Analyse des besoins pour le développement logiciel

Recueil et spécification, démarches itératives et agiles

Jacques Lonchamp
Professeur des universités

DUNOD

Toutes les marques citées dans cet ouvrage
sont des marques déposées par leurs propriétaires respectifs.

Illustration de couverture :
Grey abstract communication © iStock.com/nnnnae

5 rue Laromiguière, 75005 Paris
www.dunod.com

ISBN 978-2-10-072714-8

Table des matières

Avant-propos

POURQUOI CET OUVRAGE ?

Le présent ouvrage est l'aboutissement d'une longue pratique de l'enseignement du développement logiciel à divers publics universitaires et de formation continue. Plus précisément, ce sont les insatisfactions éprouvées à la lecture des documents pédagogiques existants, à l'occasion d'une réflexion nationale sur la formation des informaticiens, qui ont constitué l'élément déclencheur de sa rédaction.

Sa première moitié est consacrée aux *styles et processus de développement* des applications informatiques. Sa seconde moitié détaille les techniques utilisables lors de toutes les activités en amont de la conception, qui vont *du recueil des besoins à la spécification de l'application*. Dans la suite du volume, cet ensemble d'activités sera désigné par l'expression « analyse des besoins » ou, encore plus simplement, par le terme « analyse ».

Cet ouvrage reprend les principes d'un premier volume paru dans la même collection et consacré de manière complémentaire à la *conception* des applications (*Conception d'applications en Java/JEE. Principes, patterns et architectures*. Jacques Lonchamp, Dunod, 2014). Les principes communs qui sous-tendent ces deux volumes sont résumés au paragraphe suivant.

Alors qu'il est possible de parler de la conception de manière assez indépendante des styles et processus de développement logiciel, ce n'est pas du tout le cas pour l'analyse : on ne la pratique pas de la même manière dans un développement « agile », fondé sur l'adaptation continue aux évolutions des besoins grâce à un dialogue permanent avec le client et des itérations courtes, et dans un développement « classique », fondé sur le recueil initial exhaustif des besoins et la planification rigide de toutes les activités de production. C'est ce qui justifie le regroupement au sein d'un même volume de la description des styles et

processus de développement avant la présentation des connaissances théoriques et pratiques relatives à l'analyse.

LA CONCEPTION DE L'OUVRAGE

Le positionnement dans le cursus

Un cursus de formation à l'informatique ressemble à une fusée à trois étages.

L'étage *disciplinaire* vise l'acquisition des savoirs conceptuels et techniques de base, grâce à des enseignements ciblés vers des domaines bien délimités.

L'étage *intégratif* doit permettre de tisser des liens entre les savoirs disciplinaires, de les approfondir en conséquence, et de prendre conscience des enjeux réels dans le monde professionnel.

L'étage *professionnalisant*, qui s'appuie sur la compréhension globale des grandes thématiques que procure l'étage précédent, vise l'approfondissement de thèmes techniques spécialisés permettant de passer de la *compréhension* à la *maîtrise effective* de certaines facettes du métier.

Les deux ouvrages proposés, sur la conception d'une part et sur les processus et l'analyse d'autre part, relèvent clairement de la phase « intégrative », pour laquelle le manque de documents pédagogiques est criant.

Une progression logique des apprentissages

Dans le volume sur la conception, les savoirs en conception ont été ancrés dans les connaissances préalables en programmation. En effet, tout programmeur Java utilise au sein du JDK, sans en être nécessairement conscient, les patrons (*patterns*) de conception les plus connus et les plus utiles. Leur compréhension et leur mémorisation se trouvent grandement facilités quand cet ancrage est rendu explicite.

Dans ce second volume consacré aux concepts et méthodes de l'analyse, l'importance de la connaissance préalable des styles et processus de développement a déjà été évoquée. Par ailleurs, beaucoup de concepts de l'analyse prennent du sens à travers la compréhension de ce que peut être une conception et une réalisation de qualité, comme la modélisation de l'application en termes de classes frontières, de classes de contrôle et de classes entité. Une progression logique des apprentissages en développement des applications peut donc se schématiser par :

programmation $\longrightarrow$ *conception* $\longrightarrow$ *styles et processus de développement* $\longrightarrow$ *analyse*.

Pour une « culture générale » de l'analyse

Le côté très parcellaire de beaucoup de documents existants est aux antipodes de ce qu'on peut attendre pour la phase « intégrative » du cursus : l'analyse ne se réduit pas aux seuls besoins fonctionnels, la modélisation au seul langage UML et les processus aux seules approches agiles.

Sans tomber dans le travers encyclopédique des énormes « bibles » du génie logiciel (comme [Pre09] ou [Som04]), cet ouvrage cherche à donner une *présentation synthétique large* des thèmes abordés, une forme de « culture générale » de l'analyse, tout en approfondissant les points essentiels des pratiques les plus courantes.

L'importance de la mise en pratique

Beaucoup d'ouvrages de synthèse ne proposent ni exercices d'application ni études de cas, au contraire de ceux centrés sur des thématiques spécialisées. Or toute faiblesse sur ce point risque d'accréditer l'idée auprès des étudiants que ces documents se limitent à un *discours théorique*, éloigné de la pratique de terrain, et qui peut donc être négligé.

