

PRATIQUE DES TESTS LOGICIELS

**Améliorer la qualité par les tests
Préparer la certification ISTQB**



Jean-François Pradat-Peyre
Jacques Printz

3^e édition

DUNOD

© Dunod, 2017

ISBN : 9782100772483

11 rue Paul Bert, 92240 Malakoff

Visitez notre site Web : www.dunod.com

Toute reproduction d'un extrait quelconque de ce livre par quelque procédé que ce soit, et notamment par photocopie ou microfilm, est interdite sans autorisation écrite de l'éditeur.

Avant-propos

Les logiciels informatiques, indispensables au fonctionnement des entreprises et des infrastructures technologiques, ont également 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. Crées par l'homme, les logiciels sont soumis aux aléas de l'activité humaine et de ce fait contiennent des erreurs qu'il faut s'efforcer de détecter afin de les corriger avant la mise en service.

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 re-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 : 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 puissance. 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. 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 ; 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 » ou l'*eXtreme Programming* insistent à juste titre sur l'importance des tests et sur le développement guidé par les tests, *i.e.* le « *Test Driven Development* ».

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, cet ouvrage prépare au passage de la certification ISTQB du métier de testeur.

Cette seconde édition reprend ces objectifs tout en proposant un nouveau chapitre sur les outils de tests et un nouveau chapitre sur les enjeux des tests d'intégration. Enfin, un QCM permet au lecteur d'évaluer son niveau face à une possible certification.

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 JUnit, des informations de mise à niveau, des applications avancées, des logiciels recommandés, des supports didactiques...

Notes

[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.

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 (*ā*) 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.

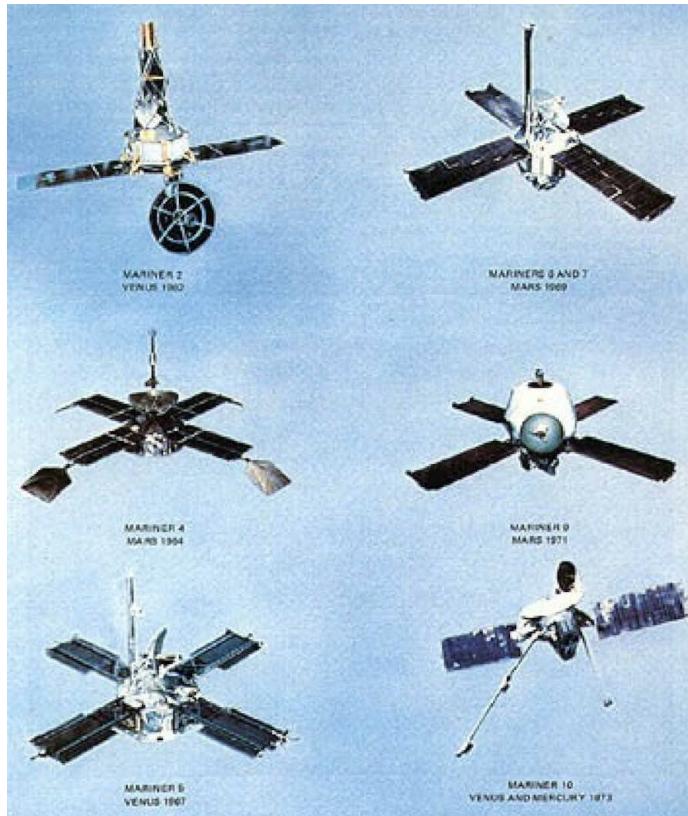


Fig. 1.1 Les sondes Mariner (image NASA)

http://upload.wikimedia.org/wikipedia/commons/7/7f/Mariner_1_to_10.jpg

Cette confusion conduit 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 à chacun peut être témoin dans sa vie professionnelle ou familiale.

Dans tous ces cas, une erreur de réalisation ou de conception d'un logiciel conduit un système automatique à faire autre chose que ce pourquoi 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 sont sans conséquences dans certains cas et dans d'autres conduisent à des comportements non prévus et sont la cause

de dysfonctionnement 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 suite à 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 malgré 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.

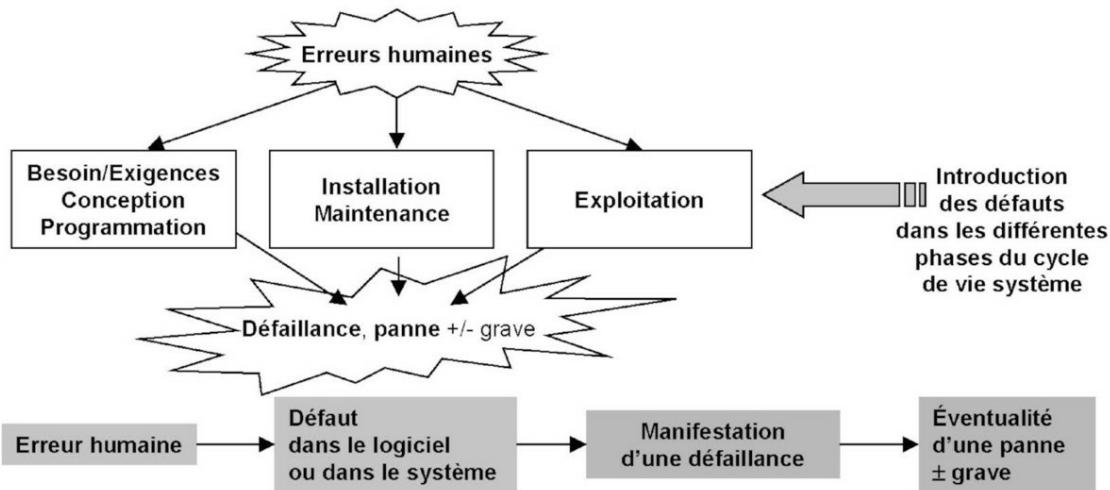


Fig. 1.2 La chaîne de l'erreur

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 insoluble). 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 permettront 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'algorithme, la plupart des programmes peuvent atteindre rapidement un nombre d'états différents supérieurs au nombre estimé d'atomes dans l'univers (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 tous les cas, il faudrait, à raison de 1 milliard d'opérations de test par seconde, près de 127 années de travail !

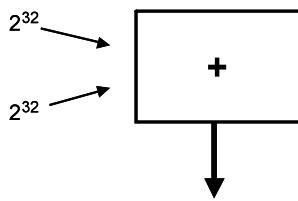


Fig. 1.3 Tester tous les cas (2^{64}) \cong 127 années à raison de 1 milliard d'opérations par seconde

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 prétera 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 lequel 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 nombreuses. 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 nouvelles erreurs ; non qu'elle soit inadaptée mais parce qu'elle a permis de dénicher toutes les erreurs pour lesquelles 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 re-testés dans ce nouvel environnement, les défauts (ou fonctionnement non prévu) 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 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 plates-formes 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 car l'activité de test est une activité complexe qui mobilise de nombreuses capacités ; 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.

Tester à chaque niveau du cycle de vie

Un logiciel ne naît pas spontanément : il est issu d'un besoin (connu ou pressenti) ou d'une idée et il sera fabriqué par l'assemblage de composants existants ou développés spécifiquement. Il évoluera ensuite en fonction des améliorations souhaitées ou nécessaires. Comme nous l'avons déjà dit, plus une erreur sera détectée tôt, plus il sera simple de la corriger. Il faudra donc tester (ou préparer les tests) le plus tôt possible ; selon le modèle de développement retenu, les tests seront plus ou moins bien intégrés et facilités à chaque étape du cycle de vie du logiciel.

2.1. Les différents modèles de développement

Le principal modèle de développement est le modèle dit du « cycle en V ». Dans ce modèle, le développement se déroule en deux grandes époques : la première, correspond à la conception et se décompose en cinq phases plus une phase de codage. Dans ce temps de la conception, il va falloir passer d'une expression abstraite de l'application (Qu'attend-on de l'application du point de vue du client ?) à une réalisation concrète de cette application (Comment réalise-t-on les mécanismes concrets permettant la satisfaction des attentes et des besoins ?). Vient ensuite le temps des tests et des recettes, temps pendant lesquels l'application est progressivement testée et mise en service avant d'en assurer un suivi en maintenance corrective ou évolutive. Ce modèle est détaillé schématiquement figure 2.1.

Dans ce modèle, déjà ancien, le client ne voit potentiellement le résultat qu'une fois l'application finie. Il va donc s'écouler un moment important entre l'expression initiale des besoins et la remise au client de l'application. Durant ce temps, des petits malentendus ont pu conduire à de grosses divergences entre l'attendu et le réalisé. Par ailleurs, dans ce modèle, les tests

ont tendance à être relégués au second plan ce qui conduit à découvrir certains défauts tardivement ; cependant, comme ces défauts peuvent entraîner d'importantes modifications il sera peut-être nécessaire de revenir très en amont dans le cycle de développement. Tout cela peut amener, dans le cas de projets difficiles, à un dépassement systématique des échéances. De plus, le logiciel obtenu à l'arrivée ne correspond généralement que partiellement aux attentes du client qui ont pu évoluer entre-temps ou qui ont été mal comprises.

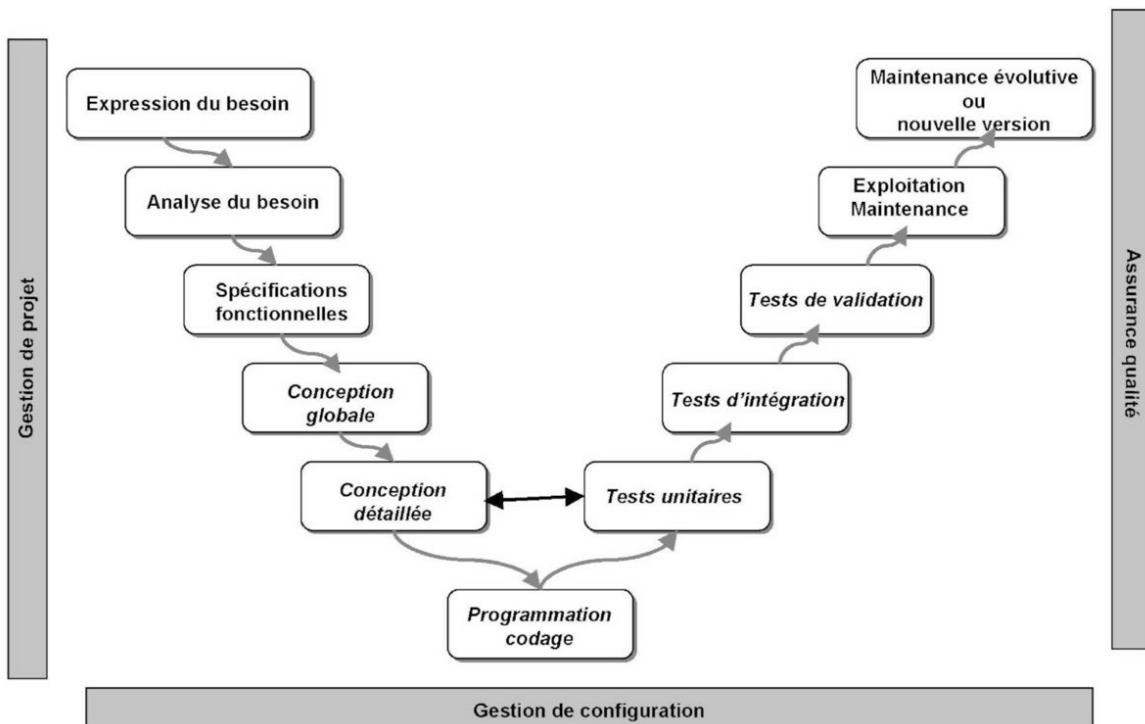


Fig. 2.1 Le cycle en V, schéma général

Aussi, peu à peu est apparu le besoin de mener les projets de façon plus souple en remettant régulièrement des versions de plus en plus complètes de l'application au client, ce qui revient à replacer implicitement ou explicitement le client et les tests au cœur du processus : c'est l'apparition des méthodes de développement itératif et incrémental qui se déclinent sous de nombreuses formes. On peut citer la méthode RAD (*Rapid Application Development*) la méthode UP (*Unified Process*) ou la méthode XP (*Extreme Programming*) et ses variantes du développement dit agile. On notera que

tous ces modèles dérivent du cycle en V, sauf peut-être certains modèles itératifs extrêmes (comme la méthode *Crystal clear*), et que donc la compréhension du cycle en V est la condition nécessaire pour la compréhension des autres modèles.

