

**UNIVERSITE LUMIERE DE BUJUMBURA
FACULTE D'INFORMATIQUE DE GESTION
TROISIEME ANNEE DE
BACCALAUREAT**

COURS D'INGENIERIE DU LOGICIEL

ANNEE : 2020 - 2021

TITULAIRE DU COURS : MSc. Jackson NIGARURA

Table des matières

INTRODUCTION.....	1
OBJECTS DU COURS.....	2
DEFINITION DES CONCEPTS CLES	2
HISTOIRE.....	3
CHAPITRE 0 : GENERALITES SUR LE GENIE (L'INGENIERIE) LOGICIELLE	5
0.1. DEFINITION ET CONTEXTE D'ETUDES	6
0.2. ÉVOLUTION DU LOGICIEL	7
0.2.1. LOIS D'EVOLUTION DU LOGICIEL.....	8
0.2.2. EVOLUTION DU LOGICIEL E-TYPE	9
0.3. LES PARADIGMES LOGICIELS	10
0.3.1. PARADIGME DE DEVELOPPEMENT LOGICIEL.....	11
0.3.2. PARADIGME DE CONCEPTION	11
0.3.3. PARADIGME DE PROGRAMMATION.....	11
0.4. BESOINS DU GENIE LOGICIEL	11
0.5. CARACTERISTIQUES D'UN BON LOGICIEL.....	12
0.5.1. CRITERES GENERAUX.....	12
0.5.2. CRITERES EXTERNES	13
0.5.3. CRITERES INTERNES	14
0.6. CATEGORIES DE LOGICIELS	15
0.7. ENJEUX DU GENIE LOGICIEL.....	15
0.8. DIMENSIONS DU GENIE LOGICIEL	16
0.9. CYCLE DE VIE D'UN LOGICIEL.....	17
a) Le modèle en cascade (waterfall model)	17
1 ⁰ Qu'est-ce que le modèle en cascade ?.....	17
2 ⁰ Comment fonctionne le modèle en cascade ?.....	17
3 ⁰ Les phases du modèle en cascade	19

□ Analyse	19
□ Conception	19
□ Implémentation	19
□ Test.....	20
4 ⁰ Évaluation du modèle en cascade	20
5 ⁰ Vue d'ensemble des avantages et inconvénients du modèle en cascade	20
□ Une vérification après chaque phase de projet	21
□ Au moins deux itérations	21
□ Des tests intégrant l'utilisateur final	21
6 ⁰ Alternatives au modèle en cascade	21
b) Le modèle en V.....	22
1 ⁰ Les différentes phases du cycle en V	22
2 ⁰ Interaction entre conception et assurance qualité	23
3 ⁰ Le V modél XT : la poursuite du développement du cycle en V.....	23
4 ⁰ Domaines d'application du cycle en V	24
5 ⁰ Avantages et inconvénients du cycle en V	24
6 ⁰ Les alternatives au cycle en V	25
c) Modèle en spirale.....	25
1 ⁰ Informations générales.....	25
2 ⁰ Comment ça fonctionne	26
3 ⁰ Avantages/inconvénients	26
4 ⁰ Importance pour la programmation	27
0.10. LES PRINCIPES DU GENIE LOGICIEL.....	27
CHAPITRE I : ANALYSE FONCTIONNELLE D'UN PRODUIT LOGICIEL (AF).....	29
I.1 : Présentation	29
I.2 : Maîtrisez vos coûts grâce à l'analyse fonctionnelle.....	31
I.3 : En quoi consiste une analyse fonctionnelle ?.....	31
I.4 : Quatre grandes étapes de l'analyse fonctionnelle selon la méthode ROCH.....	31
I.5 : Qu'est-ce que l'analyse fonctionnelle apporte à votre projet ?.....	31
I.6 : Les autres avantages.....	32
I.7 : Cahier des charges projet	32

a. Pourquoi un cahier des charges projet ?	33
b. C'est quoi au juste un cahier des charges projet ?	34
c. Quand intervient le cahier des charges de projet ?	34
d. Les techniques d'élaboration d'un cahier des charges projet	35
e. Les composantes d'un cahier des charges projet	36
1. Description de l'état actuel (Contexte).....	36
2. Les spécifications non fonctionnelles	36
3. Les spécifications fonctionnelles	37
4. Les ressources	37
5. Les délais.....	37
6. Les besoins financiers et le budget.....	37
f. Modèle de cahier des charges	38
g. Conclusion	38
CHAPITRE II : ARCHITECTURE ET/OU CONCEPTION DU LOGICIELLE	39
1. Définition	39
2. Importance.....	39
3. Objectifs	40
4. Architecture ou conception ?.....	40
5. L'activité d'architecture	41
a. Définition.....	41
b. Place dans le processus de création	41
6. L'architecture d'un logiciel	42
CHAPITRE III : PRINCIPES DE CONCEPTION.....	42
1. Séparation des responsabilités.....	42
2. Réutilisation	43
3. Encapsulation maximale	43
4. Couplage faible	44
5. Cohésion forte	44

6. DRY	44
7. KISS	46
8. YAGNI.....	46
CHAPITRE IV : PRODUCTION DU CODE SOURCE.....	46
1. Introduction	46
2. Convention de nommage.....	47
3. Langue utilisée	47
4. Formatage du code	48
5. Commentaires.....	48
CHAPITRE V : GESTION DES VERSIONS	49
1. Introduction	49
2. Les logiciels de gestion des versions.....	50
a) Principales fonctionnalités.....	50
b) Gestion centralisée Vs gestion décentralisée.....	50
c) Principaux logiciels de gestion des versions	52
3. Présentation de Git	52
a) Fonctionnement	53
b) Principales opérations.....	54
c) Notion de branche.....	54
d) Le fichier .gitignore	55
CHAPITRE VI : TRAVAIL COLLABORATIF	56
1. Les enjeux du travail collaboratif.....	56
2. Présentation de GitHub	56
a) Modèle économique	56
b) Fonctionnement	57
c) Issues	58
CHAPITRE VII : LES TESTS.....	59

1. Pourquoi tester ?.....	59
2. Comment tester ?.....	59
a) Tests de validation	59
b) Tests d'intégration.....	59
c) Tests unitaires	60
3. Création de tests doubles	60
4. Compléments.....	61
CHAPITRE VIII : DOCUMENTATION	62
1. Introduction	62
2. La documentation technique	62
a) Rôle.....	62
b) Public visé.....	62
c) Contenu.....	63
3. La documentation utilisateur	67
4. Conseils de rédaction.....	69

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 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

Parmi les méthodes de développement, les approches objet, issues de la programmation objet, sont basées sur une modélisation du domaine d'application. 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, au développement des applications informatiques de façon systématique (méthodique) et reproductible (rééritable) afin de produire, dans les conditions prévues au préalable, un outil répondant aux besoins d'utilisateurs. Et, D'une façon spécifique ce cours vise à :

- Acquérir aux étudiants qui auront suivi ce cours, les bonnes pratiques d'analyser d'abord les besoins avant toute autre tâche du développement d'un logiciel ;
- Acquérir aux étudiants l'étape de conception et architecture au sein du développement d'un logiciel ;
- Acquérir aux étudiants les bonnes pratiques et principes à tenir en compte lors du codage ;
- Acquérir aux étudiants le comportement des logiciels ainsi que les outils nécessaires pour la gestion des versions des logiciels ;
- Acquérir aux étudiants les préalables et le processus de livraison d'un produit logiciel.

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 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.
- ❖ Le **génie logiciel**, l'**ingénierie logicielle** ou l'**ingénierie du logiciel** (en [anglais](#) : *software engineering*) est une science de [génie industriel](#) qui étudie les méthodes de travail et les bonnes pratiques des ingénieurs qui [développent des logiciels](#). Le génie logiciel s'intéresse en particulier aux procédures systématiques qui permettent d'arriver à ce que des logiciels de grande taille correspondent aux attentes du client, soient fiables, aient un coût d'entretien réduit et de bonnes performances tout en respectant les délais et les coûts de construction¹

Selon l'arrêté ministériel du 30 décembre 1983 relatif à l'enrichissement du vocabulaire de l'informatique [*Journal officiel* du 19 février 1984], le génie logiciel est « l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi ».

Est aussi appelée génie logiciel l'[ingénierie](#) appliquée au logiciel informatique, c'est-à-dire l'activité par laquelle le [code source](#) d'un [logiciel](#) est spécifié puis produit et déployé. Le génie logiciel touche au [cycle de vie des logiciels](#). Toutes les phases de la création d'un logiciel informatique y sont enseignées : l'analyse du [besoin](#), l'élaboration des [spécifications](#), la conceptualisation du mécanisme interne au logiciel ainsi que les techniques de programmation, le [développement](#), la phase de [test](#) et finalement la [maintenance](#).

HISTOIRE

Le terme anglais « *Software* » est utilisé la première fois en 1958 par le statisticien [John Tukey](#)². Les premières bases du *génie logiciel*, en anglais « *software engineering* », sont attribuées à l'[informaticienne](#) et [mathématicienne](#) [Margaret Hamilton](#), la conceptrice du [système embarqué](#) du [Programme Apollo](#)^{3,4}.

« Quand j'ai proposé la première fois le terme, personne n'en avait entendu parler auparavant, du moins dans notre monde [*Ndt: à la NASA*]. Pendant assez longtemps ils aimaient me charrier sur mes idées radicales. Ce fut une journée mémorable lorsque l'un des gourous les plus respectés dans le domaine du hardware a expliqué lors d'une réunion qu'il était d'accord avec moi pour dire que le processus de construction d'un logiciel devrait également être considéré comme une discipline d'ingénierie, au même titre que le matériel électronique. Pas à cause de son acceptation du nouveau terme en tant que tel, mais parce que nous avons gagné son approbation et celle des

¹ Marylène Micheloud et Medard Rieder, *Programmation orientée objets en C++: Une approche évolutive*, PPUR presses polytechniques, 2002, ([ISBN 9782880745042](#))

² Bourque, Pierre, Fairley, Richard E. et IEEE Computer Society, *Guide to the software engineering body of knowledge (SWEBOK)* ([ISBN 978-0-7695-5166-1](#), [OCLC 1100623800](#), [lire en ligne](#) [\[archive\]](#))

autres personnes présentes dans la salle comme œuvrant dans un domaine d'ingénierie à part entière.³ »

Le terme « *software engineering* » a été mentionné publiquement pour la première fois en 1968⁴ pour une conférence organisée par l'OTAN sur le sujet⁵. Il a été repris l'année suivante à une conférence concernant la *crise du logiciel*. La crise du logiciel est une baisse significative de la qualité des logiciels dont la venue coïncide avec le début de l'utilisation des [circuits intégrés](#) dans les ordinateurs: l'augmentation de la puissance de calcul des ordinateurs a permis de réaliser des logiciels beaucoup plus complexes qu'auparavant. En 1972, [l'IEEE](#) lance un premier périodique, « *Transactions on Software Engineering* », consacrant ainsi cette discipline émergente de l'ingénierie².

Les premières tentatives de création de logiciels de grande ampleur ont vite montré les limites d'un travail informel d'ingénieurs logiciel : les produits réalisés ne sont pas terminés dans les temps, coûtent plus cher que prévu, ne sont pas fiables, sont peu performants et coûtent cher en entretien. La baisse du coût du matériel informatique s'accompagnait d'une augmentation du coût du logiciel. Des études se sont penchées sur la recherche de méthodes de travail adaptées à la complexité inhérente aux logiciels contemporains et ont donné naissance au *génie logiciel*⁶.

Aujourd'hui (en 2004), l'utilisation des méthodes de génie logiciel reste quelque chose de relativement peu répandu dans l'industrie du logiciel. Le programmeur travaille souvent comme un [artisan](#), guidé par son talent, son expérience et ses connaissances théoriques et la *crise du logiciel* s'apparente à une maladie chronique de l'industrie du logiciel⁷.

³ Snyder, Lawrence,, *Fluency with information technology : skills, concepts, & capabilities*, 2017, 816 p. ([ISBN 978-0-13-444872-5](#), [0134448723](#) et [9780133577396](#), [OCLC 960641978](#), [lire en ligne](#) [\[archive\]](#))

⁴ Wohlin, Claes, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén.

Experimentation in software engineering. Springer Science & Business Media, 2012, page 3 ([ISBN 9783642290435](#))

⁵ « [NATO Software Engineering Conference 1968](#) » [\[archive\]](#), sur [homepages.cs.ncl.ac.uk](#) (consulté le 2 juin 2019)

⁶ Ian Sommerville, *Software engineering, International computer science series*, Pearson Education, 2001, ([ISBN 9780321313799](#))

⁷ Pankaj Sharma, *Software Engineering - Volume 2*, APH Publishing, 2004, ([ISBN 9788176485401](#))

CHAPITRE 0 : GENERALITES SUR LE GENIE (L'INGENIERIE) LOGICIELLE

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 fait, 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.

0.1. DEFINITION ET CONTEXTE D'ETUDES

IEEE⁸ 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.

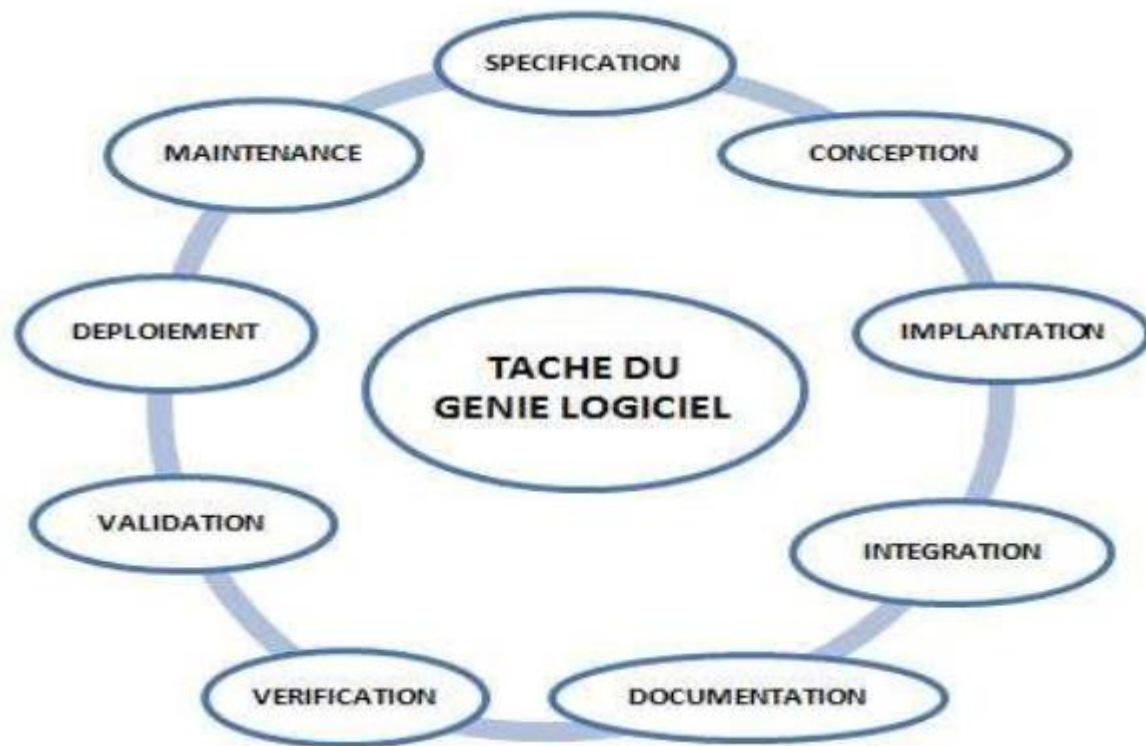
Fritz Bauer, un informaticien allemand, définit le génie logiciel comme l'établissement et l'utilisation de principes d'ingénierie afin d'obtenir des logiciels économiques, fiables et efficaces sur des machines réelles. Dans le cadre de ce cours, le génie logiciel sera considéré comme étant un ensemble 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 :

⁸ L'IEEE, Institute of Electrical and Electronics Engineers (Institut des ingénieurs électriciens et électroniciens, en français), est une association regroupant des milliers de professionnels du domaine de l'informatique et des télécommunications à travers le monde



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

0.2. 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.

