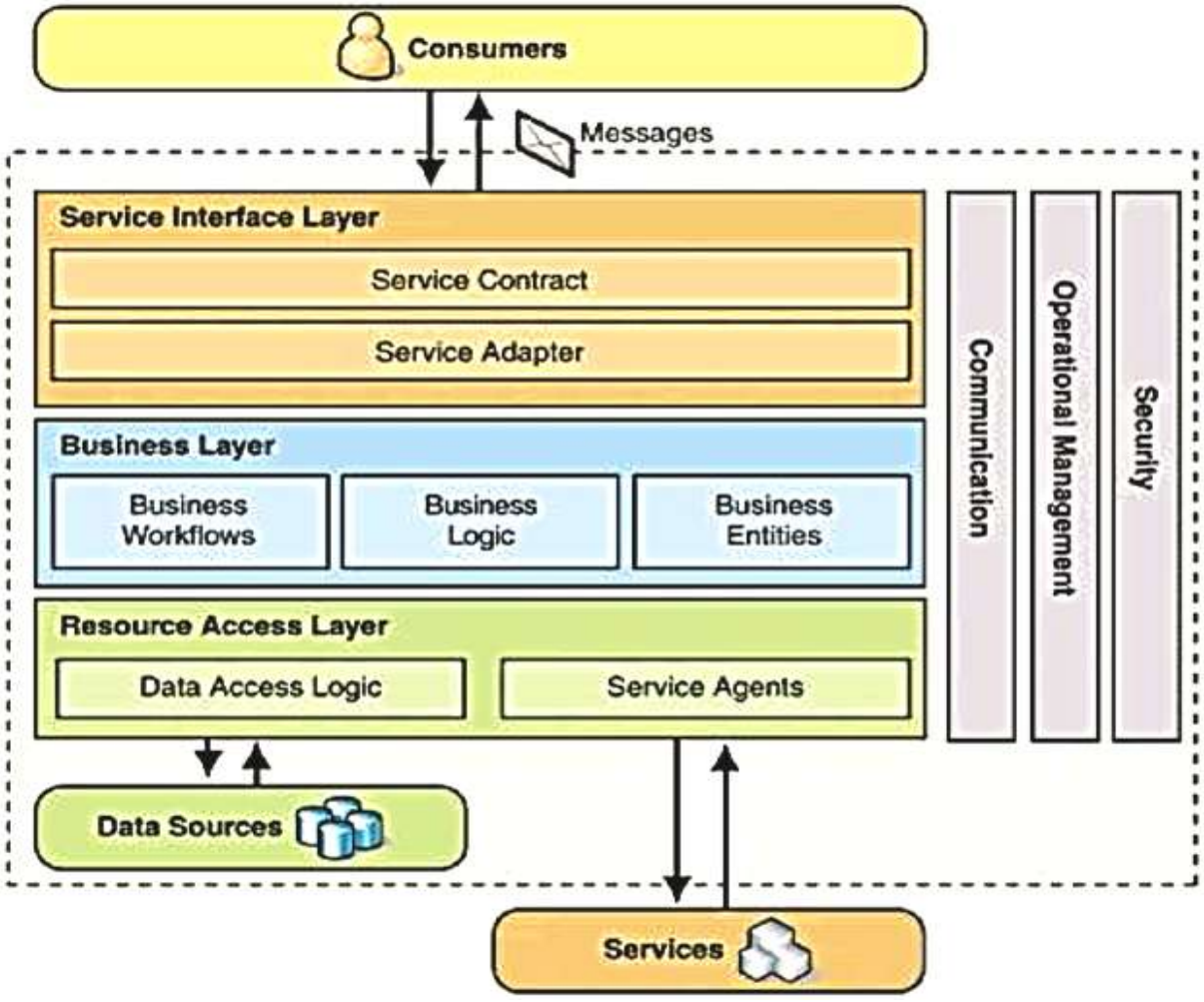
Le principal avantage de l'architecture client/serveur tient à la centralisation des données. Stockées à un seul endroit, elles sont plus faciles à sauvegarder et à sécuriser. Le serveur qui les héberge peut être dimensionné pour pouvoir héberger le volume de données nécessaire et répondre aux sollicitations de nombreux clients. Cette médaille a son revers : le serveur constitue le nœud central du système et représente son maillon faible. En cas de défaillance (*surcharge, indisponibilité, problème réseau*), les clients ne peuvent plus fonctionner. On peut classer les clients d'une architecture client/serveur en plusieurs types :

- **Client léger** : destiné uniquement à l'affichage (*exemple : navigateur web*) ;
- **Client lourd** : application native spécialement conçue pour communiquer avec le serveur (*exemple : application mobile*) ;
- **Client riche :** combinant les avantages des clients légers et lourds (*exemple : navigateur web utilisant des technologies évoluées pour offrir une expérience utilisateur proche de celle d'une application native*) ;

Le fonctionnement en mode client/serveur est très souvent utilisé en informatique. Un réseau Windows organisé en domaine, la consultation d'une page hébergée par un serveur Web ou le téléchargement d'une application mobile depuis un magasin central (*App Store, Google Play*) en constituent des exemples.

IV.3.2. ARCHITECTURE EN COUCHES

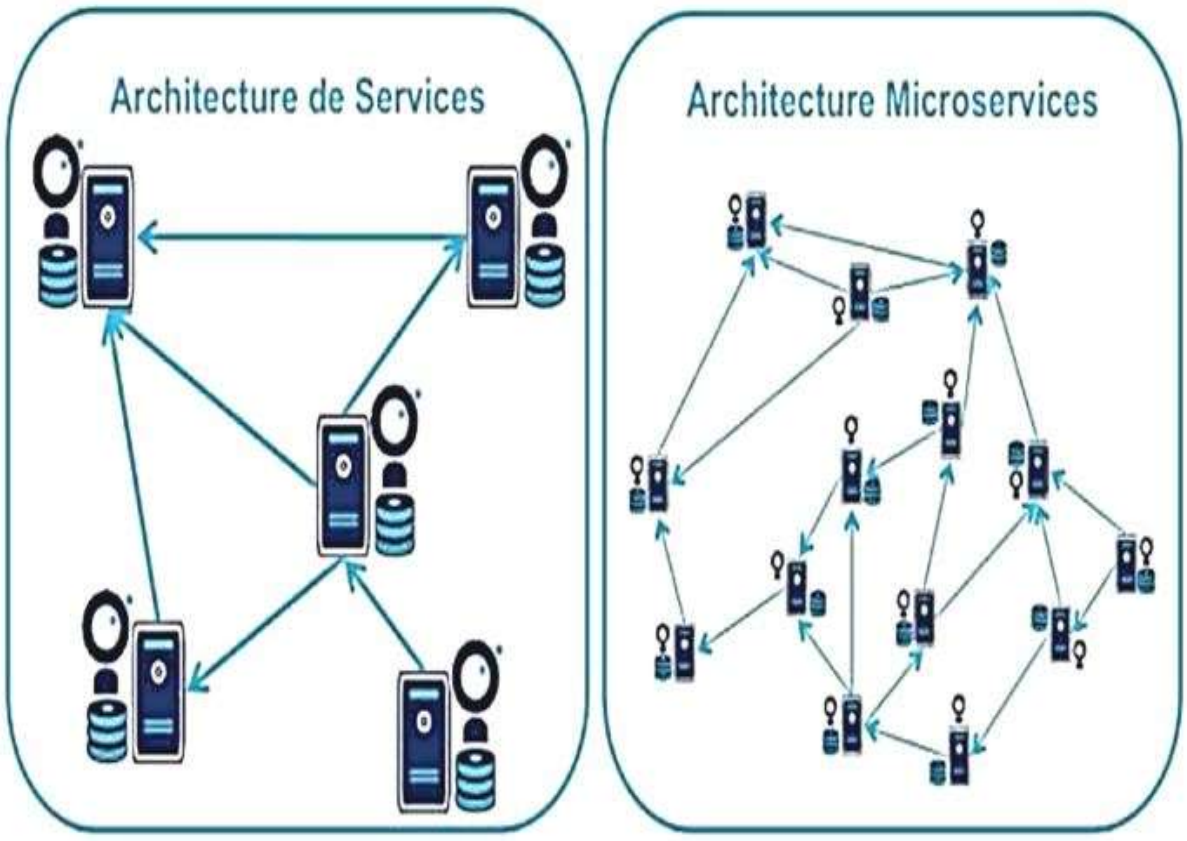
Une architecture en couches organise un logiciel sous forme de couches (*layers*). Chaque couche ne peut communiquer qu'avec les couches adjacentes.



Cette architecture respecte le principe de séparation des responsabilités et facilite la compréhension des échanges au sein de l'application. Lorsque chaque couche correspond à un processus distinct sur une machine, on parle d'architecture « ***n-tiers*** », ***n*** désignant le nombre de couches. Un navigateur web accédant à des pages dynamiques intégrant des informations stockées dans une base de données constitue un exemple classique d'architecture 3-tiers.

IV.3.3. ARCHITECTURE ORIENTEE SERVICES

Une architecture orientée services (SOA : *Service-Oriented Architecture*) décompose un logiciel sous la forme d'un ensemble de services métier utilisant un format d'échange commun, généralement XML ou JSON.

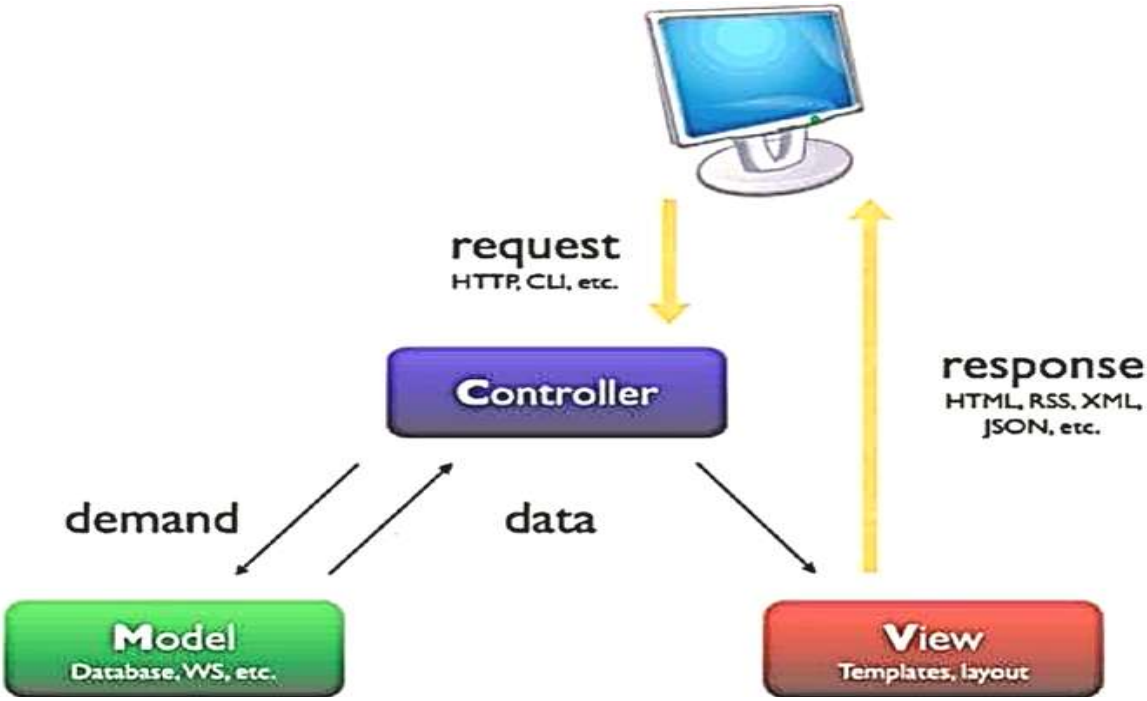


Une variante récente, *l'architecture micro-services*, diminue la granularité des services pour leur assurer souplesse et capacité à évoluer, au prix d'une plus grande distribution du système. L'image ci-dessous (*source*) illustre la déférence entre ces deux approches.