Pour cela nous détaillons maintenant, sous l'angle des tests, les différentes phases du cycle en V et en particulier les phases de conception qui, bien que n'incluant pas directement des activités de test, sont des phases primordiales pour la capacité de mener à bien ces tests une fois les étapes de réalisation finies.

2.2. Préparer les tests lors des phases de conception du cycle en V

Dans les phases de conception, il s'agit non de conduire les tests, mais de préparer leur mise en œuvre lors des phases dédiées à l'assurance qualité. Cette préparation est fondamentale car de sa bonne réalisation dépendra la possibilité de mener à bien ou non des tests efficaces aux différents niveaux de réalisation.

Nous détaillons maintenant quels sont les points clés à chacun de ces niveaux pour réussir à concevoir une application testable.

2.2.1. Préparer les tests lors des phases d'expression et d'analyse du besoin

Dès les phases d'expression et d'analyse du besoin, il est fondamental de se préoccuper des problèmes de VV&T : Validation (Fait-on le bon logiciel ?), Vérification (Le logiciel est-il bien fait ?) et Tests (Peut-on découvrir des défauts ?). Il s'agit en fait de se poser les questions de la qualité en général et de la testabilité du système en particulier. En effet des besoins mal exprimés, incomplets, ambigus ou contradictoires vont être générateurs, tout au long du cycle de vie, de perturbations et de coût de non-qualité.

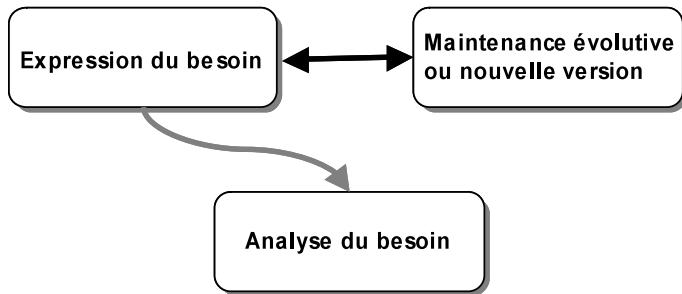


Fig. 2.2 La phase d'*expression du besoin*

Lors de la phase d'*expression du besoin*, on prépare implicitement les évolutions futures dues aux corrections ou adaptations qu'il faudra apporter lors des futures étapes de maintenances évolutives (migrations, suppressions ou ajouts de parties du système). À chaque adaptation, il faut revenir sur le système initial pour déterminer quelles parties doivent être re-testées par des tests de non-régression. Il est donc important à cette étape de conception de faire le tri entre les différents types d'exigences : exigences applicatives *versus* exigences systèmes, exigences fonctionnelles *versus* exigences non fonctionnelles. Il faut également à cette étape tenter d'éliminer les « vœux pieux » qui conduisent à des exigences difficilement testables comme « l'application pourra être utilisée par toute personne sachant manipuler une souris ».

L'exploitation du logiciel mettra en évidence des défauts. La découverte de ces défauts nécessitera de revenir sur la phase d'*analyse du besoin* afin de déterminer quelles modifications devront être effectuées en maintenance (maintenance corrective). Après correction, il sera nécessaire de vérifier que les modifications apportées n'ont pas impacté des parties du système qui fonctionnaient correctement. Ceci est fait à l'aide de tests de non-régression qu'il faut avoir prévus dès ce niveau de conception.

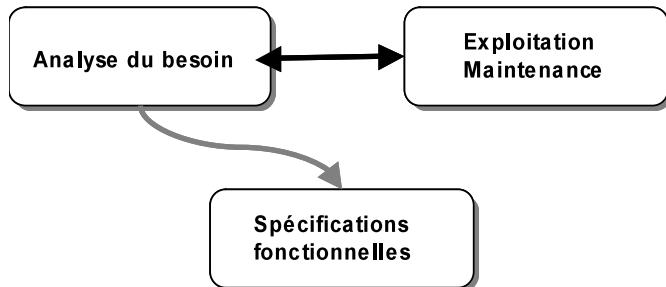


Fig. 2.3 La phase d'analyse du besoin

On notera que, dans une approche UML, les phases d'expression et d'analyse du besoin pourront utiliser les cas d'utilisation du système et la modélisation des processus métiers pour faciliter la préparation des tests.

2.2.2. Préparer les tests lors de l'écriture des spécifications fonctionnelles

La phase de spécifications fonctionnelles doit consolider le travail effectué dans les deux phases précédentes et permettre de décrire de façon exhaustive l'ensemble des fonctionnalités de la future application ainsi que son interface graphique (IHM, interface homme/machine). Là encore il faudra prendre garde d'éliminer ou de préciser des exigences fonctionnelles non testables. Afin de garantir l'exhaustivité des tests de validation, des outils plus ou moins sophistiqués de traçabilité des exigences peuvent être utilisés (on notera que bien souvent un simple tableur pourra être utilisé en lieu d'outils dédiés mais complexes).

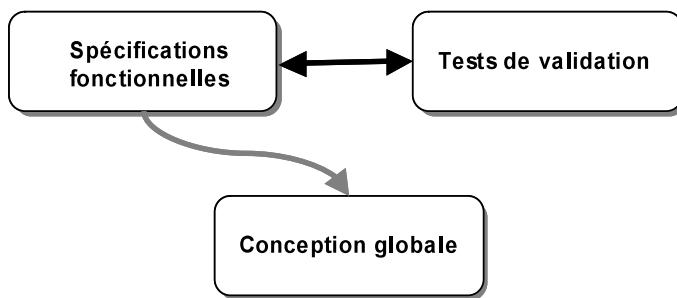


Fig. 2.4 La phase d'écriture des spécifications fonctionnelles

À cette étape également, tout effort de modélisation pourra grandement simplifier et améliorer les futurs tests. C'est ainsi qu'un cas d'emploi pourra décrire précisément un scénario de test de validation.

2.2.3. Préparer les tests lors de la conception globale

Cette phase de conception est cruciale pour la qualité générale d'une application, d'un système ou d'une fédération de système. C'est en effet lors de cette phase que l'on va passer de la « machinerie métier » (le besoin) à la « machinerie informatique » (le système) selon le schéma suivant (figure 2.6).

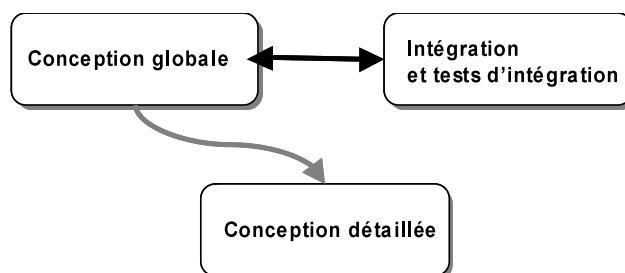


Fig. 2.5 La phase de conception globale

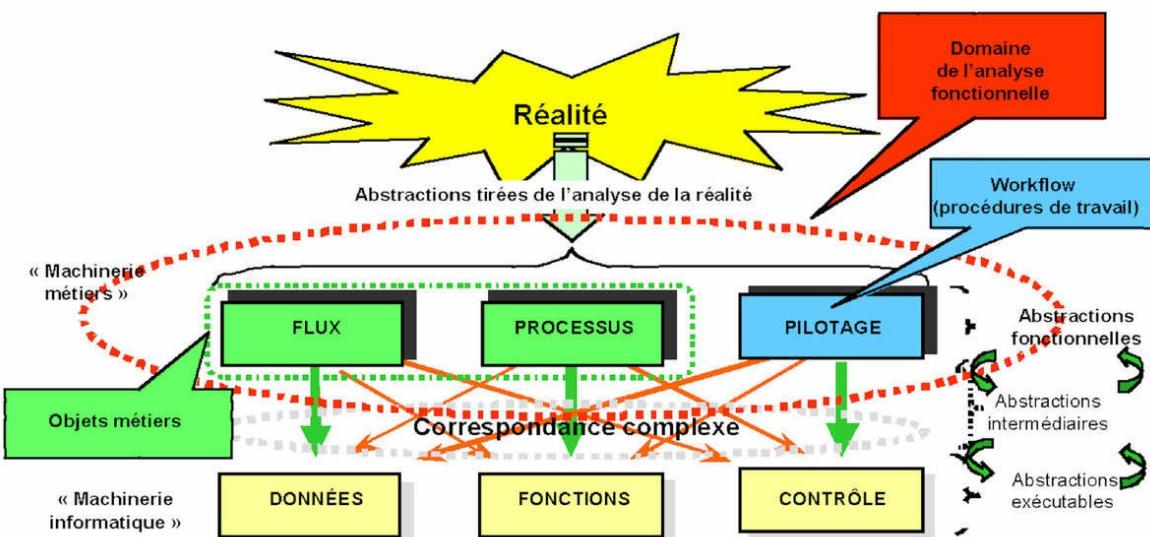


Fig. 2.6 De la « machinerie métier » à la « machinerie informatique »

La difficulté provient du fait qu'il faut représenter la réalité d'une façon nécessairement simplifiée et abstraite, en séparant autant que possible les aspects technologiques des processus et des données manipulées.

Tous les détails ne peuvent (et ne doivent) pas être pris en compte et leur représentation peut entraîner une modification de leurs propriétés (par exemple, une adresse sera représentée par une chaîne de caractères codée sur un nombre fini de bits et ne pourra donc prendre qu'un nombre limité de valeurs qui dépendra non de la réalité mais de son codage informatique). Bien que les langages de programmation aient fortement évolué au cours du temps en offrant des niveaux d'abstraction de plus en plus élevés et permettant ainsi de représenter la réalité de façon de plus en plus fidèle sans contraindre le programmeur à des exercices de contorsion, le choix de l'architecture reste un problème complexe qui n'est pas résolu par la simple application de patrons prédefinis. Cette architecture doit en effet résoudre de nombreuses contraintes parfois contradictoires. Ainsi, elle doit être simple et modulaire tout en prenant en compte une réalité complexe où les éléments sont parfois très dépendants les uns des autres. L'application du précepte « forte cohésion et faible couplage » peut guider à maîtriser cette contradiction. De même le traitement des erreurs peut être fait au niveau de leur apparition (par exemple lors d'une entrée sortie) ou au contraire au niveau du contrôle global de l'application (par exemple lors de la demande d'authentification d'une personne). Selon le niveau où l'erreur est traitée, on dispose, soit de détails concrets sur l'erreur (par exemple, problème de positionnement d'un bit sur les droits d'accès à un fichier) mais pas de possibilités de récupération car la sémantique de l'instruction n'est pas connue à ce niveau, soit au contraire, on dispose d'alternatives mais on n'a pas de possibilité de comprendre les détails.

Du fait de cette complexité de conception, et du fait de l'impact des choix architecturaux sur l'analyse du comportement de l'application, il est primordial de définir une architecture testable. Globalement, une architecture testable doit minimiser le volume de tests nécessaires à la validation du produit logiciel correspondant, et ceci dans le contexte d'exploitation réelle du logiciel et de son cycle de vie. Le critère de simplicité de l'architecture est clairement corrélé au volume de tests à produire et à l'effort nécessaire à les gérer. Il est par ailleurs indispensable de préparer à ce niveau les tests

d'intégration, c'est-à-dire prévoir des dispositifs architecturaux qui permettront de pouvoir observer les échanges au niveau des interfaces internes à une application ou des interfaces externes entre applications ou systèmes ainsi que des mécanismes de journalisation des événements à des fins d'observation ou de « re-jeu » d'un scénario. Pour résumer, les cinq principes suivants peuvent aider à préparer les tests à cette étape :

- Appliquer le principe de forte cohésion (les entités d'un même module sont fortement liées) et faible couplage (les modules sont peu interdépendants les uns des autres) ;
- « Casser » les longues chaînes de traitement de sorte que des résultats intermédiaires visibles sont produits (et récupérables au travers de « logs » appropriés) ;
- Prévoir une politique générale de traitement des erreurs avec un mécanisme de journalisation en vue d'une possibilité d'analyse ultérieure ;
- Définir précisément les interfaces des différents modules et donner une cartographie précise de l'interdépendance des modules ;
- Définir un ordre d'intégration et planifier au mieux les tests d'intégration en termes de ressources humaines et matérielles.