Au contraire, cet ouvrage propose près de 70 exercices d'application tous corrigés, associés à chacun des chapitres, et deux études de cas finales très détaillées. La majorité des exercices ont été glanés sur le Web et retravaillés. Que leurs auteurs originaux, souvent impossibles à déterminer, soient ici remerciés collectivement.

POUR QUELS LECTEURS ?

Cet ouvrage pédagogique s'adresse prioritairement aux étudiants de deuxième et troisième année des cursus spécialisés en informatique, quelle que soit leur nature (DUT/LP, L2/L3, deuxième et troisième années d'écoles d'ingénieurs).

Il peut bien entendu être également utile à tous les praticiens de l'informatique qui souhaitent rafraîchir ou consolider leurs connaissances en processus de développement et analyse des besoins.

LE PLAN DE L'OUVRAGE

La première partie s'attache à présenter de manière synthétique les différentes facettes du développement logiciel en insistant plus particulièrement sur les activités de l'ingénierie des besoins en amont de la conception des applications qui sont au cœur de l'ouvrage.

- Le **chapitre 1** définit le logiciel et la problématique de son développement au sein des organisations.
- Le **chapitre 2** est une introduction à l'ensemble des activités du développement logiciel.
- Le **chapitre 3** discute de la modélisation, qui est à la base de toute la démarche d'analyse et de conception, et donne quelques rappels sur la notation UML.
- Le **chapitre 4** présente les différentes organisations du développement logiciel, appelées communément « modèles de cycle de vie ».
- Le **chapitre 5** entre dans le détail des « méthodes » de développement objet les plus en vogue actuellement : *(Rational) Unified Process* – (R)UP, *eXtreme Programming* – XP, *Scrum*.

Les trois parties suivantes du livre décrivent les principales techniques et notations utilisables concrètement pendant l'analyse.

La deuxième partie est consacrée aux techniques de la modélisation métier.

- Le **chapitre 6** introduit la problématique de la modélisation métier et de l'utilisation du langage UML à cette fin.
- Le **chapitre 7** détaille la modélisation des processus métier.
- Le **chapitre 8** décrit la modélisation des entités métier ou modélisation du domaine.
- Le **chapitre 9** introduit la spécification formelle d'expressions avec le langage OCL (*Object Constraint Language*) qui sert souvent à préciser les modèles de classes du domaine. Mais ce langage peut bien entendu être utilisé dans d'autres contextes et avec d'autres modèles UML.

La troisième partie est dédiée aux techniques de la modélisation des besoins.

- Le **chapitre 10** présente l'utilisation des *user stories* (scénarios client).
- Le **chapitre 11** décrit l'utilisation des cas d'utilisation (*use cases*).
- Le **chapitre 12** aborde l'utilisation d'autres modèles UML en complément des cas d'utilisation : les diagrammes de séquences « système » et les diagrammes d'activités des cas.

La quatrième partie est consacrée aux techniques de la modélisation des applications.

- Le **chapitre 13** présente le modèle des classes d'analyse.
- Le **chapitre 14** traite des modélisations UML complémentaires au modèle des classes d'analyse sous la forme de diagrammes de séquences et de diagrammes d'états.
- Le **chapitre 15** aborde les aspects liés aux interfaces homme-machine (IHM) à travers le modèle de navigation.

La cinquième et dernière partie du livre présente en détail deux études de cas.

La première porte sur l'analyse d'un site de commerce électronique selon les préconisations du processus unifié (R)UP.

- Le **chapitre 16** décrit l'unique itération de la phase d'initialisation.
- Le **chapitre 17** décrit la première itération de la phase d'élaboration.

La deuxième étude de cas porte sur l'analyse d'un jeu de plateau (type Monopoly).

- Le **chapitre 18** décrit informellement les règles du jeu, puis donne une spécification de l'application correspondante sous forme de *user stories*.
- Le **chapitre 19** donne une spécification de l'application en termes de cas d'utilisation et de diagramme de cas.
- Le **chapitre 20** décrit le modèle des classes du domaine et illustre les diagrammes d'états associés aux entités complexes du jeu.

Chapitre 1

Introduction

1.1 LE LOGICIEL

1.1.1. L'importance du logiciel

Beaucoup d'aspects de notre quotidien ne peuvent être imaginés aujourd'hui sans logiciel. C'est le cas entre autres des domaines du transport, du commerce, des communications, de la médecine, des finances ou des loisirs. Cette omniprésence du logiciel fait que nos vies et le fonctionnement de nos sociétés dépendent fortement de sa *qualité*.

Les erreurs logicielles peuvent causer de vrais désastres, économiques ou sanitaires. Un exemple emblématique est l'explosion de la première fusée Ariane 5 en 1996, qui a coûté plus d'un demi-milliard de dollars, à cause du dépassement de capacité d'une variable lors d'un calcul. Dans un registre différent, tout le monde se souvient de la grande peur du « bug de l'an 2000 » et de son coût astronomique, estimé entre 300 et 5000 milliards de dollars !

1.1.2. La diversité du logiciel

Il existe une grande variété de logiciels et de nombreuses manières de les classer.

