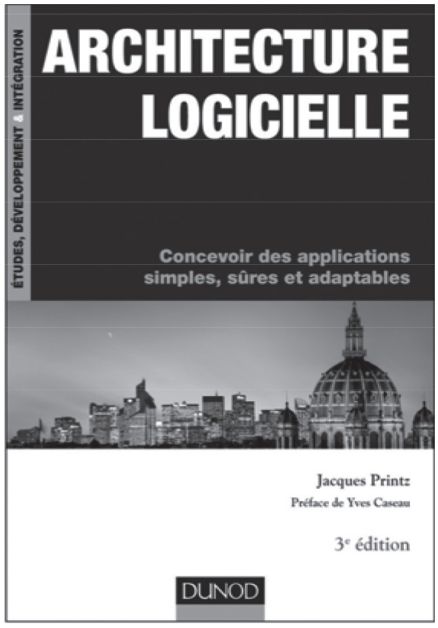
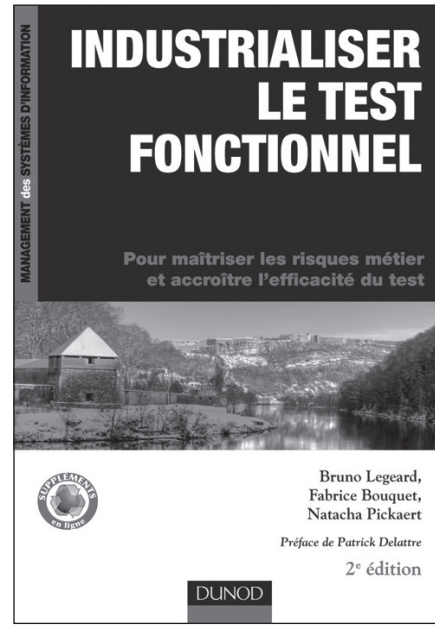
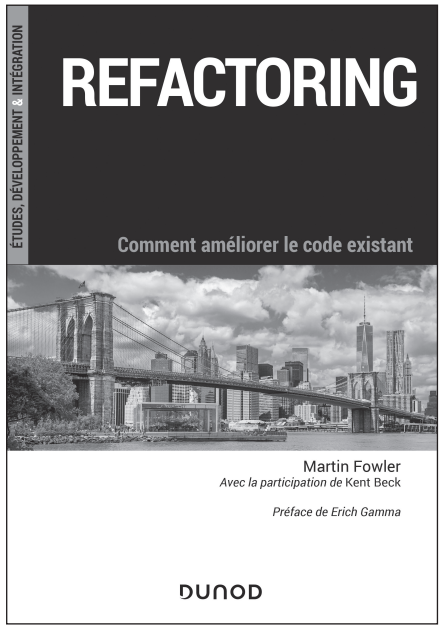
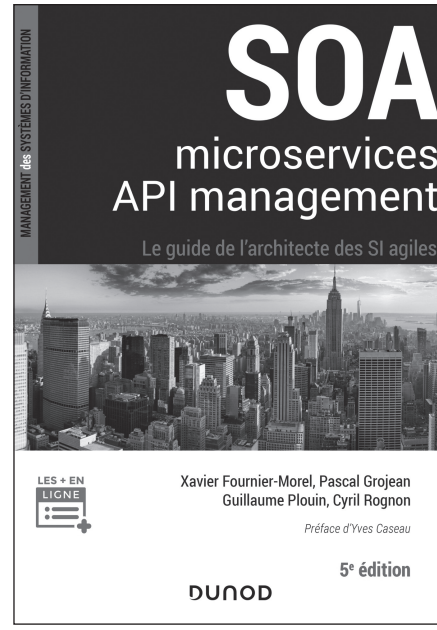
PRATIQUE
DES
TESTS LOGICIELS

Améliorer la qualité par les tests
Gérer une campagne de test
Préparer la certification ISTQB


ÉTUDES, DÉVELOPPEMENT & INTÉGRATION
ARCHITECTURE LOGICIELLE
Concevoir des applications simples, sûres et adaptables
Jacques Printz
Préface de Yves Caseau
3e édition
DUNOD


MANAGEMENT des SYSTÈMES D'INFORMATION
INDUSTRIALISER LE TEST FONCTIONNEL
Pour maîtriser les risques métier et accroître l'efficacité du test
SUPPLÉMENT en ligne
Bruno Legeard,
Fabrice Bouquet,
Natacha Pickaert
Préface de Patrick Delattre
2e édition
DUNOD


ÉTUDES, DÉVELOPPEMENT & INTÉGRATION
REFACTORING
Comment améliorer le code existant
Martin Fowler
Avec la participation de Kent Beck
Préface de Erich Gamma
DUNOD


MANAGEMENT des SYSTÈMES D'INFORMATION
SOA
microservices
API management
Le guide de l'architecte des SI agiles
LES + EN LIGNE
Xavier Fournier-Morel, Pascal Grojean
Guillaume Plouin, Cyril Rognon
Préface d'Yves Caseau
5e édition
DUNOD

PRATIQUE DES TESTS LOGICIELS

Améliorer la qualité par les tests
Gérer une campagne de test
Préparer la certification ISTQB

Jean-François Pradat-Peyre
Professeur à l'université Paris Nanterre et chercheur au LIP6 UMR 7606 Sorbonne Université – CNRS

Jacques Printz
Professeur émérite du Cnam

Préface de Klaus Lambertz
Fondateur et PDG de Verify Technology GmbH

Postface de Claude Kaiser
Professeur émérite au CNAM X57 Ancien ingénieur du Génie maritime Fondateur du département de mathématiques et d'informatique au CNAM et du laboratoire de recherche CEDRIC (cedric.cnam.fr)

4e édition

DUNOD

Illustration de couverture : © Ekaterina Pokrovsky, Shutterstock

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-081995-9

Table des matières

Préface

L'importance de la qualité des logiciels et par conséquent celle des tests logiciels, ne cesse de croître. Nos clients à travers le monde travaillent sur des projets dans lesquels la qualité du logiciel est indispensable, sur des projets « sensibles » ou même critiques, sur des cibles embarquées même très petites. Le test est bien intégré dans leur processus de développement. Ils ont besoin des outils de tests matures, qui sont qualifiés ou certifiés pour le développement des logiciels « critiques ».