2.2.4. Préparer les tests lors de la conception détaillée

La phase de conception détaillée va préciser la façon dont chaque composant (module, classe, méthode) va réaliser ce pourquoi il est prévu ; il s'agira ici de détailler les algorithmes et structures de données utilisées. Deux approches s'affrontent : l'approche « classique » où le développeur code puis teste à l'aide de tests unitaires et l'approche « développement agile » où le développeur conçoit et code les tests avant de coder le composant. Cette dernière approche est connue sous le terme « TDD » pour « *Test Driven Development* » (développement piloté par les tests). Dans les deux cas, il faudra penser lors de cette phase à prévoir les cas de tests et les environnements nécessaires à leur bonne exécution.

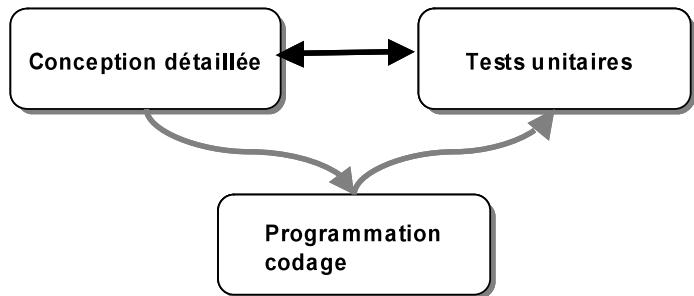


Fig. 2.7 La phase de conception détaillée

L'intérêt de préparer les tests avant de commencer la réalisation est de permettre au développeur de se concentrer sur les difficultés et les pièges potentiels liés à la réalisation : cas irréguliers, frontières entre les valeurs nominales et les valeurs hors domaine, difficultés algorithmiques, propriétés associées à la représentation des données, respect de pré et/ou post-conditions, etc.

2.3. Les tests et les modèles itératifs

Les méthodes de développement dites agiles (de l'anglais *Agile Modeling*) visent à réduire le cycle de vie du logiciel en accélérant son développement. Le principe est de développer une version minimale, puis d'intégrer les fonctionnalités par un processus itératif basé sur une écoute du client et des tests tout au long du cycle de développement. L'origine de ces méthodes dites « agiles » est liée à l'évolution permanente de l'environnement technologique et au fait que le client est souvent dans l'incapacité de définir ses besoins de manière précise et exhaustive dès le début du projet. Le terme « agile » fait ainsi référence à la capacité d'adaptation du logiciel et des équipes aux changements de contexte et aux modifications de spécifications intervenant pendant le processus de développement. En 2001, 17 personnes mirent ainsi au point le manifeste agile <http://agilemanifesto.org> dont les grandes lignes sont :

- Individus et interactions plutôt que processus et outils ;
- Développement logiciel plutôt que documentation exhaustive ;
- Collaboration avec le client plutôt que négociation contractuelle ;

- Ouverture au changement plutôt que suivi d'un plan rigide ;
- Ce manifeste s'oppose aux méthodologies dites « lourdes » comme ISO 12207 et le modèle de développement dit « en cascade », ISO 9000.

Grâce aux méthodes agiles, le client est au cœur de son projet et obtient très vite une première mise en production de son logiciel. Il est ainsi possible d'associer les utilisateurs dès le début du projet. Nous présentons maintenant brièvement les caractéristiques essentielles de deux modèles significatifs et très employés : UP pour « *Unified Process* » et XP pour « *eXtreme Programming* ».

Il est important de noter que tous ces modèles, UP compris, mis entre des mains inexpérimentées donneront des résultats catastrophiques qui ne seront pas imputables aux modèles, mais plutôt à ceux qui les mettent en œuvre. Comme l'a dit B. Boehm, il a bien longtemps : « *The method won't save you, but it can help !* ». Avant d'utiliser un modèle quelconque, nous devons comprendre il marche (*cf. Coût et durée des projets informatiques* de J. Printz chez Hermes, chapitre 1).

2.3.1. Le développement itératif « UP »

La dynamique d'un projet UP (*Unified Process*) est structurée d'une part en itérations, appelées mini-projets, et d'autre part en incrémentations qui vont constituer les composants (*building blocks*) du produit final. Au démarrage, la dynamique UP peut être représentée comme suit :

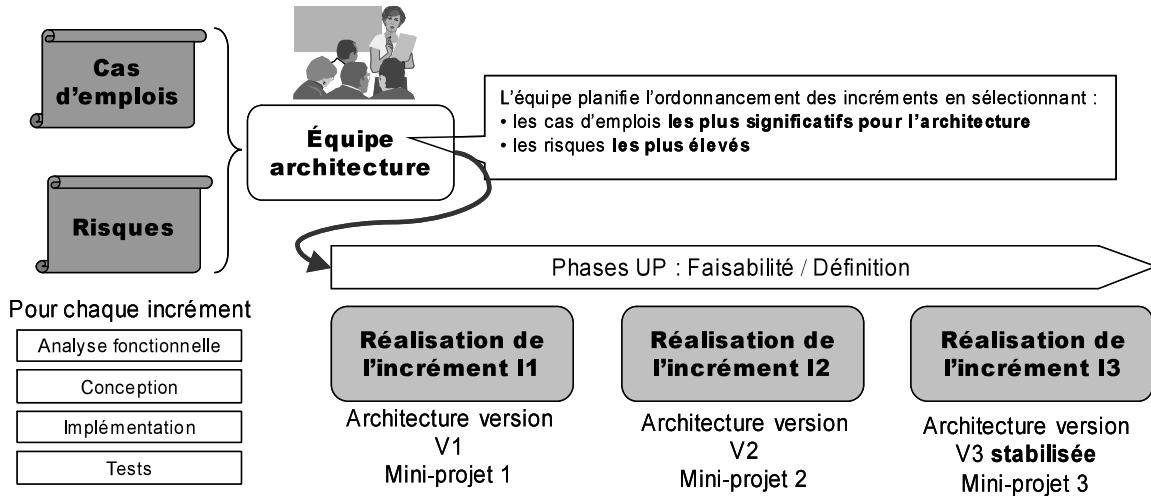


Fig. 2.8 La dynamique UP en phase de démarrage

Ce processus est conforme aux règles de la dynamique admises dans la gestion des projets de développement^[1]. Il met en évidence, sans concession à la facilité, les trois difficultés majeures de quasiment tous les projets informatiques :

- Choix des cas d'emploi permettant de stabiliser l'architecture, et choix de l'ordonnancement (NB : les caractéristiques non fonctionnelles sont fondamentales pour cadrer l'architecture ; par exemple, s'occuper très tôt des performances).
- Identification des risques principaux du projet.
- Stabilisation effective de l'architecture, afin de garantir une croissance optimale en phase de construction.

Le développement itératif UP n'est rien d'autre qu'un cycle en V répété n fois en miniature. Le terme « unifié » (*Unified Process*) vient du fait que la méthode prend en compte l'ensemble des intervenants : client, utilisateur, gestionnaire, qualiticien, etc.

L'approche UP est une approche :

- pilotée par les cas d'utilisation ;
- centrée sur l'architecture (80 % des cas d'utilisation cartographiés) ;
- itérative et incrémentale ;

- fondée sur la production de composants logiciels ;
- qui gère les besoins et les exigences ;
- qui surveille la qualité et les risques.

Les tests s'intègrent dans chaque itération, ce qui permet un contrôle continu de la qualité. Leur mise en œuvre est simplifiée car la méthode s'appuie sur des modèles et des cas d'utilisations qui permettent de dériver simplement des cas de tests fonctionnels pertinents.

2.3.2. Le développement itératif « XP »

Dans XP (*eXtrem Programming*), tout est optimisé pour la livraison rapide de versions de système utilisables immédiatement par les utilisateurs et pour la prise en compte immédiate des besoins, y compris ceux qui peuvent émaner de l'utilisation des premières versions supposées donner des idées pour de nouveaux besoins. Il est primordial de rester en permanence à l'écoute du client, d'optimiser la communication avec lui pour pouvoir rétroagir au plus vite et « sortir » le code qui convient exactement à son besoin. L'équipe projet est partie prenante de la communication.

Le personnage central de ce dispositif de pilotage est le COACH de XP qui joue le rôle de chef de projet (même si dans la nomenclature XP, il est désigné par le terme « *coach* » ou « *tracker* »). Le schéma de communication mise en œuvre dans un projet XP est donné dans la figure 2.9.

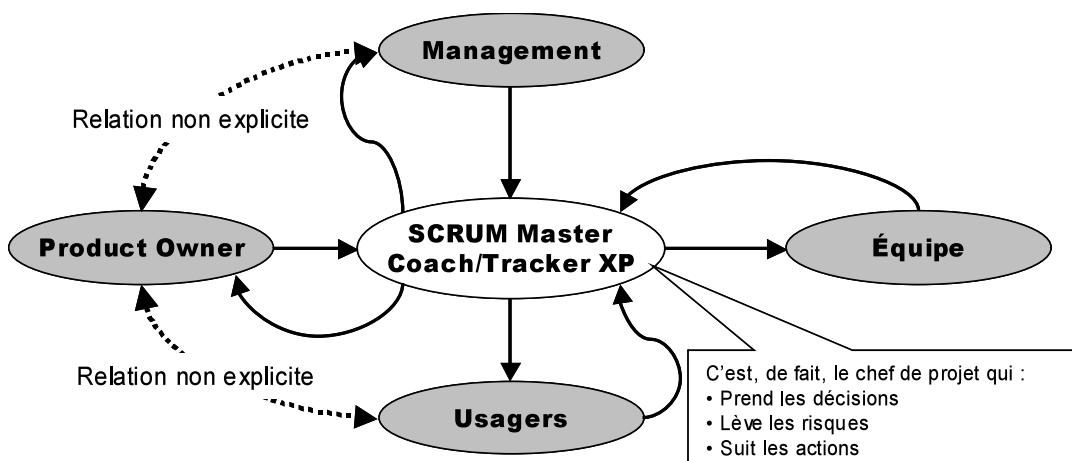


Fig. 2.9 La communication entre acteurs dans XP

Même pour un projet de taille modeste il y a toujours des difficultés de communication. Les relations non explicitées entre le management, la maîtrise d'ouvrage (ou MOA) et les usagers mériteraient quelques explications. Le chef de projet doit faire preuve de courage, plus que tout autre ; K. Beck va jusqu'à dire : « *Without courage, XP just simply doesn't work* » ; et de plus, il doit savoir dominer sa peur. Il est au centre de tous les conflits.

Symétriquement, le rôle et le comportement attendu du client est également critique : « *Being an XP customer is not easy* ». Il doit être capable d'écrire de « bons » cas d'emplois. Le client type XP est celui qui « *will actually use the system being developed, but who also have a certain perspective on the problem to be solved* » ; ce qui mêle à la fois le court terme et le long terme. Dans notre terminologie, c'est à la fois un MOA métier classique, mais doublé d'une capacité stratégique, donc un profil de MOA rare.

Pour cette méthode, le code et la programmation sont au centre du dispositif, avec, un couplage très étroit avec les tests ; « *Programming and testing together is faster than just programming* » ; ce qui est une façon d'adapter à la programmation le bon vieux slogan de R. Crosby : « *Do it right first time* ». XP préconise la programmation en binôme programmeur – testeur. Le problème est d'organiser la production de code correctement testé, en maintenant un contact étroit avec le client.

Le cycle de vie est fondé sur une succession de phases de développement, et non pas sur les processus et les activités d'ingénierie. Le cycle de vie idéal prend en compte les caractéristiques bien particulières d'un projet XP :

- Projet de petite taille sans contrainte d'interopérabilité avec d'autres systèmes
- Équipe expérimentée
- Relations de confiance entre le client et l'équipe projet

Le cycle de vie se présente comme sur la figure 2.10.