IV.3.4. ARCHITECTURE MODELE-VUE-CONTROLEUR

Le patron Modèle-Vue-Contrôleur, ou **MVC**, décompose une application en trois sous parties :

- La partie **Modèle** qui regroupe la logique métier (« *business logic* ») ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (*Modèle procédural*) ou de classes (*Modèle orienté objet*) ;
- La partie **Vue** qui s'occupe des interactions avec l'utilisateur : présentation, saisie et validation des données ;
- La partie **Contrôleur** qui gère la dynamique de l'application. Elle fait le lien entre les deux autres parties.

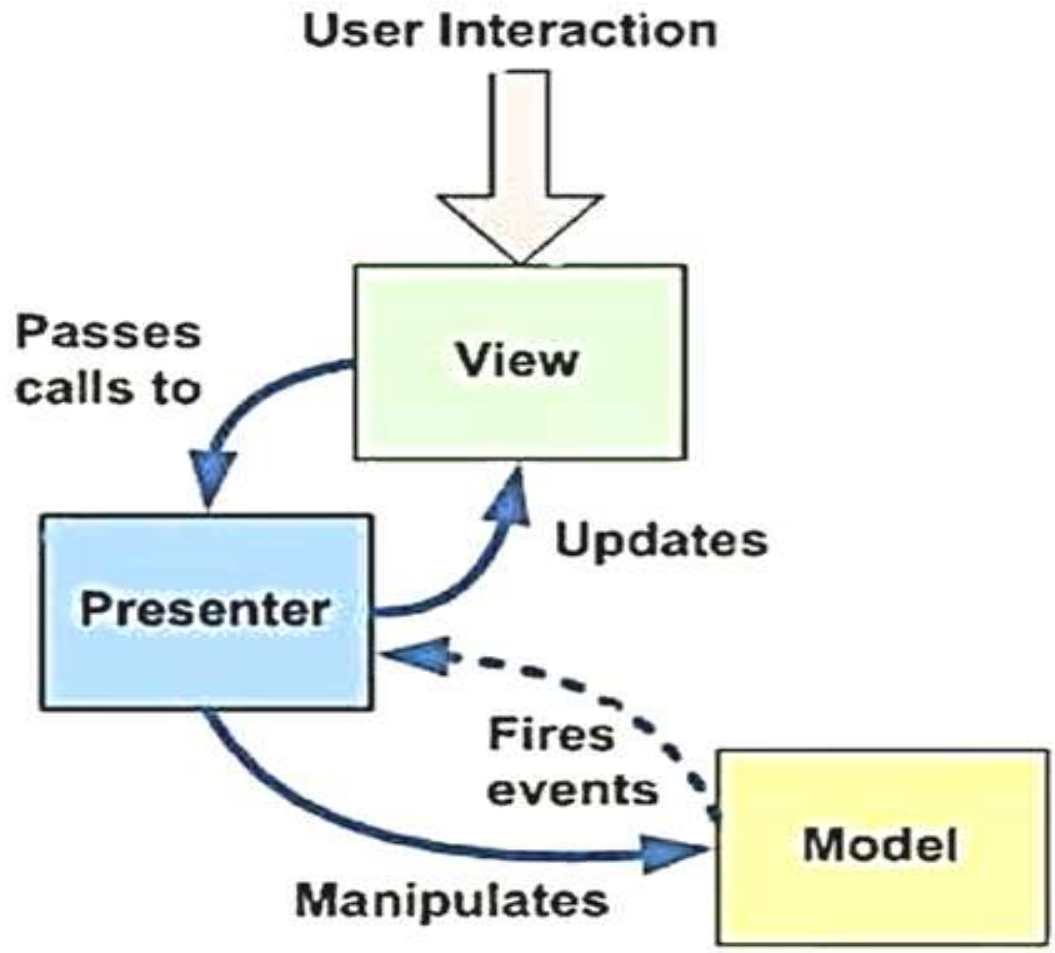


Ce patron a été imaginé à la fin des années 1970 pour le langage Small-talk afin de bien séparer le code de l'interface graphique de la logique applicative. On le retrouve dans de très nombreux langages : bibliothèques Swing et Model 2 (*JSP*) de *Java, frameworks PHP, ASP.NET MVC …* Le diagramme ci-dessous (*extrait de la documentation du framework PHP Symfony)* résume les relations entre les composants d'une architecture web MVC[17].

[17] Attention à ne pas employer le terme de « ***couche*** » à propos d'une architecture MVC. Dans une architecture en couches, chaque couche ne peut communiquer qu'avec les couches adjacentes. Les parties Modèle, Vue et Contrôleur ne sont donc pas des couches au vrai sens du mot.

IV.3.5. ARCHITECTURE MODELE-VUE-PRESENTATION

Le patron Modèle-Vue-Présentation, ou **MVP**, est un proche cousin du patron MVC surtout utilisé pour construire des interfaces utilisateurs (*UI*). Dans une architecture MVP, la partie **Vue** reçoit les évènements provenant de l'utilisateur et délègue leur gestion à la partie **Présentation**. Celle-ci utilise les services de la partie **Modèle** puis met à jour la **Vue**.

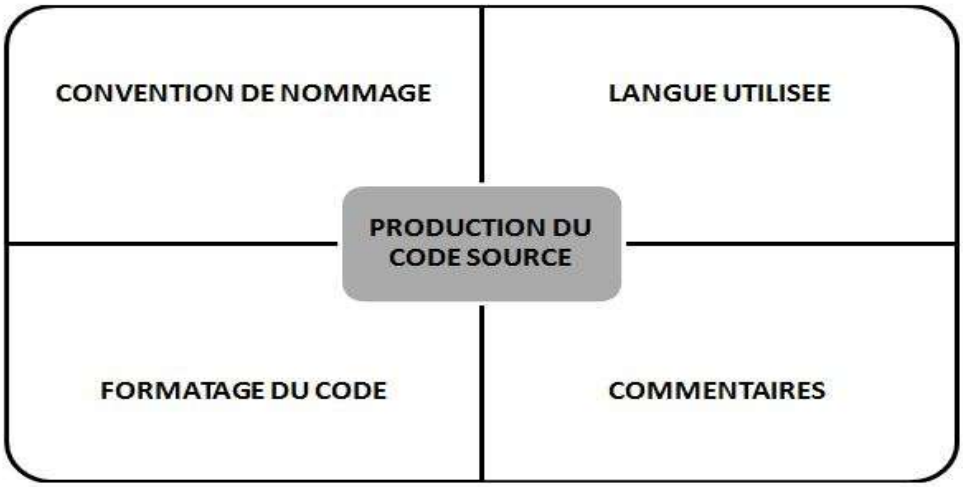


Dans la variante dite « *Passive View* » de cette architecture, la Vue est passive et dépend totalement du contrôleur pour ses mises à jour. Dans la variante dite « *Supervising Controller* », Vue et Modèle sont couplées et les modifications du Modèle déclenchent la mise à jour de la Vue.

IV.4. PRODUCTION DU CODE SOURCE

L'objectif de cette partie du cours est de présenter les enjeux et les solutions liés à la production du code source d'un logiciel. Le code source[18] est le cœur d'un projet logiciel. Il est essentiel que tous les membres de l'équipe de développement se coordonnent pour adopter des règles communes dans la production de ce code. L'objectif de ces règles est l'uniformisation de la base de code source du projet. Les avantages liés sont les suivants :

- La consultation du code est facilitée.
- Les risques de duplication ou d'erreurs liées à des pratiques disparates sont éliminés.
- Chaque membre de l'équipe peut comprendre et intervenir sur d'autres parties que celles qu'il a lui-même réalisées.
- Les nouveaux venus sur le projet mettront moins longtemps à être opérationnels.



[18] Il est important de garder à l'esprit qu'un développeur passe en moyenne beaucoup plus de temps à lire qu'à écrire du code.

IV.4.1. CONVENTION DE NOMMAGE

Une première série de règle concerne le nommage des différents éléments qui composent le code. Il n'existe pas de standard universel à ce sujet. La convention la plus fréquemment adoptée se nomme « ***camelCase*** » (ou parfois *lowerCamelCase*). Elle repose sur deux grands principes :

- Les noms des classes (*et des méthodes en C#, pour être en harmonie avec le framework .NET*) commencent par une lettre majuscule.
- Les noms de tous les autres éléments (*variables, attributs, paramètres, etc.)*commencent par une lettre minuscule.

Si le nom d'un élément se compose de plusieurs mots, la première lettre de chaque mot suivant le premier s'écrit en majuscule. Voici un exemple de classe conforme à cette convention.

```
classUneNouvelleClasse {
private in tun Attribut;
private float un Autre Attribut;
public void Une Methode(int monParam1, int monParam2) { ... }
public void Une Autre Methode(string encoreUnParametre) { ... }
}
```

On peut ajouter à cette convention une règle qui impose d'utiliser le pluriel pour nommer les éléments contenant plusieurs valeurs, comme les tableaux et les listes. Cela rend le parcours de ces éléments plus lisible[19].

```
List<string> clients = new List<string> ();
// ...
foreach(string client in clients) {
// 'clients' désigne la liste, 'client' le client courant
// ...
}
```

IV.4.2. LANGUE UTILISEE

La langue utilisée dans la production du code doit bien entendu être unique sur tout le projet. Le français (*idClientSuivant*) et l'anglais (*nextClientId*) ont chacun leurs avantages et leurs inconvénients. On choisira de préférence l'anglais pour les projets de taille importante ou destinés à être publiés en ligne.

[19] *On peut aussi utiliser des noms de la forme « **listeClients** »*

IV.4.3. FORMATAGE DU CODE

La grande majorité des IDE (*Intelligence Drive Electronic*) et des éditeurs de code offrent des fonctionnalités de formatage automatique du code. A condition d'utiliser un paramétrage commun, cela permet à chaque membre de l'équipe de formater rapidement et uniformément le code sur lequel il travaille. Les paramètres de formatage les plus courants sont :

- Taille des tabulations (*2 ou 4 espaces*) ;
- Remplacement automatique des tabulations par des espaces ;
- Passage ou non à la ligne après chaque accolade ouvrante ou fermante ;
- Ajout ou non d'un espace avant une liste de paramètres.