Pour les logiciels sensibles, des normes ont été mises en place. Le précurseur était – sans surprise – le secteur aéronautique. La première version de la norme DO 178 « Réglementation pour le développement de logiciels dans le secteur aéronautique » a été établie dans les années 1980. « Les dépenses pour la qualité et les tests pouvant atteindre jusqu'à 95 % du budget d'un projet de développement logiciel » *dixit* un responsable qualité d'un constructeur d'avions.

Plus tard, d'autres secteurs se sont dotés des normes afin de se donner des directives pour la partie test. Pour la construction automobile – le secteur le plus important en Europe pour les logiciels embarqués – la norme ISO 26262 « Véhicules routiers – Sécurité fonctionnelle » règle la façon dont les logiciels doivent être développés et testés afin de garantir la sécurité indispensable. La première version de cette norme, une adaptation de la norme CEI 61508, prenant en compte les spécificités du secteur automobile, a été publiée fin 2011.

Il y a des années, quand le test était encore considéré comme peu utile voire inutile par certains développeurs – il était très fréquent d''entendre des phrases comme « C'est notre client qui nous dit lorsque quelque chose ne fonctionne pas ». Depuis, les normes ont considérablement changé la donne. Comme la DO 178 pour l'aéronautique, l'ISO 26262 est très claire et stricte sur des procédures à suivre et des mesures de qualité à effectuer. En cas d'accident, les preuves concernant les mesures de qualité peuvent être demandées des années après la livraison des logiciels. Pour cette raison, certains de nos clients gardent les rapports de couverture de tests pendant 30 ans.

Les secteurs ferroviaire (EN-50128 « Applications ferroviaires – Systèmes de signalisation, de télécommunication et de traitement »), nucléaire (CEI 60880) et médical (CEI/EN 62304 « Logiciels de dispositifs médicaux – Processus du cycle de vie du logiciel ») disposent des normes de qualité similaires, mais adaptés aux spécificités des industries respectives. Le point commun à toutes ces normes est : plus le risque est élevé, plus les exigences concernant les tests sont élevées.

Toutes ces normes exigent – parmi d'autres mesures pour améliorer la qualité – une analyse statique et la preuve par la couverture de tests.

Les sociétés qui développent des logiciels « critiques » n'ont donc plus le choix aujourd'hui : le test est une obligation et fait partie intégrante du processus de développement d'un logiciel.

Le test est également important pour d'autres secteurs. Certes, la majorité de nos clients travaille dans le secteur dit « critique », mais ces dernières années nous avons de plus en plus d'utilisateurs de nos outils de test dans des secteurs qui ne sont pas soumis

aux normes, par exemple : l'agriculture en pleine mutation avec ses innovations technologiques. De plus en plus de solutions de récolte, des nettoyeurs d'étable, des systèmes d'alimentation automatisés et des robots de traite sont utilisés.

« Le bon fonctionnement des logiciels est primordial dans les automates pour nourrir ou traire les vaches. Les animaux qui reçoivent trop ou trop peu de nourriture peuvent tomber malade – l'agriculteur qui, après un défaut logiciel, devra traire mille vaches en pleine nuit "manuellement" subira certainement de lourdes conséquences pour la gestion de son exploitation » m'a confié un de nos clients, spécialiste d'innovations technologiques pour l'agriculture.

Notre expérience est clairement dans le test des systèmes embarqués, donc des logiciels « techniques ». Grâce aux retours d'expériences de nos plus de 600 clients, dans 40 pays dans le monde qui utilisent nos outils de test, nous pensons avoir une image précise des démarches à suivre pour produire un « bon logiciel ». Depuis 2003 nous avons également vu beaucoup d'erreurs à ne pas commettre – parfois drôles, parfois catastrophiques, toujours coûteuses. Il serait intéressant et amusant pour le lecteur de décrire ces histoires, mais cela ne serait évidemment pas dans l'intérêt des sociétés concernées. Chose positive : après ces déboires, nous observons ces dernières années, une forte croissance des efforts de nos clients pour se diriger vers des processus exemplaires. Nous sommes heureux de constater qu'il y a de plus en plus de « bons élèves » dans l'industrie – aussi en dehors des secteurs « critiques » comme l'aérospatiale et la défense, l'automobile et le transport.

Une bonne qualité des logiciels est donc indispensable. Le test n'est pas une source de coût ou un mal nécessaire, mais – quand il est fait dans les règles – un levier pour réduire les risques, augmenter la qualité et surtout pour économiser les dépenses dues aux mauvais fonctionnements des logiciels.

Cette 4e édition de la *Pratique des Tests Logiciels* fournit les bases nécessaires afin d'établir une stratégie pour le test et pour faire des tests efficaces.

Je recommanderai, à tous les acteurs et futurs acteurs de projet informatique, de lire sans modération ce livre.

Klaus LAMBERTZ
Fondateur et PDG de Verifysoft Technology GmbH

Avant-propos

Les logiciels informatiques, indispensables au fonctionnement des entreprises et des infrastructures technologiques, ont pris une place essentielle dans notre vie quotidienne : ils améliorent la qualité des images de nos appareils photos, gèrent nos annuaires téléphoniques mais participent aussi à la sécurité de nos trajets automobiles. Si le dysfonctionnement d'un appareil photo peut être désagréable, l'arrêt de l'ABS ou du contrôle dynamique de trajectoire peut avoir des conséquences dramatiques[1]. La société dite « numérique » est en fait une société où la matière première est le logiciel, une société où les programmeurs se comptent en millions et les lignes de code qu'ils écrivent en milliards. C'est une production de masse ! Le monde du développement de logiciels est donc confronté à de nouveaux défis où l'innovation ne peut se faire au détriment de la qualité. On attend, bien entendu, des logiciels qu'ils réalisent ce pourquoi ils ont été conçus mais également qu'ils ne fassent pas ce pourquoi ils n'ont pas été conçus. Créés par l'homme, les logiciels sont soumis aux aléas de l'activité humaine et de ce fait, contiennent des défauts qu'il faut s'efforcer de détecter et de corriger au plus tôt.