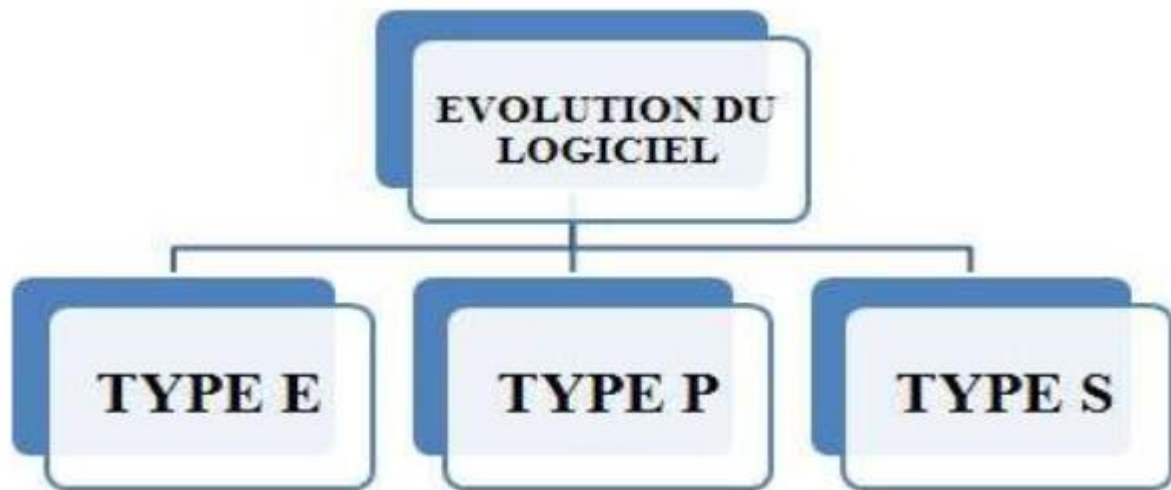


L'évolution commence par le processus de collecte des exigences. Après quoi les développeurs créent un prototype du logiciel prévu et le montrent aux utilisateurs pour obtenir leurs commentaires dès les premières étapes du développement de logiciels. Les utilisateurs suggèrent des modifications sur lesquelles 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.

0.2.1. 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.

0.2.2. EVOLUTION DU LOGICIEL E-TYPE

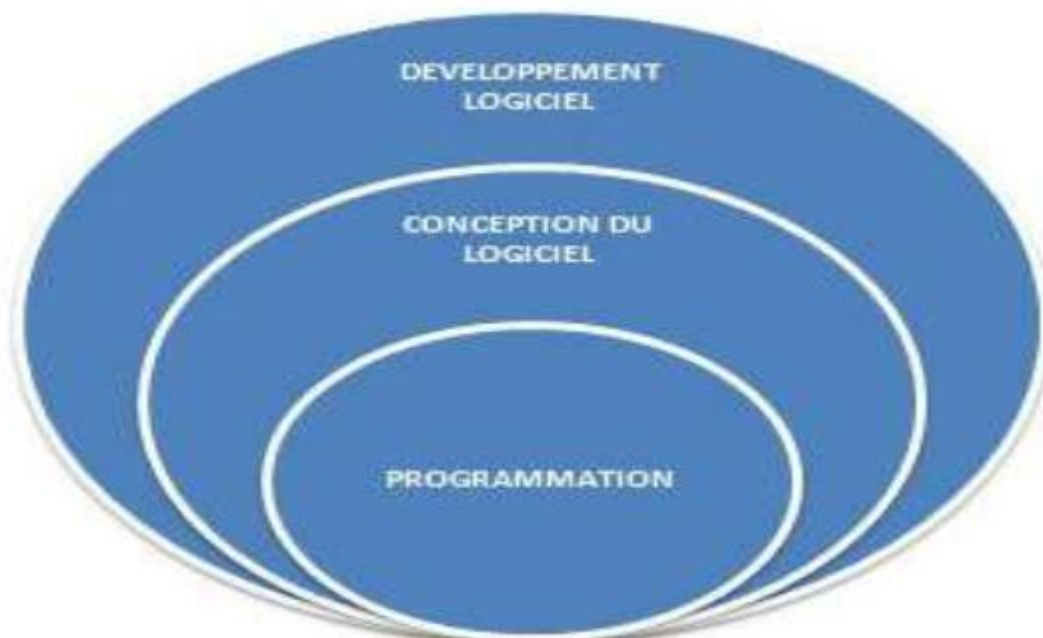
Meir Lehmana donné huit lois pour l'évolution des logiciels de type E:

- 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.

- iv) **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.
- v) **Réduire la qualité** : Un système logiciel de type E diminue en qualité, sauf si rigoureusement entretenu et adapté à un environnement opérationnel changeant.
- vi) **Systèmes de rétroaction** : C'est un processus du logiciel pouvant se déclencher automatiquement après une opération défailante, 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.
- vii) **Autorégulation** : Les processus d'évolution du système de type E s'autorégulent, la distribution des mesures du produit et du procédé étant presque normale.
- viii) **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.

0.3. 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.

0.3.1. 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.

0.3.2. PARADIGME DE CONCEPTION

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

0.3.3. 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.

0.4. 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 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.

0.5. 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 être évalué 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.

0.5.1. 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.

0.5.2. CRITERES EXTERNES

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

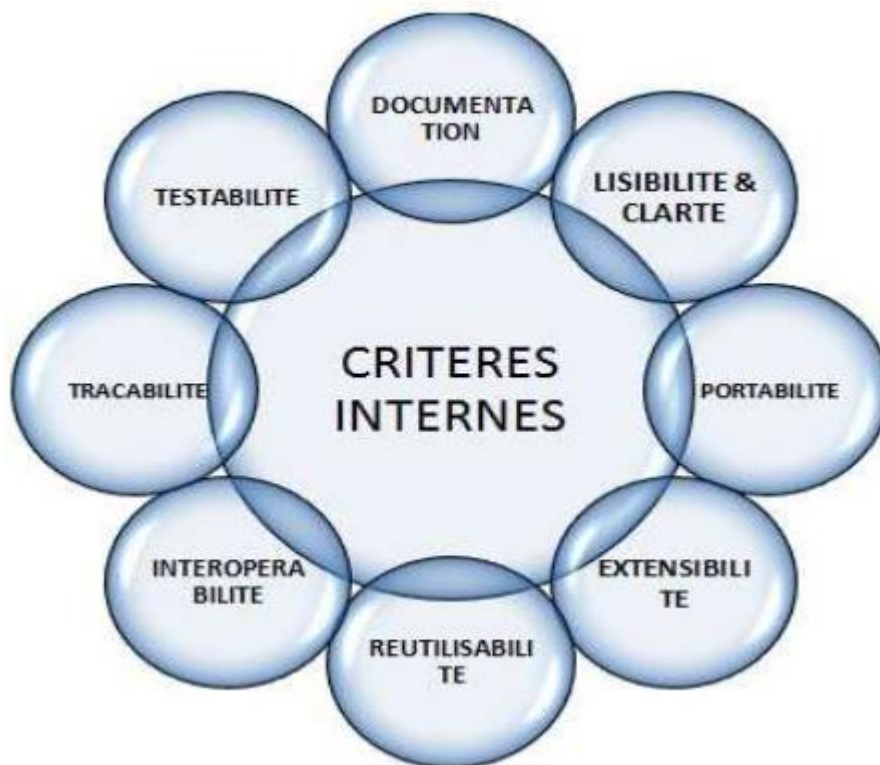


- ✓ **Fiabilité:** (correction, justesse et conformité): le logiciel est conforme à ses spécifications, les résultats sont ceux attendus.
- ✓ **Robustesse et Sureté :**(dysfonctionnements ou ne plante pas): le logiciel fonctionne raisonnablement en toutes circonstances, rien de catastrophique ne peut 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:** 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.

0.5.3. 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.

0.6. CATEGORIES DE LOGICIELS

Il existe 2 catégories des logiciels notamment:

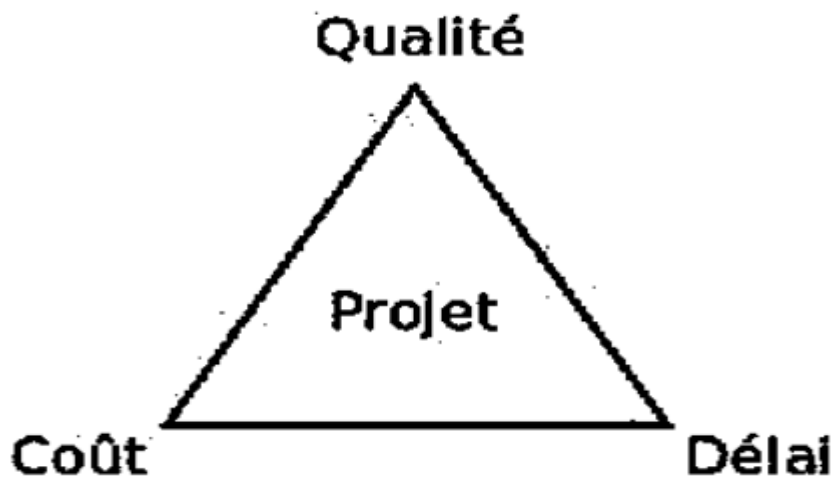
- **LOGICIELS GENERIQUES:** Ce sont des logiciels qui sont vendus comme les produits courants sur le marché informatique. Dans cette catégorie, on en distingue autant:
 - **Logiciels amateurs:** Il s'agit de logiciels développés par des « amateurs » (par exemple par des gens passionnés ou des étudiants 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.

0.7. 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.

0.8. 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.
- ✓ ...

0.9. CYCLE DE VIE D'UN LOGICIEL

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

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

On distingue différents modèles de cycle de vie d'un logiciel. Dans ce cours on va parler de ces 3 modèles couramment utilisés:

- Cycle de vie en Cascade;
- Cycle de vie en V;
- Cycle de vie en Spiral.

a) Le modèle en cascade (waterfall model)

On appelle modèle en cascade un modèle de gestion séquentiel permettant de représenter les développements à travers des phases successives.

1^o Qu'est-ce que le modèle en cascade ?

Le modèle en cascade (en anglais : *waterfall model*) est un **modèle de gestion linéaire** qui divise les processus de développement en phases de projet successives. Contrairement aux modèles itératifs, chaque phase est effectuée une seule fois. Les sorties de chaque phase antérieure sont intégrées comme entrées de la phase suivante. Le modèle en cascade est principalement utilisé dans le développement de logiciels.

2^o Comment fonctionne le modèle en cascade ?

On doit le développement du modèle en cascade classique à l'informaticien Winston Walker Royce. Royce n'en est toutefois pas l'inventeur. En effet, son essai publié en 1970 sous le titre « [Managing the Development of Large Software Systems](#) » présente plutôt une **analyse critique des modèles linéaires**. Royce proposait comme alternative un modèle itératif et incrémental dans lequel chaque phase reposerait sur la précédente et en vérifierait les résultats.

Il recommandait un modèle en sept phases qui se déroulaient en plusieurs étapes (itérations) :

1. **Exigences système**
2. **Exigences logicielles**
3. **Analyse**
4. **Conception**
5. **Implémentation**

6. Test

7. Exploitation

Le modèle de gestion que l'on appelle modèle en cascade est fondé sur les phases définies par Royce, **mais prévoit une seule itération**.

Dans son essai, Royce n'évoque pas une seule fois le terme cascade (*waterfall*).

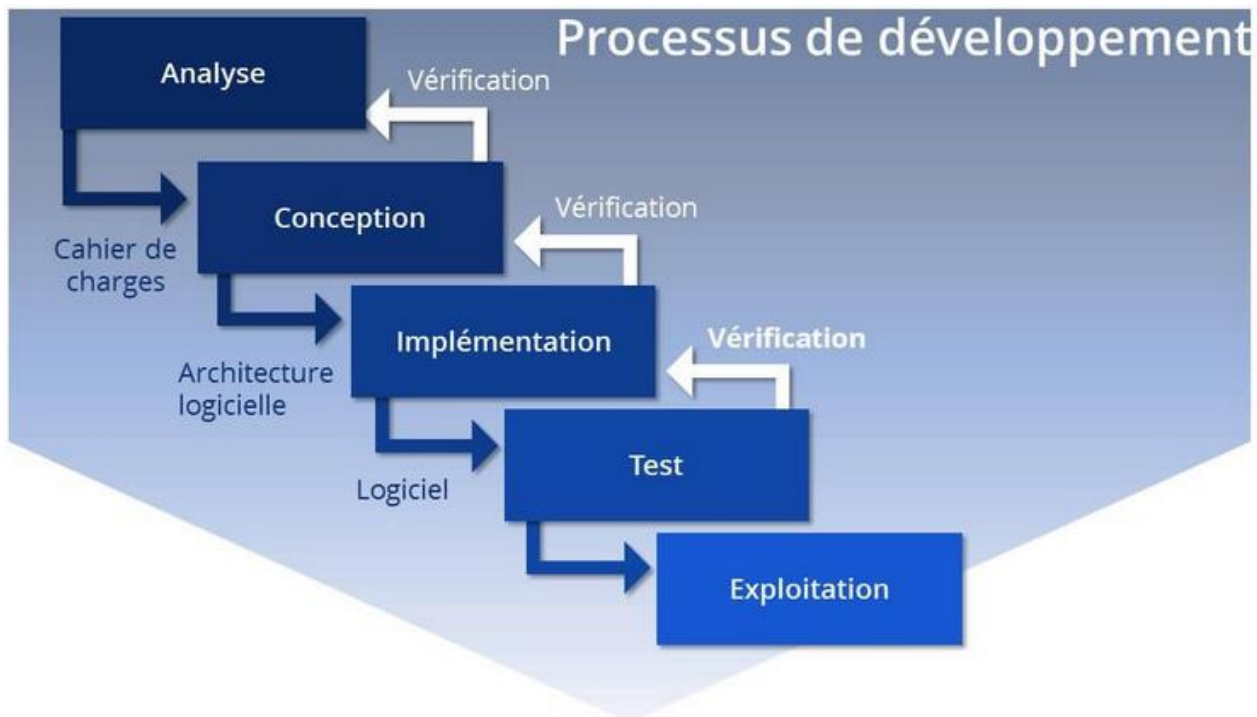
Remarque :

Le modèle en cascade doit sa grande notoriété au standard américain DoD-STD-2167. Ce dernier repose sur une forme extrêmement simplifiée du modèle de gestion développé par Royce, qui n'avait pas été suffisamment compris par les auteurs. Comme David Maibor – l'un des auteurs du standard – le concéda des années plus tard, cette erreur était due à une incompréhension des modèles itératifs et incrémentaux.

En pratique, **plusieurs versions du modèle en cascade** sont utilisées. Les modèles les plus courants divisent les processus de développement en cinq phases. Les phases 1, 2 et 3 définies par Royce sont parfois regroupées en une seule et même phase, qualifiée d'analyse des besoins.

1. **Analyse** : planification, analyse et spécification des besoins
2. **Conception** : conception et spécification du système
3. **Implémentation** : programmation et tests des modules
4. **Test** : intégration du système, tests du système et de l'intégration
5. **Exploitation** : livraison, maintenance, amélioration

Le schéma suivant illustre pourquoi le modèle de gestion linéaire est qualifié de « modèle en cascade ».



Le modèle en cascade reposant sur les exigences de Winston Walter Royce divise les processus de développement en cinq phases de projet, qui sont les suivantes : analyse, conception,

implémentation, test et exploitation. Le schéma présente déjà l'une des extensions du modèle recommandé par Royce : la vérification des résultats de chaque phase en tenant compte des exigences et des spécifications élaborées au préalable.

Des extensions du modèle en cascade simple viennent compléter le modèle de base avec des fonctions itératives, par exemple avec des retours en arrière permettant de comparer les résultats de chaque phase avec les hypothèses tirées de la phase précédente et ainsi de les vérifier.

3⁰ Les phases du modèle en cascade

Dans le modèle en cascade, les différentes phases d'un processus de développement s'enchaînent. Chaque phase se termine par un **résultat intermédiaire (étape)**, par exemple avec un catalogue d'exigences sous la forme d'un cahier des charges, avec la spécification d'une architecture logicielle ou avec une application au stade alpha ou bêta.