Une première différenciation oppose les logiciels « génériques », ou progiciels, qui sont vendus en grand nombre comme des produits de consommation, aux logiciels « spécifiques » qui sont développés pour un contexte et un client particulier.

Une seconde différenciation repose sur la nature de la dépendance entre les logiciels et leurs environnements. Elle distingue trois classes :

– les logiciels « autonomes » (*standalone*), comme les traitements de texte, les logiciels de calcul ou les jeux,
– les logiciels qui gèrent des processus (*process support*), aussi bien industriels que de gestion,
– les logiciels « embarqués » dans des systèmes (*embedded*), comme dans les transports, la téléphonie ou les satellites.

Dans cette dernière classe, on a coutume de parler plutôt de « développement système » que de « développement logiciel », car les problématiques matérielles et logicielles y sont indissociables.

Une troisième différenciation sépare les logiciels isolés des « lignes de produits logiciels ». Celles-ci regroupent une famille de logiciels présentant de fortes similarités et un même domaine d'application, comme des applications pour téléphones mobiles. Elles sont développées à partir d'éléments réutilisables (*assets*) qui peuvent être des exigences, des modèles, des codes (composants), des plans de tests, etc.

Dans les grandes organisations, on particularise souvent ce qu'on appelle les « logiciels du patrimoine » (*legacy software*). Ce sont les applications les plus anciennes, complexes et peu documentées, auxquelles on touche le moins possible tant qu'elles remplissent correctement leurs fonctions en général vitales pour l'organisation.

Enfin les « logiciels web », ou « applications web », sont souvent considérés comme une catégorie à part. Ce sont des logiciels hébergés sur un serveur web et manipulables via les réseaux, Internet ou réseaux locaux, grâce à un navigateur web. On peut citer comme particularités notables :

– les accès concurrents qu'ils subissent, avec une charge difficilement prédictible,
– leurs exigences élevées de performance, de disponibilité et de sécurité,
– la prédominance des données par rapport aux traitements,
– leurs besoins fréquents d'évolution,
– l'importance de l'ergonomie et de l'esthétique dans leur conception.

On comprend que la notion de qualité et les exigences qui s'y attachent peuvent varier considérablement pour toute cette diversité de logiciels.

1.1.3. La complexité du logiciel

Techniquement, le terme « logiciel » désigne un riche ensemble d'artefacts incluant :

– des codes sources et des exécutables,
– des programmes et des données de test,
– des fichiers de configuration,
– des documentations « externes » à destination des utilisateurs,
– des documentations « internes » à destination de l'équipe de développement, etc.

1.2 LE DÉVELOPPEMENT LOGICIEL

1.2.1. Les acteurs du développement

Le développement logiciel est une activité intellectuelle à forte composante humaine (*human-intensive*). En 2013, le secteur du conseil, des études et des services informatiques en France comptait 575 000 emplois, hors fonction publique d'État.

Pendant le développement d'un logiciel, outre les membres de l'équipe de développement proprement dite, les autres acteurs à considérer sont le client (l'investisseur), les utilisateurs finaux et d'éventuels sous-traitants.

1.2.2. Les contextes du développement

Le logiciel peut être développé dans plusieurs contextes assez différents.

– Chez des éditeurs de logiciels qui commercialisent les progiciels qu'ils développent.

– Dans des Sociétés de service en ingénierie informatique (SSII), également nommées Entreprises de services du numérique (ESN), qui répondent aux besoins d'externalisation des projets informatiques, en développant des logiciels pour le compte de leurs clients.

– En interne, dans des organisations de toutes natures, entreprises, administrations et associations.

– Sur Internet, au moins en partie par des développeurs bénévoles, pour les logiciels libres.

1.2.3. Les activités du développement

Schématiquement, le développement logiciel consiste à transformer une idée ou un besoin en un logiciel fonctionnel. L'idée est produite par un client ou *maître d'ouvrage* (MOA). Le logiciel correspondant est développé par un fournisseur ou *maître d'œuvre* (MOE), puis exploité par des utilisateurs finaux. Lorsqu'il s'agit d'entités on parle de *Maîtrise d'ouvrage* (MOA) et de *Maîtrise d'œuvre* (MOE). La MOA peut être assistée en externe et en interne, souvent par d'anciens informaticiens ayant une bonne pratique de la conduite des projets : on parle de MOAD, pour *maîtrise d'ouvrage déléguée*.

FIGURE 1.1 Le développement logiciel

Comme cela sera détaillé dans la suite de l'ouvrage, le développement logiciel comprend en amont une phase centrée sur les *besoins* (ou exigences). Il s'agit pour toutes les parties prenantes d'un projet informatique de s'accorder sur l'ensemble des besoins auxquels devra répondre le système ou logiciel à construire. Le terme d'« ingénierie des besoins » ou « ingénierie des exigences » (*requirements engineering*) caractérise toutes les activités liées

à la gestion des besoins, y compris celles qui se poursuivent tout au long du développement, comme la gestion des évolutions des besoins et de leur traçabilité.

Le présent ouvrage est plus précisément centré sur les phases *en amont du développement*, à savoir le recueil, l'analyse, la spécification, la revue et le classement par priorités des besoins, qui précèdent la conception du logiciel. Cet ensemble d'activités est désigné dans la suite de l'ouvrage par l'expression « analyse des besoins » ou encore plus simplement par le terme « analyse ».