IV.4.4. COMMENTAIRES

L'ajout de commentaires permet de faciliter la lecture et la compréhension d'une portion de code source. L'ensemble des commentaires constitue une forme efficace de documentation d'un projet logiciel. Il n'y a pas de règle absolue, ni de consensus, en matière de taux de commentaires dans le code source. Certaines méthodologies de développement agile (*eXtremeProgramming*) vont jusqu'à affirmer qu'un code bien écrit se suffit à lui-même et ne nécessite aucun ajout de commentaires.

Dans un premier temps, il vaut mieux se montrer raisonnable et commenter les portions de code complexes ou essentielles : en-têtes de classes, algorithmes importants, portions atypiques, etc. Il faut éviter de paraphraser le code source en le commentant, ce qui alourdit sa lecture et n'est d'aucun intérêt. Voici quelques exemples de commentaires inutiles :

```
// Initialisation de i à 0
int i = 0;
// Instanciation d'un objet de la classe Random
Randomrng = new Random();
// Appel de la méthode Next sur l'objet rng
intnombreAlea = rng.Next(1, 4);
```

Voici comment le code précédent pourrait être mieux commenté.

```
inti = 0;
Random rng = new Random();
// Génération aléatoire d'un entier entre 1 et 3
intnombreAlea = rng.Next(1, 4);
```

IV.5. GESTION DES VERSIONS

L'objectif de cette partie est de découvrir ce qu'est la gestion des versions d'un logiciel, aussi parfois appelé « *la gestion des codes sources ou SCM*[20] ». Nous avons déjà mentionné qu'un projet logiciel d'entreprise a une durée de vie de plusieurs années et subit de nombreuses évolutions au cours de cette période. On rencontre souvent le besoin de livrer de nouvelles versions qui corrigent des bogues ou apportent de nouvelles fonctionnalités. Le code source du logiciel « ***vit*** » donc plusieurs années. Afin de pouvoir corriger des problèmes signalés par un utilisateur du logiciel, on doit savoir précisément quels fichiers source font partie de quelles versions.

En entreprise, seule une petite minorité de logiciels sont conçus par un seul développeur. La grande majorité des projets sont réalisés et/ou maintenus par une équipe de plusieurs personnes travaillant sur la même base de code source. Ce travail en parallèle est source de complexité :

- Comment récupérer le travail d'un autre membre de l'équipe ?
- Comment publier ses propres modifications ?
- Comment faire en cas de modifications conflictuelles (travail sur le même fichier source qu'un ou plusieurs collègues) ?
- Comment accéder à une version précédente d'un fichier ou du logiciel entier?

Pour les raisons précédentes, tout projet logiciel d'entreprise (même mono-développeur) doit faire l'objet d'une **gestion des versions** (*Revision Control System* ou *versioning*). La gestion des versions vise les objectifs suivants :

- Assurer la pérennité du code source d'un logiciel ;
- Permettre le travail collaboratif ;
- Fournir une gestion de l'historique du logiciel.

La gestion des versions la plus basique consiste à déposer le code source sur un répertoire partagé par l'équipe de développement. Si elle permet à tous de récupérer le code, elle n'offre aucune solution aux autres complexités du développement en équipe et n'autorise pas la gestion des versions. Afin de libérer l'équipe de développement des complexités du travail collaboratif, il existe une catégorie de logiciels spécialisés dans la gestion des versions.

[20] ***SCM*** *: Source Code Management*

IV.5.1. LES LOGICIELS DE GESTION DES VERSIONS

Selon les Principales fonctionnalités**,** Un logiciel de gestion des versions est avant tout un **dépôt de code** qui héberge le code source du projet. Chaque développeur peut accéder au dépôt afin de récupérer le code source, puis de publier ses modifications. Les autres développeurs peuvent alors récupérer le travail publié.

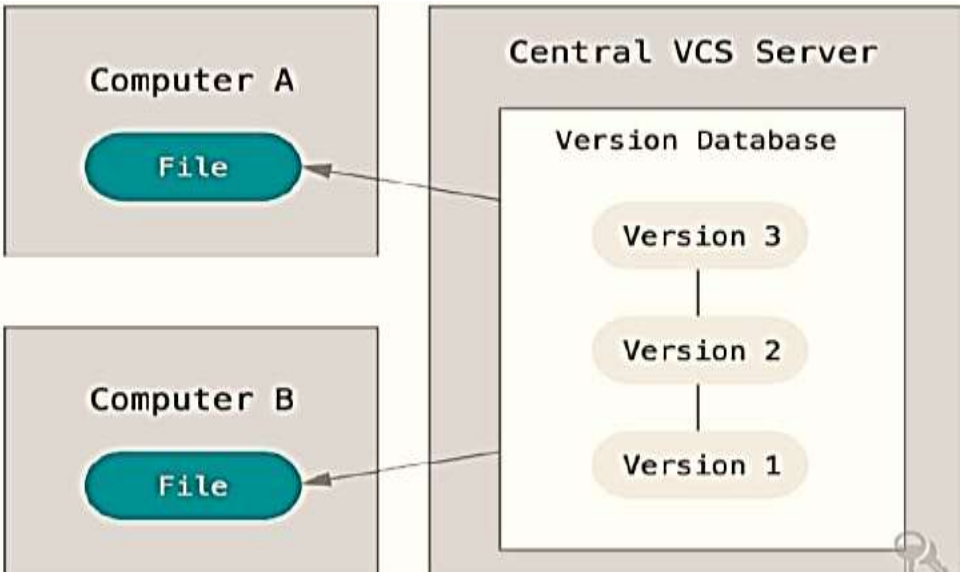
Le logiciel garde la trace des modifications successives d'un fichier. Il permet d'en visualiser l'**historique** et de revenir à une version antérieure. Un logiciel de gestion des versions permet de travailler en parallèle sur plusieurs problématiques (*par exemple, la correction des bogues de la version publiée et l'avancement sur la future version*) en créant des **branches**. Les modifications réalisées sur une branche peuvent ensuite être intégrées (*merging*) à une autre.

En cas d'apparition d'un **conflit** (*modifications simultanées du même fichier par plusieurs développeurs),* le logiciel de gestion des versions permet de comparer les versions du fichier et de choisir les modifications à conserver ou à rejeter pour créer le fichier fusionné final. Le logiciel de gestion des versions permet de regrouper logiquement des fichiers par le biais du *tagging*: il ajoute aux fichiers source des tags correspondant aux différentes versions du logiciel.

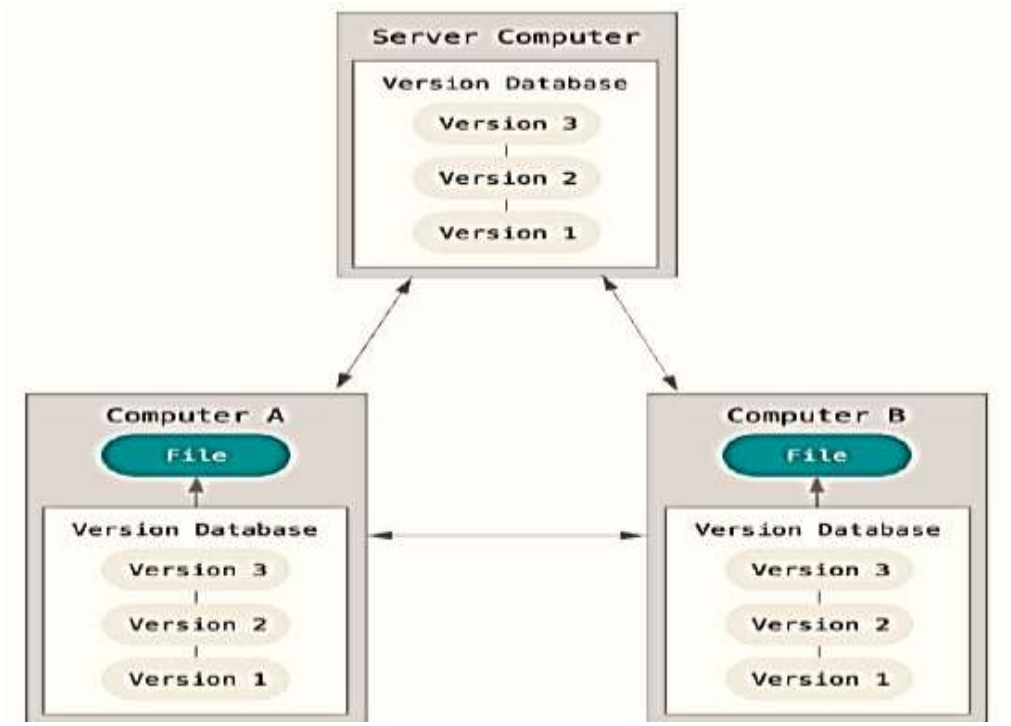
IV.5.2. GESTION CENTRALISEE ET DECENTRALISEE

On peut classer les logiciels de gestion des versions en deux catégories :

- **La gestion centralisée du code source** : Dans le cas où, il n'existe qu'un seul dépôt qui fait référence. Les développeurs se connectent au logiciel de gestion des versions suivant le principe du **client/serveur**. Cette solution offre les avantages de la centralisation (*administration facilitée*) mais handicape le travail en mode déconnecté : une connexion au logiciel de SCM est indispensable.



- **La gestion décentralisée du code source** : c'est une catégorie qui est apparu, il y a peu. Elle consiste à voir le logiciel de gestion des versions comme un outil individuel permettant de travailler de manière décentralisé (*hors ligne*). Dans ce cas, il existe autant de dépôts de code que de développeurs sur le projet. Le logiciel de gestion des versions fournit heureusement un service de synchronisation entre toutes ces bases de code. Cette solution fonctionne suivant le principe du « ***pair-à-pair*** ». Cependant, il peut exister un dépôt de référence contenant les versions livrées.



IV.5.3. PRINCIPAUX LOGICIELS DE GESTION DES VERSIONS

Selon Wikipédia, Il existe de très nombreux logiciels de gestion des versions. Cependant, dans le cadre de cours, nous ne citerons que les principaux :

- **CVS** (*Concurrent Versioning System*) : Assez ancien mais toujours utilisé, il fonctionne sur un principe centralisé, de même que son successeur **SVN** (*Subversion*). Tous deux sont souvent employés dans le monde du logiciel libre (*ce sont eux-mêmes des logiciels libres*).
- Les logiciels de SCM décentralisés sont apparus plus récemment. On peut citer « ***Mercurial*** *et* ***Git*** **»**, Ce sont également des logiciels libres.
- Microsoft fournit un logiciel de SCM développé sur mesure pour son environnement. Il se nomme **TFS** (*Team Foundation Server*) et fonctionne de manière centralisée. TFS est une solution payante.