➤ Analyse

Chaque projet logiciel commence par une phase d'analyse comprenant une étude de faisabilité et une définition des besoins. Les coûts, le rendement et la faisabilité du projet logiciel sont estimés lors de l'**étude de faisabilité**. Celle-ci permet de créer un cahier des charges (une description grossière des besoins), un plan de projet, une budgétisation du projet et, le cas échéant, un devis pour le client.

Les **besoins sont ensuite définis de façon détaillée**. Cette définition comprend une analyse réelle et un concept cible. Alors que les analyses réelles décrivent les problèmes, le concept cible permet de définir quelles fonctionnalités et quelles propriétés le produit logiciel doit offrir afin de répondre aux besoins. La définition des besoins permet notamment d'obtenir un cahier des charges, une description détaillée de la façon dont les exigences du projet doivent être remplies ainsi qu'un plan pour le test d'acceptation.

Enfin, la première phase du modèle en cascade prévoit une **analyse de la définition des besoins**, au cours de laquelle les problèmes complexes sont décomposés en sous-tâches de moindre ampleur et des stratégies de résolution correspondantes sont élaborées.

➤ Conception

La phase de conception sert à l'élaboration d'un concept de résolution concret sur la base des besoins, des tâches et des stratégies déterminées au préalable. Au cours de cette phase, les développeurs élaborent l'**architecture logicielle** ainsi qu'un **plan de construction détaillé du logiciel** et se concentrent ainsi sur les éléments concrets tels que les interfaces, les frameworks ou les bibliothèques. Le résultat de la phase de conception inclut un document de conception avec un plan de construction logicielle, ainsi que des plans de test pour les différents éléments.

➤ Implémentation

L'architecture logicielle élaborée pendant la phase de conception est réalisée lors de la **phase d'implémentation** qui comprend la **programmation du logiciel**, la **recherche d'erreurs** et les **tests de modules**. Lors de la phase d'implémentation, le projet de logiciel est transposé dans la langue de programmation souhaitée. Les différents composants logiciels sont développés séparément, contrôlés dans le cadre de **tests de modules** et intégrés étape par étape dans le

produit global. Le résultat de la phase d'implémentation est un produit logiciel qui sera testé pour la première fois en tant que produit global lors de la phase suivante (test alpha).

➤ Test

La phase de test comprend l'intégration du logiciel dans l'environnement cible souhaité. En règle générale, les produits logiciels sont tout d'abord livrés à une sélection d'utilisateurs finaux sous la forme d'une **version bêta** (bêta-tests). Il est alors déterminé si le logiciel répond aux besoins préalablement définis à l'aide des **tests d'acceptation** développés lors de la phase d'analyse. Un produit logiciel ayant passé avec succès les bêta-tests est prêt pour la mise à disposition.

Après avoir réussi la phase de tests, le logiciel est **mis en production** pour exploitation. La dernière phase du modèle en cascade inclut la **livraison**, la **maintenance** et l'**amélioration du logiciel**.

4⁰ Évaluation du modèle en cascade

Le modèle en cascade offre une **structure hiérarchique claire** pour les projets de développement dans laquelle les différentes phases du projet sont clairement délimitées. Étant donné que chaque phase se termine par une étape, le processus de développement peut être suivi facilement. Le point fort du modèle se trouve dans la documentation des étapes du processus. Les connaissances acquises sont consignées dans les documents d'exigences et de conception.

En théorie, le modèle en cascade doit créer les conditions pour une réalisation rapide et peu coûteuse des projets par une planification minutieusement élaborée au préalable. Toutefois, **les avantages du modèle en cascade sont sujets à controverse dans la pratique**. D'une part, les phases du projet sont rarement délimitées clairement dans le développement logiciel. Pour les projets logiciels complexes notamment, les développeurs sont souvent confrontés au fait que les différents composants d'une application se trouvent dans différentes phases de développement au même moment. D'autre part, le déroulement linéaire du modèle en cascade ne correspond souvent pas aux conditions réelles.

Le modèle en cascade ne prévoit pas à proprement parler des adaptations en cours de projet. Un projet de logiciel, dans lequel l'ensemble des détails du développement sont déjà définis au début du projet, peut toutefois uniquement aboutir lorsque beaucoup de temps et d'argent ont été investis dès le départ dans l'analyse et la conception. S'y ajoute le fait que les projets logiciels plus vastes s'étendent parfois sur plusieurs années et sans un ajustement régulier aux développements actuels, ils fourniraient des résultats **déjà obsolètes lors de leur introduction**.

5⁰ Vue d'ensemble des avantages et inconvénients du modèle en cascade

Avantages	Inconvénients
Une structure simple grâce à des phases de projet clairement délimitées.	Les projets complexes ou à plusieurs niveaux ne peuvent que rarement être divisés en phases de projet clairement définies.
Une bonne documentation du processus de développement par des étapes clairement définies.	Une faible marge pour les ajustements du déroulement du projet en raison d'exigences modifiées.
Les coûts et la charge de travail peuvent être estimés dès le début du projet.	L'utilisateur final est uniquement intégré dans le processus de production après la programmation.

Avantages	Inconvénients
Les projets structurés d'après le modèle en cascade peuvent être représentés facilement sur un axe temporel.	Les erreurs sont parfois détectées uniquement à la fin du processus de développement.

Le modèle en cascade est principalement utilisé dans les projets pour lesquels les besoins et les processus peuvent être définis de façon précise dès la phase de planification et pour lesquels on peut supposer que les hypothèses changeront peu tout au long du déroulement du projet. Les modèles de gestion strictement linéaires conviennent ainsi principalement aux **projets logiciels plus petits, simples et clairement structurés**. Royce était déjà parvenu à cette conclusion dans les années 1970. L'alternative au modèle de gestion linéaire qu'il proposait, plus tard connue sous le nom de modèle en cascade, comprenait par conséquent trois extensions essentielles :

❖ Une vérification après chaque phase de projet

D'après Royce, les résultats de chaque phase de projet doivent être comparés et vérifiés immédiatement à l'aide des documents élaborés au préalable. Il serait par exemple nécessaire de contrôler qu'un module répond aux exigences définies au préalable immédiatement après son développement et non uniquement à la fin du processus de développement.

❖ Au moins deux itérations

Selon Royce, le modèle en cascade devrait être effectué au minimum à deux reprises : une première fois pour le **développement d'un prototype** et une seconde pour le **développement du produit logiciel à proprement parler**.

❖ Des tests intégrant l'utilisateur final

Comme troisième extension du modèle en cascade, Royce recommandait dans son essai une mesure qui fait aujourd'hui partie de la procédure standard dans le développement de produit : l'intégration de l'utilisateur final dans le processus de production. Royce proposait d'intégrer l'utilisateur à trois reprises dans le processus de développement logiciel : lors de la planification du logiciel dans le cadre de la phase d'analyse, entre la conception du logiciel et son implémentation et lors de la phase de test avant la mise à disposition du logiciel.

6⁰ Alternatives au modèle en cascade

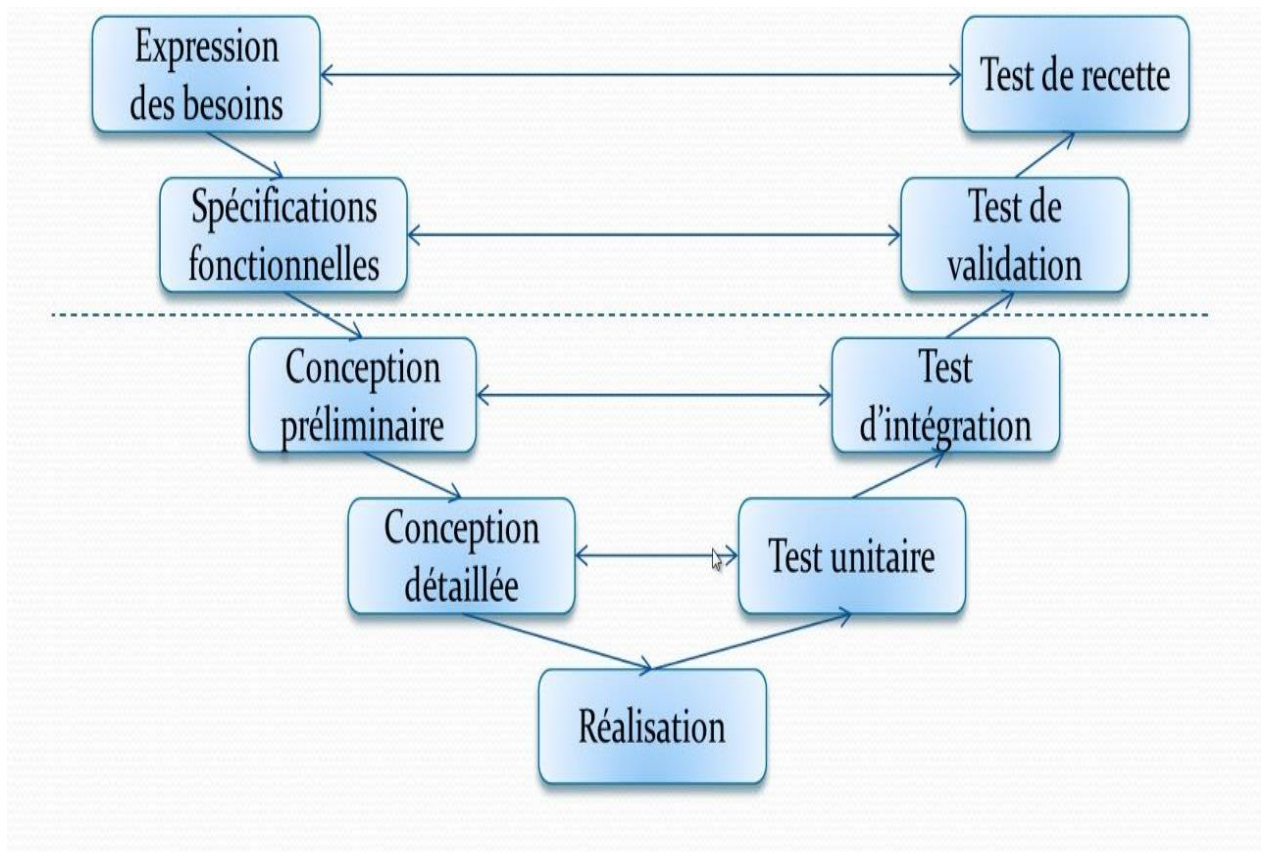
En raison du déroulement strictement linéaire des phases de projet qui se suivent de façon séquentielle, le modèle en cascade convient uniquement à de petits projets logiciels, si tant est qu'il convienne à un quelconque projet. En revanche, les processus complexes et à plusieurs niveaux ne sauraient être représentés avec ce modèle. C'est la raison pour laquelle différentes approches alternatives ont été développées au fil du temps.

Alors que des modèles comme le modèle en spirale ou le modèle du cycle en V sont considérés comme **des améliorations du modèle en cascade classique**, des concepts tels que l'« extreme programming », le développement logiciel agile ou le prototypage itératif reposent sur une tout autre approche et permettent généralement une adaptation flexible aux modifications actuelles et aux nouveaux besoins.

b) Le modèle en V

Le cycle en V ou V model en anglais est un modèle utilisé dans différents processus de développement, notamment dans le **développement de logiciels**. Élaboré dans les années 90 sous sa forme originale, il est perfectionné au fil des ans et adapté aux méthodes de développement contemporaines. L'idée de départ remonte cependant aux années 70 et a été imaginée comme une sorte de prolongement du [modèle en cascade](#).

Outre les différentes phases de développement d'un projet, le cycle en V définit parallèlement les démarches afférentes à mettre en place en termes d'assurance qualité et détaille la façon dont les différentes phases doivent interagir entre elles. Le cycle en V doit son nom à **sa forme qui rappelle la lettre V**.



1^o Les différentes phases du cycle en V

Dans un premier temps, le cycle en V définit le **déroulement d'un projet** en phases distinctes qui sont tour à tour détaillées :

- En début de projet, le modèle prévoit une analyse de l'ensemble des besoins relatifs au système envisagé.
- Le projet est ensuite enrichi par l'expression des besoins fonctionnels et non fonctionnels liés à l'architecture du système.
- Puis on passe à la phase de conception du système, lors de laquelle les composants et les interfaces du système sont planifiés.
- Une fois ces étapes franchies, on peut passer à la conception de l'architecture logicielle en détail.

On entre alors dans la phase effective de développement du logiciel en fonction de ce qui a été planifié. Ensuite ce sont les phases d'**assurance qualité**, qui se réfèrent toujours aux étapes du développement. Le modèle prévoit les tâches suivantes :

- Tests unitaires ;
- Tests d'intégration ;
- Intégration système ;
- La « recette » (ou test d'acceptation).

2⁰ Interaction entre conception et assurance qualité

La lettre « V » provient du fait que le modèle **associe chaque phase de développement avec la phase de validation qui lui correspond**. Le bras gauche du « V » désigne les tâches relatives à la conception et au développement du système, le bras droit les mesures d'assurance qualité qui lui sont associées. La phase de mise en œuvre du produit se trouve au centre des deux bras, entre les phases de développement et d'assurance qualité. Dans le cas d'un projet informatique, on parlera plus volontiers de programmation logicielle.

Les **tests unitaires** permettent de vérifier la bonne mise en œuvre de l'**architecture logicielle** prévue. Lors de cette étape, on vérifie si les différents modules du logiciel répondent en tous points aux fonctionnalités requises et s'ils sont bien conformes aux résultats attendus. Dans l'idéal, ces modules de tests devraient être réalisés si possible parallèlement à la conception, afin d'éviter les erreurs.

La **conception du système** est soumise à des **tests d'intégration**. Il s'agit ici de vérifier si les différents composants interagissent comme prévu, et si par exemple, tous les processus génèrent les résultats attendus. À ce stade, des résultats non conformes peuvent mettre en avant des problèmes liés à certaines étapes.

Le **test système** vérifie si l'ensemble des exigences exprimées lors de la conception de l'**architecture système** ont été respectées. En règle générale, les simulations ont lieu dans un environnement qui reproduit aussi fidèlement que possible les situations rencontrées par le client.

En fin projet, l'**analyse des besoins** de l'ensemble du système est comparée avec la réception du produit fini. Lors de la **réception définitive**, le client s'assure que ses exigences ont bien été prises en compte durant le fonctionnement. En règle générale, on teste uniquement le fonctionnement du logiciel en surface, autrement dit, on teste ce que le client peut voir au quotidien lorsqu'il utilise son logiciel. On parle ici également de test de recette.

3⁰ Le V modell XT : la poursuite du développement du cycle en V

La dernière modification du cycle en V date de 2006. L'objectif était de pouvoir intégrer de nouvelles démarches comme le [développement agile](#). Cela a donné naissance à la méthodologie « V modell XT ». XT signifie ici « **extreme tailoring** » (« flexibilité extrême »). Cette approche explique comment modéliser le modèle sur mesure en fonction des différentes exigences liées au projet.

En continuant à développer ce système, l'idée de base était de fournir un modèle qui puisse **s'adapter facilement à des projets d'envergures différentes**. En effet, l'ancienne méthode s'est avérée extrêmement coûteuse, notamment pour les petits projets et donc inefficace. Dans le cas de petits projets, le V modell XT permet de supprimer certaines phases qui nécessiteraient un investissement trop élevé.

De plus, le nouveau modèle intègre des blocs de tâches qui font directement référence au client. L'ancien modèle se contentait de faire piloter le projet par le prestataire avant la réception définitive. Le nouveau modèle **implique le client de manière beaucoup plus directe**.

Le modèle est mis à jour régulièrement afin de prendre en compte les nouveautés en termes de processus de développement logiciel et d'améliorer la prise en main. La version 2.3 est la version actuelle du V modél XT.

4⁰ Domaines d'application du cycle en V

Le V modél XT est une **approche très répandue** dans le secteur industriel. Le recours au cycle en V est presque même devenu la norme pour la plupart des appels d'offres des marchés publics relatifs à des projets informatiques. Par conséquent, c'est un paramètre très important pour les entreprises qui conçoivent des logiciels pour les pouvoirs publics et les ministères. Il convient aux projets informatiques de toute taille, que ce soit au niveau des entreprises, de l'armée ou du secteur public. C'est un outil qui permet de faciliter l'organisation et la réalisation liées au développement, à la maintenance et au développement continu des différents systèmes d'information.