Comme la manière de conduire une analyse dépend fortement du style de développement suivi, l'ouvrage commence par une description en profondeur des différents styles et processus de développement logiciel.

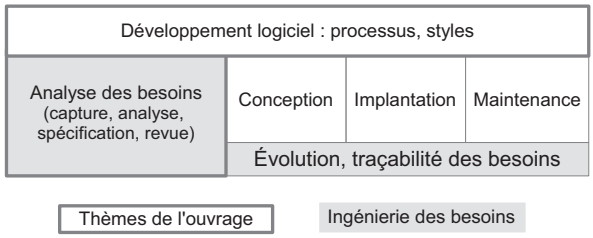


FIGURE 1.2 Les thèmes principaux de l'ouvrage

1.3 LA QUALITÉ DU LOGICIEL

La notion de « qualité du logiciel » n'est pas simple à définir car elle est multiforme et dépend du point de vue adopté, client ou fournisseur. Pour le client, le logiciel doit répondre à ses besoins (adéquation), être efficace, facile d'apprentissage et d'utilisation (ergonomie), être fiable, robuste et sûr, être peu coûteux et livré dans les délais. Pour le fournisseur, le coût et la durée du développement sont généralement primordiaux, de même que la facilité d'extension, de maintenance, d'adaptation et d'évolution, ainsi que la portabilité et l'interopérabilité.

Il est important de ne pas confondre les termes qui caractérisent les différentes formes de qualité. La signification des plus courants d'entre eux est rappelée dans le tableau suivant.

Terme français	Terme anglais	Signification
Disponibilité	*Availability*	Capacité à délivrer le service attendu
Fiabilité	*Reliability*	Capacité à maintenir la continuité du service attendu
Sécurité	*Security*	Absence de conséquences catastrophiques
Sûreté	*Safety*	Protection contre les actions illicites
Intégrité	*Integrity*	Absence de d'altérations inapropriées
Confidentialité	*Confidentiality*	Absence de divulgations non autorisées
Maintenabilité	*Maintainability*	Aptitude aux réparations et aux évolutions

On résume parfois la variété de ce qui est attendu d'un logiciel par le sigle CQFD, pour Coûts, Qualité, Fonctionnalités, Délais. Cette vision simplifiée est suffisante pour montrer les tensions qui peuvent exister entre les principaux critères. Idéalement, il faudrait pouvoir faire du logiciel « chic et pas cher » (F élevé et C faible), ce qui n'est pas simple, et de le faire « vite et bien » (D faible et Q élevé), ce qui n'est également pas évident !

Comme cela a déjà été évoqué, tous ces critères sont bien entendu à pondérer en fonction du type de logiciel à développer : du logiciel critique pour la sécurité (*safety critical*), dont les défaillances peuvent avoir des conséquences désastreuses en termes humains, économiques ou environnementaux, jusqu'au logiciel « jetable », beaucoup moins exigeant.

1.4 LA « CRISE DU LOGICIEL »

1.4.1. Le constat

L'expression « crise du logiciel » est employée depuis la fin des années 1960 pour rendre compte de toutes les difficultés associées à sa production : délais de livraison et budgets non respectés, non-satisfaction de certains besoins des clients ou utilisateurs, difficultés d'utilisation et de maintenance, etc.

De nombreuses études rendent compte de ce phénomène en classant les développements en trois grandes catégories :

– *Succès* : quand le logiciel est livré à temps, sans dépassement de budget et avec toutes les fonctionnalités initialement prévues.

– *Mitigé* : quand le logiciel est livré et opérationnel, mais avec moins de fonctionnalités qu'attendu ou un dépassement notable de budget ou des délais.

– *Échec* : quand le logiciel est abandonné en cours de développement ou livré et non utilisé.

Une des toutes premières études, réalisée par les militaires du Department of Defense (DoD) américain dans les années 1980, donnait des résultats absolument catastrophiques : succès 5 % (dont utilisation « tel que » 2 % et avec de faibles modifications 3 %), mitigé 19 %, échec 76 % (dont jamais utilisé 47 % et non livré 29 %). Depuis, si on se base sur les études annuelles du Standish Group, connues sous le nom de *Chaos Report* et résumées dans le tableau suivant, les progrès semblent à la fois réels et assez lents.

Année	Succès	Mitigé	Échec
1995	16 %	53 %	31 %
2000	28 %	49 %	23 %
2006	35 %	46 %	19 %
2010	37 %	42 %	21 %
2013	39 %	43 %	19 %

Des exemples d'abandon de logiciels extrêmement coûteux continuent encore et toujours à être dénoncés dans la presse, comme celui du Logiciel unique à vocation interarmées de la solde (LOUVOIS) lancé en France en 1996 et dont l'abandon a été annoncé en novembre

2013. Son développement a été marqué par plusieurs crises, relances et réorientations entre 2001 et 2011, avant l'abandon final en raison de son incapacité à assurer son rôle de manière fiable avec des dizaines de millions d'euros de moins-perçus et de trop-perçus dans les soldes des militaires français…

1.4.2. L'influence de la complexité

Les progrès, bien qu'indéniables, sont atténués car la *complexité* des applications ne cesse de s'accroître au fil des années ainsi que les exigences des utilisateurs.