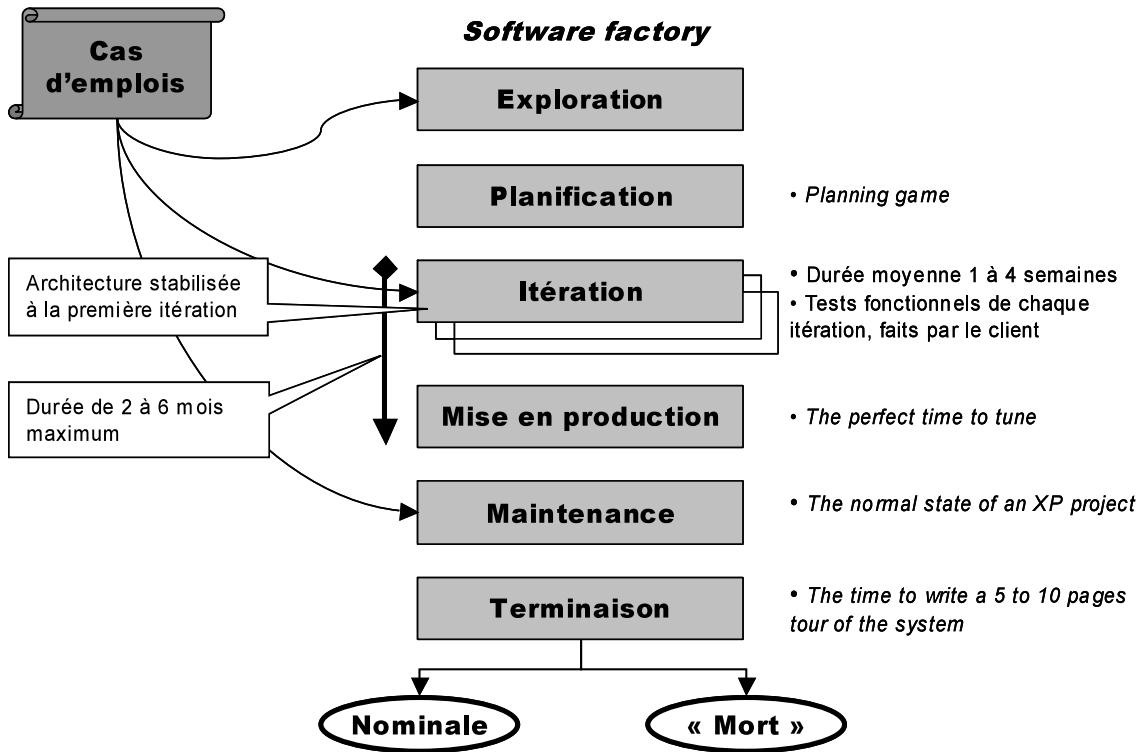


Fig. 2.10 Le cycle de vie idéal d'un projet XP

Dans l'approche XP, la « *software factory* » distingue six phases, que l'on pourrait comparer aux phases de UP. L'ensemble du dispositif est alimenté à partir de cas d'emplois élaborés en coopération avec le client. Les cas d'emploi sont la base de la relation avec le client.

La phase d'exploration, comme son nom l'indique est une phase exploratoire dont le but est de cerner les grandes lignes du système à réaliser.

La phase de planification, dite aussi phase de « *planning game* », est faite en collaboration avec le client, et K. Beck va jusqu'à dire : « *you will have to educate your customer about the rules of the game* », ce qui a toutes les apparences d'un vœu pieux. Toutes les deux semaines, et pour les deux prochaines semaines, les développeurs estiment le coût des fonctionnalités candidates, et le client choisit les fonctionnalités à implémenter en fonction du coût et de la valeur ajoutée. La durée pour réaliser ce qui constitue la première version ou *release* ne doit pas excéder 6 mois, car au-delà le risque de divergence est jugé excessif ; *a contrario*, cela donne une indication sur la taille des projets envisagés qui sont des petits, voire très petits, projets.

La phase d’itération livre une succession d’incrément, dont le premier doit mettre en place l’architecture du logiciel (*The whole system in a skeletal form*) ce qui peut nécessiter de « bouchonner » au maximum les composants qui seront délivrés ultérieurement.

Ceci n’est possible que si le chef de projet-architecte a une grande expérience. Le client est mis à contribution pour les tests fonctionnels correspondant à chacun des incrément livrés.

La phase de mise en production, correspondant à la première *release* du système, correspond à la fin du « *planning game* », avec des itérations extrêmement courtes de l’ordre de 1 semaine, au lieu des 3 ou 4 d’une itération nominale. K. Beck nous dit que c’est le « *perfect time to tune* », ce qui est pour le moins ambigu ; cela presuppose que la performance a été intégralement anticipée lors de la première itération qui définit l’architecture. C’est une nouvelle évidence que cela ne peut marcher qu’avec du personnel très expérimenté ; dans ce cas seulement on n’aura à effectuer que quelques réglages dans la phase de mise en production. Avec une équipe inexpérimentée, le risque de dérapage est maximum.

La phase de maintenance, considérée comme l’état normal d’un projet XP, dure aussi longtemps qu’il y a des besoins nouveaux à satisfaire. L’équipe XP a en charge la maintenance de la *release* en cours d’exploitation et de développement de la prochaine *release* dans laquelle seront intégrées les corrections dues aux défaillances constatées. Implicitement l’équipe XP fait l’ingénierie complète au sens ISO 12207.

La phase de terminaison marque l’arrêt du projet XP qui n’est pas forcément l’arrêt de l’exploitation. La terminaison nominale correspond à la satisfaction de tous les besoins, ce qui laisse présager une période d’exploitation de plusieurs années, sans modification. C’est comme le dit K. Beck « *The time to write a 5 to 10 pages tour of the system* », ce qui est pour le moins léger, pour ne pas dire irresponsable car cela revient à dire que la documentation, c’est le code !

La terminaison par « mort par entropie » (*Entropic death*) correspond à la situation où l’architecture initiale n’ayant pas résisté aux chocs successifs des besoins à satisfaire ces derniers ont fini par la détruire ; il est plus sage d’arrêter le projet. Le projet succombe à l’excès de complexité qu’il a lui-

même créé, et de laquelle rien n'émerge, que du chaos. Dans ce cas, il est recommandé que le client et l'équipe XP qui ont opéré de concert tirent les leçons de l'échec pour envisager un avenir meilleur en faisant un bilan de projet.

Sans vouloir glosser sur la terminologie, K. Beck nous dit que « *XP is not magic. Entropy eventually catches XP projects, too. You just hope that it happens much later than sooner* ». Cela fait du développement un pari plutôt qu'un acte d'ingénierie raisonné et assumé. Ceci n'est jouable que pour de petits projets, sans enjeu stratégique véritable pour l'entreprise. La complexité est toujours un vrai risque, et aucune méthode ne garantit le succès.

2.4. Les différents niveaux de test

Quelle que soit la méthode de développement utilisée, la réalisation d'une application comprend trois grandes étapes : la réalisation des composants élémentaires (ou leurs choix dans une bibliothèque), l'intégration de ces composants afin de constituer l'application, puis enfin son déploiement sur un environnement d'exécution. Selon la taille de l'application, les deux premières étapes peuvent être répétées afin de constituer des composants de plus en plus complexes.

À chacune de ces étapes, il va être nécessaire de tester au niveau adéquat : tester le composant au niveau unitaire, tester la bonne interaction des composants au niveau de l'intégration puis tester l'ensemble au niveau système tout d'abord du point de l'équipe projet puis enfin du point de vue du client. À chaque défaut ou manquement découvert, il y aura une modification corrective ou évolutive ce qui nécessitera de s'assurer à l'aide de tests de non-régression que la modification n'a pas impacté les parties de l'application qui n'ont pas été modifiées.

Même si l'objectif général reste la découverte de défauts, à chacun de ces niveaux, les tests auront leurs propres spécificités. Ce sera bien entendu l'objet à tester, mais aussi, les documents en entrée des tests, les défauts et défaillances typiques à rechercher, les approches méthodologiques possibles ou le partage des responsabilités dans ces activités de test.

Étudions maintenant pour chacun des niveaux mentionnés (unitaire, intégration, système) ces différentes spécificités ; nous ne détaillerons pas ici, mais aux chapitres suivants, les techniques et les outils pouvant être employés.

2.4.1. Le test de composants ou tests unitaires

Les tests unitaires concernent le test d'éléments élémentaires appelés composants et donc on parle aussi de test de composants. C'est le premier niveau des tests au sens du cycle de développement.

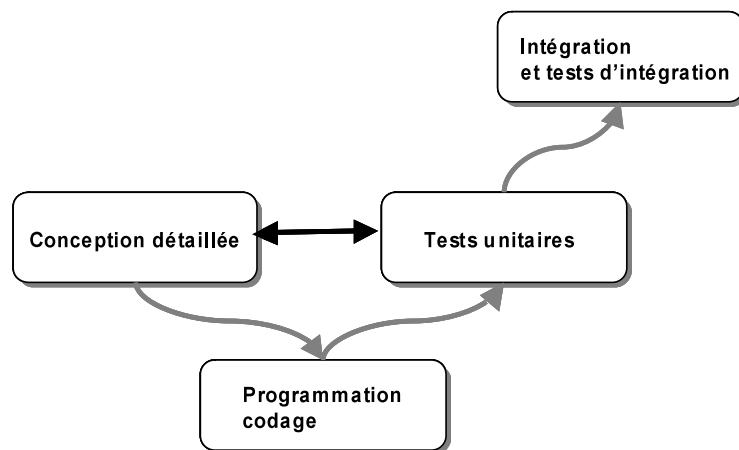


Fig. 2.11 Les tests unitaires dans le cycle en V

Il n'y a pas de définition universelle de la notion de composants mais on peut la comprendre comme étant une unité logicielle clairement identifiable, relativement autonome et avec des interfaces clairement définies. Un composant doit donc être utilisable et testable séparément du reste de l'application moyennant l'utilisation éventuelle de « bouchons » ou de « pilotes » (figure 2.12). Ainsi, selon le niveau de granularité auquel on observera l'application, un composant pourra être une fonction, une méthode, une classe, une bibliothèque ou encore un sous-système (comme par exemple un système de gestion de bases de données, un sous-système de cartographie, etc.) lorsque l'application est complexe et se décompose en plusieurs sous systèmes clairement identifiables et plus ou moins autonomes.

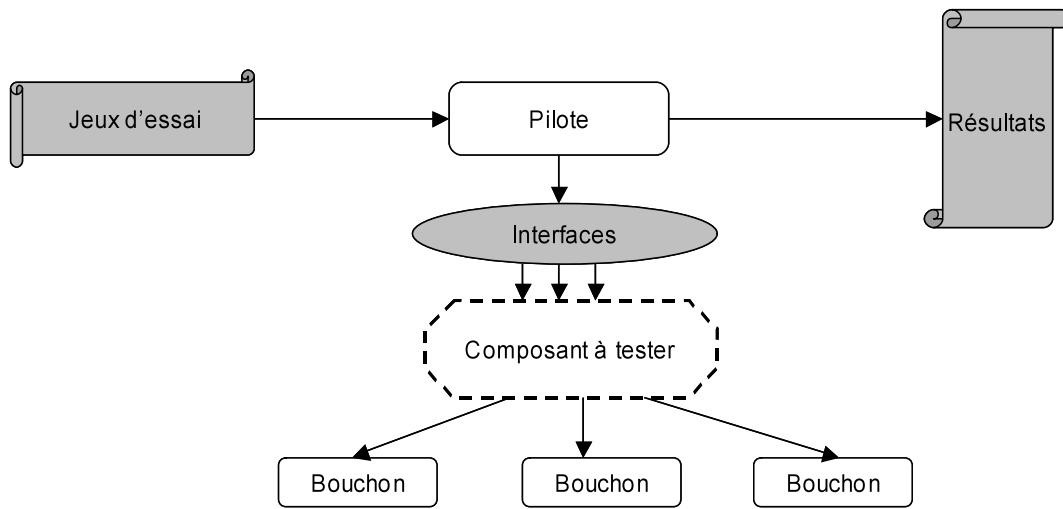


Fig. 2.12 Pilotes et bouchons dans les tests unitaires

L’indépendance d’un composant n’est que relative. Dans le cas général, un composant doit être utilisé par d’autres composants selon une logique prédéfinie et il s’appuie généralement sur d’autres composants. Pour le tester, il faut recréer un environnement d’exécution qui simule l’utilisation de ce composant dans le contexte d’utilisation pour lequel il a été défini. Par exemple, si le composant est « à état » (comme un objet au sens programmation par objet), le pilote pourra avoir comme rôle de placer le composant dans l’état où il peut effectuer une certaine action que l’on veut tester. Par ailleurs, le pilote peut devoir également récupérer les données à passer au composant d’un autre composant en suivant sa logique (comme par exemple la transmission d’une suite de bits aléatoires). Enfin, le pilote devra stocker ou transmettre à un processus d’impression les résultats fournis par le composant lors des tests.