Le cycle en V peut également être utilisé dans d'autres secteurs de développement, comme par exemple les systèmes électroniques ou mécaniques dans les domaines de la recherche et des sciences. Pour ces domaines d'application, il existe des **versions légèrement modifiées** qui tiennent compte des stades spécifiques propres à chaque secteur.

5⁰ Avantages et inconvénients du cycle en V

Cette démarche est largement répandue, car elle garantit avant tout un degré élevé de transparence ainsi que des processus clairement définis et compréhensibles. Vous trouverez ci-après un récapitulatif des avantages et des éléments de critique les plus importants.

➤ Les avantages du cycle en V

- Optimisation de la communication entre les parties prenantes grâce à des modalités et des responsabilités clairement définies.
- Risques maîtrisés et meilleure planification grâce à des fonctions, des structures et des résultats bien définis en amont.
- Amélioration de la qualité du produit grâce à l'intégration de mesures liées à l'assurance qualité.
- Réduction des coûts grâce à un processus transparent de l'ensemble du cycle de vie du produit.
- Dans l'ensemble, ce modèle peut permettre d'éviter **les malentendus ainsi que les tâches inutiles**. De plus, il permet de s'assurer que toutes les tâches soient exécutées en temps voulu, dans le bon ordre en réduisant les temps morts au maximum.

➤ Les inconvénients du cycle en V

Du point de vue des développeurs, cette démarche s'avère souvent trop simpliste, car elle ne reflète pas intégralement le processus de développement. **La gestion de projet** occupe une place de plus en plus importante. En outre, la rigidité relative de la structure ne permet guère de réagir avec souplesse aux modifications en cours de développement et favorise donc un déroulement relativement linéaire du projet. Pourtant, il est possible de pratiquer la méthode agile avec le cycle en V, si le modèle a bien été compris et est correctement utilisé.

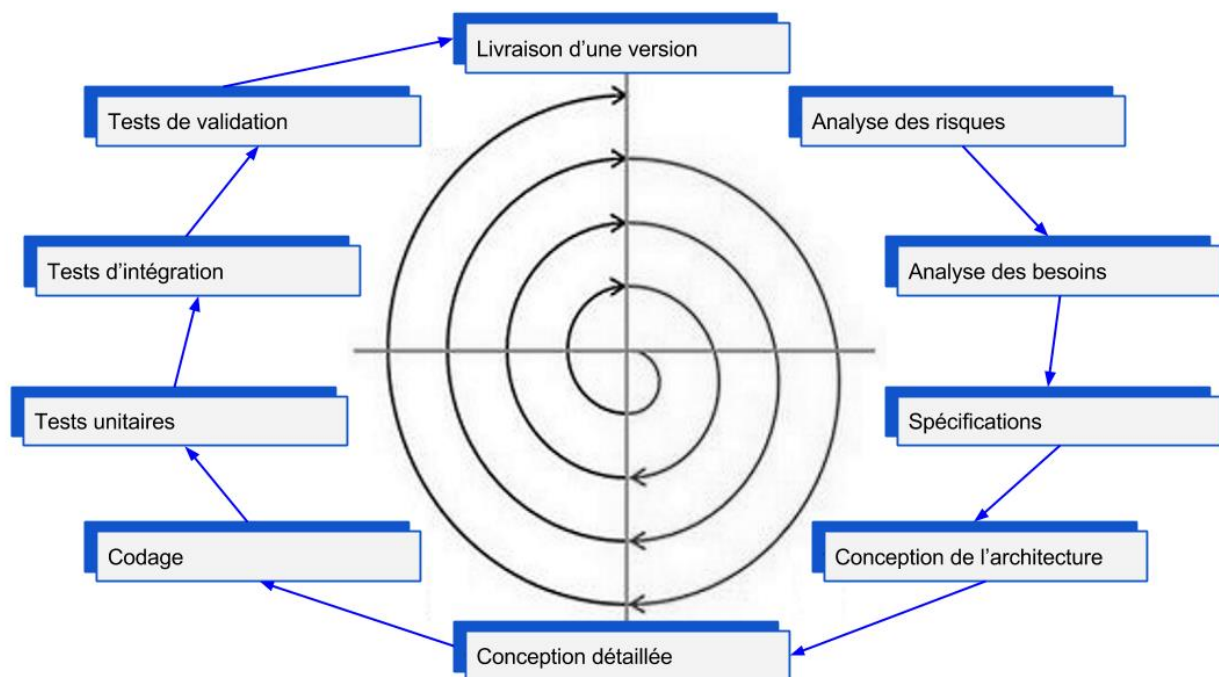
6⁰ Les alternatives au cycle en V

Il existe différents types d'approches propres au secteur du développement de logiciels qui peuvent convenir en fonction des projets et de la structure des équipes. Le choix des approches est relativement large, même si le modèle en cascade ou **le modèle en spirale** sont particulièrement répandus. Le modèle en cascade est particulièrement adapté aux projets de petite envergure, à caractère linéaire, tandis que le modèle en spirale se prête plutôt à des projets itératifs.

c) Modèle en spirale

Le modèle en spirale est une approche du développement logiciel qui peut être considérée comme une réponse aux inconvénients du [modèle en cascade](#). Le modèle en spirale décrit le cycle de développement d'un logiciel au moyen de spirales, qui sont répétées jusqu'à ce que le produit fini puisse être livré. Le modèle en spirale est également appelé modèle incrémental. Le produit est continuellement travaillé et les améliorations se déroulent souvent en très petites étapes.

Une caractéristique clé du modèle en spirale est la minimisation des risques dans le développement logiciel, ce qui peut entraîner une augmentation des coûts globaux, ainsi que plus d'efforts et un lancement différé. Ces risques sont contrôlés par l'approche progressive en réalisant d'abord des prototypes, qui sont répétés dans les spirales ou les cycles de développement de logiciels au moins une fois. Le modèle en spirale est également générique et peut être combiné avec d'autres méthodes de développement [agile](#) classiques, c'est pourquoi il est également appelé modèle de second ordre.



1⁰ Informations générales

Le modèle en spirale a été présenté par Barry W. Boehm dans son essai *A Model of Software Development and Enhancement*. À l'heure actuelle, le *waterfall model* ou modèle en cascade et les inconvénients associés ont été fréquemment discutés. Contrairement à d'autres modèles tels

que le "code and fix" ou le modèle en cascade, ce modèle-là s'axe sur les risques. L'identification et la résolution des risques jouent un rôle important dans les différents cycles du projet après la définition des objectifs et des conditions. L'accent est mis sur les facteurs possibles qui peuvent causer des incertitudes pour le logiciel ou l'ensemble du projet. Si les risques peuvent être contrôlés de manière rentable, rien n'empêche la réussite du projet selon l'hypothèse. Les approches pour minimiser ces risques sont, par exemple, les prototypes, les simulations, les tests de référence ou les entretiens avec les utilisateurs. Pour certains types de risques, toute la procédure peut même être révisée et différemment structurée. Des interventions dans la gestion sont possibles à chaque cycle du projet et d'autres approches de développement peuvent être adaptées.

2⁰ Comment ça fonctionne

Le modèle en spirale se caractérise par les cycles suivants (aussi appelés quadrants) :^[1]

- **objectif et décision alternative** : les objectifs sont déterminés conjointement avec le client. Dans le même temps, les alternatives possibles sont discutées et les conditions cadres sont spécifiées (par exemple le système d'exploitation, l'environnement et langage de programmation).
- **analyse et évaluation des risques** : les risques potentiels sont identifiés et évalués. Les alternatives en question sont également évaluées, tandis que les risques sont enregistrés, estimés puis réduits à l'aide de prototypes, des simulations et des logiciels d'analyse. Dans ce cycle, plusieurs prototypes existent sous forme de modèles de conception ou de composants fonctionnels.
- **développement et test** : les prototypes sont encore plus étendus et des fonctionnalités sont ajoutées. Le code réel est écrit, testé et migré vers un environnement de test plusieurs fois jusqu'à ce que le logiciel puisse être implémenté dans un environnement productif.
- **planification du cycle suivant** : le cycle à venir est planifié à la fin de chaque cycle. Si des erreurs se produisent, les solutions sont recherchées. Si une meilleure alternative est une solution envisageable, elle sera préférée au sein du cycle suivant.

La force motrice la plus importante du modèle en spirale est l'analyse et l'évaluation des risques. Tout risque qui menace le projet est censé être identifié dès le début. L'avancement du projet dépend de manière décisive de la manière dont les risques peuvent être limités ou supprimés. Le projet ne sera considéré comme réussi quand il n'y aura plus de risque. Le but du cycle est de produire un produit qui est en amélioration continue. Le logiciel ou l'application est constamment affiné. Le modèle en spirale est incrémental, mais pas nécessairement répétitif. Les répétitions ne se produisent que lorsque des risques, des erreurs ou des conflits menacent le projet. Ensuite seulement, le produit passera à un cycle suivant, ce qu'on appellera une répétition ou une itération.

3⁰ Avantages/inconvénients

- Le modèle en spirale est souvent utilisé pour des projets plus importants qui sont soumis à des risques. Comme ces risques ont un impact monétaire direct, le contrôle des budgets pour les clients et les entreprises de développeurs est central. Le modèle en spirale est surtout utilisé dans les nouveaux environnements techniques, car le risque est réel.

- Les conflits entre les prérequis d'un logiciel et sa conception sont effectivement évités grâce à l'approche cyclique : ces prérequis peuvent être constamment vérifiés et si nécessaire, modifiés.
- Retours et commentaires peuvent être obtenus auprès des utilisateurs, des développeurs et des clients lors des premières phases du projet. Cependant, cette structure nécessite également une gestion qui prend en compte les cycles du produit et peut donc répondre rapidement aux risques. Le contrôle de ces projets est donc relativement complexe et nécessite également une bonne documentation pour que tous les changements soient enregistrés. ^[2]
- Bien que le logiciel soit testé sous divers aspects pendant le cycle de développement logiciel et du test (test des unités, d'acceptation et d'intégration), il arrive souvent que les prototypes soient transférés dans le système de production. Il existe donc un risque que d'autres erreurs et des incohérences conceptuelles soient saisies dans le produit final.
- Dans les cas où des décisions doivent être prises sur les cycles suivants, il existe un risque que des boucles se forment et que le projet prenne plus de temps si de mauvais choix sont faits. Pour cette raison, alternatives et évaluations sont importantes.

4^e Importance pour la programmation

Contrairement à un modèle séquentiel (comme le modèle en cascade) disposé en phases successives, le modèle en spirale esquisse le cycle de développement d'un logiciel qui doit traverser plusieurs étapes. Cette approche ressemble donc davantage à un prototypage qu'avec les approches classiques. Le modèle en spirale est censé éviter les inconvénients, tandis que d'autres modèles mettent l'accent sur les avantages. En se concentrant sur la réduction des risques, ce modèle comporte une composante financière qui peut être pertinente pour les preneurs de décisions. Grâce aux discussions avec les clients, aux analyses et aux études de faisabilité, des projets de grande envergure peuvent être mis en œuvre et leur impact économique peut être surveillé. Dans quelle mesure les méthodes agiles comme le [scrum](#), la programmation extrême ou le DevOps sont de meilleurs choix dépendent de différents facteurs, notamment de la portée du projet, du budget ou du niveau requis de soutien et de maintenance. Plus la flexibilité est requise, plus les méthodes agiles peuvent entrer en jeu.

0.10. 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 Ghezzi⁹:

- 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, . . .
- La Généralisation:** regroupement d'un ensemble de fonctionnalités semblables en une fonctionnalité paramétrable (généricité¹⁰ et héritage)

⁹ C. Ghezzi, —Fundamentals of Software Engineering‖ Prentice Hall, 2nd edition, 2002.

- c. **La Structuration**: c'est la manière de décomposer un logiciel (utilisation d'une méthode bottom-up ou top-down). c'est la décomposition des problèmes en sous-problèmes indépendants¹¹. 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.
- d. **La modularité**¹²: 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).
- e. **L'abstraction**¹³: 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++, . . .
- f. **La construction incrémentale**: Un développement logiciel a plus de chances d'aboutir s'il suit un cheminement incrémental¹⁴ (baby-steps)
- g. **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.
- h. **La Vérification** : c'est la détermination du respect des spécifications établies sur la base des besoins identifiés dans la phase précédente du cycle de vie.

¹⁰ Un logiciel est générique lorsqu'il est adaptable.

¹¹ Edsger W. Dijkstra On the role of scientific thought.

<http://cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>

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

¹³ 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

¹⁴ 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

CHAPITRE I : ANALYSE FONCTIONNELLE D'UN PRODUIT LOGICIEL (AF)

I.1 : Présentation

D'après la norme AFNOR NF X 50-151, l'analyse fonctionnelle est une démarche qui consiste à rechercher, ordonner, caractériser, hiérarchiser et / ou valoriser les fonctions du produit attendu par l'utilisateur.

L'analyse fonctionnelle s'applique à la création ou à l'amélioration d'un produit, elle est dans ce cas l'étape fondamentale de l'analyse de la valeur.

Appliquée au seul besoin, elle est la base de l'établissement du [Cahier des Charges Fonctionnel](#) Besoin.

Elle doit permettre de **répondre aux questions** :

- Quel est le besoin auquel doit satisfaire le produit fini ?
- Quels sont les objectifs attendus ?

C'est également durant cette étape que s'identifient les principales parties prenantes du futur projet

- Première **mise en relation des intervenants** ayant une implication sur le projet ou ayant un intérêt dans le projet ;
- **Intégration des futurs utilisateurs** car les bénéficiaires peuvent être hostiles aux changements engendrés par un nouveau projet.

La structuration de tout projet en différentes étapes hiérarchiques permet d'assurer une véritable réflexion sur trois questions fondamentales que doit se poser tout porteur de projet avant tout passage à l'acte :

- Pourquoi le projet ? étape 1 : Emergence
- Quoi, que souhaite t'on réaliser ? étape 2 : Faisabilité
- Comment souhaite-t-on le réaliser ? étape 3 : Conception

Certes la recherche de réponses à ces trois questions peut sembler être une perte de temps, mais comme le disait Abraham Lincoln :

« Si j'ai six heures pour couper un arbre, j'en prends 5 pour affûter ma hache ! »

En effet, la préparation du projet permet de gagner du temps dans la phase de réalisation. L'absence de préparation, ou une faible préparation, fera systématiquement perdre du temps en phase de réalisation, comme le démontre le graphe suivant :



La période de préparation du projet inclut les trois premières phases (émergence, faisabilité et conception) alors que la phase de réalisation inclut les deux dernières étapes : réalisation et terminaison.

I.2 : Maîtrisez vos coûts grâce à l'analyse fonctionnelle

Un projet informatique est souvent au cœur de nombreuses ambitions. Afin de le rendre performant tout en optimisant son temps de développement et donc son coût, il est parfois nécessaire de faire une analyse fonctionnelle. L'[analyse fonctionnelle](#) sert à identifier vos besoins ou ceux de vos clients pour ensuite les hiérarchiser, les prioriser et associer à chaque élément un coût de réalisation cohérent.

I.3 : En quoi consiste une analyse fonctionnelle ?

L'analyse fonctionnelle d'un projet informatique est une étape qui s'avère très souvent nécessaire pour mener à bien ce dernier. Elle permet de concevoir un système pour lequel toutes les options seront parfaitement conçues, orientées vers une **satisfaction client maximale**.

Ainsi, l'analyse fonctionnelle repose sur **l'étude des besoins** liés au produit final pour l'adapter aux attentes du client mais aussi et surtout des futurs utilisateurs. En exprimant les besoins des utilisateurs sous la forme de résultats à atteindre, l'analyse fonctionnelle donne la possibilité de **mieux appréhender la conception du produit**.

I.4 : Quatre grandes étapes de l'analyse fonctionnelle selon la méthode ROCH

- **Recensement:** lister les objectifs à atteindre et les contraintes à respecter. On parle ici de « fonctions de services » pour formuler l'ensemble des éléments liés de façon directe au produit et aux fonctionnalités qu'il devra proposer ;
- **Ordonnement:** permet de structurer les fonctions de services en deux types distincts : « fonctions de finalités » et « fonctions de réalisations » afin de déterminer par quel moyen répondre à chaque besoin ;
- **Caractérisation:** identifier et quantifier les besoins de votre produit en les alliant à l'objectif de satisfaction client. Lors de cette étape on attribue à chaque fonction des critères précis et mesurables pour déterminer ensuite leur coût de réalisation ;
- **Hiérarchisation:** analyser l'importance de chaque fonction par rapport à son degré de priorité et son coût de réalisation estimé. On parle alors à ce stade d'analyse de la valeur.