Or plus la taille s'accroît plus les échecs sont fréquents. Dans l'étude du Standish Group de 2012 on trouve une comparaison entre « petits projets » de moins de 1 million de dollars et « gros projets » de plus de 10 millions de dollars qui le démontre clairement.

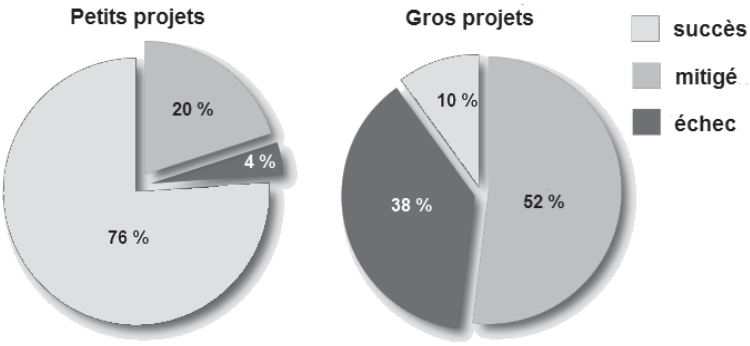


FIGURE 1.3 Influence de la complexité sur la réussite des projets

Pour donner quelques ordres de grandeur à propos de la complexité des logiciels, on peut indiquer qu'un compilateur simple correspond à 20 ou 30 KLS (milliers de lignes source), un logiciel de modélisation 3D correspond à 200 KLS, le logiciel de contrôle d'un sous-marin nucléaire exige 1 000 KLS et celui de la navette spatiale plus de 2 200 KLS. Soit un facteur 100 entre ces extrêmes.

1.4.3. Les causes

Parmi les multiples causes avancées, beaucoup sont liées à la nature même du logiciel et donc très difficiles à combattre :

– Le logiciel est un objet *immatériel*, une structure d'information.

– Le logiciel est très *malléable*, au sens de facile à modifier, avec parfois des conséquences majeures pour des modifications infimes.

– Les défaillances et erreurs ne proviennent ni de défauts dans les matériaux ni de phénomènes d'usure dont on peut connaître les lois d'occurrence, mais d'*erreurs humaines*, peu prévisibles. Beaucoup de ces erreurs sont difficiles à découvrir avant la livraison du produit chez le client.

– Le logiciel ne s'use pas et ne se répare pas. Il n'y a pas de pièces de rechange ! Par contre, le logiciel peu devenir assez rapidement obsolète par rapport aux concurrents, par rapport au contexte technique, par rapport aux logiciels qui l'entourent. Il s'agit d'un domaine en *évolution* extrêmement rapide.

D'autres causes sont plus spécifiquement liées à la difficulté de comprendre les besoins des clients et des utilisateurs. Elles sont extrêmement importantes et au cœur de la problématique abordée dans cet ouvrage :

– La trop faible *implication* des clients et des utilisateurs dans les projets de développement logiciel.
– La difficulté pour les clients à *décrire leurs besoins* de façon claire et exhaustive pour ceux qui développent les applications.
– L'inévitable *évolutivité* des besoins, résultant par exemple des évolutions fréquentes dans l'environnement des applications.

Enfin, d'autres causes plus générales résultent de la difficulté à gérer des projets et des équipes, en particulier quand ils sont de grande taille, comme par exemple :

– la différence de langage et de culture entre les personnels techniques et non techniques,
– le renouvellement rapide du personnel (*turnover*),
– la difficulté à motiver les personnes sur le long terme, etc.

1.4.4. Les « mythes » du développement logiciel

Beaucoup de croyances erronées, souvent qualifiées de « mythes », gênent la bonne compréhension du développement logiciel et peuvent contribuer à aggraver les difficultés. Les cinq mythes suivants sont parmi les plus connus [Pre09] :

– *Un vague énoncé est suffisant pour commencer à coder. Il sera toujours temps de voir les détails plus tard.*
  En réalité, la connaissance insuffisante des besoins est une source reconnue de retards et de coûts élevés.
– *Les besoins changent sans cesse, mais ce n'est pas grave car le logiciel est flexible.*
  En réalité, le coût d'une modification augmente considérablement avec l'avancement du projet. Si on part d'un facteur 1 pour la phase de définition, on estime à un facteur 1,5 à 6 le coût d'une modification en cours de développement et à un facteur 60 à 100, le coût d'une modification après installation du logiciel.
– *Quand un programme est écrit et tourne, le travail du développeur est terminé.*
  En réalité, la maintenance des applications requiert de 50 à 70 % de l'effort total.
– *Tant qu'un programme ne tourne pas, il n'y a aucun moyen d'en vérifier la qualité.*
  En réalité, on peut procéder à des revues ou inspections des produits du développement (cf. paragraphe 2.9.4.) qui sont très efficaces pour en améliorer la qualité.
– *Si un projet prend du retard, il suffit d'ajouter des programmeurs.*
  En réalité, le développement du logiciel n'est pas une activité industrielle classique. Ajouter des programmeurs peut contribuer, au contraire, à aggraver la situation. Ce mythe,

énoncé dans le célèbre ouvrage *The mythical Man-Month* de Frederick Brooks [Bro75] est analysé en exercice à la fin du chapitre.