Ceci ne peut se faire sans tester régulièrement le logiciel et/ou les parties qui le composent. Régulièrement car il ne suffit pas de tester uniquement le produit final. En effet, une erreur découverte en bout de chaîne peut entraîner la refonte totale et le développement de tout ou de presque tout. Il faut donc tester en amont lors des phases d'assemblages des composants du logiciel (il s'agit des tests dits d'intégration), mais également lors du développement des composants (il s'agit des tests dits unitaires) car assembler des composants truffés d'erreurs ne peut que compliquer le diagnostic et réduire à néant le rendement de l'effort des tests suivants.

Les tests doivent donc rechercher des erreurs de conception ou de réalisation : des erreurs dites fonctionnelles, le logiciel ne fait pas ce qu'il devrait faire ; mais aussi des erreurs non fonctionnelles : le logiciel fait ce qu'il faut mais pas dans des temps acceptables ou en devant être relancé fréquemment ou ne supportant pas la montée en charge. Le nombre de cas d'utilisation possibles d'un logiciel étant en général très grand, il est tout à la fois illusoire de penser mener cette recherche d'erreur de manière empirique, mais également de vouloir prétendre à l'exhaustivité ; la bonne approche sera de nature statistique, avec des stratégies basées sur les contextes d'emploi. Il faudra savoir construire efficacement les cas de tests pertinents, c'est-à-dire ceux permettant de mettre en évidence rapidement les erreurs commises lors des différentes phases de conception. Il faudra également savoir déterminer à quel moment on peut arrêter les tests sans crainte de ne pas avoir assez testé afin de garantir la qualité de service conforme au contrat de service. Ces jeux de tests peuvent être construits à l'aide des textes sources du logiciel et/ou des détails de son assemblage ou, uniquement à l'aide de ses spécifications. On parlera, selon les cas, de tests *Boîtes blanches* ou *Boîtes noires*.

On le voit, tester un logiciel est un challenge aussi excitant que de le programmer, et qui, tout comme la programmation, ne peut se faire sans une bonne assise technique ;

---

1. On pourra également penser aux énormes soucis humains créés par le dysfonctionnement du logiciel de gestion de paie de l'armée « Louvois » mis en place en 2011.

tester c'est mener une expérimentation scientifique et il faut pour cela enthousiasme mais aussi rigueur et méthode. Les méthodes les plus récentes comme les méthodes dites « agiles » insistent à juste titre sur l'importance des tests et sur le développement guidé par les tests, c'est-à-dire le « *Test Driven Development* » (les tests sont conçus avant la réalisation).

◆ *À qui s'adresse ce livre ?*

Ce livre a un triple objectif. Tout d'abord, il vise à donner aux étudiants des universités et des grandes écoles d'ingénieurs, c'est-à-dire aux futurs concepteurs, développeurs, intégrateurs de logiciels ou aux futurs chefs de projets, les bases indispensables pour concevoir et mener à bien les tests tout au long du cycle de vie du logiciel et du système. Deuxièmement, ce livre vise à donner aux équipes de testeurs une référence en termes de vocabulaire, de méthodes et de techniques permettant un dialogue plus efficace entre les donneurs d'ordre, les maîtrises d'œuvre et les maîtrises d'ouvrage. Enfin, conforme au Syllabus niveau fondation de l'ISTQB et posant une vraie réflexion sur la gestion des tests et les tests d'intégration, cet ouvrage prépare au passage de la certification ISTQB du métier de testeur : niveau fondation mais également niveau avancé.

Cette quatrième édition, entièrement revue et mise à jour, reprend ces objectifs tout en détaillant plus en profondeur certaines parties présentes dans les précédentes éditions : automatisation des tests, test des systèmes embarqués, amélioration des processus de tests avec le référentiel TMMi. Cette édition contient également une postface du Professeur Claude Kaiser (X'57) qui détaille la naissance des tests logiciels dans la mise au point du logiciel de commande du périscope de visée astrale dans les années 1960, utilisé par la suite dans le sous-marin nucléaire le Redoutable.

◆ *Suppléments en ligne*

Retrouvez sur *www.dunod.com* les suppléments en ligne qui complètent cet ouvrage. Il s'agit d'exercices corrigés, du corrigé commenté du QCM, de compléments sur les tests unitaires avec différents outils, des informations de mise à niveau, des applications avancées, des logiciels recommandés, des supports didactiques…