I.5 : Qu'est-ce que l'analyse fonctionnelle apporte à votre projet ?

L'analyse fonctionnelle a pour but de traduire un besoin client en axant son étude sur des résultats à atteindre, et non pas sur des moyens techniques à employer. C'est cette opération de réflexion, centrée sur un angle d'attaque stratégique, qui permet de **tirer des conclusions précises et adaptées**. En effet, grâce à une vision pragmatique, l'analyse fonctionnelle **donne une réponse complète aux phases de production du produit final** et limite ainsi les risques de pertes de temps et/ou d'argent.

En d'autres termes, l'analyse fonctionnelle vous assure de traiter point par point chacune de vos exigences et de déterminer comment y répondre, avec quelles priorités et combien cela vous coûtera.

L'opération d'analyse fonctionnelle est de ce fait indispensable à la bonne réussite d'un projet informatique complexe. C'est une démarche incontournable non seulement dans l'analyse de sa valeur, car elle se base sur l'optimisation, mais c'est aussi une excellente manière de **mesurer le [rapport performance/coût](#)**.

I.6 : Les autres avantages

Les avantages de l'analyse fonctionnelle sont nombreux. Parmi ceux-là, et au-delà de la maîtrise des coûts qu'elle induit, cette méthode permet de **réfléchir en amont aux points bloquants** qui pourraient être rencontrés lors de la phase de production et d'y apporter dès le début des réponses adéquates.

De plus, l'analyse fonctionnelle permet de **faire travailler en collaboration différents corps de métiers**, comme ceux relatifs aux achats, au marketing ou encore à la conception technique. Travailler en équipe permet à chacun de donner son point de vue, d'apporter ses compétences, ses savoirs et ses savoir-faire au projet.

Pour mener à bien cette phase d'analyse fonctionnelle, il est nécessaire qu'**un chef de projet soit entièrement dédié à la conduite et à l'animation** de cette étude, et qu'il mette ensuite en forme les conclusions obtenues. Contrôler les étapes, conduire les réflexions et guider les professionnels qui travailleront en synergie sur le même projet constitue en effet une tâche relativement compliquée, qu'il faut confier à un expert en analyse fonctionnelle.

La réussite de l'étape d'analyse fonctionnelle permet la faisabilité d'un outil de conduite très important appelé **CAHEIR DES CHARGES PROJET**.

I.7 : Cahier des charges projet¹⁵

Savez-vous qu'une définition **claire du besoin et du périmètre de projet** est l'une des tops trois critères de réussite d'un projet ?

Effectivement, les résultats de plusieurs études ont révélé à l'unanimité que **l'énoncé clair du cahier des charges de projet et son périmètre** font partie des premiers facteurs clés de réussite des projets.

En effet, prendre en charge et gérer un projet sans énoncé ou cahier des charges clair, c'est essayer d'atteindre un objectif sans avoir de point de référence, et après, ça revient à tenter de tirer dans une cible les yeux fermés.

Réussir le projet dans ce cas de figure ne peut être que résultat d'hasard!

¹⁵ <https://blog-gestion-de-projet.com/cahier-des-charges-projet/>

A travers cette partie, je vise à vous fournir une meilleure compréhension des tenants et aboutissants des cahiers des charges projet, leurs raisons d'être, les bonnes pratiques à adopter. Avant de finir par la présentation d'un exemple de modèle de cahier des charges à adapter pour vos projets.

Il n'existe pas de format prédéfini pour un cahier des charges. L'important est qu'il soit précis, adapté à la situation et compréhensible par chacun des intervenants. Pour ce qui est du fond, le cahier des charges contiendra un ensemble de supports écrits permettant de décrire le résultat ou l'objectif attendu (éléments techniques, organigrammes, schémas techniques, spécificités, photos, ...). **Il fera référence à la réglementation en vigueur dont il rappellera les principaux éléments notamment en termes de sécurité.**

a. Pourquoi un cahier des charges projet ?

Gérer un projet est un acte rationnel qui consiste à éliminer des prises de décision impulsives, les rêveries des parties prenantes et les spéculations de l'équipe de projet.

Gérer un projet est une activité complexe qui ne tolère pas l'approximation.

Pour discipliner les réflexions des parties prenantes, quoi de plus mieux que d'encadrer les comportements et les attentes par des processus et un document de référence pour l'ensemble des acteurs ?

Vous ne pouvez pas gérer un projet sans documenter les besoins métiers et seul ce qui est explicite peut-être mis en œuvre et mesuré.

Un cahier des charges métier est l'antidote à l'ambiguïté, il oblige à être clair sur les concepts et les attentes vagues.

L'absence d'un énoncé clair des besoins à travers un cahier des charges projet et des spécifications métiers, ne peut qu'à mener à un dérapage ou une dérive de périmètre.

Plus de détail sur [la gestion de contenu du projet](https://blog-gestion-de-projet.com/contenu-et-perimetre-de-projet/) (<https://blog-gestion-de-projet.com/contenu-et-perimetre-de-projet/>)

En effet, la plupart des projets semblent souffrir d'une dérive de périmètre, ainsi, les équipes de projet et les parties prenantes en sont constamment frustrées.

Dérive du périmètre (Scope Sreep en Anglais) : C'est l'expansion non contrôlée du contenu du produit ou du périmètre du projet, sans ajustement des délais, du coût ou des ressources.

Lorsqu'une dérive de périmètre se produit dans un projet, elle devient une source considérable de gaspillage de l'argent et d'insatisfaction.

Une dérive de périmètre pourrait facilement priver le projet de produire le bénéfice et le résultat attendu.

b. C'est quoi au juste un cahier des charges projet ?

Le cahier des charges est le recueil des **exigences fonctionnelles** et **non-fonctionnelles**, demandées par la **maîtrise d'ouvrage**. Il exprime la demande en termes de besoins à satisfaire et de résultats ou services attendus du projet.

Il décrit des règles de gestion et de traitement dans le langage du métier (vue externe du système, c'est la vision utilisateur).

En effet, un cahier des charges projet bien fait exprime un besoin métier:

- **Nécessaire** : Le cahier des charges indique tous les paramètres requis pour la conception et réalisation du projet.
- **Réalisable** : Le cahier des charges exprime un besoin réaliste et qui est techniquement et financièrement réalisable dans le cadre du planning du cahier des charges, et des moyens disponibles.
- **Clair** : cela veut dire que l'idée ou le besoin que le cahier des charges projet exprime n'est pas susceptible d'être interprété différemment de ce que pense le maître d'ouvrage.
- **Vérifiable** : l'équipe de projet ou l'[AMOA](#) doit s'assurer qu'il existe des moyens de vérifier ce qui est requis.
- **Cohérent** : le cahier des charges projet doit exprimer des besoins qui ne se contredisent pas entre eux.

Attention : un cahier des charges est une description du besoin (le quoi) et ne pas de la solution (le comment). Ainsi, lors du montage du cahier des charges, concentrez-vous sur ce qu'il faut faire plutôt que comment il faut le faire !

A cet effet, il est fortement recommandé de se poser la question pourquoi ? à chaque affirmation d'une exigence pour éviter de décrire un produit ou des mises en œuvre au lieu d'exprimer un besoin métier.

Le risque, entre autres, quand on décrit un produit au lieu d'exprimer des besoins, est de croire que tous les besoins sont couverts par le produit indiqué.

Le cahier de charge (CDC) sera utilisé en interne par l'équipe projet. Mais sera aussi utilisé en externe lors d'interventions éventuelles de tierce parties, pour définir de manière détaillée le cadre de leur mission.

Le CDC ne peut donc laisser la place à aucune interprétation qui pourrait négativement influencer le budget ou le résultat final : il doit décrire en détail tous les éléments dont il faudra tenir compte lors de la réalisation du projet.

c. Quand intervient le cahier des charges de projet ?

En début de phase Planification, vous avez rencontré les bénéficiaires du projet et documenté l'Expression des besoins.

Découvrez plus sur *l'expression des besoins*.(<https://blog-gestion-de-projet.com/expression-des-besoins/>)

Vous avez donc maintenant une bonne idée des attentes fonctionnelles du projet ainsi que de certaines contraintes, exigences, et risques identifiés lors des réunions et interviews.

Cette expression des besoins est la base du cahier des charges.

En effet, le cahier des charges du projet s'appuie sur les activités de recueil et d'analyse des besoins et des exigences métiers.

Ces deux activités comprennent les tâches suivantes:

- **Spécifier et modéliser les exigences:** décrit un ensemble de spécifications métiers ou plans en détail.
- **Vérifier les exigences:** s'assurer que l'ensemble des exigences ou de conceptions retenues pour le projet sont suffisamment développées pour être utilisables par la maîtrise d'œuvre.
- **Valider les exigences:** vérifier que les exigences retenues pour le projet offrent une valeur à l'organisation et soutiennent les objectifs. Par la suite, faire valider ces exigences par les parties prenantes concernées.
- **Définir l'architecture des exigences:** structurer toutes les exigences et conceptions afin qu'ils soutiennent l'objectif du projet et qu'ils fonctionnent efficacement comme un ensemble cohérent.

En s'appuyant sur le cahier des charges, la maîtrise d'œuvre analyse et compare les différentes options, pour identifier et recommander la solution de mise en œuvre, qui offre la plus grande valeur à l'organisation.

Attention : les spécifications fonctionnelles détaillées ou les plans d'exécutions, sont élaborées par la maîtrise d'œuvre après réception et analyse du cahier des charges projet.

Les spécifications fonctionnelles détaillées est la traduction du cahier des charges projet en termes plus techniques (données en entrée, données en sortie, ...) dont le but est de décrire de façon détaillée comment les exigences métiers du projet vont être implémentées dans la solution.

d. Les techniques d'élaboration d'un cahier des charges projet

Dans cette section, je vous cite quelques outils et techniques de référence pour mener à bien le travail d'élaboration du cahier des charges de projet.

1- Observations

2- Entretiens individuels

- Questionnaires;
- Entretien directif ou semi-directif.

3- Réunions de réflexion collective

- Brainstorming;
- Ateliers de travail ou workshop;
- Groupe de discussion ou focus group.

4- Analyses

- Analyse des données : revue documentaire ou autre;
- Benchmark;
- Présentation de maquettes;
- Tests utilisateurs exploratoires.

e. Les composantes d'un cahier des charges projet

Dans cette section, et pour vous faciliter la structuration de votre cahier des charges projet, je vous propose les 6 composantes principales à intégrer absolument dans votre cahier des charges projet.

L'exemple traité ici dans cette section porte sur un projet informatique de mise en place d'un nouveau système.

Par la suite, je vous propose un modèle de cahier des charges à télécharger et adapter à vos projets spécifiques.

En effet, les six composantes indispensables à prévoir dans un cahier des charges projet sont les suivantes :

1. Description de l'état actuel (Contexte)

Pour avoir une bonne vue d'ensemble, l'équipe de projet ou le représentant du maître d'ouvrage doit s'assurer de décrire le fonctionnement général de l'entreprise et des processus métiers couverts par le projet; ainsi que les équipements et les systèmes actuellement utilisés.

Cela permettra de mieux comprendre l'importance du nouveau système ainsi que l'ampleur du changement pour l'organisation.

2. Les spécifications non fonctionnelles

Une exigence non-fonctionnelle est une exigence qui caractérise une propriété ou qualité intrinsèque désirée du système telle que sa performance, sa sécurité sa convivialité, sa maintenabilité, etc.

Une exigence non-fonctionnelle peut porter, aussi, sur le plan préconisé de management de projet, le niveau de support attendu après la mise en place du système et autres exigences vis-à-vis de la maîtrise d'œuvre.

Attention : l'ensemble des exigences doit être vérifiables, à défaut elles ne sont que des buts.

3. Les spécifications fonctionnelles

Une exigence fonctionnelle, comme déjà vu plus tôt dans cet article, est une exigence définissant une fonction du système à développer.

L'utilité d'une solution est déterminée par ses exigences fonctionnelles et ses caractéristiques non-fonctionnelles.

L'ensemble des fonctionnalités n'est pas utilisable sans certaines caractéristiques non fonctionnelles.

4. Les ressources

Maintenant que vous avez défini les objectifs, sous-objectifs et livrables, vous pouvez estimer de manière plus fine les besoins en ressources.

Une première estimation des ressources a été faite lors de la création de la Charte du Projet.

En savoir plus sur la charte de projet. (<https://blog-gestion-de-projet.com/realiser-la-charte-de-projet/>)

Grâce aux informations fonctionnelles et techniques récoltées, vous pouvez maintenant répondre à certaines questions restées ouvertes lors de l'initialisation du projet.

- ✚ Les compétences internes sont-elles suffisantes ?
- ✚ Avez-vous besoin d'aide externe et laquelle ?
- ✚ Les estimations de la charte correspondent-elles aux besoins maintenant détaillés ?
- ✚ Qui va faire quoi et quand ?

Ce dernier point est important : le besoin en ressources va évoluer en fonction de la charge du projet et de la phase en cours.

Pour cela, vous allez travailler en parallèle à la planification du projet en ajoutant aux livrables les ressources nécessaires à leur réalisation.

5. Les délais

En fonction des assignations de ressources aux différents livrables, vous pouvez confirmer ou corriger les estimations ayant servi à l'établissement de la Charte Projet.

La marge d'erreur acceptée sera fortement réduite, car vous pouvez maintenant travailler avec un niveau d'information beaucoup plus proche de la réalité.

En effet, vous connaissez maintenant les besoins, les contraintes et les exigences de manière détaillée.

6. Les besoins financiers et le budget

Grâce à la définition des besoins, du calcul des ressources et du plan détaillé, vous pouvez maintenant estimer de manière plus précise le budget.

Et ce en répondant aux questions suivantes :

Investissements:

- ✓ Quels matériels, logiciels ou équipements devez-vous acheter et quel est leur prix ?
- ✓ Quels amortissements appliquer ? (règles légales et spécifiques à l'entreprise)

Ressources:

- ✓ Le coût journalier de chaque ressource impliquée dans le projet (y compris chef de projet) multiplié par le nombre de jours nécessaires et le grand total. Ce montant peut aussi faire part des investissements;
- ✓ Ressources externes (consultants, experts métier, installateurs etc...)
- ✓ Coût opérationnel, (Quels sont les coûts récurrents?) : Maintenance, réseau, licences, location locaux ou matériel, etc...

f. Modèle de cahier des charges

Je vous propose ce modèle de cahier des charges, que vous pouvez adapter ou vous en inspirer. A télécharger au lien suivant : (<https://blog-gestion-de-projet.com/wp-content/uploads/2020/09/BGDP-Template-Cahier-des-charges.docx>)

g. Conclusion

Enfin, j'espère que vous avez saisi les tenants et les aboutissants du cahier des charges projet, et que la nuance entre l'expression du besoin métier et la conception de la solution est claire.

Aussi, il est important de noter que le Cahier des Charges est utilisé en interne comme en externe si vous faites appel à des prestataires de services.

Le Cahier des Charges devra être approuvé officiellement par le sponsor (maître d'ouvrage) et par les commanditaires/bénéficiaires principaux de votre projet lors d'un comité de direction projet.

CHAPITRE II : ARCHITECTURE ET/OU CONCEPTION DU LOGICIELLE

L'objectif de ce chapitre est de présenter ce qu'est l'architecture logicielle.

1. Définition

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.

2. Importance

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.

3. Objectifs

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.

4. Architecture ou conception ?

Il n'existe pas de vrai consensus concernant le sens des mots "architecture" et "conception" 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 dit, la distinction suivante est généralement admise et sera utilisée dans la suite de ce livre :

- **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 plus fin 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...

5. L'activité d'architecture

a. Définition

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. Cette phase correspond à l'activité d'architecture. 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.

b. Place dans le processus de création

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.

Répartition des problématiques

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

- Gérer les interactions avec l'extérieur, en particulier l'utilisateur : saisie et contrôle de données, affichage. C'est la problématique de **présentation**.
- Effectuer sur les données des opérations (calculs) en rapport avec les règles métier ("business logic"). C'est la problématique des **traitements**.
- Accéder et stocker les informations qu'il manipule, notamment entre deux utilisations. C'est la problématique des **données**.