1.4.5. Les solutions

L'étude des différents éléments de solution mis en œuvre aujourd'hui pour combattre la « crise du logiciel » est au cœur du présent ouvrage.

Pour la phase en amont du développement logiciel, cela recouvre pour l'essentiel quatre approches complémentaires qui seront étudiées en détail dans la suite de l'ouvrage.

(1) Le recueil des besoins avec les clients et utilisateurs et leur expression initiale à travers des formes d'expression adaptées.

(2) L'analyse approfondie des besoins à l'aide de langages de modélisation, souvent visuels, comme UML.

(3) La manière de gérer les développements. L'étude du Standish Group, déjà citée, montre que la loi de Pareto du 20/80 est en gros respectée pour le logiciel, avec 20 % des fonctionnalités implantées qui sont toujours (7 %) ou souvent (13 %) utilisées et 80 % des fonctionnalités implantées qui sont parfois (16 %), rarement (19 %) ou jamais (45 %) utilisées. Il est donc très important et rentable de cibler les développements en priorité sur les fonctionnalités que l'on pense les plus utiles ou les plus critiques. D'où le fort développement des méthodes itératives et agiles.

(4) Les bonnes pratiques pour la qualité, comme les revues et inspections ou le développement dirigé par les tests.

1.5 LA MATURITÉ DES ORGANISATIONS

À côté des aspects techniques, les aspects organisationnels jouent également un grand rôle pour le développement logiciel.

Le modèle CMMI (*Capability Maturity Model + Integration* [1]), mis au point depuis 1987 par le Software Engineering Institute (SEI) sur la base des travaux de Watts Humphrey, décrit les stades de maturité que peuvent atteindre les organisations qui développent du logiciel, en termes de *qualité de leur processus*. La norme distingue cinq niveaux de maturité croissants :

– *Initial* : L'organisation ne dispose pas de procédures standardisées ni de suivi de performance. Le succès dépend essentiellement des efforts et des compétences des individus.

– *Piloté* : Il y a un consensus dans l'organisation sur la manière dont les choses doivent être gérées, mais cela n'est ni formalisé ni écrit. Le rôle des chefs de projets est essentiel pour la capitalisation de l'expérience. Les coûts et les délais sont gérés.

– *Standardisé* : Le processus de développement est formalisé, documenté et appliqué. La capitalisation de l'expérience est centralisée. Une structure chargée de la qualité et des méthodes est en place.

1. La version la plus récente s'appelle CMMI-DEV et date de 2006.

– *Quantifié* : Les projets sont pilotés sur la base d'indicateurs objectifs de qualité des produits et des processus. Un processus formel de collecte des informations est en place.
– *Optimisé* : L'amélioration continue des processus devient une préoccupation centrale. L'organisation se donne les moyens d'identifier et de mesurer les faiblesses de ses processus. Une cellule de veille identifie puis met en œuvre les technologies innovantes et les pratiques d'ingénierie logicielle les plus efficaces.

Ces niveaux constituent les étapes conduisant à des processus matures, c'est-à-dire conformes aux meilleures pratiques observées à travers le monde dans les entreprises réputées bien gérer leurs processus.

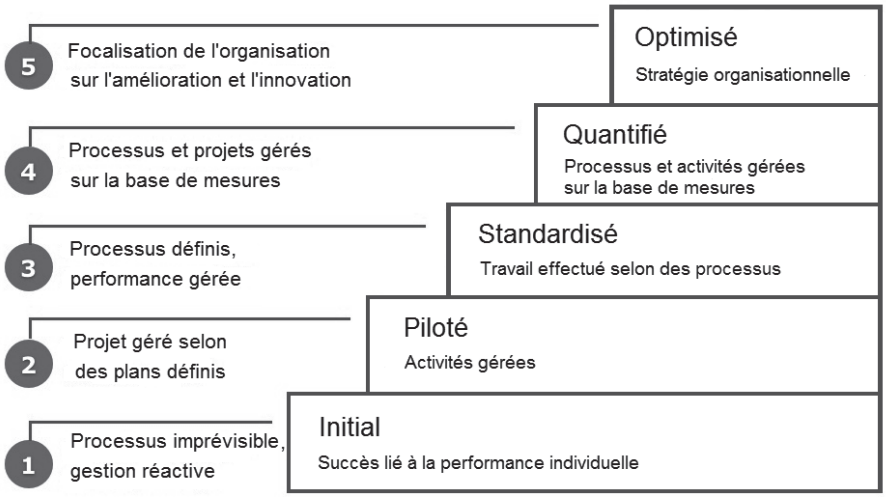


FIGURE 1.4 Les niveaux de maturité

Chaque niveau de maturité, sauf le niveau initial, définit une liste de thèmes majeurs à considérer.