IV.6. TRAVAIL COLLABORATIF DANS LE PROJET LOGICIEL

L'objectif de ce chapitre est de présenter les enjeux du travail collaboratif dans le cadre de la réalisation d'un logiciel. La très grande majorité des projets logiciels sont menés par des équipes de plusieurs développeurs. Il est de plus en plus fréquent que ces développeurs travaillent à distance ou en mobilité. Le travail en équipe sur un projet logiciel nécessite de pouvoir :

- Partager le code source entre membres de l'équipe ;
- Gérer les droits d'accès au code ;
- Intégrer les modifications réalisées par chaque développeur.
- Signaler des problèmes ou proposer des améliorations qui peuvent ensuite être discutés collectivement.

Pour répondre à ces besoins, des plates-formes de publication et de partage de code enligne sont apparues. On les appelle parfois des **forges logicielles**. La plus importante à l'heure actuelle est la « ***plate-forme GitHub***[21] ».

[21] GitHub est une plate-forme web d'hébergement et de partage de code. Comme son nom l'indique, elle se base sur le logiciel Git. Le principal service proposé par GitHub est la fourniture de dépôts Git accessibles en ligne. Elle offre aussi une gestion des équipes de travail (*organisations* et *teams*), des espaces d'échange autour du code (*issues* et *pull requests*), des statistiques, … GitHub est utilisée par pratiquement toutes les grandes sociétés du monde du logiciel, y compris Microsoft, Facebook et Apple. Pour un développeur, GitHub peut constituer une vitrine en ligne de son travail et un atout efficace pour son employabilité.

IV.7. TESTS DU LOGICIEL

L'objectif de cette partie est de présenter le rôle des tests dans le processus de création d'un logiciel. La problématique des tests est souvent considérée comme secondaire et négligée par les développeurs. C'est une erreur : lorsqu'on livre une application et qu'elle est placée en production (*offerte à ses utilisateurs*), il est essentiel d'avoir un maximum de garanties sur son bon fonctionnement afin d'éviter au maximum de coûteuses mauvaises surprises.

Le test d'une application peut être manuel. Dans ce cas, une personne effectue sur l'application une suite d'opérations prévue à l'avance (*navigation, connexion, envoi d'informations...)* pour vérifier qu'elle possède bien le comportement attendu. C'est un processus coûteux en temps et sujets aux erreurs (*oublis, négligences, etc*.). En complément de ces tests manuels, on a tout intérêt à intégrer à un projet logiciel des tests automatisés qui pourront être lancés aussi souvent que nécessaire. Ceci est d'autant plus vrai pour les méthodologies agiles basées sur un développement itératif et des livraisons fréquentes, ou bien lorsque l'on met en place une intégration continue.

On peut classer les tests logiciels en différentes catégories :

1. **Tests de validation :** Ces tests sont réalisés lors de la *recette* (*validation*) par un client d'un projet livré par l'un de ses fournisseurs. Souvent écrits par le client lui-même, ils portent sur l'ensemble du logiciel et permet de vérifier son comportement global en situation. De par leur spectre large et leur complexité, les tests de validation sont le plus souvent manuels. Les procédures à suivre sont regroupées dans un document associé au projet, fréquemment nommé plan de validation.

2. **Tests d'intégration :** Dans un projet informatique, l'intégration est de fait d'assembler plusieurs composants (*ou modules*) élémentaires en un composants de plus haut niveau. Un ***test d'intégration*** valide les résultats des interactions entre plusieurs composants permet de vérifier que leur assemblage s'est produit sans défaut. Il peut être manuel ou automatisé. Un nombre croissant de projets logiciels mettent en place *un processus d'intégration continue*. Cela consiste à vérifier que chaque modification ne produit pas de régression dans l'application développée. L'intégration continue est nécessairement liée à une batterie de tests qui se déclenchent automatiquement lorsque des modifications sont intégrées au code du projet.

3. **Tests unitaires :** Contrairement aux tests de validation et d'intégration qui testent des pans entiers d'un logiciel, un test unitaire na valide qu'une portion atomique du code source (*exemple : une seule classe*) et est systématiquement automatisé. Le test unitaire offre les avantages suivants :

   - Il est facile à écrire. Dédié à une partie très réduite du code, le test unitaire ne nécessite le plus souvent qu'un contexte minimal, voire pas de contexte du tout ;
   - Il offre une granularité de test très fine et permet de valider exhaustivement le comportement de la partie du code testée (*cas dégradés, saisie d'informations erronées*...) ;
   - Son exécution est rapide, ce qui permet de le lancer très fréquemment (*idéalement à chaque modification du code testé*) ;
   - Il rassemble les cas d'utilisation possibles d'une portion d'un projet et représente donc une véritable documentation sur la manière de manipuler le code testé ;

L'ensemble des tests unitaires[22] d'un projet permet de valider unitairement une grande partie de son code source et de détecter le plus tôt possibles d'éventuelles erreurs. En pratique, très peu de parties d'un projet fonctionnent de manière autonome, ce qui complique l'écriture des tests unitaires. Par exemple, comment tester unitairement une classe qui collabore avec plusieurs autres pour réaliser ses fonctionnalités ? La solution consiste à créer des éléments qui simulent le comportement des collaborateurs d'une classe donnée, afin de pouvoir tester le comportement de cette classe dans un environnement isolé et maîtrisé. Ces éléments sont appelés des « *tests doubles* ». Selon la complexité du test à écrire, un *test double* peut être :

- *Un* ***dummy*** : élément basique sans aucun comportement, juste là pour faire compiler le code lors du test ;
- *Un* ***stub*** *:* qui renvoie des données permettant de prévoir les résultats attendus lors du test ;
- *Un* ***mock*** : qui permet de vérifier finement le comportement de l'élément testé (*ordred'appel des méthodes, paramètres passés, etc*.).

[22] Certaines méthodologies de développement logiciel préconisent l'écriture des tests unitaires avant celle du code testé (TDD : *Test Driven Development*).

IV.8. LA DOCUMENTATION D'UN LOGICIEL

Précisons que l'objectif de cette partie est de découvrir les différents aspects associés à la documentation d'un logiciel. Nous avons déjà mentionné à maintes reprises qu'un logiciel a une durée de vie de plusieurs années et subit de nombreuses évolutions au cours de cette période. En entreprise, seule une petite minorité de logiciels sont conçus par un seul développeur. La grande majorité des projets sont réalisés et maintenus par une équipe de plusieurs personnes travaillant sur la même base de code source. Il est fréquent que les effectifs changent et que des développeurs soient amenés à travailler sur un logiciel sans avoir participé à sa création. L'intégration de ces nouveaux développeurs doit être aussi rapide et efficace que possible.

Cependant, il est pénible, voire parfois très difficile, de se familiariser avec un logiciel par la seule lecture de son code source. En complément, un ou plusieurs documents doivent accompagner le logiciel. On peut classer cette documentation en deux catégories : *La documentation technique* et *La documentation utilisateur*.

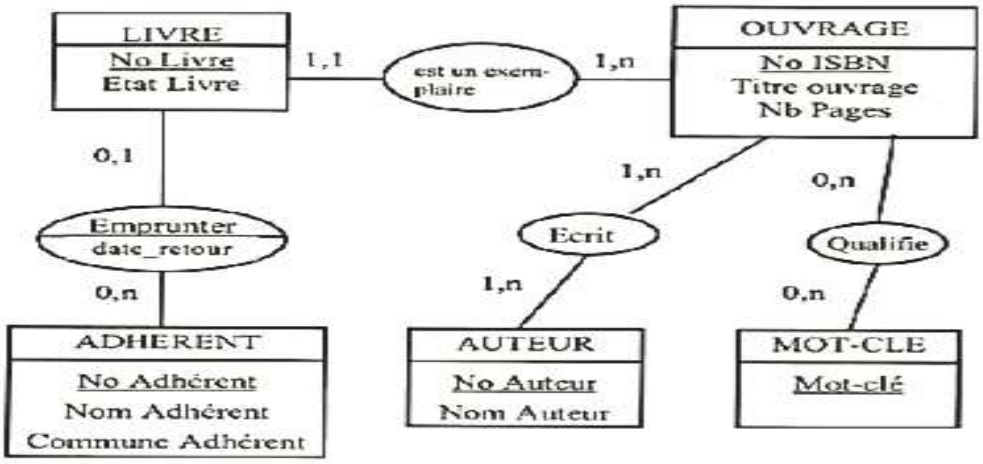
IV.8.1. LA DOCUMENTATION TECHNIQUE

Le mot-clé de la documentation technique est « *comment* ». Il ne s'agit pas ici de dire pourquoi le logiciel existe, ni de décrire ses fonctionnalités attendues. Ces informations figurent dans d'autres documents comme le cahier des charges. Il ne s'agit pas non plus d'expliquer à un utilisateur du logiciel ce qu'il doit faire pour effectuer telle ou telle tâche : c'est le rôle de la documentation utilisateur. La documentation technique doit expliquer ***comment*** fonctionne le logiciel.

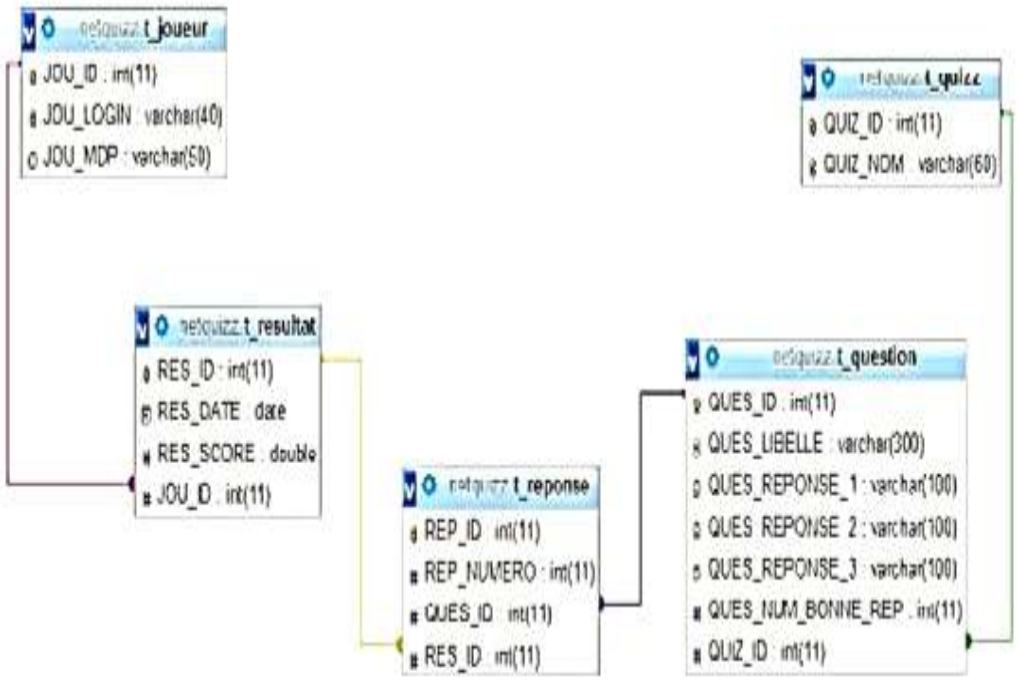
La documentation technique est écrite par des informaticiens, pour des informaticiens. Elle nécessite des compétences techniques pour être comprise. Le public visé est celui des personnes qui interviennent sur le logiciel du point de vue technique : « *développeurs, intégrateurs, responsables techniques, éventuellement chefs de projet* ». Dans le cadre d'une relation maîtrise d'ouvrage ou maîtrise d'œuvre pour réaliser un logiciel, la responsabilité de la documentation technique est à la charge de la maîtrise d'œuvre. Le contenu de la documentation technique varie fortement en fonction de la structure et de la complexité du logiciel associé. Néanmoins, nous allons décrire les aspects qu'on y retrouve le plus souvent :

1. **La Modélisation :** La documentation technique inclut les informations liées au domaine du logiciel. Elle précise comment les éléments - métiers ont été modélisés informatiquement au sein du logiciel.