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

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.

6. L'architecture d'un logiciel

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 l'architecture d'un logiciel selon différents points de vue. Entre autres, une vue **logique** mettra 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.

CHAPITRE III : PRINCIPES DE CONCEPTION

Ce chapitre présente les grands principes qui doivent guider la création d'un logiciel, et plus généralement le travail du développeur au quotidien.

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

1. Séparation des responsabilités

Le **principe de séparation des responsabilités** ([*separation of concerns*](#)) vise à organiser un logiciel en plusieurs sous-parties, chacune ayant une responsabilité bien définie.

C'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-partie(s) concernée(s). 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. "A class should have only one reason to change" (Robert C. Martin).

Exemples d'applications de ces deux principes :

- Une sous-partie qui s'occupe des affichages à l'écran participe 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.

2. Réutilisation

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 bibliothèques, 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](#) a 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 événements dans des fichiers.
- ...

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 **public** 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.

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 implementations").

5. Cohésion 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.

6. DRY

DRY est l'acronyme de **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 dûs 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.

```
1 function A() {
2   // ...
3
4   // Code dupliqué
5
6   // ...
7 }
8
9 function B() {
10  // ...
11
12  // Code dupliqué
13
14  // ...
15 }
```

La solution classique consiste à factoriser les lignes auparavant dupliquées.

```
1 function C() {
2   // Code auparavant dupliqué
3 }
4
5 function A() {
6   // ...
7
8   // Appel à C()
9
10  // ...
11 }
12
13 function B() {
14   // ...
15
16   // Appel à C()
17
18   // ...
19 }
```

Le principe DRY est important mais ne doit pas être appliqué de manière trop zélée. 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.

7. KISS

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 :

1. *Make it work.*
2. *Make it right.*
3. *Make it fast.*

8. YAGNI

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.

CHAPITRE IV : PRODUCTION DU CODE SOURCE

L'objectif de ce chapitre est de présenter les enjeux et les solutions liés à la production du code source d'un logiciel.

1. Introduction

Le code source 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.

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.

2. 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.

```
1 class UneNouvelleClasse {  
2     private int unAttribut;  
3     private float unAutreAttribut;  
4  
5     public void UneMethode(int monParam1, int monParam2) { ... }  
6     public void UneAutreMethode(string encoreUnParametre) { ... }  
7 }
```

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.

```
1 List<string> clients = new List<string>();  
2 // ...  
3 foreach(string client in clients) {  
4     // 'clients' désigne la liste, 'client' le client courant  
5     // ...  
6 }
```

On peut aussi utiliser des noms de la forme listeClients

3. Langue utilisée

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.

4. Formatage du code

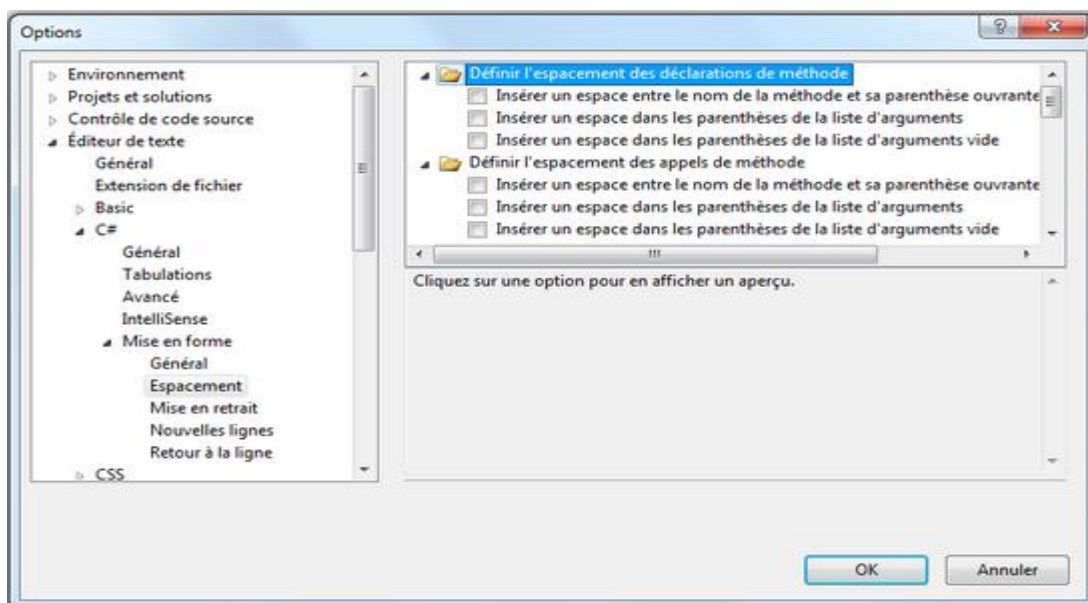
La grande majorité des [IDE](#) (Integrated Development Environment/ Environnement de Développement Intégré) et des éditeurs de code offrent des fonctionnalités de formatage automatique du code. A condition d'utiliser un paramétrage commun, cela permet à chaque membre de l'équipe de formater rapidement 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.
- ...

Sous Visual Studio, la commande de formatage automatique du code est **Edition->Avancé->Mettre le document en forme**.

Voici quelques exemples de paramétrages possibles du formatage (menu **Outils->Options**).



5. 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 (*eXtreme Programming*) 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 : autant lire directement les instructions décrites.

```
1 // Initialisation de i à 0
2 int i = 0;
3
4 // Instanciation d'un objet de la classe Random
5 Random rng = new Random();
6
7 // Appel de la méthode Next sur l'objet rng
8 int nombreAlea = rng.Next(1, 4);
```

Voici comment le code précédent pourrait être mieux commenté.

```
1 int i = 0;
2 Random rng = new Random();
3
4 // Génération aléatoire d'un entier entre 1 et 3
5 int nombreAlea = rng.Next(1, 4);
```

CHAPITRE V : GESTION DES VERSIONS

L'objectif de ce chapitre est de découvrir ce qu'est la gestion des versions d'un logiciel, en utilisant comme exemple l'outil Git.

1. Introduction

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 quelle(s) version(s).

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 est parfois appelée gestion du code source (SCM, Source Code Management).

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.

2. Les logiciels de gestion des versions

a) 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.

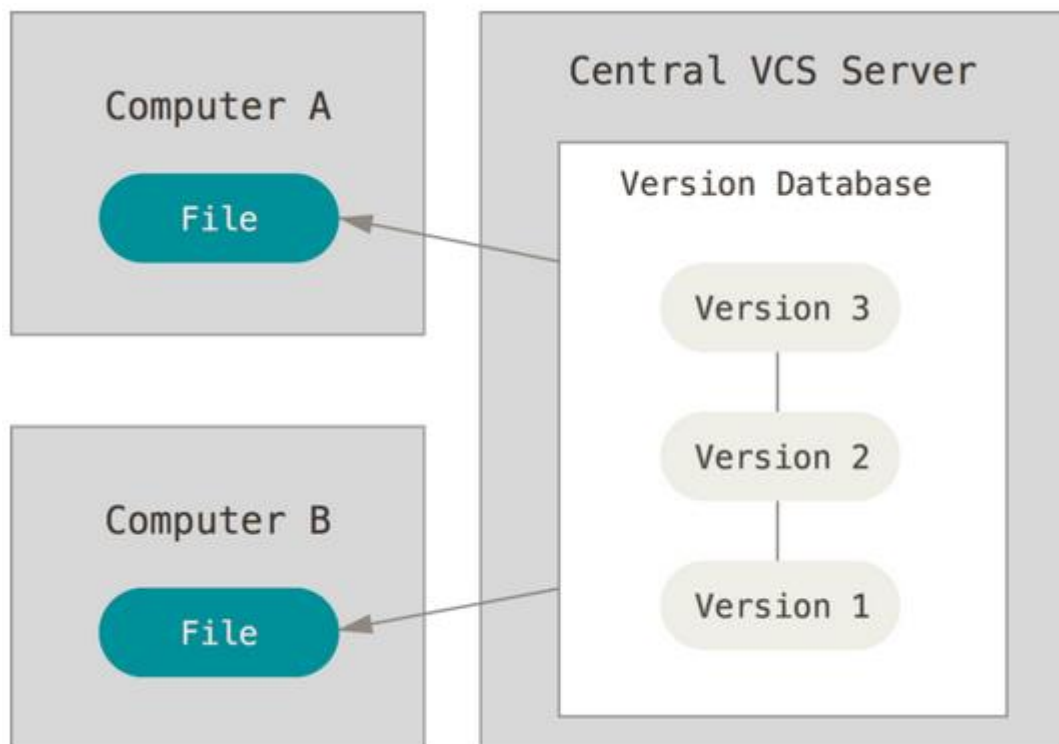
Le logiciel de gestion des versions permet de regrouper logiquement des fichiers par le biais du *tagging* : il ajoute aux fichiers source des tags correspondant aux différentes versions du logiciel.

b) Gestion centralisée Vs gestion décentralisée

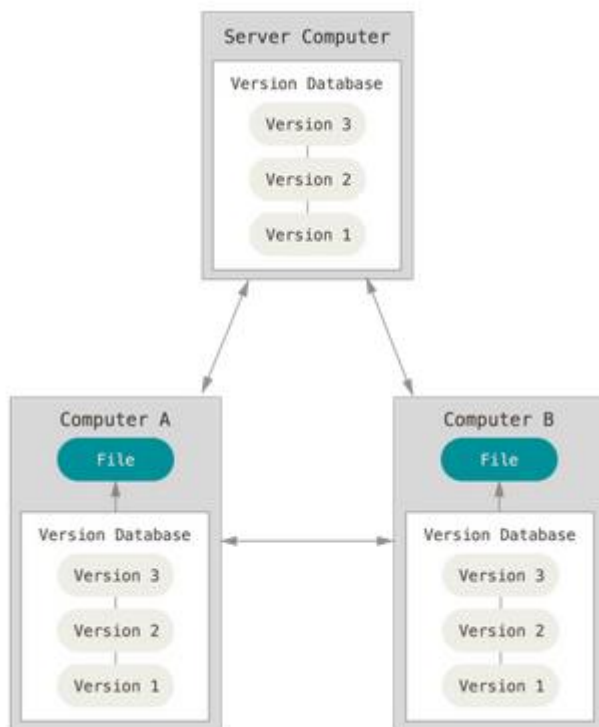
On peut classer les logiciels de gestion des versions en deux catégories.

La première catégorie offre une gestion centralisée du code source. Dans ce cas de figure, 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.



Une seconde catégorie est apparue il y a peu d'années. Elle consiste à voir le logiciel de gestion des versions comme un outil individuel permettant de travailler de manière décentralisée (hors ligne). Dans ce cas de figure, 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.



c) Principaux logiciels de gestion des versions

Il existe de très nombreux logiciels de gestion des versions. Nous n'allons citer que les principaux.

Assez ancien mais toujours utilisé, **CVS** (*Concurrent Versioning System*) 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 surtout **Git**, que nous utiliserons dans la suite de ce chapitre. 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.

Il est tout à fait possible de gérer le code source d'un projet .NET avec un autre outil de gestion des versions que TFS.

3. Présentation de Git

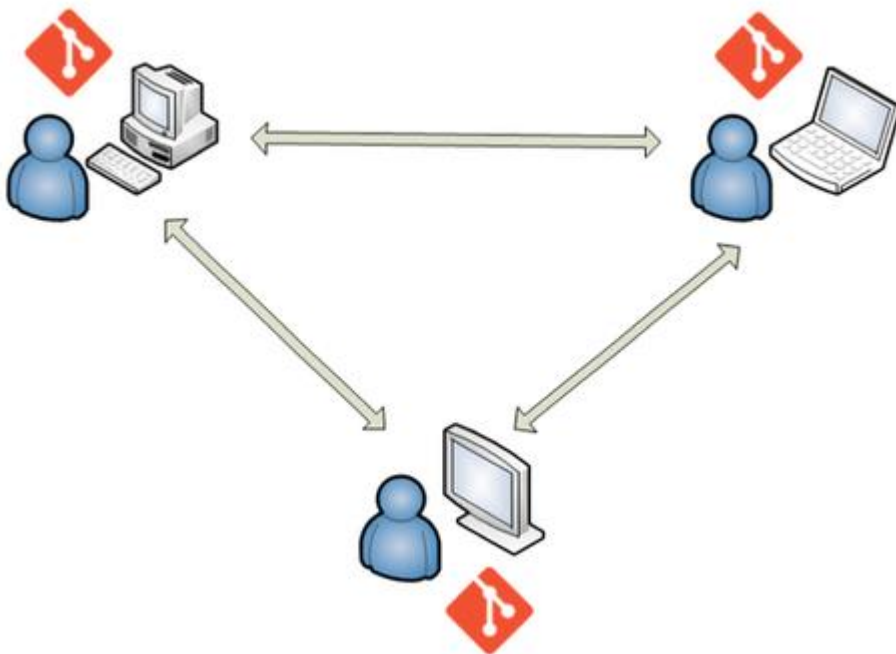
[Git](#) est un [logiciel libre](#) de [gestion des versions](#). C'est un outil qui permet d'archiver et de maintenir les différentes versions d'un ensemble de fichiers textuels constituant souvent le code source d'un projet logiciel. Créé à l'origine pour gérer le code du noyau Linux, il est multi-langages et multiplateformes. Git est devenu à l'heure actuelle un quasi-standard.



Le nom "Git" se prononce comme dans "guitare" et non pas comme dans "jitsu".

a) Fonctionnement

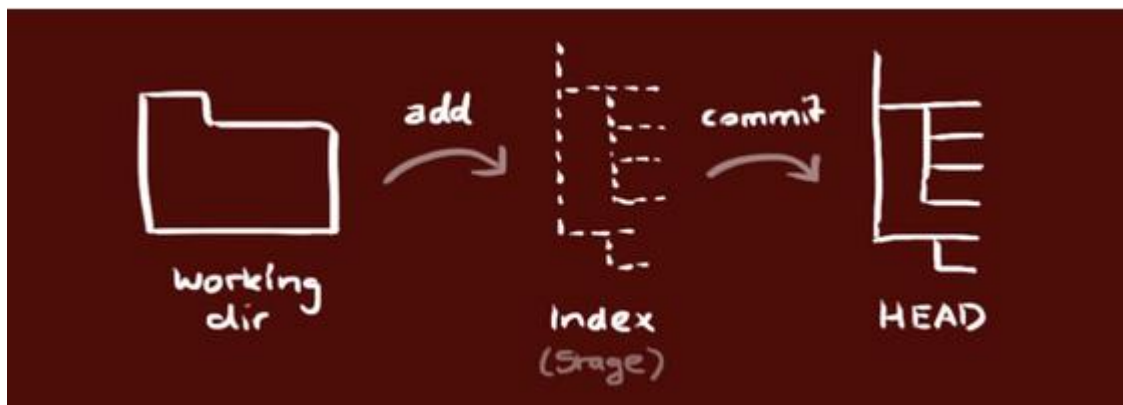
Git rassemble dans un **dépôt** (*repository* ou *repo*) l'ensemble des données associées au projet. Il fonctionne de manière décentralisée : tout dépôt Git contient l'intégralité des données (code source, historique, versions, etc). Chaque participant au projet travaille à son rythme sur son dépôt local. Il existe donc autant de dépôts que de participants. Git offre des mécanismes permettant de synchroniser les modifications entre tous les dépôts.



Un dépôt Git correspond physiquement à un ensemble de fichiers rassemblés dans un répertoire `.git`. Sauf cas particulier, il n'est pas nécessaire d'intervenir manuellement dans ce répertoire.