En résumé, le rôle du pilote est de lire les données de test, les passer au module à tester et imprimer les résultats ou les stocker dans un fichier de référence.

De même que le composant ne peut pas toujours être utilisé directement, ce qui nécessite la mise en place de pilotes, il est fréquent qu’un composant utilise d’autres composants pour mener à bien ses travaux. Afin de détecter de façon fiable des erreurs du composant, il faut être sûr des composants utilisés (en termes de comportement et de leurs interfaces). Si les composants

utilisés ne sont pas sûrs (ou pas finis) ce qui est presque tout le temps le cas, il va donc falloir les remplacer par des composants partiels, donc plus simples à écrire que le composant total, et dont on est sûr du travail effectué. Ces composants qui simulent partiellement d'autres composants sont appelés des « bouchons » ou « souches » ou « stubs ». Ces souches ou bouchons simuleront la logique de comportement et se contenteront en général de retourner une valeur fixe conforme aux paramètres passés. Pour les souches complexes, des simulateurs pourront être développés et dans le cadre de projets complexes, les premiers prototypes ou d'anciennes versions pourront servir à construire à moindres coûts ces souches.

Si le composant est lui-même composé de plusieurs composants plus simples (comme une classe qui est à la fois une unité en termes de compilation ou d'application mais qui se décompose en éléments plus simples que sont les méthodes) le composant aura été testé unitairement lorsque toutes les fonctions ou méthodes qui le composent auront été testées unitairement.

Les composants étant par définition les briques élémentaires de l'application, il y a une nécessité absolue d'effectuer correctement et systématiquement les tests unitaires sinon les tests futurs s'appuieront sur des éléments mal maîtrisés et c'est, en particulier, l'activité d'intégration qui se trouvera détournée de son objectif principal, c'est-à-dire les tests concernant les interfaces entre modules.

On notera que le principe architectural énoncé plus haut de forte cohésion et de faible couplage permettra de simplifier cette étape des tests en ayant permis de définir des composants facilement identifiables, aux interfaces bien définies, rendant ainsi l'écriture des pilotes et des souches plus simple voire inutile.

Par ailleurs, lors de ces tests, il sera possible de mesurer certains critères qualitatifs (maintenabilité, évolutivité, etc.) obtenus par interprétation et consolidation de critères quantitatifs simples (nombre de lignes par méthodes, nombre d'appels de méthodes par méthodes, etc.).

Pour mener à bien ces tests, le testeur va utiliser les documents de conception détaillée du composant et/ou le code du composant (dans un développement classique). Dans une démarche agile (très itérative), les tests peuvent être

écrits avant le composant lui-même donc le testeur n'utilise alors que les documents de conception.

Les défauts recherchés seront des erreurs de codage au sens large :

- Variables utilisées et non initialisées (erreur de codage la plus fréquente)
- Conditions inversées
- Sortie de boucle prématuée
- Absence de code défensif concernant des conditions d'entrée dans le composant non prévues (non-respect de préconditions)
- Utilisation d'un pointeur nul, etc.

Ces erreurs seront découvertes par la présence de défaillances typiques :

- Message d'erreur sorti à tort
- Sortie du composant sur des valeurs non prévues
- Violation mémoire
- Dépassement de capacité de représentation (*overflow*), etc.

Pour cela on prévoira des tests qui couvrent dans un premier temps les cas nominaux afin de vérifier que le code chargé de traiter le comportement normal du composant est réalisé correctement. Ensuite, il faudra prévoir des tests pour vérifier que la prise en compte des cas irréguliers est correcte. Avec ces deux premières catégories de tests, on aura testé (au sens de la recherche d'erreurs) que le composant fait ce pourquoi il est prévu. Après cela il faudra tester que le composant réagit correctement lorsqu'on lui passe des données clairement hors du domaine de définition et enfin lorsqu'on lui passe des données limites au sens de ces domaines ; c'est en effet pour ces valeurs limites qu'apparaissent fréquemment des erreurs de codage ou d'interprétation des spécifications.

En procédant de la sorte, on découvre en premier lieu les erreurs ou oublis sur ce que doit faire le composant puis on découvre les erreurs concernant le traitement des cas pathologiques et enfin on vérifie que le composant est protégé (selon ce qui est défini) contre une utilisation anormale.

Ces tests seront sous la responsabilité du développeur ou de l'équipe en charge des tests au sein de l'équipe projet.

2.4.2. Le test d'intégration

L'intégration est fondamentalement une activité de test et de planification de test, au sens large du terme. La première difficulté de l'intégration est qu'elle est totalement tributaire de la qualité des activités et des processus amont. Selon le modèle d'estimation COCOMO (acronyme de « *Constructive COst MOdel* »), l'intégration consomme de 20 à 35 % de l'effort total de développement, selon la complexité du projet.

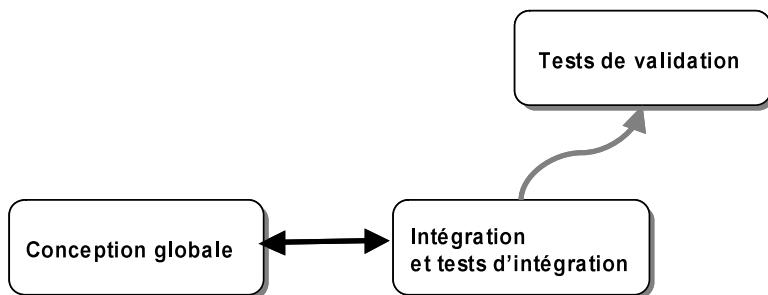


Fig. 2.13 Les tests d'intégration dans le cycle en V

Les tests d'intégration se concentrent sur les interfaces entre les composants, les interactions entre les différentes parties du système ou les interfaces entre systèmes (pour les fédérations de systèmes). Il est important de noter que l'on ne teste pas le résultat de l'ensemble (ce qui sera le rôle des tests de validation) mais que l'on se concentre sur les échanges au niveau des interfaces. Il va sans dire que l'impact des choix architecturaux est très important à cette étape car, sans interface clairement définie, pas de tests d'intégration possibles !

Une application étant généralement constituée de multiples composants, et les interactions entre ceux-ci pouvant être complexes il va falloir choisir dans quel ordre réaliser ces tests d'intégrations. Si l'on considère l'exemple suivant (figure 2.14), le composant 2 utilise les composants 4 et 5 et est utilisé par le composant 1.

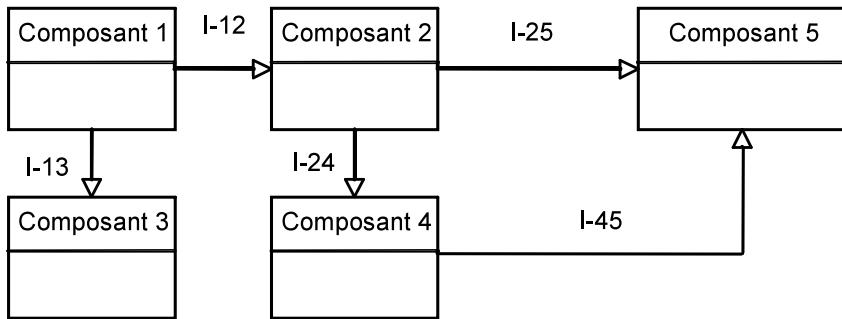


Fig. 2.14 Un schéma d'architecture logicielle

Selon l'ordre dans lequel ces quatre composants seront intégrés il sera plus ou moins facile de tester leurs interactions. On distingue généralement, trois stratégies :

- La stratégie du Big Bang
- La stratégie par lots fonctionnels
- Les stratégies incrémentales

Avant de détailler plus précisément ces stratégies, il faut remarquer que celles-ci sont liées à la disponibilité des composants qui conditionne le choix de la stratégie que l'on peut réellement appliquer.

La stratégie du Big Bang consiste à intégrer tous les composants disponibles en même temps. Il va sans dire que tout étant « branché » en même temps il est difficile de localiser précisément une erreur. Ainsi, dans notre petit exemple précédent, les cinq composants sont intégrés simultanément et un problème sur l'interface I-25 (entre 2 et 5) pourra en fait être la conséquence d'une défaillance sur l'interface I-12 ou même sur l'interface I-45 (entre 4 et 5) qui place le composant 5 dans un état incompatible à un échange avec le composant 2. La recherche d'erreur peut donc s'avérer ténue et paradoxalement les tests peuvent tromper sur la localisation des erreurs. On notera, que bien que peu efficace, cette stratégie optimiste a le mérite de la simplicité, et qu'elle peut tout de même être employée dans le cas de très petites applications.

La stratégie par lots fonctionnels tente de pallier aux défauts de la stratégie du Big Bang en intégrant les composants selon une logique « métiers »,

permettant ainsi de tester l’interaction de composants prévus pour travailler dans un même contexte — probablement réalisés par le même service ou la même entreprise — et facilitant ainsi l’analyse des défauts. Il s’agit donc d’une intégration Big Bang raisonnée.

Les stratégies les plus construites d’intégration se basent sur les dépendances entre les composants pour déterminer l’ordre dans lequel sera faite l’intégration et sont appelées stratégies incrémentales. Pour cela on distingue les composants de « bas niveau », c’est-à-dire ceux sur lesquels les autres composants s’appuient, et ceux de « haut niveaux » qui implémentent la logique de l’application et s’appuient sur des composants de plus bas niveau pour réaliser leur mission. Dans notre exemple, le composant 5 est le composant de plus bas niveau (il ne s’appuie sur aucun autre composant) et le composant 1 est le composant de plus haut niveau (aucun composant ne l’utilise). La première possibilité dans une intégration incrémentale est de procéder de bas en haut en intégrant les composants de bas niveau avant les composants de haut niveau. De la sorte, il est relativement simple de trouver et de localiser les erreurs d’interaction car l’on ne s’appuie que sur des composants testés au niveau unitaire et au niveau intégration car déjà intégrés. La contrepartie est qu’il est nécessaire de créer des pilotes qui ne pourront pas être réutilisés lors de futures phases d’intégration ou au niveau des tests de validation car ils sont trop spécifiques. De plus, les parties les mieux testées seront les parties de bas niveau qui ne sont pas forcément les plus complexes.

C’est pour cela que l’on adopte parfois une stratégie d’intégration incrémentale « descendante » dans laquelle on intègre en premier lieu les composants de « haut niveau ». En procédant de la sorte, les défauts de structure seront très vite découverts et l’architecture générale du logiciel sera testée tôt. Les jeux de tests pourront resservir pour plusieurs étapes d’intégration et certains pourront également être repris pour les tests de validation. On peut noter que cette stratégie permet de mieux répartir l’effort des tests d’intégration. En contrepartie, il va falloir réaliser un certain nombre de bouchons qui simuleront le comportement des composants de bas niveau non encore intégrés. Ce comportement dépend naturellement du contexte d’utilisation du composant et il peut alors s’avérer complexe de simuler ces

composants induisant le risque de tester incomplètement l’interaction d’un composant avec un autre du fait de l’utilisation de bouchons incomplets.

Il est alors tentant de combiner les approches descendante et ascendante pour obtenir une stratégie d’intégration incrémentale « mixte » qui permet d’optimiser l’effort de test. Il est en effet possible d’intégrer en premier lieu les composants réutilisables, ceux difficilement simulables ou ceux dont on maîtrise parfaitement la logique et les interfaces indépendamment de leur caractéristique bas ou haut niveau. On peut également chercher à intégrer les composants sensibles en premier de sorte qu’ils soient exercés (et donc testés) plus que les autres. Il s’agit d’essayer de minimiser les inconvénients des deux stratégies précédentes tout en maximisant les avantages. Cela ne peut être obtenu qu’avec une forte expérience du chef de projet et une bonne connaissance des équipes impliquées.