Quelques idées essentielles sur les tests

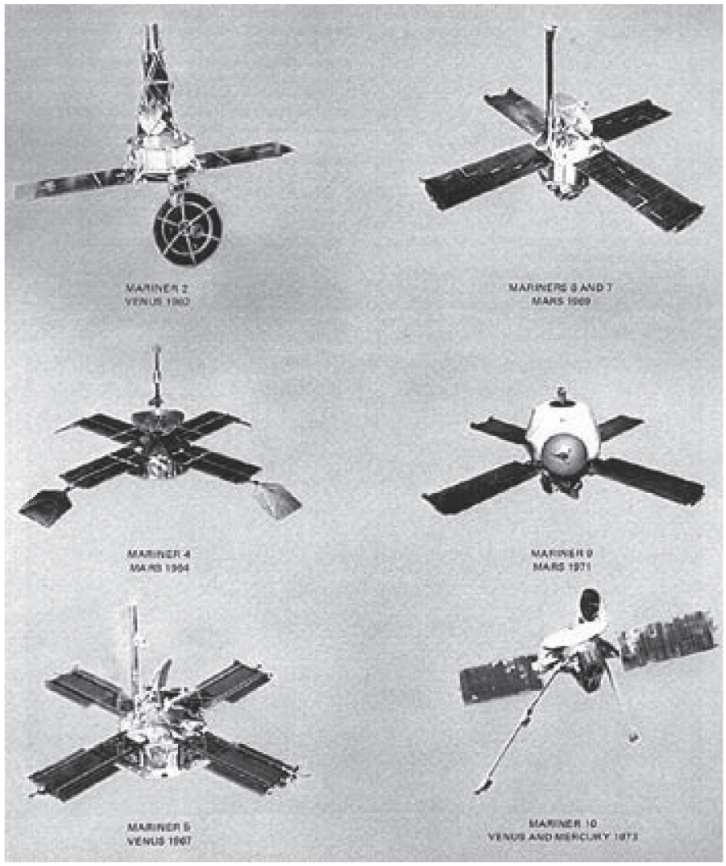
Le **samedi** 21 juillet 1962, la première sonde spatiale interplanétaire du programme Mariner de la NASA, Mariner I, est lancée depuis Cap Canaveral pour une mission d'analyse de Vénus. Quatre minutes après son lancement, le lanceur dévie de sa trajectoire et doit être détruit.

Un article du *New York Times* daté du 27 juillet 1962 relate cet épisode malheureux de la conquête de l'espace et donne une première explication de cet échec : « *The hyphen, a spokesman for the laboratory explained, was a symbol that should have been fed into a computer, along with a mass of other coded mathematical instructions. The first phase of the rocket's flight was controlled by radio signals based on this computer's calculations. The rocket started out perfectly on course, it was stated. But the inadvertent omission of the hyphen from the computer's instructions caused the computer to transmit incorrect signals to the spacecraft…* »

L'explication donnée à l'époque de cet échec et maintenue jusqu'à récemment est qu'une instruction du programme de guidage (écrit en Fortran) contenait une virgule à la place d'un point. L'instruction « *DO 10 I = 1.100* » aurait dû être « *DO 10 I = 1,100* ». Dans le premier cas, il s'agit d'une déclaration de variable de nom « *DO 10 I* » de type réel auquel on donne la valeur 1,1 alors que dans le second cas il s'agit d'une boucle qui répète de 100 fois la suite d'instructions qui suit la ligne ; la différence de comportement résultant de l'interprétation de ces deux instructions est radicale !

En fait, la cause du problème la plus probable est plus subtile car elle proviendrait non d'une erreur de codage mais d'une erreur d'interprétation des spécifications : la transcription manuelle du symbole surligné ($\overline{a}$) sur une variable, écrit rapidement dans la marge des spécifications, correspondant au lissage de la variable en question, fut pris pour une apostrophe, ('a), notant la dérivée de la variable. Il s'agirait donc d'une apostrophe au lieu d'une barre et non d'une virgule au lieu d'un point.

Figure 1.1 – Les sondes Mariner (image NASA)
http://upload.wikimedia.org/wikipedia/commons/7/7f/Mariner_1_to_10.jpg



Cette confusion conduisit à un code non conforme aux spécifications initiales ce qui s'est traduit par une différence de comportement importante entre la version codée et la version souhaitée. Ainsi, lors des tentatives de stabilisation du lanceur, le système de contrôle lui envoya des ordres incohérents causant sa perte.

Quelle que ce soit la version de ce terrible échec parmi ces deux hypothèses, il est la conséquence d'une erreur de réalisation d'un logiciel informatique : une erreur d'interprétation des spécifications ou une erreur de codage.

De nombreux autres et malheureux exemples sont là pour nous rappeler l'importance du logiciel dans le monde actuel ; nous ne citons que les plus (tristement) célèbres :

- ✓ convocation à l'école de personnes âgées de 106 ans. Cause : codage de l'âge sur deux caractères ;
- ✓ navire de guerre anglais coulé par un Exocet français, au cours de la guerre des Malouines, le vaisseau anglais n'ayant pas activé ses défenses. Plusieurs centaines de morts. Cause : les missiles de type Exocet n'étaient pas répertoriés comme des missiles ennemis ;
- ✓ passage de la ligne : au passage de l'équateur un F16 se retrouve sur le dos. Cause : changement de signe de la latitude mal pris en compte ;

- ✓ station MIR : deux jours sans courant (14 au 16 novembre 1999). Cause : arrêt d'un ordinateur qui contrôlait l'orientation des panneaux solaires ;
- ✓ hôpital : décès d'un malade. Cause : erreur logicielle dans un programme de contrôle d'appareil de radiothérapie ;
- ✓ missile : en URSS, fusée pointant Hambourg au lieu du Pôle Nord. Cause : erreur de signe entraînant une erreur de 180° du système de navigation ;
- ✓ inondation de la vallée du Colorado en 1983. Cause : mauvaise modélisation du temps d'ouverture du barrage ;
- ✓ perte de la sonde Mars Climate Orbiter (anciennement Mars Surveyor Orbiter) le 23 septembre 1999 après 9 mois de voyage. Cause : confusion entre pieds et mètres ;
- ✓ et bien d'autres dont tout un chacun peut être témoin dans sa vie professionnelle ou familiale (penser par exemple au nombre de fois où il est nécessaire de redémarrer un smartphone pour retrouver un fonctionnement normal).

Dans tous ces cas, une erreur de réalisation ou de conception d'un logiciel conduit un système automatique à faire autre chose que ce pour quoi il est fait. Différents termes sont employés pour relater ces problèmes et il nous faut préciser ici ce que l'on entend par erreur, défaut, défaillance ou panne.