Lorsqu'on travaille avec Git, il est essentiel de faire la distinction entre trois zones :

- Le **répertoire de travail** (*working directory*) correspond aux fichiers actuellement sauvegardés localement.
- L'**index** ou *staging area* est un espace de transit.
- **HEAD** correspond aux derniers fichiers ajoutés au dépôt.



b) Principales opérations

Bien qu'il existe de nombreux plugins et autres interfaces graphiques, Git est à la base prévu pour être utilisé sous forme de commandes textuelles. Les principales commandes sont :

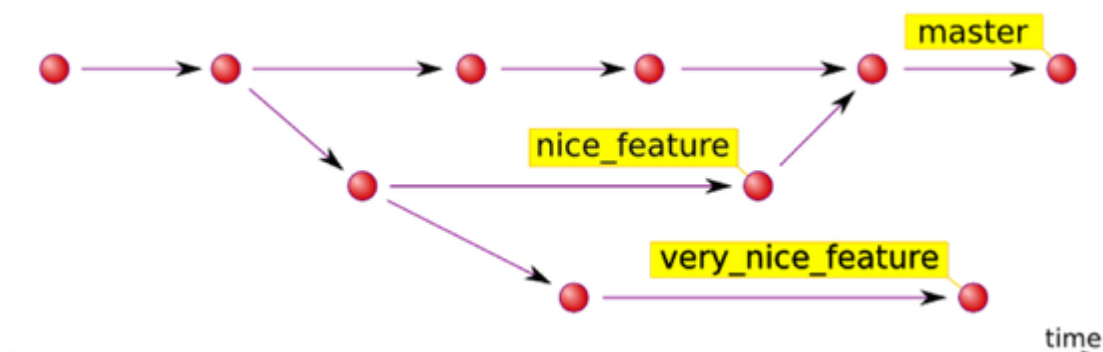
- `git init`: crée un nouveau dépôt vide à l'emplacement courant.
- `git status`: affiche les différences entre le répertoire de travail, l'index et HEAD.
- `git add`: ajoute des fichiers depuis le répertoire de travail vers l'index.
- `git commit`: ajoute des fichiers depuis l'index vers HEAD.
- `git clone`: clone un dépôt existant local ou distant.
- `git pull`: récupère des modifications depuis un dépôt distant vers HEAD.
- `git push`: publie des modifications depuis HEAD vers un dépôt distant.

Pour plus de détails sur les autres opérations possibles, consultez [ce mémento](#).

c) Notion de branche

Une branche permet de travailler de manière isolée sur une problématique particulière. Leur gestion par Git est particulièrement simple et efficace.

Tout dépôt Git possède une branche par défaut nommée `master`. On peut ensuite créer de nouvelles branches (`git branch`), y effectuer des modifications, puis fusionner cette branche avec la branche par défaut (`git merge`).



d) Le fichier .gitignore

Lorsqu'il est présent à la racine d'un répertoire géré par Git, le fichier .gitignore permet d'ignorer certains fichiers qui n'ont pas à être versionnés. C'est typiquement le cas des fichiers créés lors de la génération du logiciel ou des composants externes téléchargés par un gestionnaire de dépendances.

La plate-forme GitHub maintient en ligne une [liste de fichiers .gitignore](#) pour de nombreux types de projets. Voici un extrait du fichier .gitignore pour l'environnement Visual Studio.

```
1  # User-specific files
2  *.suo
3  *.user
4  *.userosscache
5  *.sln.docstates
6
7  # User-specific files (MonoDevelop/Xamarin Studio)
8  *.userprefs
9
10 # Build results
11 [Dd]ebug/
12 [Dd]ebugPublic/
13 [Rr]elease/
14 [Rr]eleases/
15 x64/
16 x86/
17 bld/
18 [Bb]in/
19 [Oo]bj/
20 [Ll]og/
21
22 # ...
```


CHAPITRE VI : TRAVAIL COLLABORATIF

L'objectif de ce chapitre est de présenter les enjeux du travail collaboratif dans le cadre de la réalisation d'un logiciel.

1. Les enjeux du travail collaboratif

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 en ligne sont apparues. On les appelle parfois des **forges logicielles**. La plus importante à l'heure actuelle est la plate-forme GitHub.

2. Présentation de GitHub

[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 (*organizations* et *teams*), des espaces d'échange autour du code (*issues* et *pull requests*), des statistiques, etc.

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.

a) Modèle économique

Le *business model* de GitHub est le suivant :

- La création d'un compte et de dépôts publics sont gratuites.
- La création de dépôts privés est payante.

Ce modèle économique est à l'origine de l'énorme popularité de GitHub pour la publication de projets *open source*.

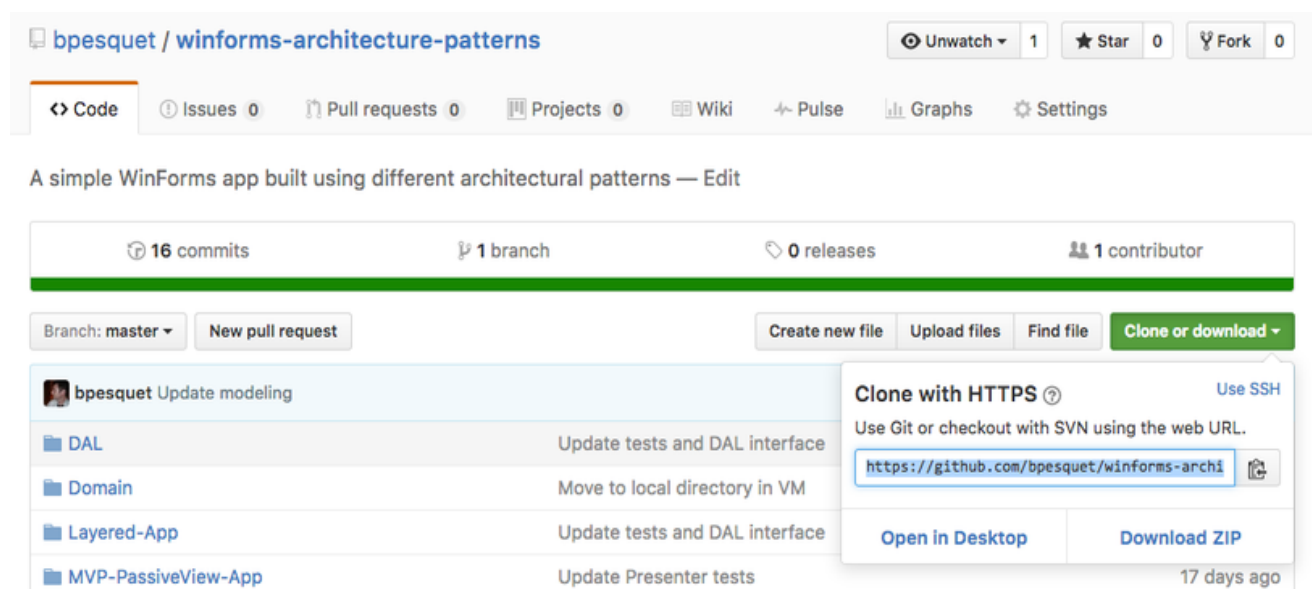
GitHub peut offrir des dépôts privés gratuits aux étudiants et aux établissements scolaires.

D'autres plates-formes alternatives se basent sur un modèle différent :

- [BitBucket](#) autorise les dépôts privés gratuits mais limite la taille de l'équipe à 5 personnes.
- [GitLab](#) est une solution *open core* pour installer sur ses propres serveurs une plate-forme similaire à GitHub.

b) Fonctionnement

Sur GitHub, un dépôt est associé à un utilisateur individuel ou à une organisation. Il est ensuite accessible via une URL de la forme <https://github.com/.../nomDuDepot.git>.

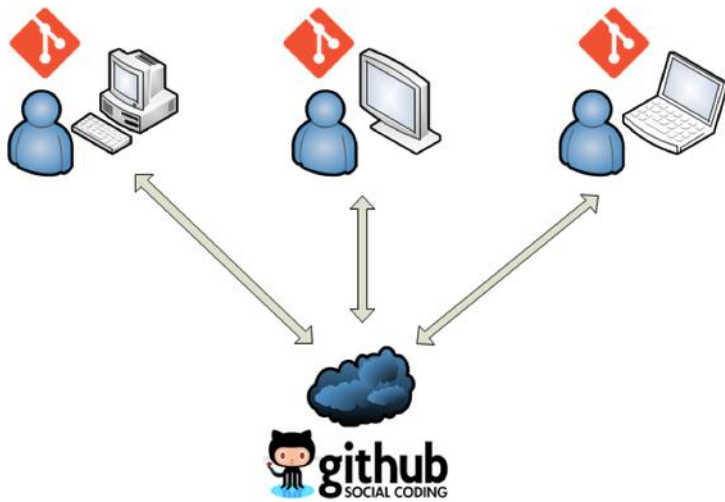


L'organisation du travail en équipe autour de GitHub peut se faire suivant deux modèles distincts.

➤ Modèle "dépôt partagé"

Ce modèle de travail est bien adapté aux petites équipes et aux projets peu complexes.

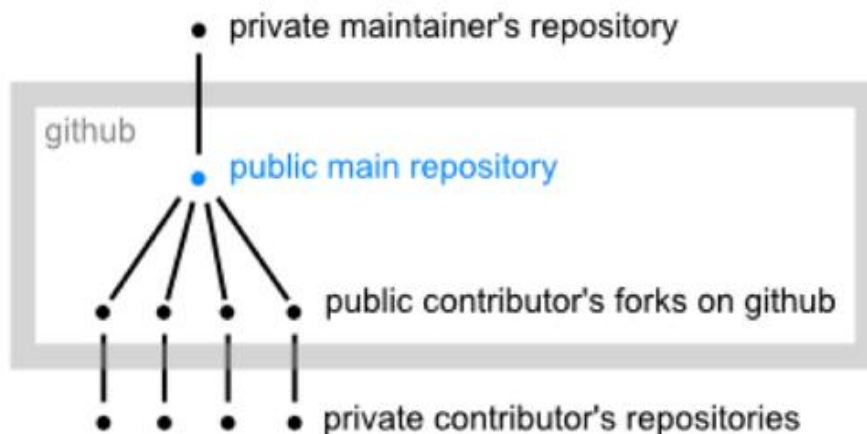
Un dépôt GitHub fait office de dépôt central pour l'équipe. Après avoir obtenu les droits nécessaires, chaque développeur clone ce dépôt (`git clone`) sur sa machine de travail. Ensuite, chacun peut envoyer ses modifications locales vers le dépôt commun (`git push`) et récupérer celles des autres membres de l'équipe (`git pull`).



➤ Modèle "fork and pull"

Ce modèle plus complexe est utilisé sur des projets de taille importante.

Un dépôt GitHub principal est *forké* par chaque développeur pour en obtenir une copie exacte sur son compte GitHub, puis cloné sur sa machine locale. Les modifications sont *pushées* sur GitHub, puis le développeur émet une demande d'intégration (*pull request*) pour signaler au responsable du dépôt commun qu'il a effectué des améliorations de son côté. Ce responsable étudie le code ajouté et décide de l'intégrer ou non dans le dépôt principal.



c) Issues

Parmi les autres services proposés par GitHub, les *issues* ("sujets de discussion") permettent de communiquer autour du projet. Elles sont souvent utilisées pour signaler des problèmes ou proposer des idées.

CHAPITRE VII : LES TESTS

L'objectif de ce chapitre est de présenter le rôle des tests dans le processus de création d'un logiciel.

1. Pourquoi tester ?

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 sujet 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](#).

2. Comment tester ?

On peut classer les tests logiciels en différentes catégories.

a) 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 permettent 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.

b) Tests d'intégration

Dans un projet informatique, l'intégration est le fait d'assembler plusieurs composants (ou modules) élémentaires en un composant de plus haut niveau. Un [test d'intégration](#) valide les résultats des interactions entre plusieurs composants et 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.

c) Tests unitaires

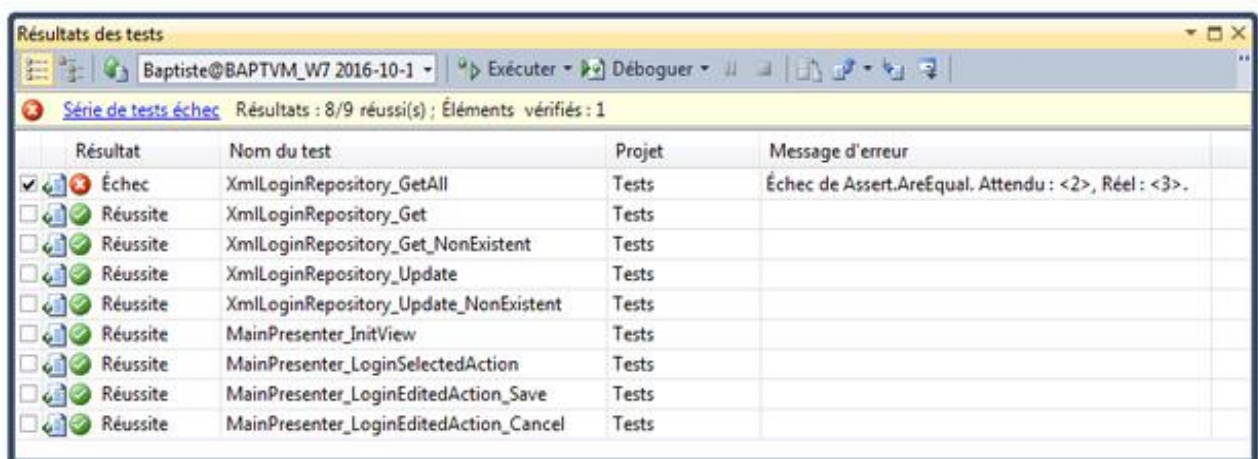
Contrairement aux tests de validation et d'intégration qui testent des pans entiers d'un logiciel, un test unitaire ne valide qu'une portion atomique du code source (exemple : une seule classe) et est systématiquement automatisé.

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 d'un projet permet de valider unitairement une grande partie de son code source et de détecter le plus tôt possible d'éventuelles erreurs.

L'image ci-dessous illustre le résultat du lancement de tests unitaires sous Visual Studio.



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

3. Création de tests doubles

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**.

Un couplage faible au sein du projet (voir [ce chapitre](#)) facilite l'écriture de *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 (ordre d'appel des méthodes, paramètres passés, etc.).

4. Compléments

Un projet écrit en langage C# et utilisant la technologie WinForms illustre certaines notions d'architecture et de test présentées dans ce cours. Son code source est [disponible en ligne](#).

CHAPITRE VIII : DOCUMENTATION¹⁶

L'objectif de ce chapitre est de découvrir les différents aspects associés à la documentation d'un logiciel.

1. Introduction

Nous avons déjà mentionné à plusieurs 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/ou 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 malaisé, 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
- La documentation utilisateur.

2. La documentation technique

a) Rôle

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.

b) Public visé

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 / maîtrise d'oeuvre pour réaliser un logiciel, la responsabilité de la documentation technique est à la charge de la maîtrise d'oeuvre. Bien sûr, le ou les document(s) associé(s) sont fournis à la MOA en fin de projet.

¹⁶ <https://ensc.gitbook.io/genie-logiciel/09-documentation>

c) Contenu

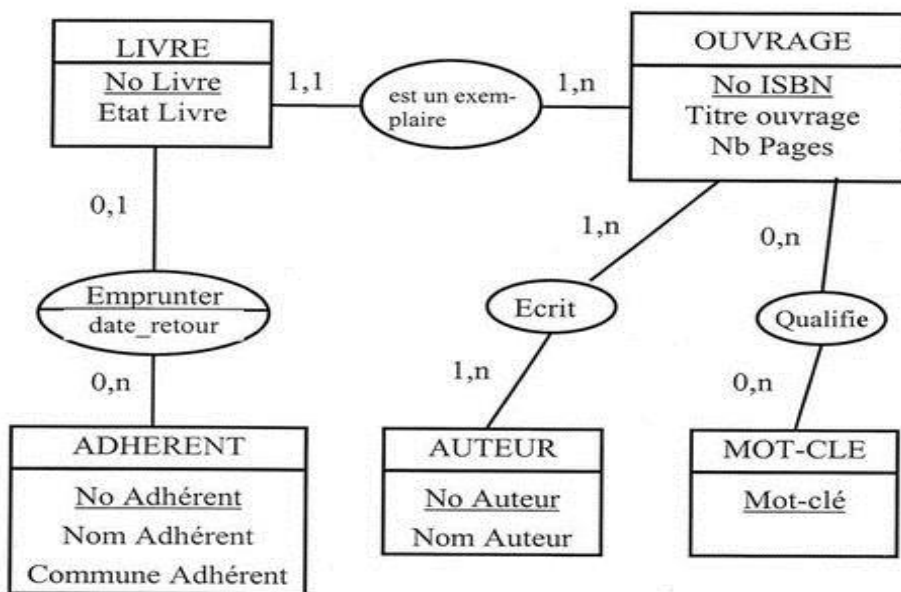
Le contenu de la documentation technique varie fortement en fonction de la structure et de la complexité du logiciel associé. Néanmoins, nous allons décrire les aspects qu'on y retrouve le plus souvent.