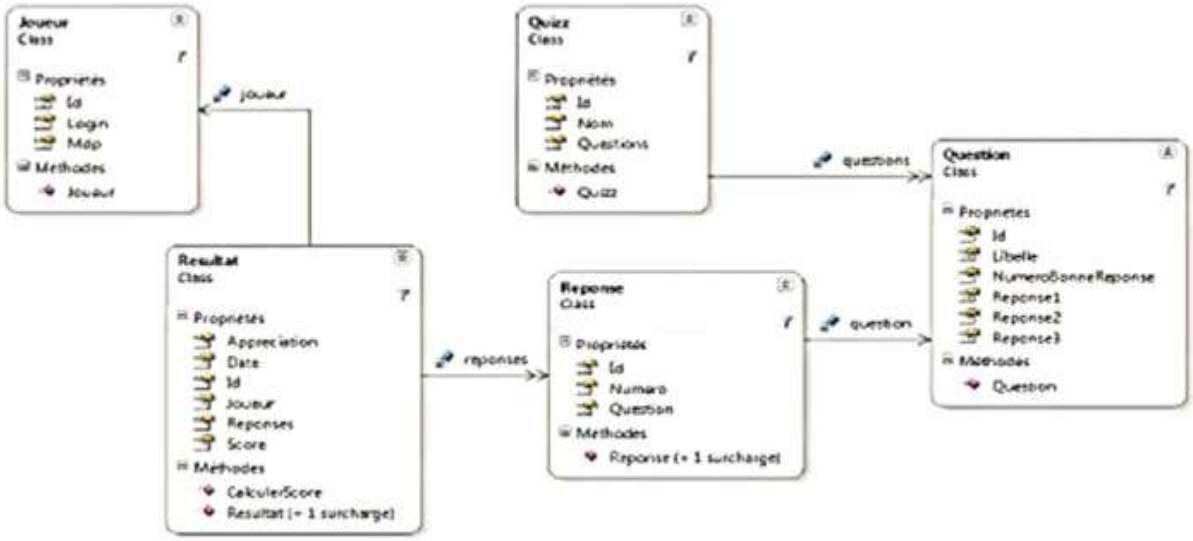
    - Si le logiciel a fait l'objet d'une modélisation de type entité-association, la documentation technique présente le modèle conceptuel résultat.



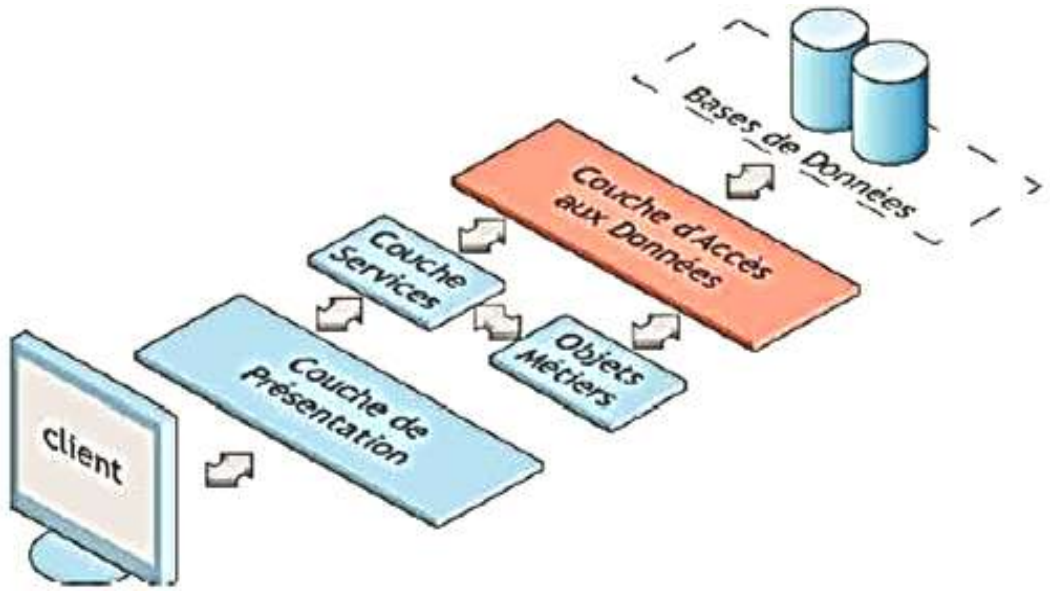
    - Si le logiciel a fait l'objet d'une modélisation orientée objet, la documentation technique inclut une représentation des principales classes (*souvent les classes métier*) sous la forme d'un diagramme de classes respectant la norme UML.



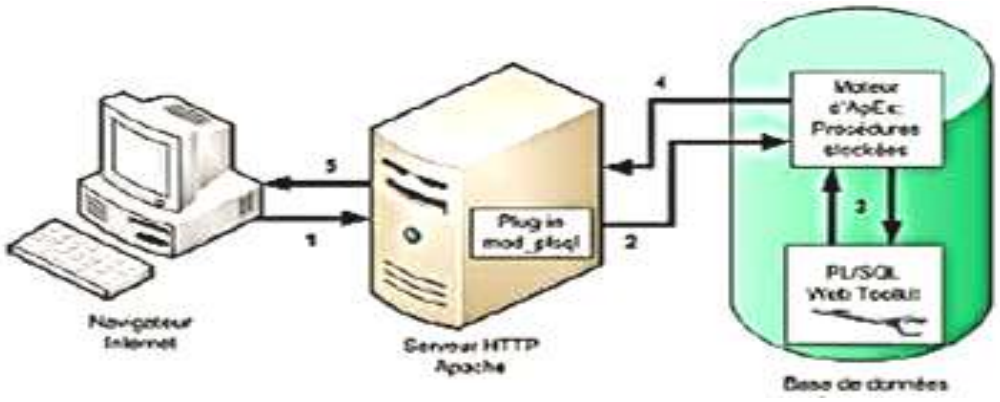
- Si le logiciel utilise une base de données, la documentation technique doit présenter le modèle d'organisation des données retenu, le plus souvent sous la forme d'un modèle logique sous forme graphique



2. **L'Architecture :** La phase d'architecture d'un logiciel permet, en partant des besoins exprimés dans le cahier des charges, de réaliser les grands choix qui structureront le développement : technologies, langages, patrons utilisés, découpage en sous-parties, outils, etc. La documentation technique doit décrire tous ces choix de conception. L'ajout de schémas est conseillé, par exemple pour illustrer une organisation logique multicouche.



L'implantation physique des différents composants (*appelés parfois tiers*) sur une ou plusieurs machines doit également être documentée.



3. **La Production du code source :** Afin d'augmenter la qualité du code source produit, de nombreux logiciels adoptent des normes ou des standards de production du code source : conventions de nommage, formatage du code, etc. Certains peuvent être internes et spécifiques au logiciel, d'autres sont reprises de normes existantes (*exemples : normes PSR-x pour PHP*).Afin que les nouveaux développeurs les connaissent et les respectent, ces normes et standards doivent être présentés dans la documentation technique.
4. **La Génération :** Le processus de génération (« ***build*** ») permet de passer des fichiers sources du logiciel aux éléments exécutables. Elle inclut souvent une phase de compilation du code source. Sa complexité est liée à celle du logiciel. Dans les cas simples, toute la génération est effectuée de manière transparente par l'IDE utilisé. Dans d'autres, elle se fait en plusieurs étapes et doit donc être documentée.
5. **Le Déploiement :** La documentation technique doit indiquer comment s'effectue le déploiement du logiciel, c'est-à-dire l'installation de ses différents composants sur la ou les machines nécessaires. Là encore, cette étape peut être plus ou moins complexe et nécessiter plus ou moins de documentation.
6. **La Documentation du code source :** Il est également possible de documenter un logiciel directement depuis son code source en y ajoutant des commentaires. Certains langages disposent d'un format spécial de commentaire permettant de créer une documentation auto-générée. C'est par exemple, *Le langage Java* qui a été le premier langage à introduire une technique de documentation du code source basée sur l'ajout de commentaires utilisant un format spécial. En exécutant un outil du JDK appelé « ***javadoc*** », on obtient une documentation du code source au format HTML.

```
/**
* Valide un mouvement de jeu d'Echecs. *
* @paramleDepuisFile File de la pièce à déplacer
* @paramleDepuisRangée Rangée de la pièce à déplacer
* @paramleVersFile File de la case de destination
* @paramleVersRangée Rangée de la case de destination
* @return vrai si le mouvement d'échec est valide ou faux si invalide */
booleanestUnDeplacementValide(intleDepuisFile, intleDepuisRangée, intleVersFile, i
ntleVersRangée) {
// ...
}
```

Voici un exemple de la méthode Java documentée au format javadoc.

L'avantage de cette approche est qu'elle facilite la documentation du code par les développeurs, au fur et à mesure de son écriture. Depuis, de nombreux langages ont repris l'idée.

```
/// <summary>
/// Classe modélisant un adversaire de Chifoumi
/// </summary>
public class Adversaire {
/// <summary>
/// Générateur de nombres aléatoires
/// Utilisé pour simuler le choix d'un signe : pierre, feuille ou ciseau
/// </summary>
private static Random rng = new Random();
/// <summary>
/// Fait choisir aléatoirement un coup à l'adversaire
/// Les coups possibles sont : "pierre", "feuille", "ciseau"
/// </summary>
/// <returns>Le coup choisi</returns>
public string ChoisirCoup()
{
// ...
}}
```

Voici un exemple de documentation d'une classe C#, qui utilise une syntaxe légèrement différente.

IV.8.2. LA DOCUMENTATION UTILISATEUR

Contrairement à la documentation technique, la documentation d'utilisation ne vise pas à faire comprendre comment le logiciel est conçu. Son objectif est d'apprendre à l'utilisateur à se servir du logiciel. La documentation d'utilisation doit être :

- ***Utile*** : une information exacte, mais inutile, ne fait que renforcer le sentiment d'inutilité et gêne la recherche de l'information pertinente ;
- ***Agréable*** : sa forme doit favoriser la clarté et mettre en avant les préoccupations de l'utilisateur et non pas les caractéristiques techniques du produit.