Ainsi, une stratégie de choix, dans le cas général, est donc certainement la stratégie incrémentale. Il existe cependant une difficulté à sa mise en pratique : l’existence de cycle au niveau des dépendances (comme, par exemple, A dépend de B qui dépend de C qui dépend de A). Si l’on s’appuie sur une règle fixe d’intégration, par exemple ascendante, il faudrait dans l’exemple précédent intégrer A avant A ! Il faut donc trouver une façon de tenir compte de ces cycles tout en maintenant une stratégie cohérente. Une façon simple de procéder^[2] est, partant d’un graphe de dépendance entre modules, d’ordonner les nœuds de ce graphe (les composants) par rapport au nombre de cycles passant par ce nœud. Notons C_m , le composant en tête de ce classement. La seconde étape consiste à remplacer C_m par un bouchon unique ou par autant de bouchons spécifiques que de composants s’appuyant sur C_m . Dans le premier cas, le bouchon simulera l’ensemble des fonctionnalités offertes par C_m , dans l’autre cas, chaque bouchon spécifique ne simulera que les fonctionnalités utilisées par le composant utilisateur. Le choix de l’une ou l’autre stratégie sera dicté par l’effort à réaliser et la présence ou non de bouchons potentiels comme d’anciens prototypes ou de versions préliminaires. En itérant ce procédé, on finit par obtenir un graphe sans cycle qui permet de procéder à une intégration incrémentale simple.

Cette étape d’intégration étant fondamentale dans l’activité des tests, elle est détaillée plus à fond au chapitre 7. Néanmoins, un certain nombre de conseils peuvent déjà être donnés pour réussir les tests d’intégration :

- **Au niveau de la préparation des tests**, il faudra chercher à optimiser le nombre de tests, à faciliter la réalisation des tests et la localisation des défauts et à optimiser le nombre de pilotes et de souches et minimiser l'utilisation de machines spécifiques (qui peuvent pénaliser la réalisation des tests).
- **Au niveau de la gestion de projet**, on cherchera à minimiser la taille des équipes et à diminuer la complexité organisationnelle et à faciliter le lissage de la charge et des affectations.

Le plan d'intégration, élaboré durant la phase de conception globale, doit être suivi car il définit la stratégie dominante d'intégration et l'ordre dans lequel les composants ou éléments doivent être intégrés ; il conditionne donc les priorités de développement et de tests unitaires sur chacun des constituants. Néanmoins on pourra chercher à intégrer dès que possible (afin de minimiser les risques) les modules devant être fortement testés, les modules à forte probabilité de défauts et les modules à fonctionnalités les plus critiques (vitales et centrales pour le système).

Les défauts et défaillances typiques à découvrir dépendent du type de l'interface. S'il s'agit d'une interface par messages, on recherchera des défauts dans la description du format (dans le composant émetteur ou récepteur) ou des défauts de routage (interface avec un MOM^[3] messagerie asynchrone, par exemple). Si, par contre, il s'agit d'une interface par événements, on recherchera plutôt des défauts sur les arguments des événements (type, nombre) ou des événements inexistant. Dans le cas d'interfaces par structures de données, on privilégiera la recherche de valeurs erronées. Enfin, dans le cas d'interfaces basées sur des transactions, on visera à détecter des transactions mal construites ou des valeurs erronées caractéristiques du mauvais enchaînement d'une transaction.

En ce qui concerne la répartition des rôles, pour les projets de taille importante (75 à 100 KLOC), il est préférable que l'équipe d'intégration soit indépendante car elle permettra d'apporter un regard externe et la différence de culture entre l'équipe projet et l'équipe responsable des tests d'intégration permettra de détecter des erreurs difficiles à imaginer par une personne proche de l'équipe de développement. De plus, une équipe externe aura moins peur de « brusquer » le logiciel, peur qui limite parfois

inconsciemment un testeur dans sa recherche d'erreur. Il faudra cependant maintenir un circuit court de remontée des anomalies entre l'équipe d'intégration et l'équipe de développement sous peine d'accumuler les anomalies.

En termes de couverture, l'objectif doit être de tendre vers une couverture à 100 % des interfaces en incluant tous les contextes d'utilisation même si, pour des systèmes complexes, ceci est très difficile à obtenir.

Enfin, pour les logiciels supérieurs à un million de lignes, l'équipe d'intégration devra dédier une personne à la gestion de configuration (composants, jeux de tests, anomalies).

2.4.3. Les tests système et les tests de validation

Une fois l'application intégrée il va être possible de la tester dans sa globalité. La phase de validation consiste ainsi à tester une application, un système (ou une fédération de systèmes) par rapport à ses spécifications fonctionnelles et non fonctionnelles (au sens ISO 9126 : performance, utilisabilité, maintenabilité, évolutivité, etc.). Ce sont par nature des tests Boîte noire.

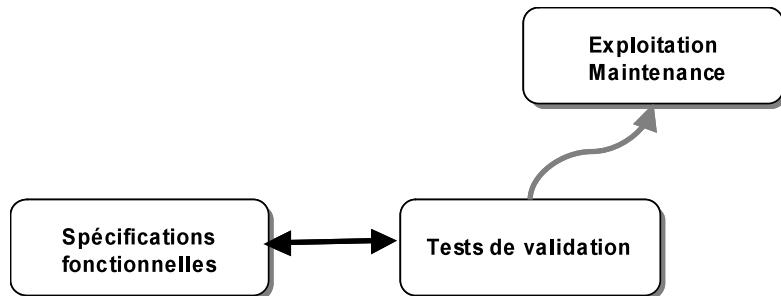


Fig. 2.15 Les tests de validation dans le cycle en V

Les tests de validation doivent être effectués dans un environnement le plus proche possible de l'environnement cible de façon à minimiser les risques de défaillances spécifiques de l'environnement utilisé et à réduire les risques d'absence de découverte de défauts qui ne peuvent être mis en évidence que dans l'environnement final. Cependant, il n'est pas toujours possible de mettre en œuvre réellement l'environnement cible pour des raisons de coût ou de mise en œuvre : centrale nucléaire, avion, systèmes d'armes, etc.

Dans ce cas, les tests de validation ont recours eux-mêmes à la simulation (en fait une double simulation : le logiciel réel est simulé et, pour chaque jeu de données ou événements, les résultats sont comparés par rapport à un modèle idéal, lui-même simulé). Il faudra alors résoudre le problème de la tolérance acceptable sur les écarts éventuels (comme la trajectoire d'un missile, par exemple) entre valeur attendue et valeur constatée.

Les défauts et défaillances typiques à découvrir concernent de façon générale tout comportement du logiciel non conforme au comportement décrit dans les différents documents d'entrée. À ce niveau de tests, on recherche encore des défauts dans le logiciel, étant entendu que des exigences ont été exprimées et que celles-ci soient bien testables, c'est-à-dire que l'on peut, entre autres, interpréter sans ambiguïté le résultat d'un test.

Les documents en entrée des tests de validation comprennent :

- Les spécifications fonctionnelles qui décrivent ce que l'on attend du logiciel, que l'on nomme exigences fonctionnelles ;
- Les spécifications non fonctionnelles qui décrivent la façon dont doit être faite la mission du logiciel (performance, tolérance aux pannes, sécurité, etc.) et la façon dont doit être conçu le logiciel (maintenabilité, évolutivité) ;
- Le manuel utilisateur qui décrit comment utiliser le logiciel ;
- Les cas d'utilisation qui apportent une aide précieuse pour la mise au point de scénarios de tests pertinents. Pour cela ils doivent avoir été conçus dans un objectif de test et, si nécessaire, il peut être utile de les retravailler avant de les utiliser ;
- Tout autre document (par exemple des documents sur l'histoire du logiciel en termes de versions et d'anomalies constatées) permettant d'apprécier quels sont les tests utiles à effectuer à ce niveau de tests.

Lorsque les tests se concentrent sur la découverte de défauts par rapport aux spécifications ou exigences fonctionnelles, on emploie le terme de « tests de validation ». Lorsque les tests se focalisent sur la découverte de défauts concernant des spécifications non fonctionnelles, on parle plutôt de « tests système » même si ce terme est également utilisé pour des tests de validation à partir du moment où ils sont effectués dans un environnement système proche ou identique au système cible.

Dans tous les cas (petits ou gros projets), les tests de validation seront menés par une équipe indépendante qui cherchera à obtenir une couverture maximale des différentes fonctionnalités et cas d'utilisation. Pour cela, on aura recours à des outils, qui peuvent être un simple tableur, pour garantir cette exhaustivité et assurer une traçabilité des tests vis-à-vis des exigences exprimées lors de la phase de spécification.

Pour la validation d'une fédération de systèmes, les choses sont légèrement plus complexes car les phases d'intégration et de validation vont s'alterner dans une progression qui se calque sur le niveau de granularité, du plus fin au plus gros : composants logiciels, application, système, fédération de systèmes.

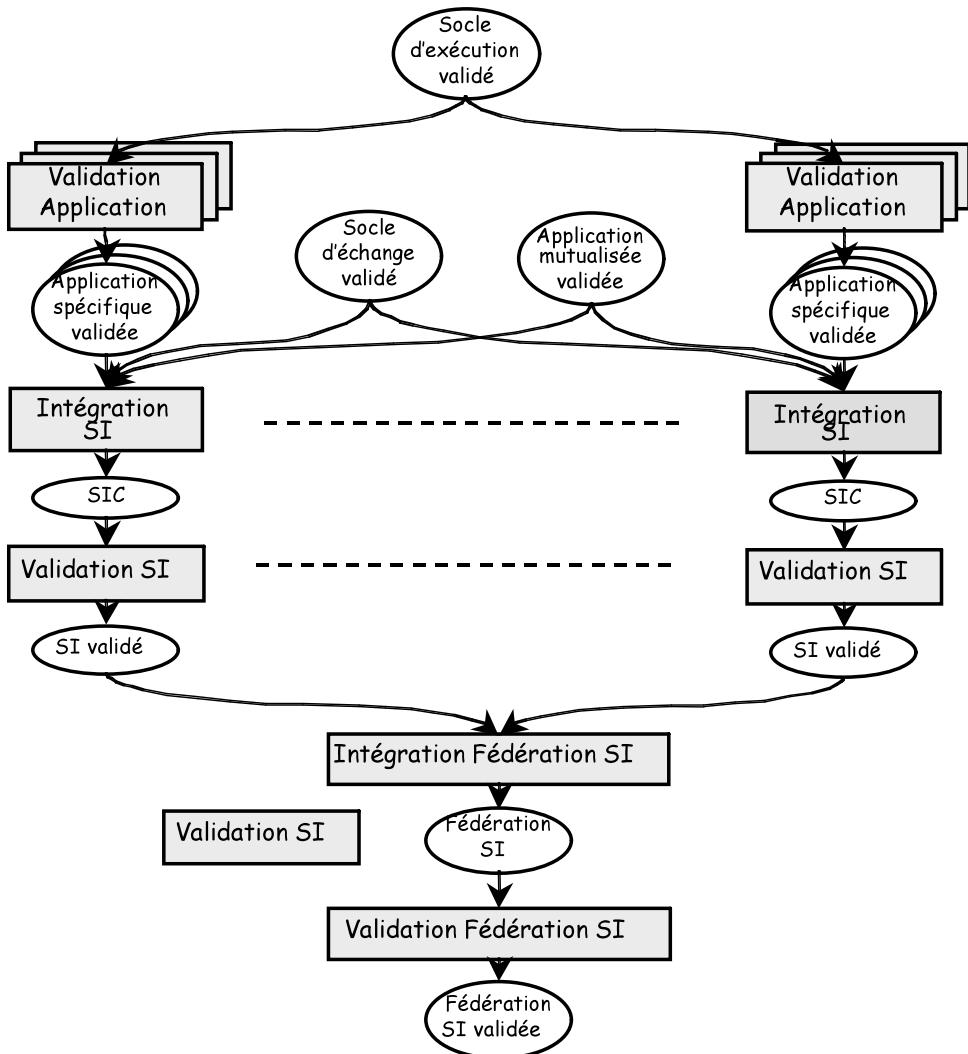


Fig. 2.16 Étapes d'intégration et de validation pour une fédération de systèmes

2.4.4. Les tests d'acceptation, tests alpha et tests bêta

Les tests d'acceptation se situent au même niveau que les tests de validation ou les tests système mais n'ont pas les mêmes objectifs. Ceux-ci ont en effet pour objectif de s'assurer du fonctionnement du système en conformité avec les besoins et les spécifications (dans le vocabulaire qualité, il s'agit de la conformité aux exigences) et non de découvrir des défauts ou des manques, même si ces tests peuvent en révéler. Les tests d'acceptation peuvent évaluer aussi le niveau de préparation du système préalablement à son déploiement et

à son utilisation (installation, administration, par exemple). Ces tests font partie des tests non fonctionnels (facilité d'installation et/ou d'administration).