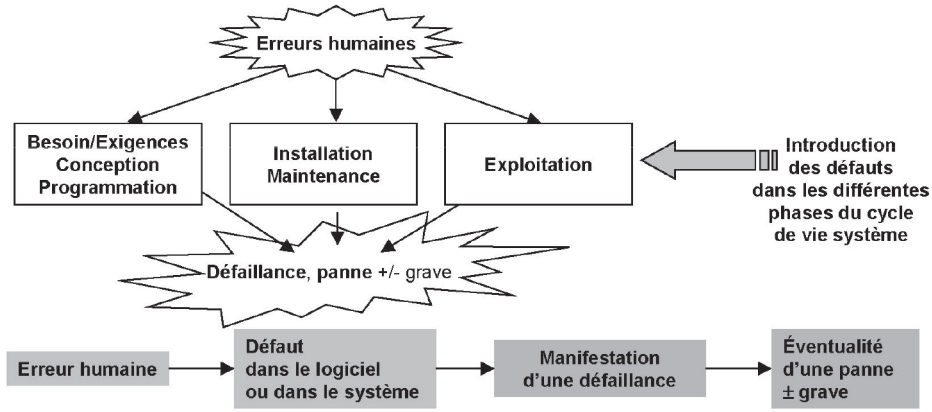
1. 1 CHAÎNE DE L'ERREUR

Comme toute activité humaine, la conception et le développement de logiciel sont sujets à des imperfections qui peuvent être sans conséquences dans certains cas et dans d'autres cas, conduire à des comportements non prévus et être la cause de dysfonctionnements plus ou moins graves. En tout état de cause, lorsque les tests sont correctement réalisés et utilisés, ils permettent de découvrir des erreurs. Si l'on veut être précis, les tests mettent en avant des défaillances du logiciel, c'est-à-dire des fonctionnements anormaux aux vues de ses spécifications. Une défaillance provient d'un défaut de réalisation ou de conception du logiciel, qui à la suite d'une exécution du code impliqué engendre un comportement fautif. Il faut noter que tout défaut ne conduit pas systématiquement à une défaillance et que, au contraire, il est fréquent qu'un logiciel se comporte correctement alors qu'il contient un grand nombre de défauts mais qui ne sont jamais exercés en fonctionnement ; cette constatation implique donc d'adopter une stratégie de grande prudence lorsque l'on réutilise une partie d'un logiciel fonctionnant parfaitement.

Ces défauts présents dans les logiciels proviennent d'erreurs humaines qui sont le fait de méprises sur la compréhension ou l'interprétation des spécifications ou encore d'erreurs de réalisation lors du codage ou également d'oublis lors des phases de conception.

Une suite de défaillances peut entraîner une panne du logiciel ou du système, mais il ne faut pas confondre une défaillance et une panne. Une défaillance se caractérise par des résultats inattendus (calculs erronés, fenêtre mal placée, message non affiché, etc.) ou un service non rendu (données non stockées dans une base, fenêtre non ouverte, etc.) ; cependant le logiciel peut continuer bon gré mal gré son fonctionnement normal. Une panne a un caractère plus franc et se révélera par un arrêt total ou partiel du logiciel qui conduit à un fonctionnement en mode (très) dégradé. La seule manière de sortir de ce mode est de redémarrer le logiciel ou le système avec toutes les conséquences que cela peut avoir.

Figure 1.2 – La chaîne de l'erreur



Afin de mesurer (*a posteriori*) le fonctionnement sans défaillance ou panne d'un logiciel et l'impact de ces défaillances ou panne sur la disponibilité du logiciel deux indicateurs sont utilisés :

- ✓ le MTBF (*Mean Time Between Failure*) qui définit le temps moyen de bon fonctionnement entre deux défaillances ou pannes, temps de réparation compris. Ce nombre doit être le plus grand possible ;
- ✓ le MTTR (*Mean Time To Repair*) qui définit le temps moyen de remise en route après une panne.

Il faut noter que seul un modèle précis, et donc difficile à produire, permet d'essayer de calculer *a priori* ces indicateurs. Le plus souvent ils sont estimés à l'aide d'anciens logiciels ou des logiciels concurrents, ce qui donne une idée de ce qui est possible, et ils sont affinés en fonction des exigences métiers, ce qui donne un objectif de ce qu'il faudrait atteindre.

1. 2 RÔLE DES TESTS

Le rôle des tests est multiple. En effet, aujourd'hui, de très nombreuses activités ne peuvent être réalisées sans l'aide de logiciels et ceci dans des domaines très variés : gestion de la paye, gestion de la carrière des personnels, suivi des clients, contrôle des centrales nucléaires, aide au pilotage des avions civils et militaires, amélioration du fonctionnement des appareils ménagers, services offerts sur les mobiles ou les tablettes, etc. *« Il y a plus d'informatique dans la Volvo S80 que dans le chasseur F15 »* déclarait en janvier 2000 Denis Griot, responsable de l'électronique automobile chez Motorola.

Or, on ne sait pas, par construction, fabriquer des logiciels sans défaut : l'homme commet des erreurs et aucun programme ne peut générer de façon sûre un autre programme ou vérifier qu'un programme fait exactement ce pour quoi il est fait. En effet, les limites théoriques montrent qu'il est impossible dans le cas général, de construire un algorithme permettant de dire si deux programmes réalisent les mêmes fonctions (il s'agit d'un problème indécidable). La base de ce résultat provient de l'impossibilité à fabriquer un programme qui statue, sans se tromper, sur la terminaison d'un programme quelconque. Si tel était le cas, et si l'on notait Stop ce programme et Stop(P), le résultat de l'analyse

de la terminaison d'un programme P par Stop, on pourrait construire un programme B contenant l'instruction « si Stop(B) alors rentrer dans une boucle sans fin, sinon arrêter ». Ce programme B mettrait en défaut systématiquement ce programme oracle Stop.

Devant ce constat d'impossibilité, différentes approches sont possibles :

1. se limiter à construire des programmes très simples que l'on sait analyser de façon certaine mais qui ne pourront résoudre que des problèmes limités ;

2. construire des programmes complexes dont on ne pourra prédire de façon exacte le comportement mais qui permettront dans la plupart des cas de résoudre des problèmes ambitieux.