Le public visé est l'ensemble des utilisateurs du logiciel. Selon le contexte d'utilisation, les utilisateurs du logiciel à documenter peuvent avoir des connaissances en informatique (*exemples : cas d'un IDE ou d'un outil de SCM*). Cependant, on supposera le plus souvent que le public visé n'est pas un public d'informaticiens. La Conséquence essentielle est que toute information trop technique est à bannir de la documentation d'utilisation. Pas question d'aborder, l'architecture MVC ou les design patterns employés : ces éléments ont leur place dans la documentation technique. D'une manière générale, s'adapter aux connaissances du public visé constitue la principale difficulté de la rédaction de la documentation d'utilisation.

LES FORMES DE LA DOCUMENTATION UTILISATEUR

1. **Le Manuel utilisateur :** La forme la plus classique de la documentation d'utilisation consiste à rédiger un manuel utilisateur, le plus souvent sous la forme d'un document bureautique. Ce document est structuré et permet aux utilisateurs de retrouver les informations qu'ils recherchent. Il intègre très souvent des captures d'écran afin d'illustrer le propos. Un manuel utilisateur peut être organisé de deux façons :
   - **Le Guide d'utilisation** : ce mode d'organisation décompose la documentation en grandes fonctionnalités décrites pas à pas et dans l'ordre de leur utilisation. Exemple pour un logiciel de finances personnelles : création d'un compte, ajout d'écritures, pointage...Cette organisation plaît souvent aux utilisateurs car elle leur permet d'accéder facilement aux informations essentielles. En revanche, s'informer sur une fonctionnalité avancée ou un détail complexe peut s'avérer difficile.
   - **Manuel de référence** : dans ce mode d'organisation, on décrit une par une chaque fonctionnalité du logiciel, sans se préoccuper de leur ordre ou de leur fréquence d'utilisation. Par exemple, on décrit l'un après l'autre chacun des boutons d'une barre de boutons, alors que certains sont plus « *importants* » que d'autres. Cette organisation suit la logique du créateur du logiciel plutôt que celle de son utilisateur. Elle est en général moins appréciée de ces derniers.

2. **Le Tutoriel :** De plus en plus souvent, la documentation d'utilisation inclut un ou plusieurs tutoriels, destinés à faciliter la prise en main initiale du logiciel. Un tutoriel est un guide pédagogique constitué d'instructions détaillées pas à pas en vue d'objectifs simples. Le tutoriel a l'avantage de "prendre l'utilisateur par la main" afin de l'aider à réaliser ses premiers pas avec le logiciel qu'il découvre, sans l'obliger à parcourir un manuel utilisateur plus ou moins volumineux. Il peut prendre la forme d'un document texte, ou bien d'une vidéo ou d'un exercice interactif. Cependant, il est illusoire de vouloir documenter l'intégralité d'un logiciel en accumulant les tutoriels.
3. ***Le FAQ* :** Une Foire Aux Questions (en anglais *Frequently Asked questions*) est une liste de questions-réponses sur un sujet. Elle peut faire partie de la documentation d'utilisation d'un logiciel. La création d'une FAQ permet d'éviter que les mêmes questions soient régulièrement posées.
4. **L'Aide en ligne :** L'aide en ligne est une forme de documentation d'utilisation accessible depuis un ordinateur. Il peut s'agir d'une partie de la documentation publiée sur Internet sous un format hypertexte. Quand une section de l'aide en ligne est accessible facilement depuis la fonctionnalité d'un logiciel qu'elle concerne, elle est appelée aide contextuelle ou aide en ligne contextuelle.

Les principaux formats d'aide en ligne sont le *HTML* et le *PDF*. Microsoft en a publié plusieurs formats pour l'aide en ligne des logiciels tournant sous Windows : *HLP*, *CHM* ou encore *MAML*. Un moyen simple et efficace de fournir une aide en ligne consiste à définir des infos bulle (*tooltips*). Elles permettent de décrire succinctement une fonctionnalité par survol du curseur.

IV.8.3. LES RECOMMANDATIONS DE REDACTION D'UNE DOCUMENTATION

Quelle que soit la documentation à rédiger : *technique* ou *d'utilisation*, toute documentation doit absolument être écrite de manière **structurée**, afin de faciliter l'accès à une information précise. Ce qui signifie :

- Le décomposer en paragraphes suivant une organisation hiérarchique ;
- Et Utiliser des styles de titre, une table des matières, des références...

Comme dit jadis, le public visé n'a pas forcément d'expérience informatique, il est indispensable de bannir les explications trop techniques et penser à définir les principaux termes et le « *jargon* » utilisé. Une documentation doit être rédigée dans une langue simple, pour être compris de tous, y compris de personnes étrangères apprenant la langue de rédaction.

CINQUIEME CHAPITRE LES METHODES DE DEVELOPPEMENT LOGICIEL

Une méthode définit une démarche en vue de produire des résultats. Par exemple, les cuisiniers utilisent des recettes de cuisine, les pilotes déroulent des check-lists avant de décoller, les architectes font des plans avant de superviser des chantiers. *Une méthode* permet d'assister une ou plusieurs étapes du cycle de vie du logiciel. Les méthodes d'analyse et de conception fournissent des notations standards et des conseils pratiques qui permettent d'aboutir à des conceptions « *raisonnables* », mais nous ferons toujours appel à la créativité du concepteur. Il existe différentes manières pour classer ces méthodes, dont :

- **La distinction compositionnelle / décompositionnelle** : met en opposition d'une part les méthodes ascendantes qui consistent à construire un logiciel par composition à partir de modules existants et, d'autre part, les méthodes descendantes qui décomposent récursivement le système jusqu'à arriver à des modules programmables « *simplement* ».
- **la distinction fonctionnel / orientée objet** : Dans *la stratégie fonctionnelle* un système est vu comme un ensemble d'unités en interaction, ayant chacune une fonction clairement définie. Les fonctions disposent d'un état local, mais le système a un état partagé, qui est centralisé et accessible par l'ensemble des fonctions. *Les stratégies orientées objet* considèrent qu'un système est un ensemble d'objets interagissant. Chaque objet dispose d'un ensemble d'attributs décrivant son état et l'état du système est décrit (*de façon décentralisé*) par l'état de l'ensemble. La décomposition fonctionnelle du haut vers le bas a été largement utilisée aussi bien dans de petits projets que dans de très grands, et dans divers domaines d'application. La méthode orientée objet a eu un développement plus récent. Elle encourage la production de systèmes divisés en composants indépendants, en interaction.

Dans le cadre de cours, nous distinguerons alors quatre (4) types des méthodes à savoir :

- *les méthodes fonctionnelles*, basées sur les fonctionnalités du logiciel ;
- *les méthodes objet*, basées sur différents modèles (*statiques, dynamiques et fonctionnels*) de développement logiciel.
- *Les méthodes adaptatives ou Agiles*, basées sur le changement des besoins ;
- *Les méthodes spécifiques*, basées sur les découpages temporels particuliers.

V.1. LES METHODES FONCTIONNELLES

Les méthodes fonctionnelles ont pour origine la programmation structurée. Cette approche consiste à décomposer une fonctionnalité (*ou fonction*) du logiciel en plusieurs sous fonctions plus simples. Il s'agit d'une conception « top-down », basée sur le principe « *diviser pour mieux régner*». L'architecture du système est le reflet de cette décomposition fonctionnelle. La programmation peut ensuite être réalisée soit à partir des fonctions de haut niveau (***développement « top-down »),*** soit à partir des fonctions de bas niveau (développement « ***bottom-up*** »).

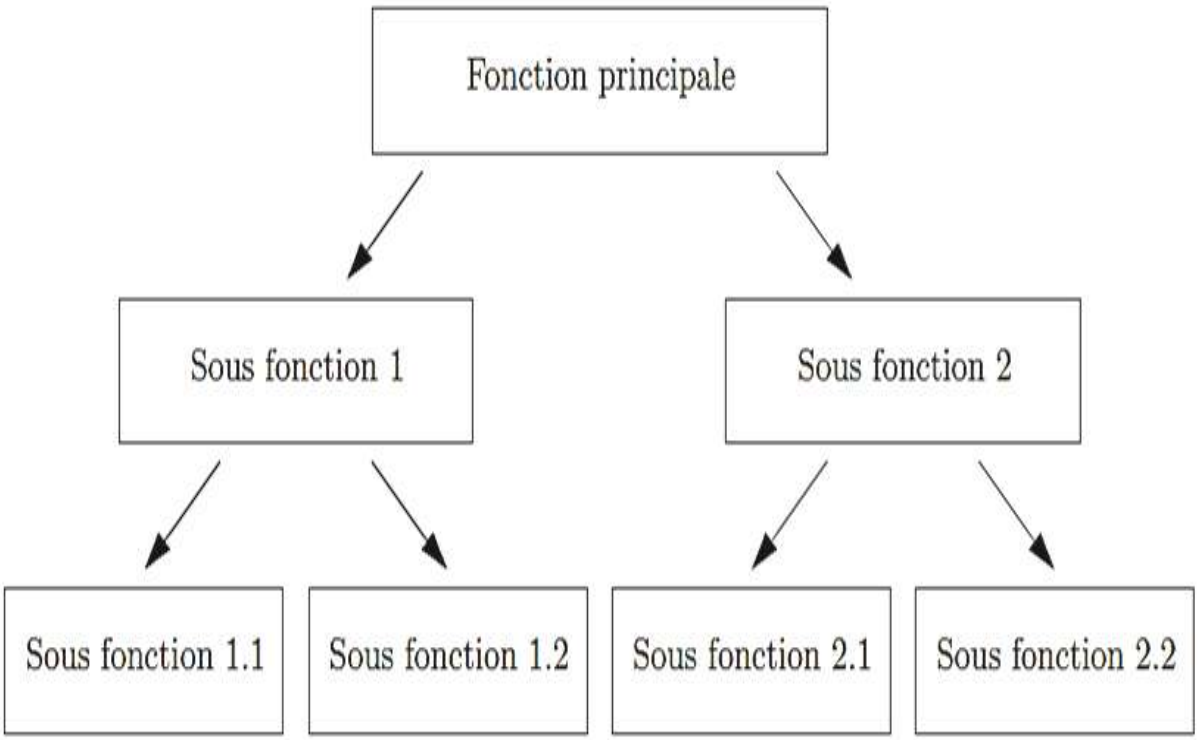


Schéma de la décomposition fonctionnelle

Cette méthode présente comme les inconvénients suivants :

- L'architecture étant basée sur la décomposition fonctionnelle, une évolution fonctionnelle peut remettre en cause l'architecture. Cette méthode supporte donc mal l'évolution des besoins.
- Cette méthode ne favorise pas la réutilisation de composants, car les composants de bas niveau sont souvent ad hoc et donc peu réutilisables.