On distingue généralement différents types de tests d'acceptation :

- Les tests d'acceptation par les utilisateurs visent à vérifier l'aptitude et l'utilisabilité du système par des utilisateurs dans l'environnement cible (l'utilisabilité de l'interface homme/machine (IHM) aura été normalement testée sur une maquette ou un prototype lors de la phase de spécification)
- Les tests d'acceptation opérationnelle qui concernent principalement les administrateurs du système et analysent des opérations comme les sauvegardes et les restaurations du système, la reprise après sinistre, la gestion des utilisateurs, la gestion de la sécurité, etc.
- Les tests d'acceptation contractuelle qui concernent l'acceptation par le client d'un logiciel développé « sur mesure » et qui sont généralement associés au versement d'une partie des montants dus par le client. Les critères d'acceptation ont normalement été définis contractuellement lors de la rédaction du contrat. Lorsque le logiciel n'est pas accepté, l'ensemble des acteurs (le client et la société ou le service qui a développé le logiciel) se trouvent dans une situation complexe qu'il est difficile de quitter sans l'avis d'une tierce personne ou d'une équipe d'audit qui sera amenée à évaluer la situation. En particulier, cette équipe devra répondre à la question suivante : « Des améliorations ou des corrections pourront-elles être apportées pour passer cette étape d'acceptation ou faut-il jeter le logiciel car les défauts sont trop importants comme par exemple, une architecture totalement inadaptée à la mission du logiciel ou aux exigences contractuelles ? ». Cette question est fondamentale car il est en effet préférable de passer à pertes et profits un logiciel qui, de par sa structure, ou du fait de sa réalisation, ne peut absolument pas garantir qu'une évolution favorable est envisageable.
- Les tests d'acceptation réglementaires qui sont définis par rapport aux règlements et législations en cours, comme les obligations légales, gouvernementales ou de sécurité. Les tests d'acceptation réglementaires

peuvent, implicitement ou explicitement, induire un rejet au niveau de l'acceptation contractuelle.

Dans le cas de logiciel à forte diffusion, on désignera par « tests alpha » des tests d'acceptation réalisés par une équipe différente de l'équipe de développement mais de la même entreprise. Ces tests permettront à l'équipe de développement d'avoir un premier retour sur le logiciel en situation réelle par un client plus tolérant et moins vindicatif en cas de problèmes importants qu'un client réel. Une fois cette étape d'acceptation passée, l'entreprise pourra passer aux « tests bêta » qui reproduisent cette mise en situation mais, cette fois-ci, avec de « vrais » clients qui auront accepté d'être cobayes en échange de la possibilité d'utiliser des versions récentes et non encore commercialisées d'un logiciel donné. Ces clients sont alors dénommés « bêta-testeurs ». Ils participent alors volontairement et sans coûts pour la société éditrice du logiciel à l'amélioration de celui-ci. Leur motivation est soit l'envie de posséder une version récente d'un logiciel à la mode, soit le besoin de passer à une version qui permet de sortir d'une impasse en corrigeant ou en apportant une fonctionnalité par rapport à la version précédente, soit encore l'envie de participer d'une façon peu contraignante au développement d'un logiciel libre. Le schéma suivant résume le positionnement des différents tests étudiés dans les phases de développement ainsi que les documents et données en entrées de chacun.

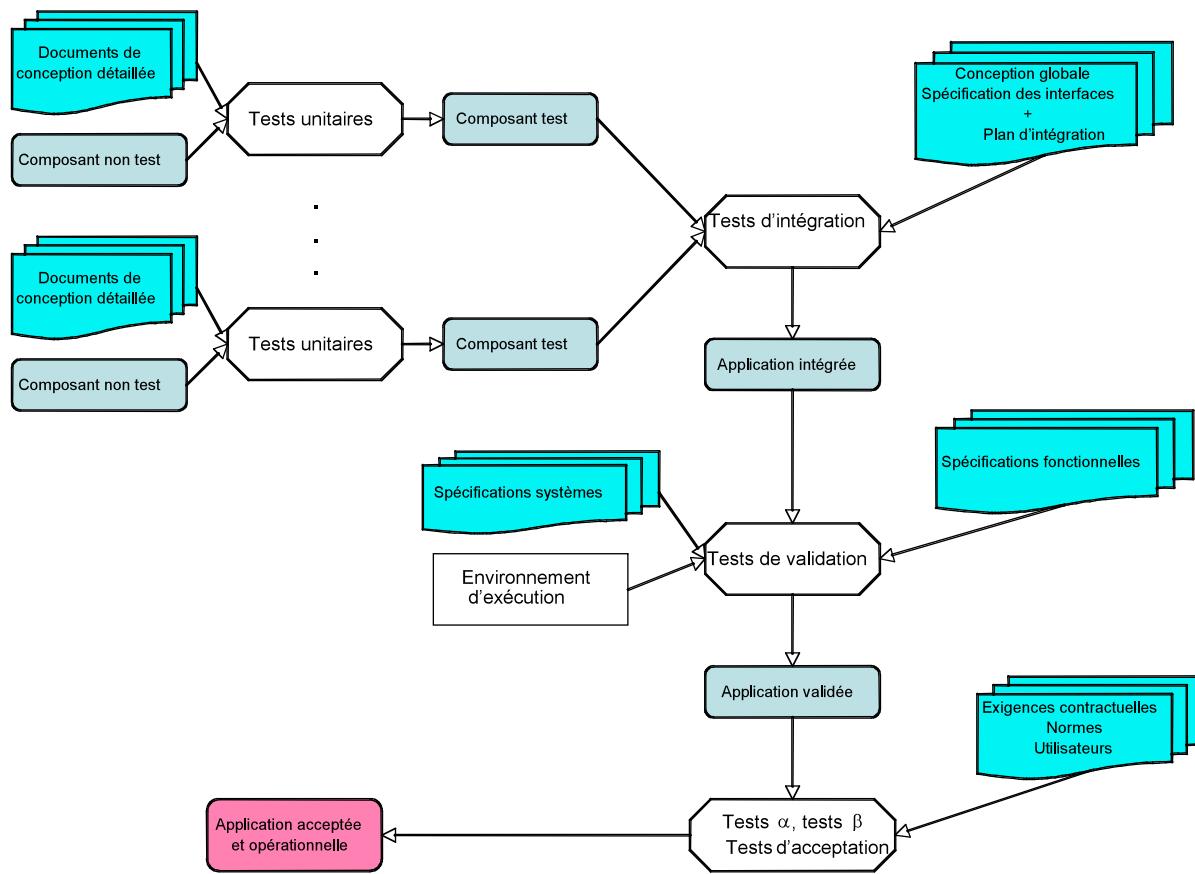


Fig. 2.17 Positionnement des types de tests dans un cycle en V

2.5. Les différents types de tests

À chaque niveau du développement d'un logiciel correspondent des activités de tests. Ces tests sont spécifiques du niveau auquel ils s'appliquent et c'est ainsi que les tests unitaires sont différents des tests d'intégration et des tests de validation. Cependant, certaines caractéristiques sont communes à plusieurs phases de tests : ce sont des types de tests transverses au cycle de vie, présents à différents niveaux et ayant des objectifs communs. Le diagramme suivant (figure 2.18) permet de visualiser les différents critères selon lesquels les tests peuvent être catalogués.

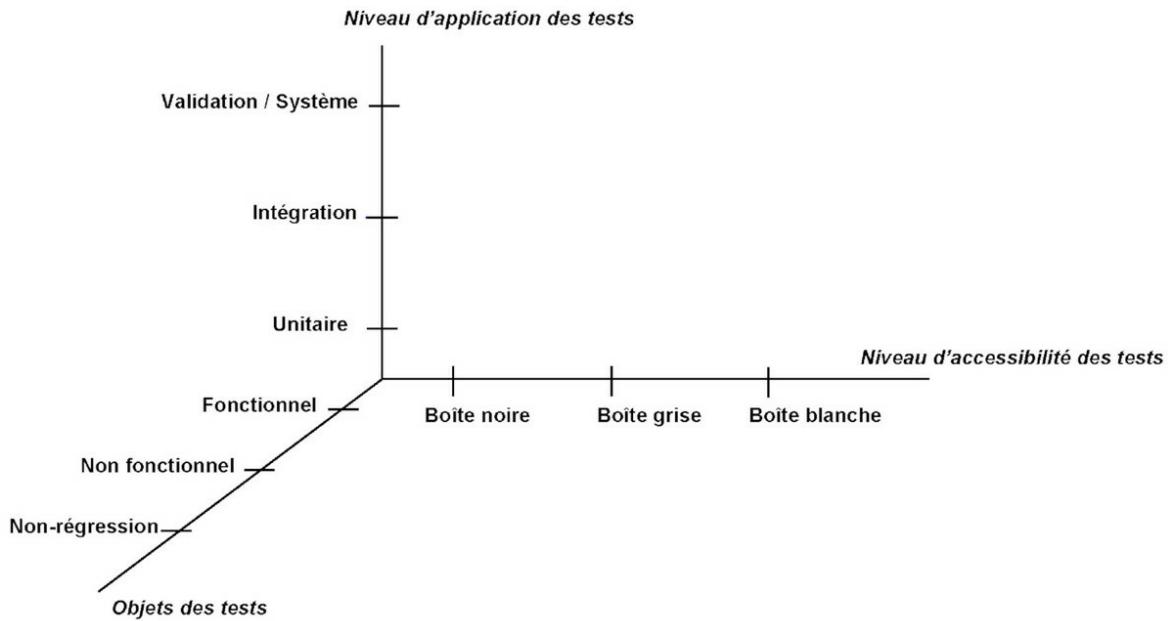


Fig. 2.18 Typologie des tests

Sur ce diagramme, l'axe vertical représente le niveau auquel peut être appliqué un test : l'échelle va des tests unitaires qui portent sur un composant élémentaire aux tests de validation qui porte sur une application ou un système complet. L'axe horizontal définit l'accès possible aux détails de représentation. Lorsque l'accès est total, on parle de tests Boîte blanche, lorsque l'accès est impossible ou non voulu, on parle de tests Boîte noire et lorsque seule une partie des détails de représentation est accessible, on parle de tests Boîte grise. Enfin le troisième axe propose une classification selon l'objet des tests : le test vise-t-il à rechercher un défaut vis-à-vis d'une fonctionnalité, d'un critère non fonctionnel comme la performance ou la sûreté de fonctionnement, ou cherche-t-il à vérifier qu'une modification du logiciel n'a pas fait régresser son comportement ?

Chaque point de cet espace définit une catégorie de tests pouvant exister, même si l'usage et les contraintes habituelles veulent que certaines catégories soient plus utilisées que d'autres. C'est ainsi qu'il est plus commun que les tests unitaires soient des tests Boîte blanche dont l'objet est la recherche d'erreur sur l'aspect fonctionnel, même s'il est possible d'écrire des tests

unitaires Boîte noire à visée non fonctionnelle, comme l'évaluation de la sécurité ou de la performance d'un composant.

2.5.1. Tests fonctionnels versus tests non fonctionnels

Les tests fonctionnels visent à tenter de démontrer que le logiciel effectue ce pourquoi il est fait en recherchant d'éventuelles défaillances lorsque le testeur le sollicite sur des valeurs nominales. On peut ainsi chercher à vérifier qu'une fenêtre s'ouvre bien après que l'on ait tapé un mot de passe correct dans un menu de connexion. Il s'agit là d'un test fonctionnel. Cependant si la fenêtre ne s'ouvre qu'après plusieurs longues secondes la fonction sera satisfaite mais d'une manière imparfaite. Il s'agit là d'un critère non fonctionnel. Ces critères non fonctionnels ne cherchent pas à qualifier le logiciel sur ce qu'il fait mais sur la façon dont il le fait. Le modèle ISO 9126, présenté figure 2.19, définit les exigences non fonctionnelles pouvant être utilisées pour définir les attentes sur le comportement et la conception d'un logiciel.