Comme les besoins en termes de logiciels évoluent plus vite que les possibilités théoriques de construction sûre, l'approche 2 est celle qui est choisie dans la majorité des cas. Il faut donc accepter que le logiciel produit contienne des défauts ; l'enjeu est alors d'éliminer les défauts qui conduisent à des défaillances avant que le logiciel entre en service. Si nécessaire, il faudra également s'assurer que le logiciel satisfait un certain nombre de normes légales ou contractuelles, nécessaires par exemple pour une certification du logiciel. Enfin, une fois le logiciel déployé, il faudra, lors des phases de maintenance, vérifier que les évolutions ou améliorations n'ont pas entamé les parties non modifiées du logiciel et, dans le cas de maintenances correctives, que la nouvelle version corrige bien les défauts de l'ancienne. Toutes ces étapes sont réalisées à l'aide de différents types de tests : tests unitaires ou tests d'intégration lors des phases de développement, tests système et tests de validation lors des phases de déploiement, tests d'acceptation et tests de recette lors des phases d'acceptation ou de certification, et enfin tests de correction et de non-régression lors des phases de maintenance.

On le voit, les objectifs des tests peuvent être multiples ; cependant, en aucun cas, et même si cela est choquant, il faut être conscient que leur but n'est pas d'éliminer tous les défauts. L'objet principal est d'analyser le comportement d'un logiciel dans un environnement donné : ainsi, il ne sert *a priori* à rien de tester le bon fonctionnement de la gestion du système de freinage d'une automobile pour une vitesse de 1 000 km/h.

Par ailleurs, les tests vont permettre de découvrir un certain nombre d'erreurs, mais il est faux de penser que l'amélioration de la fiabilité est proportionnelle au nombre de défauts détectés puis corrigés. En effet, un logiciel mal conçu, en particulier du point de vue de son architecture, va faire apparaître un grand nombre de défauts. La correction de ces défauts pourra avoir pour conséquence, non de réduire leur nombre, mais au contraire d'en créer de nouveaux. Si le ratio entre le nombre de défauts créés et le nombre de défauts corrigés est supérieur à 1, les tests ne cesseront de découvrir des défauts sans pour autant qu'il y ait convergence vers un logiciel fiable. De même, tester beaucoup en quantité et en temps n'est pas nécessairement un gage de qualité. Exercer sans cesse le même code avec des jeux de valeurs similaires ne donne aucune garantie quant à la capacité à découvrir des défauts de comportement. Tester est une activité complexe qui demande expérience et méthode et nous allons essayer maintenant de cerner les quelques principes qui doivent guider toute activité de test.

1. 3 LES SEPT PRINCIPES GÉNÉRAUX DES TESTS

Quelques grands principes nous permettent de mieux cerner et de définir intuitivement ce que sont et ce que ne sont pas les tests.

1.3.1 Principe 1 – Les tests montrent la présence de défauts

Il est important de se convaincre que les tests ne peuvent pas prouver l'absence d'erreur de conception ou de réalisation. Leurs objets au contraire sont de mettre en évidence la présence de défaut ; aussi, lorsqu'une série de tests, conçus et appliqués avec méthode, ne trouve aucun défaut, il est important de se poser la question de la pertinence des jeux de tests avant de conclure à une absence de défauts. En tout état de cause, si aucun défaut n'est découvert, ce n'est pas une preuve qu'il n'en reste pas.

1.3.2 Principe 2 – Les tests exhaustifs sont impossibles

Sauf pour des cas triviaux, la combinatoire d'un programme est impossible à explorer de façon exhaustive. En effet, du fait des différentes valeurs possibles des paramètres des méthodes ou sous-programmes et de la combinatoire engendrée par l'algorithmique, la plupart des programmes peuvent atteindre rapidement un nombre d'états différents supérieurs au nombre estimé d'atomes dans l'univers observable (de l'ordre de 10^{80}). Pour mieux nous rendre compte de la complexité intrinsèque d'un programme informatique, considérons une fonction d'addition de deux nombres entiers ; on ne peut plus simple. Si les entiers sont des valeurs codées sur 32 bits, chaque paramètre peut prendre 2^{32} valeurs distinctes (un peu plus de 4 milliards). Le nombre de cas différents d'exécution de cet additionneur est donc égal à 2^{64}, ce qui correspond à plus de 4 milliards de milliards. Si l'on veut tester « naïvement » tous les cas possibles, il faudrait, à raison de 1 milliard d'opérations de test par seconde, près de 136 années de travail (ce qui est impossible et idiot) !

1.3.3 Principe 3 – Tester tôt

Les activités de tests doivent commencer le plus tôt possible dans le cycle de développement du logiciel ou du système. Elles doivent être focalisées sur des objectifs définis compatibles avec les risques et les exigences de qualité. En effet, plus on retarde les activités de tests, plus le coût associé aux erreurs est important : celles-ci sont de plus en plus difficiles à localiser et impactent une partie du logiciel de plus en plus importante. De plus, la difficulté de leurs corrections augmente avec le temps de détection et donc le coût et l'effort associés. Les études menées sur le coût associé à la détection d'une erreur montre que si une erreur décelée lors de la phase de l'élaboration du cahier des charges coûte 1 alors la même erreur décelée en phase de conception coûte 10 et une erreur décelée en phase d'exploitation coûte 100. Il est donc important d'essayer de détecter au plus tôt les erreurs commises lors des phases de conception puis de développement. Ces erreurs ne pourront être découvertes que si le management a donné les moyens en dégageant du temps et des ressources aux équipes concernées par les activités de test. Ainsi, les efforts consacrés à la maintenance corrective pourront être reportés sur la conception et le développement, ce qui aura pour effet d'améliorer la réactivité de ces équipes face aux demandes des clients plaçant les tests au cœur d'une dynamique vertueuse.