– *Au niveau piloté*, on trouve la planification de projet, le suivi et la supervision de projet, la gestion de la sous-traitance, l'assurance qualité, la gestion de configuration. Au cœur de la thématique de cet ouvrage, on trouve aussi la *gestion des exigences* qui implique :
  - la compréhension et la prise en compte des exigences par les développeurs,
  - la gestion des évolutions des exigences et leur traçabilité,
  - la conformité des produits du projet aux exigences.
– *Au niveau standardisé*, on trouve la définition du processus organisationnel, la focalisation sur le processus organisationnel, la formation à l'organisation, les revues par les pairs, la coordination entre les groupes. Au cœur de la thématique de cet ouvrage, on trouve aussi le *développement des exigences* qui implique :

  - l'élucidation des souhaits et des besoins des clients et leur expression en exigences client,
  - la dérivation des exigences techniques sur l'application, ses composants et son interface,
  - l'analyse et la validation des exigences.

– *Au niveau quantifié*, on trouve la gestion quantitative des processus et de la qualité logicielle.

– *Au niveau optimisé*, on trouve la gestion des changements technologiques, l'innovation organisationnelle et la prévention des défauts.

Chaque thème majeur est décrit par un ensemble de bonnes pratiques qu'il convient d'adopter si l'on veut satisfaire à ses exigences.

Pour passer d'un niveau de maturité à l'autre, il faut obtenir une certification d'un organisme habilité par le SEI.

Peu d'entreprises françaises se sont engagées dans ce type de démarche de certification spécifique au développement logiciel. Ont été citées les sociétés de service Atos Origin, Axlog, Capgemini, Silogic, Sogeti, Steria et SQLI, en général aux niveaux 2 et 3. Beaucoup d'entreprises du secteur préfèrent s'engager dans des certifications qualité plus généralistes et moins lourdes, comme celles de la famille ISO 9000.

EXERCICES

Exercice 1.1. Les défaillances logicielles de systèmes complexes

Lire l'extrait suivant du communiqué officiel expliquant la désintégration de la première fusée Ariane 5. Analyser la cascade d'erreurs commise et l'inefficacité des mesures de sécurité.

```
 Communiqué de presse conjoint ESA-CNES
 Paris, le 19 juillet 1996

            Rapport de la Commission d'enquête Ariane 501
                      Echec du vol Ariane 5O1

Président de la Commission : Professeur J.-L. LIONS

Avant-propos
1. L'échec
  1. Description générale
  2. Informations disponibles
  3. Récupération des débris
  4. Autres anomalies observées sans rapport avec l'accident
2. Analyse de l'échec
  1. Séquence des événements techniques
  2. Commentaires du scénario de défaillance
  3. Procédures d'essai et de qualification
  4. Autres faiblesses éventuelles des systèmes incriminés
3. Conclusions
  1. Résultats de l'enquête
  2. Cause de l'accident
  4. Recommandations

                                          (...)
2. ANALYSE DE LÉCHEC
2.1. SEQUENCE DES EVENEMENTS TECHNIQUES
De manière générale la chaîne de pilotage d'Ariane 5 repose sur un concept
standard. L'attitude du lanceur et ses mouvements sont mesurés par un
système de référence inertielle (SRI) dont le calculateur interne calcule
les angles et les vitesses sur la base d'informations provenant d'une
plate-forme inertielle à composants liés, avec gyrolasers et accéléromètres.
Les données du SRI sont transmises via le bus de données, au calculateur
embarqué (OBC) qui exécute le programme de vol et qui commande les tuyères
des étages d'accélération à poudre et du moteur cryotechnique Vulcain, par
l'intermédiaire de servovalves et de vérins hydrauliques.

Pour améliorer la fiabilité, on a prévu une importante redondance au niveau
des équipements. On compte deux SRI travaillant en parallèle; ces systèmes
sont identiques tant sur le plan du matériel que sur celui du logiciel.
L'un est actif et l'autre est en mode "veille active"; si l'OBC détecte que
```

le SRI actif est en panne, il passe immédiatement sur l'autre SRI à condition que ce dernier fonctionne correctement. De même on compte deux OBC et un certain nombre d'autres unités de la chaîne de pilotage qui sont également dupliquées.

La conception du SRI d'Ariane 5 est pratiquement la même que celle d'un SRI qui est actuellement utilisé à bord d'Ariane 4, notamment pour ce qui est du logiciel.

Sur la base de la documentation et des données exhaustives relatives à l'échec d'Ariane 501 qui ont été mises à la disposition de la Commission, on a pu établir la séquence suivante d'événements ainsi que leurs interdépendances et leurs origines, depuis la destruction du lanceur jusqu'à la cause principale en remontant dans le temps.

Le lanceur a commencé à se désintégrer à environ HO + 39 secondes sous l'effet de charges aérodynamiques élevées dues à un angle d'attaque de plus de 20 degrés qui a provoqué la séparation des étages d'accélération à poudre de l'étage principal, événement qui a déclenché à son tour le système d'auto destruction du lanceur.

Cet angle d'attaque avait pour origine le braquage en butée des tuyères des moteurs à propergols solides et du moteur principal Vulcain; le braquage des tuyères a été commandé par le logiciel du calculateur de bord (OBC) agissant sur la base des données transmises par le système de référence inertielle actif (SRI2). A cet instant, une partie de ces données ne contenait pas des données du vol proprement dites mais affichait un profil de bit spécifique de la panne du calculateur du SRI2 qui a été interprété comme étant des données de vol.