Les paragraphes ci-dessous ne constituent pas un plan-type de documentation technique.

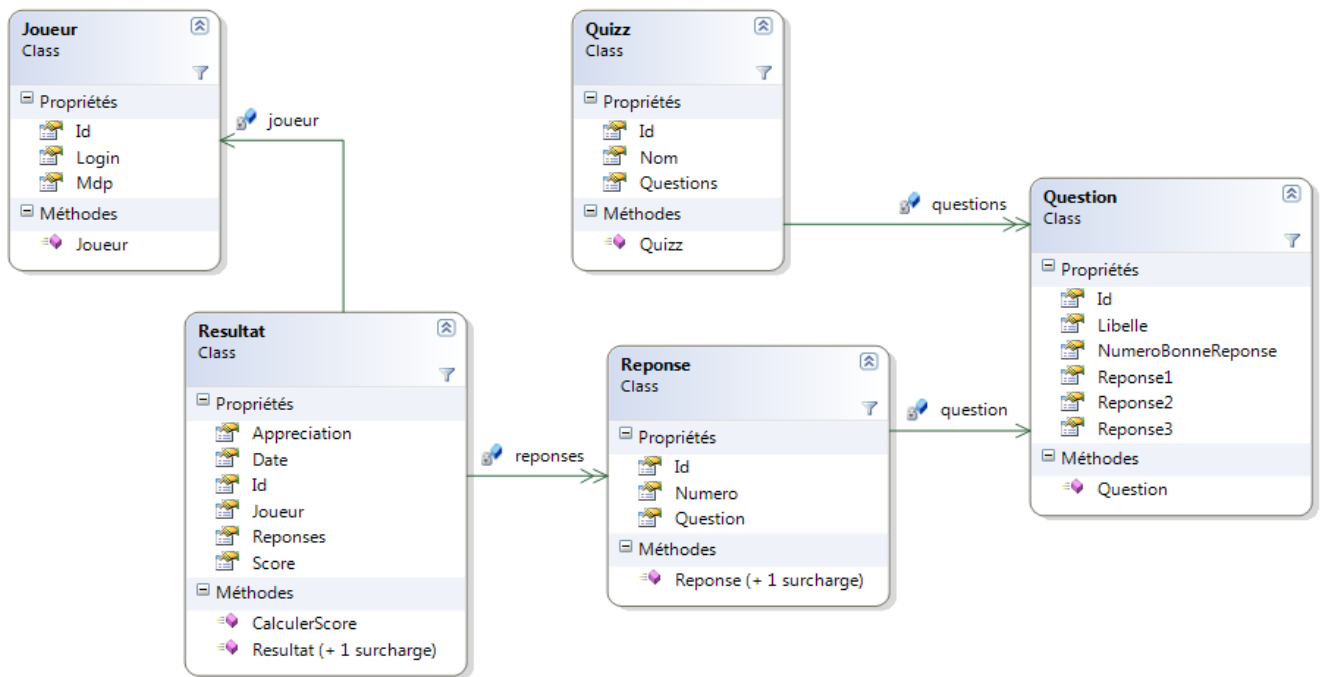
➤ Modélisation

La documentation technique inclut les informations liées au domaine du logiciel. Elle précise comment les éléments métier ont été modélisés informatiquement au sein du logiciel.

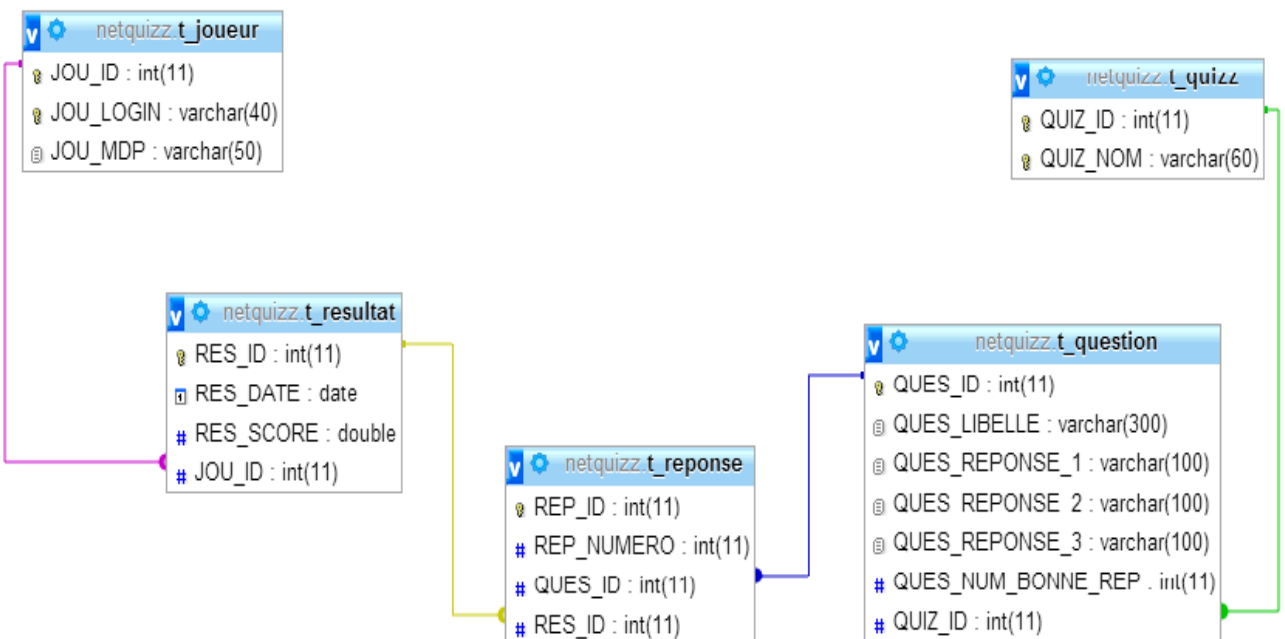
Si le logiciel a fait l'objet d'une modélisation de type entité-association, la documentation technique présente le modèle conceptuel résultat.



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.



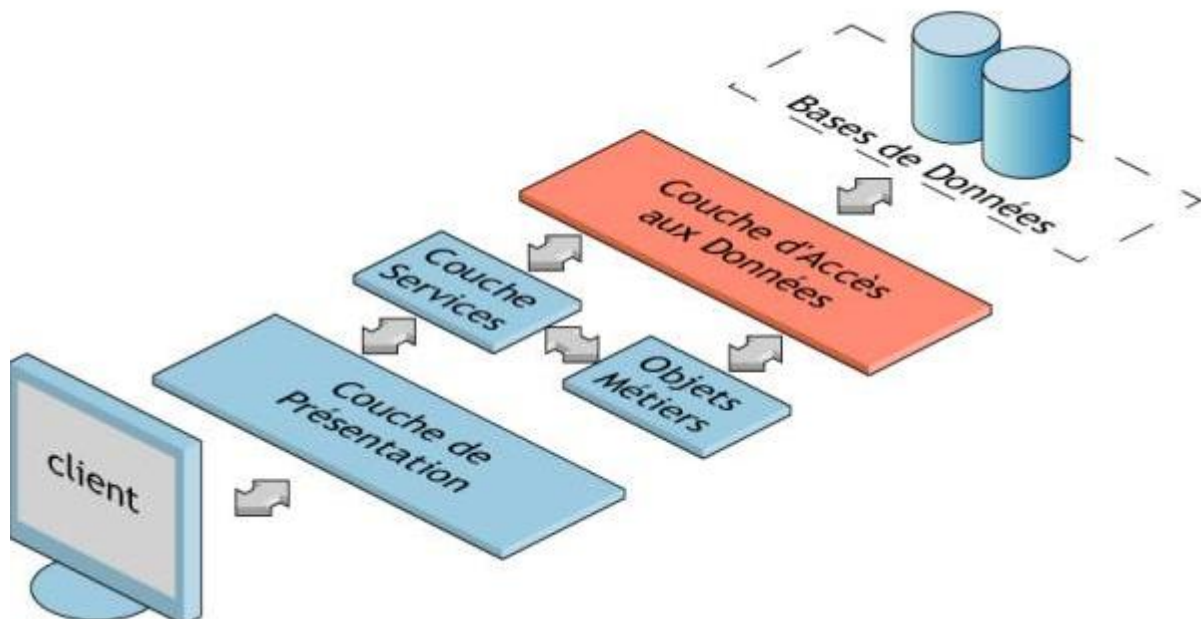
Si le logiciel utilise une base de données, la documentation technique doit présenter le modèle d'organisation des données retenu, le plus souvent sous la forme d'un modèle logique sous forme graphique.



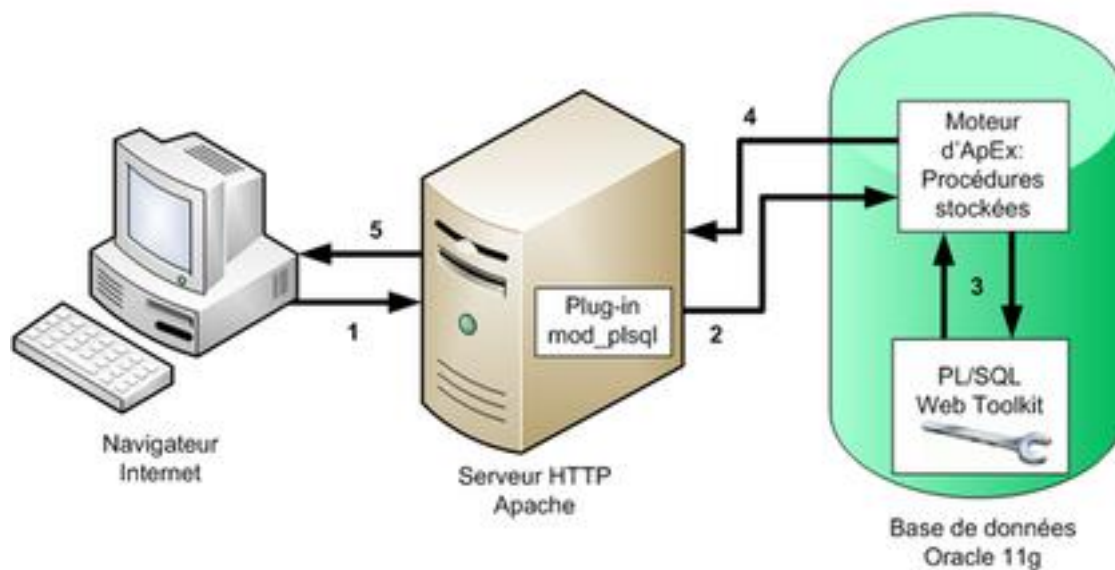
➤ Architecture

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.



➤ 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.

➤ 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.

➤ Déploiement

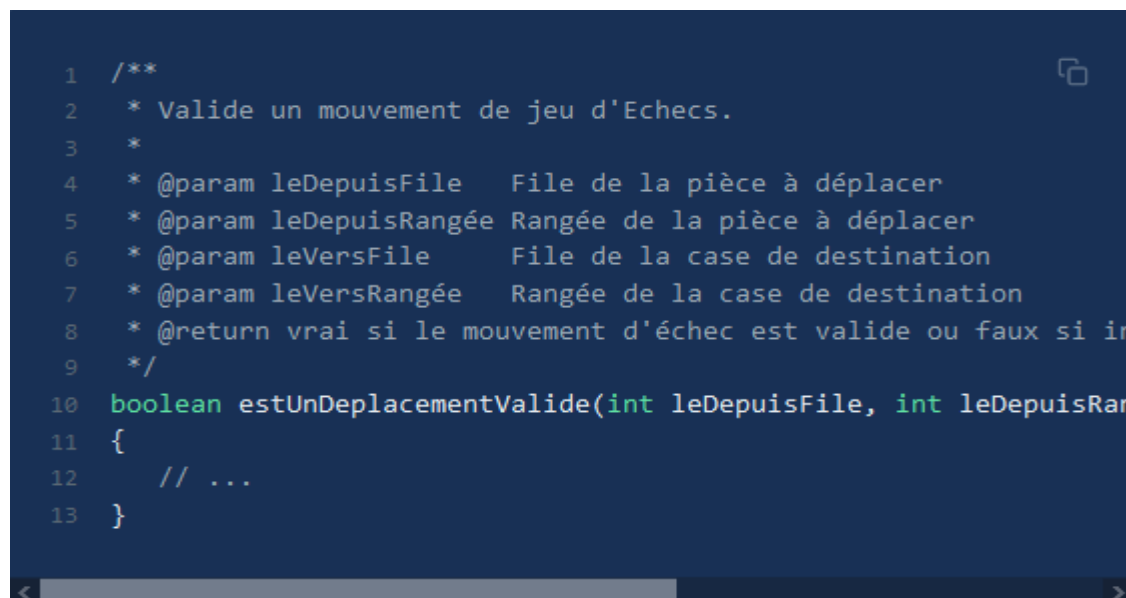
La documentation technique doit indiquer comment s'effectue le déploiement du logiciel, c'est-à-dire l'installation de ses différents composants sur la ou les machines nécessaire(s). Là encore, cette étape peut être plus ou moins complexe et nécessiter plus ou moins de documentation.

Documentation du code source

Il est également possible de documenter un logiciel directement depuis son code source en y ajoutant des commentaires (voir plus haut). Certains langages disposent d'un format spécial de commentaire permettant de créer une documentation auto-générée.

Le langage Java a le premier introduit 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.

Voici un exemple de méthode Java documentée au format javadoc.

A screenshot of a code editor with a dark blue background. It displays a Java method named `estUnDeplacementValide` with its Javadoc comments. The comments describe the method's purpose (validating a chess move) and its parameters: `leDepuisFile` (starting file), `leDepuisRangée` (starting rank), `leVersFile` (destination file), and `leVersRangée` (destination rank). The return value is a boolean indicating if the move is valid. The method signature is `boolean estUnDeplacementValide(int leDepuisFile, int leDepuisRangée, int leVersFile, int leVersRangée)`. The method body starts with a curly brace and a comment `// ...`.

```
1  /**
2   * Valide un mouvement de jeu d'Echecs.
3   *
4   * @param leDepuisFile   File de la pièce à déplacer
5   * @param leDepuisRangée Rangée de la pièce à déplacer
6   * @param leVersFile     File de la case de destination
7   * @param leVersRangée  Rangée de la case de destination
8   * @return vrai si le mouvement d'échec est valide ou faux si in
9   */
10 boolean estUnDeplacementValide(int leDepuisFile, int leDepuisRangée, int leVersFile, int leVersRangée)
11 {
12     // ...
13 }
```

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.

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

```
1  /// <summary>
2  /// Classe modélisant un adversaire de Chifoumi
3  /// </summary>
4  public class Adversaire
5  {
6      /// <summary>
7      /// Générateur de nombres aléatoires
8      /// Utilisé pour simuler le choix d'un signe : pierre, feuil
9      /// </summary>
10     private static Random rng = new Random();
11
12     /// <summary>
13     /// Fait choisir aléatoirement un coup à l'adversaire
14     /// Les coups possibles sont : "pierre", "feuille", "ciseau"
15     /// </summary>
16     /// <returns>Le coup choisi</returns>
17     public string ChoisirCoup()
18     {
19         // ...
20     }
21 }
```

3. La documentation utilisateur

❖ Rôle

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.

❖ Public visé

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.

Conséquence essentielle : 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.

❖ Formes possibles

➤ 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 :

- **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.

➤ Tutoriel

De plus en plus souvent, la documentation d'utilisation inclut un ou plusieurs tutoriel(s), 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.

➤ 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.

➤ 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 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 infobulles (*tooltips*). Elles permettent de décrire succinctement une fonctionnalité par survol du curseur.



4. Conseils de rédaction

❖ Structure

Qu'elle soit technique ou d'utilisation, toute documentation doit absolument être écrite de manière **structurée**, afin de faciliter l'accès à une information précise.

Structurer un document bureautique signifie :

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

❖ Niveau de langage

Comme dit plus haut, le public visé n'a pas forcément d'expérience informatique : il faut 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.