1.3.4 Principe 4 – Regroupement des défauts

Comment porter l'effort de test ? Voici une question importante devant une tâche complexe et dont le succès dépend en grande partie de la méthode employée et des efforts consentis. Une première possibilité, *a priori* pleine de bon sens, est de répartir uniformément l'effort

de test sur l'ensemble du logiciel afin d'essayer de n'oublier aucune part de celui-ci. Cependant, le bon sens et les évidences sont souvent mis en défaut par une analyse poussée de la réalité. Ainsi, si l'on cartographie les défauts au sein d'un logiciel, on constate qu'une grande partie de ceux-ci se concentre dans une petite part du logiciel : c'est la règle des « 80/20 » qui énonce que 80 % des défauts sont localisés dans 20 % du logiciel. Aussi, l'effort de test devrait respecter cette répartition et porter à 80 % sur la partie qui contient la majorité des erreurs et à 20 % sur le reste du logiciel. Malheureusement, cette répartition n'est réellement connue qu'après avoir conduit la campagne de test ; il faut donc se baser sur un modèle d'erreurs pour essayer de prédire quelle part du logiciel contient la majorité des défauts. Bien que chaque logiciel soit unique, il existe des domaines qui, par nature, présentent des difficultés de réalisation et auxquels on prêtera plus attention : on peut citer les composants transactionnels, multitâches ou temps réel. Chacun, à sa façon, présente des difficultés de conception ou de réalisation dues en grande partie à la concurrence d'accès à des zones mémoires communes, à la difficulté de synchroniser des entités concurrentes et aux contraintes induites par des préoccupations système (au sens système d'exploitation) qui font remonter au niveau de l'application des problèmes et des contraintes de bas niveaux techniques et difficiles à maîtriser.

1.3.5 Principe 5 – Le paradoxe du pesticide

La question « Quand doit-on arrêter de tester ? » est tout aussi essentielle que la question « Où et comment porter l'effort de test ? ». En particulier après avoir testé longtemps, peu à peu, le nombre d'erreurs découvertes va en décroissant. C'est un fait normal de convergence qui doit être constaté quand le logiciel est conçu sur une architecture correcte. Cependant, il faut prendre garde que le nombre d'erreurs découvertes peut décroître du fait de la cible des techniques de tests utilisées. En effet, tout test vise à découvrir un type d'erreur donnée, même s'il peut parfois découvrir des erreurs pour lesquelles il n'est pas conçu. Par exemple, un test fonctionnel de la partie interface graphique d'un système de réservation de places de trains a peu de chance de découvrir une erreur concernant le respect de contraintes portant sur la prise en compte des réductions offertes pour les familles de la SNCF. De même, une série de tests aux limites permettra de trouver des erreurs sur le comportement d'un composant lorsqu'on le fait fonctionner à ses limites mais pas nécessairement sur des erreurs de comportement dans sa partie nominale. Aussi, au bout d'un moment, la même série de tests ne permettra plus de découvrir de nouveaux défauts ; non qu'elle soit inadaptée mais parce qu'elle a permis de dénicher tous les défauts pour lesquels elle était faite. Toute méthode de tests laisse un résidu d'erreurs contre lesquelles elle est inefficace Ce principe est connu sous le nom du « paradoxe du pesticide » : à trop appliquer le même produit, il devient inefficace. Il faut donc constamment renouveler les tests en changeant de point de vue (objectif et stratégie de test) afin de maintenir une bonne efficacité des tests menés.

1.3.6 Principe 6 – Les tests dépendent du contexte

Certains logiciels peuvent être utilisés dans des contextes très différents. Par exemple un système d'information pour un poste de commandement militaire peut être utilisé pour des missions très différentes : maîtrise de la violence dans une zone « pacifiée » ou aide au déploiement sur une zone de combats de haute intensité, par exemple. De même, un système de gestion de base de données peut être utilisé dans un logiciel de commerce électronique pour mémoriser des transactions bancaires ou dans un système de contrôle

aérien pour permettre une identification d'un contact radar. En fonction de l'utilisation du composant, les objectifs et les types de tests ne seront pas forcément les mêmes : tests fonctionnels, tests de performances, test de la sécurité, etc. ; de même, l'environnement d'exécution, la possibilité ou non d'arrêt brutal du composant vont modifier drastiquement les objectifs et moyens des tests mis en œuvre. L'oubli de ce principe fondamental peut entraîner de graves conséquences sur la qualité du logiciel produit. On pourra ainsi se souvenir de l'exemple malheureux du lanceur européen Ariane 5, qui fut détruit le 4 juin 1996 lors de son premier vol (vol 501) causant un discrédit sur le programme Ariane ainsi qu'une perte financière due à la destruction du lanceur (et de son chargement) et au renchérissement des primes d'assurance associées aux vols suivants. L'analyse mit en cause le système de guidage et plus précisément un dysfonctionnement des centrales inertielles, composant logiciel utilisé avec succès pendant de nombreuses années sur le lanceur Ariane 4. Ce composant avait donc été testé lors des phases de conception et de fonctionnement du lanceur Ariane 4, ne présentant aucun défaut visible. Malheureusement, son utilisation dans le cadre du lanceur Ariane 5 le faisait fonctionner dans un environnement complètement différent concernant les accélérations latérales. N'ayant pas été testés dans ce nouvel environnement, les défauts de ce composant n'ont pu être mis en avant.

1.3.7 Principe 7 – L'illusion de l'absence de défaut

Enfin, les tests sont là pour découvrir des défauts par rapport à ce que l'on attend du logiciel. Plus précisément, on recherche des défauts par rapport à ce qui a été exprimé comme attentes théoriques du logiciel, qui peuvent être différentes des attentes réelles du logiciel : il ne suffit donc pas de détecter puis de corriger tous les défauts trouvés pour rendre un logiciel utilisable ! Rappelons que les différents attendus d'un logiciel peuvent être définis en se référant au modèle qualité de la norme ISO 9126 (revisitée par la norme ISO 25010) dont les critères principaux sont : *Fonctionnality* (capacité fonctionnelle), *Usability* (facilité d'utilisation), *Reliability* (fiabilité), *Portability* (portabilité), *Efficiency* (performance), *Maintainability* (maintenabilité).