La raison pour laquelle le SRI2 actif n'a pas transmis des données d'attitude correctes tient au fait que l'unité avait déclaré une panne due à une exception logicielle.

L'OBC n'a pas pu basculer sur le SRI1 de secours car cette unité avait déjà cessé de fonctionner durant le précédent cycle de données (période de 72 millisecondes) pour la même raison que le SRI2.

L'exception logicielle interne du SRI s'est produite pendant une conversion de données de représentation flottante à 64 bits en valeurs entières à 16 bits. Le nombre en représentation flottante qui a été converti avait une valeur qui était supérieure à ce que pouvait exprimer un nombre entier à 16 bits. Il en est résulté une erreur d'opérande. Les instructions de conversion de données (en code Ada) n'étaient pas protégées contre le déclenchement d'une erreur d'opérande bien que d'autres conversions de variables comparables présentes à la même place dans le code aient été protégées.

L'erreur s'est produite dans une partie du logiciel qui n'assure que l'alignement de la plate-forme inertielle à composants liés. Ce module

de logiciel calcule des résultats significatifs avant le décollage seulement. Dès que le lanceur décolle, cette fonction n'est plus d'aucune utilité.

La fonction d'alignement est active pendant 50 secondes après le démarrage du mode vol des SRI qui se produit à HO - 3 secondes pour Ariane 5. En conséquence, lorsque le décollage a eu lieu, cette fonction se poursuit pendant environ 40 secondes de vol. Cette séquence est une exigence Ariane 4 mais n'est pas demandé sur Ariane 5.

L'erreur d'opérande s'est produite sous l'effet d'une valeur élevée non prévue d'un résultat de la fonction d'alignement interne appelé BH (Biais Horizontal) et lié à la vitesse horizontale détectée par la plate-forme. Le calcul de cette valeur sert d'indicateur pour la précision de l'alignement en fonction du temps.

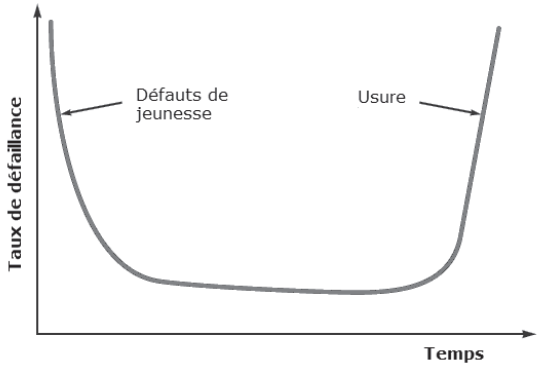
La valeur BH était nettement plus élevée que la valeur escomptée car la première partie de la trajectoire d'Ariane 5 diffère de celle d'Ariane 4, ce qui se traduit par des valeurs de vitesse horizontale considérablement supérieures.

Les événements internes du SRI qui ont conduit à l'accident ont été reproduits par simulation. En outre, les deux SRI ont été récupérés pendant l'enquête de la Commission et le contexte de l'accident a été déterminé avec précision à partir de la lecture des mémoires. De plus, la Commission a examiné le code logiciel qui s'est avéré correspondre au scénario de l'échec. Les résultats de ces examens sont exposés dans le Rapport technique.

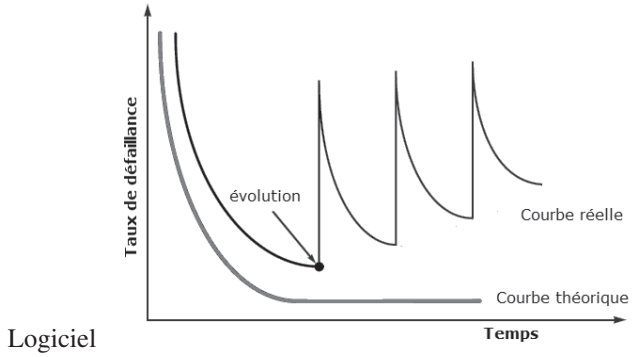
> On peut donc raisonnablement affirmer que la séquence d'événements ci-dessus traduit les causes techniques de l'échec d'Ariane 501.

Exercice 1.2. Les courbes de défaillance du matériel et du logiciel

Commenter les courbes suivantes, tirées de [Pre09], qui donnent l'évolution au cours du temps du nombre de défaillances respectivement du matériel et du logiciel.



Matériel



Exercice 1.3. La diversité des applications

Donner les principales qualités attendues :
a) d'un site de commerce électronique,
b) d'un système temps réel (par exemple de guidage au sol d'une fusée),
c) d'un système embarqué (par exemple de contrôle du freinage d'une voiture).

Exercice 1.4. *The mythical man-month*

L'homme-mois est souvent utilisé comme unité de mesure de l'effort de développement. Frederick Brooks a critiqué cette unité dans son ouvrage *The mythical Man-Month* [Bro75]. L'hypothèse qu'un homme pendant deux mois équivaut à deux hommes pendant un mois, les deux valant deux hommes-mois, est le plus souvent fausse. Expliquer pourquoi en commentant les figures suivantes.