IV.2. LES METHODES OBJET

Les approches objet sont basées sur une modélisation du domaine d'application. Les « *objets* »sont une abstraction des entités du monde réel. De façon générale, la modélisation permet de réduire la complexité et de communiquer avec les utilisateurs. Plus précisément un modèle :

- Aide à visualiser un système tel qu'il est ou tel qu'on le souhaite ;
- Permet de spécifier la structure ou le comportement d'un système ;
- Fournit un guide pour la construction du système ;
- Documente les décisions prises lors de la construction du système.

Ces modèles peuvent être comparés avec les plans d'un architecte : suivant la complexité du système on a besoin de plans plus ou moins précis. Pour construire une niche, on n'a pas besoin de plans, pour construire un chalet il faut un plan, pour construire un immeuble, on a besoin d'un ensemble de vues (*plans au sol, perspectives, maquettes*). Dans les méthodes objet, on distingue trois aspects :

- **Un aspect statique** : Dans lequel, on identifie les objets, leurs propriétés et leurs relations ;
- **Un aspect dynamique** : Dans lequel, on décrit les comportements des objets, en particuliers leurs états possibles et les évènements qui déclenchent les changements d'état ;
- **Un aspect fonctionnel** : qui, à haut niveau, décrit les fonctionnalités du logiciel, ou, a plus bas niveau, décrit les fonctions réalisées par les objets par l'intermédiaire des méthodes.

Les intérêts des approches objet sont les suivants :

- Les approches objet sont souvent qualifiées de « *naturelles* » car elles sont basées sur le domaine d'application. Cela facilite en particulier la communication avec les utilisateurs.
- Ces approches supportent mieux l'évolution des besoins que les approches fonctionnelles car *la modélisation est plus stable*, et *les évolutions fonctionnelles ne remettent par l'architecture du système en cause*.
- Les approches objet facilitent la réutilisation des composants (*qui sont moins spécifiques que lorsqu'on réalise une décomposition fonctionnelle*).

V.3. LES METHODES ADAPTATIVES

Les méthodes dites « adaptatives » sont subdivisées en 2 parties notamment : *les méthodes prédictives* et *les méthodes agiles (adaptatives)*.

V.3.1. LES METHODES PREDICTIVES

Ce sont des méthodes qui correspondent à un cycle de vie du logiciel en cascade ou en V, sont basées sur une planification très précise et très détaillée, qui a pour but de réduire les incertitudes liées au développement du logiciel. Cette planification rigoureuse ne permet pas d'évolutions dans les besoins des utilisateurs, qui doivent donc être figes dès l'étape de définition des besoins.

V.3.2. LES METHODES AGILES

Ce sont des méthodes qui correspondent à un cycle de vie itératif, qui considèrent que les changements (*des besoins des utilisateurs, mais également de l'architecture, de la conception, de la technologie*) sont inévitables et doivent être pris en compte par les modèles de développement. Ces méthodes privilégient la livraison de fonctionnalités utiles au client à la production de documentation intermédiaire sans intérêt pour le client.

Ainsi, Toutes les méthodes agiles prennent en compte dans leur modèle de cycle de vie trois exigences :

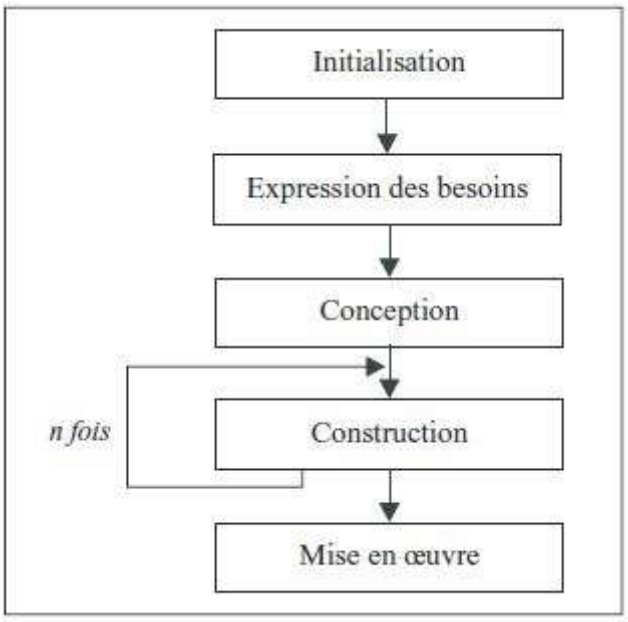
- Une forte participation entre développeurs et utilisateurs,
- Des livraisons fréquentes de logiciel et ;
- une prise en compte de possibles changements dans les besoins des utilisateurs au cours du projet.

C'est pourquoi toutes font appel, d'une façon ou d'une autre, à un modèle itératif et incrémental. De plus, elles préconisent en général des durées de cycle de vie des projets ne dépassant pas un an. Parmi les méthodes agiles, les plus usuelles ; on peut citer :

- La méthode RAD (*Rapid Application Development*) ;
- La méthode DSDM (*Dynamic Systems Development Method*);
- La méthode XP (*Programmation eXtrême*);
- La méthode SCRUM.

V. 3.2.1. La méthode RAD

La méthode RAD, est un modèle linéaire (présentoir), structuré en cinq phases, et dont le modèle itératif intervient à la phase Construction du logiciel en vu de la séquencer en plusieurs modules Successivement livrés.

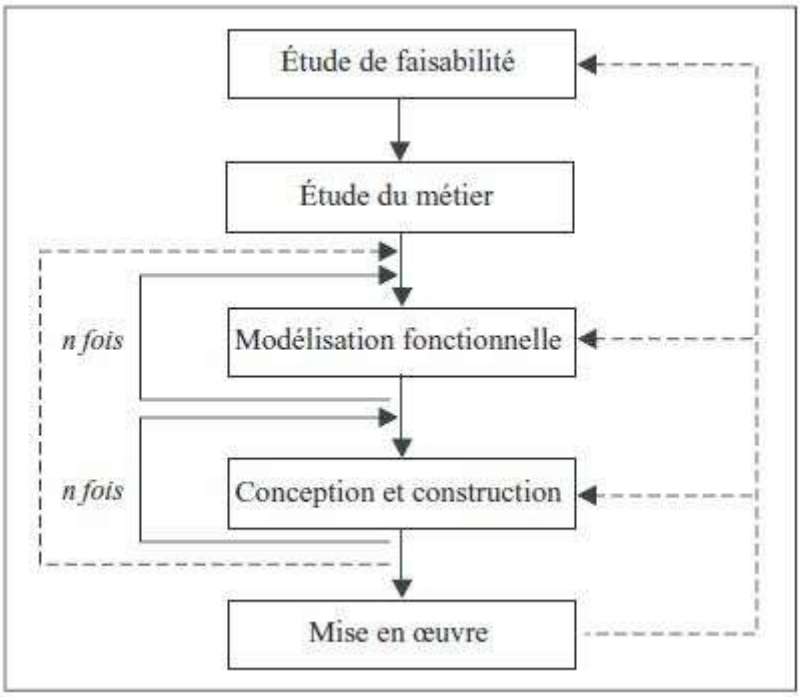


La participation des utilisateurs est placée au cœur du cycle. En effet, le déroulement d'une phase comprend une ou plusieurs sous-phases, et chaque sous-phase présente une structure à trois temps, dans laquelle la tenue d'une session participative joue un rôle central. Des travaux préparatoires rassemblent et construisent le matériau (modèle ou prototype) qui sera ensuite discuté par les différents acteurs et ajusté.

V.3.2.2. La méthode DSDM

La méthode DSDM a évolué au cours des années. L'actuelle version distingue *le cycle de vie du système* et *le cycle de vie du projet*. Le premier comprend, outre les phases du projet lui-même, une phase de pré-projet qui doit conduire au lancement du projet et une phase post-projet qui recouvre l'exploitation et la maintenance de l'application.

Le cycle de vie du projet comprend cinq phases, dont deux sont cycliques. Les flèches pleines indiquent un déroulement normal. Les flèches en pointillé montrent des retours possibles à une phase antérieure, soit après la phase Conception et construction, soit après celle de Mise en œuvre.

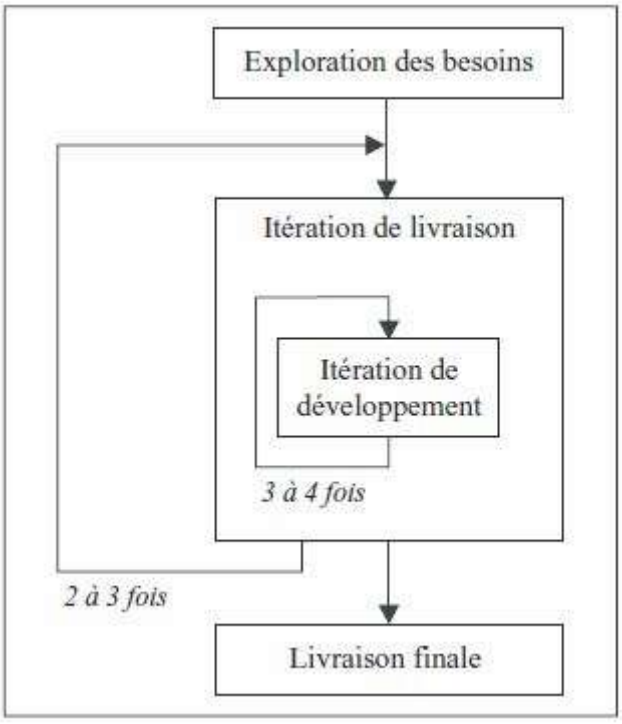


Après une Étude de faisabilité, la phase Étude du métier permet, à travers des ateliers (*workshops*) entre équipe de projet et manageurs, de définir le périmètre du projet, avec une liste d'exigences prioritaires et une architecture fonctionnelle et technique du futur système.

La phase Modélisation fonctionnelle est une suite de cycles. Chacun permet de définir précisément les fonctionnalités souhaitées et leur priorité. L'acceptation par toutes les parties prenantes d'un prototype fonctionnel, sur tout ou partie du périmètre, permet de passer à la phase Conception et construction. L'objectif de cette phase est de développer un logiciel testé, par des cycles successifs de développement/acceptation par les utilisateurs.

V.3.2.3. Le modèle XP

La méthode XP, focalisée sur la partie programmation du projet, propose un modèle itératif avec une structure à deux niveaux : d'abord des itérations de livraison (*release*), puis des itérations de développement. Les premières conduisent à livrer des fonctionnalités complètes pour le client, les secondes portent sur des éléments plus fins appelés scénarios qui contribuent à la définition d'une fonctionnalité.