Ces différents critères permettent d'exprimer précisément les qualités attendues du logiciel dans son cahier des charges. Les tests permettront de chercher des erreurs dans la réalisation vis-à-vis des critères ciblés. L'absence de défauts sous angle particulier (par exemple sous l'angle fonctionnel) ne garantira pas une qualité acceptable sous un autre angle (par exemple ergonomie et simplicité d'utilisation) ; les premiers utilisateurs en 1993 du logiciel de réservation de la SNCF Socrate peuvent témoigner de ceci.

1. 4 PROCESSUS ET PSYCHOLOGIE LIÉS AUX TESTS

Pour terminer ce chapitre introductif, il faut se rappeler que les activités liées au test sont nombreuses et variées. En particulier, tester n'est pas uniquement « passer des tests ». Il s'agit également de :

- ✓ **planifier les tests** : c'est-à-dire prévoir où et en quelle quantité porter les efforts en personne et en temps ; il s'agit également de prévoir et de réserver l'utilisation de plateformes de tests et les outils nécessaires ;
- ✓ **spécifier les tests** : c'est-à-dire préciser ce que l'on attend des tests en termes de détection d'erreurs (fonctionnel ou non fonctionnel par exemple), les techniques et

les outils à utiliser ou à ne pas utiliser, les parties du logiciel sur lesquelles porteront les tests ;

- ✓ **concevoir les tests :** c'est-à-dire définir les scénarios de tests, appelés aussi cas de test, permettant de mettre en évidence des défauts recherchés par les tests, en fonction de leurs spécifications ; il s'agit aussi de préciser les environnements d'exécution de ces tests ;
- ✓ **établir les conditions de tests** : c'est-à-dire prévoir pour chaque scénario de tests les jeux de valeurs à fournir au logiciel pour réaliser ce scénario ;
- ✓ **définir les conditions d'arrêt d'une campagne de tests** : c'est-à-dire définir, en relation avec la phase de planification, ce qui permettra de décider l'arrêt ou la poursuite des tests ; l'arrêt par épuisement du temps ou des moyens n'est pas un critère d'arrêt pertinent !
- ✓ **contrôler les résultats** : c'est-à-dire comparer les données fournies par l'exécution des tests par rapport aux données attendues ou constatées lors de phases précédentes de tests ;
- ✓ **tracer les tests vis-à-vis des exigences grâce à une matrice de traçabilité** : c'est-à-dire analyser en quoi les tests fournissent une exhaustivité de la recherche d'erreurs dans les attendus du logiciel. Cette matrice permet de vérifier qu'à toute exigence correspond au moins un scénario de test (mesure de la complétude des tests) et, à l'opposé, elle permet également de vérifier qu'un scénario de test sert à valiser au moins une exigence (mesure de l'efficacité des tests).

Bien que décomposable en différentes activités, l'objectif principal du testeur reste de découvrir des erreurs dans un logiciel. Son activité peut alors être vue comme une activité destructrice : il met en avant des problèmes et peut alors être associé à une vision négative et pessimiste des réalisations qu'on lui soumet pour analyse.

Néanmoins, avec du recul et avec le soutien de la hiérarchie, son activité sera perçue comme un moyen efficace d'améliorer la qualité des logiciels. En effet, sans tests, un logiciel risque d'être livré « brut de fonderie » de sorte que même les défauts les plus visibles, ne pourraient être découverts qu'en exploitation avec toutes les conséquences que cela peut avoir. Les tests sont donc le dernier rempart contre les défauts résiduels. En cela, leur intérêt du point de vue des équipes de développement est maintenant reconnu et ils sont donc vus de plus en plus comme une activité créatrice de profits.

Néanmoins, la psychologie du testeur reste quelque peu particulière : l'échec de son travail est l'absence de découverte de défaut. Il va donc à contre-courant des équipes de conception et de développement pour qui l'échec est la découverte d'erreurs. En cela, un testeur devra :

- ✓ être curieux, car il devra rechercher des erreurs dans des parties de logiciel non nécessairement suspectes au premier abord ;
- ✓ faire preuve d'un peu de « pessimisme professionnel » puisqu'il part du constat que tout logiciel, aussi bien conçu soit-il, contiendra un certain nombre de défauts et il faudra mener les tests en vue de détecter ces défauts ;
- ✓ posséder un œil critique car les commentaires, justificatifs ou explications de choix erronés de conception ou de réalisation peuvent convaincre également le testeur et l'induire à son tour en erreur ;
- ✓ prêter attention aux détails puisque c'est souvent dans les détails que se cachent les défauts ;

- ✓ adopter une attitude neutre et factuelle car il doit en quelque sorte annoncer des mauvaises nouvelles et, sans une attitude neutre, sa mission risquerait d'être mal perçue ou mal vécue de la part des équipes de développement ;
- ✓ avoir une bonne aptitude à communiquer puisqu'il faudra rechercher de l'information sur les spécifications manquantes, sur les besoins mal exprimés, etc., qu'il faudra transmettre aux équipes de projets des rapports d'anomalies constatées et qu'il faudra informer régulièrement de l'état d'avancement des tests ;
- ✓ posséder de l'expérience et une formation spécifique car mener des tests est une activité complexe qui mobilise de nombreuses compétences ; sans expérience, le risque est grand de ne pas savoir combattre la combinatoire potentielle des tests autrement qu'en ne se remettant au hasard pour le choix des scénarios et des valeurs de tests pertinents.

Du point de vue de l'encadrement, il est important de se convaincre que l'activité de test est une activité créatrice de richesse et qu'il faut poursuivre l'effort tant que l'objectif n'est pas atteint.