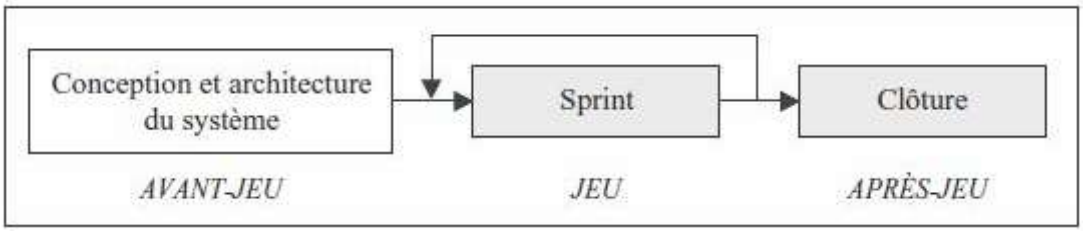


Après une phase initiale d'Exploration des besoins, un plan de livraison est défini avec le client. Chaque livraison, d'une durée de quelques mois, se termine par la fourniture d'une version opérationnelle du logiciel. Une itération de livraison est découpée en plusieurs itérations de développement de courte durée (deux semaines à un mois), chacune donnant lieu à la livraison d'une ou plusieurs fonctionnalités pouvant être testées, voire intégrées dans une version en cours.

De façon plus détaillée, chaque itération de développement commence par l'écriture de cas d'utilisation ou scénarios (*user stories*), c'est-à-dire des fonctions simples, concrètement décrites, avec les exigences associées, qui participent à la définition d'une fonctionnalité plus globale. Utilisateurs et développeurs déterminent ensemble ce qui doit être développé dans la prochaine itération. Une fonctionnalité est ainsi découpée en plusieurs tâches. Les plans de test sont écrits, les développeurs sont répartis en binôme ; ils codent les tâches qui leur sont affectées, puis effectuent avec les utilisateurs des tests d'acceptation. En cas d'échec, on revoit les scénarios et on reprend la boucle. Sinon, on continue jusqu'à avoir développé tous les scénarios retenus. Une version livrable est alors arrêtée et mise à disposition, ainsi que la documentation.

V.3.2.4. La méthode SCRUM

La méthode SCRUM emprunte au vocabulaire du jeu le qualificatif des trois phases du cycle préconisé.



- La phase d'Avant-jeu (*pre-game*), Conception et architecture du système, se déroule de façon structurée, en général linéaire, et permet de déterminer le périmètre, la base du contenu du produit à développer et une analyse de haut niveau.

- La phase de Jeu (*game*) est itérative et qualifiée d'empirique, dans la mesure où le travail effectué ne fait pas l'objet d'une planification. Une itération, dont la durée oscille entre une et quatre semaines, est appelée un Sprint, en référence à ces poussées rapides et fortes que les joueurs de rugby peuvent effectuer sur le terrain. Un Sprint est découpé en trois sous-phases :

  → Développement (*develop*) : il s'agit de déterminer l'objectif visé au terme de l'itération, de le répartir en « paquets » de fonctions élémentaires, de développer et tester chaque paquet.

  → Emballage (*wrap*) : on referme les « paquets » et on les assemble pour faire une version exécutable. • Revue (*review*) : une revue élargie permet de faire le point sur les problèmes et l'avancement.

  → Ajustement (*adjust*) : ajusté le travail restant.

- La phase d'Après-Jeu (*postgame*), Clôture, vise à livrer un produit complet et documenté. Comme dans la première phase, on peut en planifier les tâches et les dérouler de façon linéaire.

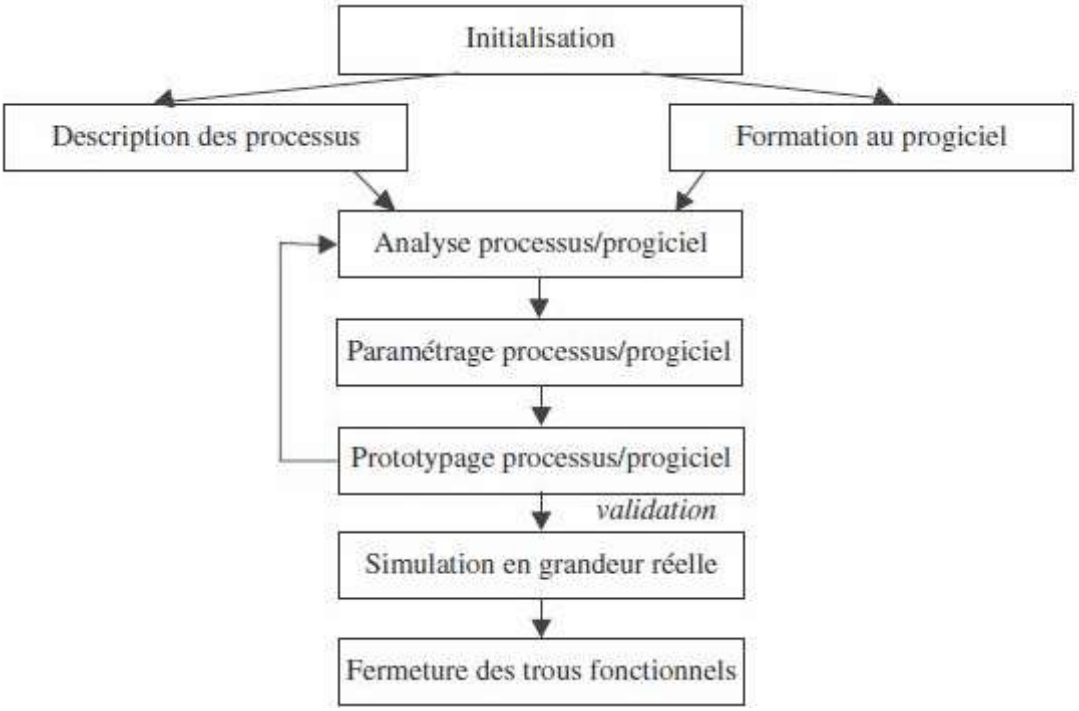
V.4. LES METHODES SPECIFIQUES

Certains découpages temporels sont liés soit à une méthode, soit à un type de projet bien particulier. Nous en proposons deux exemples : le découpage préconisé pour mettre en place un progiciel intégré et le modèle RUP proposé par la société Rational Software. A ce stade, nous citerons les méthodes telles que :

- La méthode ERP ;
- La méthode RUP ;

V.4.1. Le cycle ERP

La mise en place d'un progiciel de gestion intégré, souvent appelé du terme anglo-saxon ERP (*Enterprise Resource Planning*) s'appuie sur un découpage spécifique.



En effet, il s'agit de construire, en tirant le meilleur parti du progiciel, un système améliorant la performance de l'entreprise. Deux étapes doivent donc être menées en parallèle : Description des processus et Formation au progiciel. Ensuite, il y a autant de *cycles d'analyse — paramétrage — prototypage* qu'il y a de processus. La validation par le Comité de pilotage permet une simulation en grandeur réelle. Il faut alors prendre en compte ce qui est resté en dehors du champ couvert par le progiciel.

V.4.2. Le modèle RUP

Le modèle RUP (*Rational Unified Process*) est représentatif d'une approche combinant plusieurs modèles. Sa structure fait l'objet d'un assez large accord, notamment parmi les praticiens (figure 2.17). Il peut être lu de la façon suivante :

1. Le cycle est constitué de quatre phases principales, que l'on retrouve globalement dans toutes les approches descendantes : étude préalable (opportunité), conception de la solution détaillée (élaboration), développement de la solution (*construction*) et mise en œuvre (transition).

2. Il existe six types de tâches qui, au lieu d'être affectées exclusivement à une phase, se retrouvent à des degrés divers dans chacune des phases. Par exemple, l'étude des besoins peut apparaître jusqu'à la fin du projet, mais la plus grande partie est effectuée dans les deux premières phases. L'implémentation (développement) a principalement lieu dans la phase de construction, mais on peut réaliser un prototype dès la première phase. Certaines tâches, comme la direction de projet, s'effectuent sur toute la durée du projet.

3. Certaines phases peuvent être menées de façon cyclique. Ainsi, l'élaboration se fait en deux cycles, conduisant par exemple à la production de spécifications externes (vision utilisateur) et spécifications techniques (vision développeur). La construction est itérative et incrémentale. De plus, l'ensemble du modèle représente un tour de spirale, dans le cas d'une approche globale en spirale.

CONCLUSION

Le génie logiciel est la science des bonnes pratiques de développement de logiciel. Cette science étudie en particulier la répartition des phases dans le temps, les bonnes pratiques concernant les documents clés que sont le cahier des charges, le diagramme d'architecture ou le diagramme de classes. Le but recherché est d'obtenir des logiciels de grande ampleur qui soient fiables, de qualité, et correspondent aux attentes de l'usager.

Par contre, *Le développement de logiciel* consiste à étudier, concevoir, construire, transformer, mettre au point, maintenir et améliorer des logiciels. Ce travail est effectué par les employés d'éditeurs de logiciels, de sociétés de services et d'ingénierie informatique (SSII), des travailleurs indépendants (*freelance*) et des membres de la communauté du logiciel libre. Le développement d'une application exige de :

- *Procéder par étapes,* en prenant connaissance des besoins de l'utilisateur ; effectuer l'analyse ; trouver une solution informatique ; réaliser ; tester ; installer et assurer le suivi.

- *Procéder avec méthode*, en partant du général aux détails et aux techniques ; fournir une documentation ; s'aider de méthodes appropriées ; et Savoir se remettre en question.

- *Choisir une bonne équipe,* en trouvant les compétences adéquates ; définir les missions de chacun et coordonner les différentes actions pour la construction du logiciel.

- *Contrôler les coûts et délais,* en considérant l'aspect économique ; maîtriser de la conduite du projet ; et investissements aux bons moments.

- *Garantir le succès du logiciel,* en répondant à la demande et assurer la qualité du logiciel.

- Envisager l'évolution du logiciel, du matériel et de l'équipe.

Ainsi, Un logiciel est créé petit à petit par une équipe d'ingénieurs conformément à un cahier des charges établi par un client demandeur ou une équipe interne. Le logiciel est décomposé en différents modules et un chef de projet, ou *architecte*, se charge de la cohérence de l'ensemble. Différentes activités permettent de prendre connaissance des attentes de l'usager, créer un modèle théorique du logiciel, qui servira de plan de construction, puis construire le logiciel, contrôler son bon fonctionnement et son adéquation au besoin. La planification et la répartition des travaux permettent d'anticiper le délai et le coût de fabrication.

Le logiciel est accompagné d'une procédure d'installation, d'une procédure de vérification de bonne installation, de documentation (parfois créé automatiquement à partir de commentaires placés à cet effet dans le code source) et d'une équipe d'assistance au déploiement et à la maintenance, désignée sous le nom de support. Outre les travaux d'analyse, de conception, de construction et de tests, une procédure de *recette*, permettra de déterminer si le logiciel peut être considéré comme utilisable.

Nous espérons que le support de cours de l'introduction au Génie logiciel, vous a été utile et vous permettra d'approfondir vos connaissances informatiques relatives à la conception et à la production des logiciels. L'auteur de cet ouvrage, le Docteur YENDE RAPHAEL Grevisse, vous remercie prestement du temps que vous consacrerez à l'exploitation de support de cours. N'hésitez pas à nous contacter pour plus d'information. Bonne lecture et digestion.