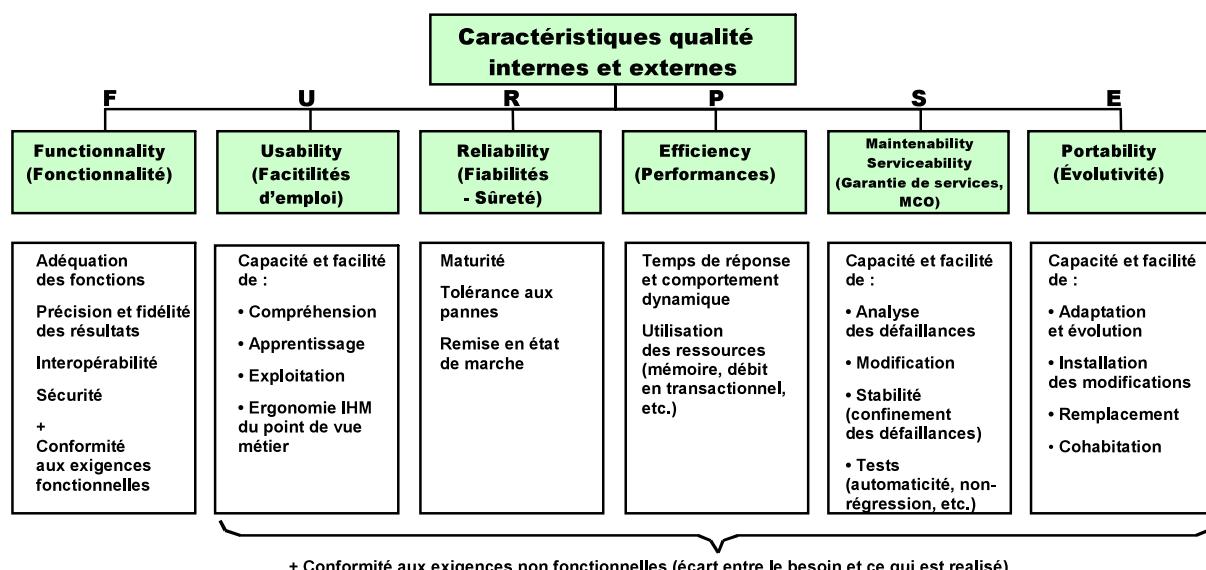


Fig. 2.19 Caractéristiques qualité et référentiel FURPSE

Les tests non fonctionnels sont généralement plus complexes à mettre en œuvre car ils s'appuient sur des analyses plus poussées que la simple vérification de la validité d'une réponse ou sur la cohérence d'un échange entre deux composants. Il s'agit par exemple de calculer et d'interpréter des

analyses statistiques sur des ensembles de données obtenues sous certaines conditions. Par exemple, lors de tests de charge, on peut chercher à calculer le temps moyen d'attente d'un processus client lors de la primo-demande d'un service à un serveur et le comparer au temps moyen d'attente des clients sur les demandes suivantes de ce même service.

Pour ces tests, les données ne sont pas simples à interpréter et ne sont pas simples à obtenir sans perturber le fonctionnement normal du composant ou des interfaces ou du système observé. Quel que soit le niveau auquel le testeur sera amené à réaliser des tests non fonctionnels, le succès de ces tests sera conditionné par la qualité des spécifications et des modèles ayant guidé la conception. De façon caricaturale, pas de modèle, pas de tests non fonctionnels possibles.

Parmi les tests non fonctionnels, les tests structurels sont à part car ils vont analyser le logiciel non du point de vue de son comportement mais du point de vue de sa conception. On ne se pose pas la question sur ce que fait le logiciel (tests fonctionnels) ni sur la façon dont ce qui est fait est fait (tests non fonctionnels) mais sur la façon dont le logiciel est fait (la façon dont ce qui fait que ce qui doit être fait est fait). Il s'agit donc nécessairement de tests Boîte blanche. Ils peuvent être réalisés au niveau des composants, au niveau des interfaces ou au niveau du système.

2.5.2. Tests liés au changement et tests de non-régression

Le test a pour vocation de découvrir des défauts et de faciliter leur localisation. Quand un défaut est détecté puis corrigé, le logiciel doit être retesté avec les mêmes tests ou des tests ciblés sur le défaut original pour confirmer que celui-ci a bien été corrigé. Ce type de test est appelé « test de confirmation ». Il ne faut pas confondre cette activité de recherche et correction d'erreur, qui est une activité de test, avec le débogage qui est une activité de développement, et non une activité de tests.

De façon générale, quand une partie du logiciel est modifiée, suite à une correction ou suite à un ajout (par exemple en maintenance corrective ou en maintenance évolutive), rien ne garantit que la partie du logiciel non modifiée n'ait pas été impactée par la modification, c'est fréquemment le contraire qui

se produit. Il est en effet tentant, pour des raisons d'efficacité ou de facilité, d'utiliser un effet de bords d'un composant non documenté, car à usage interne ou en vue de tests (mise à jour d'un fichier de log^[4] par exemple). Un programmeur utilisant cet effet de bord dans sa logique algorithmique s'expose à la modification de celui-ci sans notification car prévu à seul usage interne. Seuls des tests ciblés sur ce composant *a priori* non impacté par la modification permettront de mettre en avant ce nouveau problème. L'autre cas fréquent est qu'un premier défaut, dans un composant A, masque la présence d'un défaut dans un composant B par suite d'une espèce de modification de trajectoire dans le comportement initial : d'une certaine façon, les deux défauts se compensent. La correction de l'un fait apparaître l'autre sans raison simple et prévisible. Ces tests, qui recherchent des défauts dans des parties déjà testées, sont appelés tests de non-régression et peuvent être définis comme tout test qui est la répétition de tests sur un logiciel déjà testé, après des modifications, pour mettre à jour tout défaut introduit ou découvert en résultat de ces changements.

L'étendue des tests de non-régression est basée sur le risque de ne pas trouver d'anomalie dans un logiciel qui fonctionnait auparavant. Ils doivent avoir été prévus lors de la conception initiale des tests et s'appliquent à tous les niveaux et à tous les types de tests : tests fonctionnels, tests non fonctionnels et tests structurels

Les suites de tests de non-régression seront exécutées de nombreuses fois et elles évoluent, en général, lentement (les besoins et exigences ne sont pas modifiés de façon incessante). Les tests de non-régression sont donc de bons candidats à l'automatisation.

2.6. Conclusion

Comme nous l'avons vu, les tests peuvent (et doivent) être un guide lors des différentes phases d'un projet logiciel et ceci quel que soit le cycle de développement retenu, qu'il soit « classique » avec le cycle en V ou « agile » avec les méthodes itératives et incrémentales. Habillement préparés et intelligemment prévus lors des phases de conception, ils seront plus simples à concevoir et à exécuter durant les phases de réalisation, de déploiement ou de maintenance.

Le point crucial est la définition d'une architecture testable permettant à la fois un développement de composants fiables, simples à intégrer et rendant possible l'automatisation du jeu et des tests lors des différentes phases de déploiement. En effet, sans automatisation, l'effort consacré aux tests sera grand lors des phases initiales de développement, ce qui peut être acceptable au vu de l'amélioration de la qualité constatée, mais deviendra pesant au fil des *releases* sans réelle justification.

Le succès actuel des démarches agiles est sans aucun doute dû en grande partie à cette insertion au cœur du développement de l'automatisation des tests.

Notes

- [1] Voir, par exemple le chapitre 4 de l'ouvrage *Architecture logicielle - Concevoir des applications simples, sûres et adaptables* de J. Printz.
- [2] Voir par exemple l'article de Yves Le Traon et de [Benoit Baudry](#) « Test d'intégration d'un système à objets - planification de l'ordre d'intégration » dans les actes de la conférence LMO 2006, pages 217-230.
- [3] Middleware orienté messages ou « *Messages Oriented Middleware* ».
- [4] Un fichier de log ou « *Log File* » est un fichier qui regroupe, généralement de façon chronologique, les différents événements survenus au niveau d'une application ou d'un système.

Tester efficacement : les différentes stratégies

Dans le cadre du test des logiciels, deux grandes familles de techniques de tests se complètent : les techniques dites « statiques » qui vont s'appuyer sur les sources du logiciel à tester sans le faire fonctionner et les techniques dites « dynamiques » qui s'appuieront sur l'exécution du logiciel. Dans le premier cas, on peut tester, dans le sens « trouver des erreurs de conception de réalisation ou de spécification » sans que le logiciel soit complet ou sans que l'on dispose de la plateforme cible, puisque l'on ne fait pas réellement fonctionner le logiciel ; c'est un véritable plus pour ce type de techniques. Par contre, celles-ci ne pourront être mises en œuvre que par des testeurs ayant une bonne connaissance du langage de programmation utilisé. En effet, ceux-ci devront lire et interpréter le code manuellement ou, tout du moins, comprendre les messages produits par les outils d'analyse utilisés. La deuxième famille de tests nécessite que le logiciel soit exécutable ; ceci peut repousser l'activité de test à la fin du projet, sauf dans le cas, de plus en plus fréquent où le logiciel est développé selon le paradigme de l'agilité qui préconise (entre autres) de travailler par itérations incrémentales. De cette sorte, on dispose en permanence d'un logiciel exécutable et donc testable au sens test dynamique. D'un point de vue pratique, l'automatisation possible des tests « dynamiques » combiné au fait d'être basé sur un résultat d'exécution apportera deux intérêts majeurs de ce type de tests vis-à-vis des tests statiques :

- Les tests dynamiques pourront être mis en œuvre par des testeurs « non-informaticiens » ayant une bonne connaissance métier mais pas forcément au fait de des techniques utilisées pour la réalisation du logiciel ;
- Les tests dynamiques pourront facilement être utilisés dans le cadre des tests de non-régression.

Les tests statiques sont donc fondamentalement différents des tests dynamiques. Cependant, il ne faut pas oublier que de nombreuses caractéristiques sont communes à ces deux familles ; il s'agit par exemple des propriétés pouvant être recherchées :

- Propriétés fonctionnelles
- Propriétés au sens de la norme ISO 9126 (fiabilité, la performance, l'utilisabilité, etc.)

Il s'agit également de la possibilité de pouvoir être utilisées à différents niveaux de détails :

- Au niveau des composants élémentaires
- Au niveau des interfaces
- Au niveau du système

Et enfin, dans les deux cas, ces techniques cherchent à découvrir des erreurs ! Il sera donc profitable de combiner autant que possible ces deux types d'approche des tests qui sont complémentaires.

3.1. Aperçu des stratégies de tests dynamiques

Dans le cadre des tests dynamiques, le testeur va exécuter le logiciel qui n'est peut-être que partiellement réalisé. Ce logiciel a un comportement qui dépend de son interaction avec son environnement. Dans le cas le plus simple, il s'agit d'un composant qui prend des paramètres en entrées et qui réalise un calcul ou une action en fonction de la valeur de ces paramètres. Dans des cas plus complexes, il peut s'agir d'un système qui interagit avec un autre système (par exemple une base de données) ou qui interagit avec un utilisateur (par exemple à travers un formulaire). Dans tous ces cas, le rôle du testeur, qui cherche à découvrir des erreurs, est de trouver les « bonnes » valeurs d'exécution, c'est-à-dire les valeurs d'environnement qui auront le plus de chance de mettre en défaut le logiciel. Il ne peut en effet espérer les essayer toutes, du fait de la combinatoire présente dans le comportement des logiciels – il faut garder à l'esprit que le test exhaustif d'un simple opérateur d'addition de deux nombres entiers codés sur 32 bits prend plus de

100 années à raison d'un milliard d'opérations de test par seconde. Il lui faut donc une stratégie pour choisir des valeurs de test. Nous abordons ici à grands traits, les différentes stratégies possibles et pour simplifier notre discours, nous nous plaçons volontairement dans le cas du test d'un composant dont le comportement dépend uniquement de paramètres passés lors de son lancement.

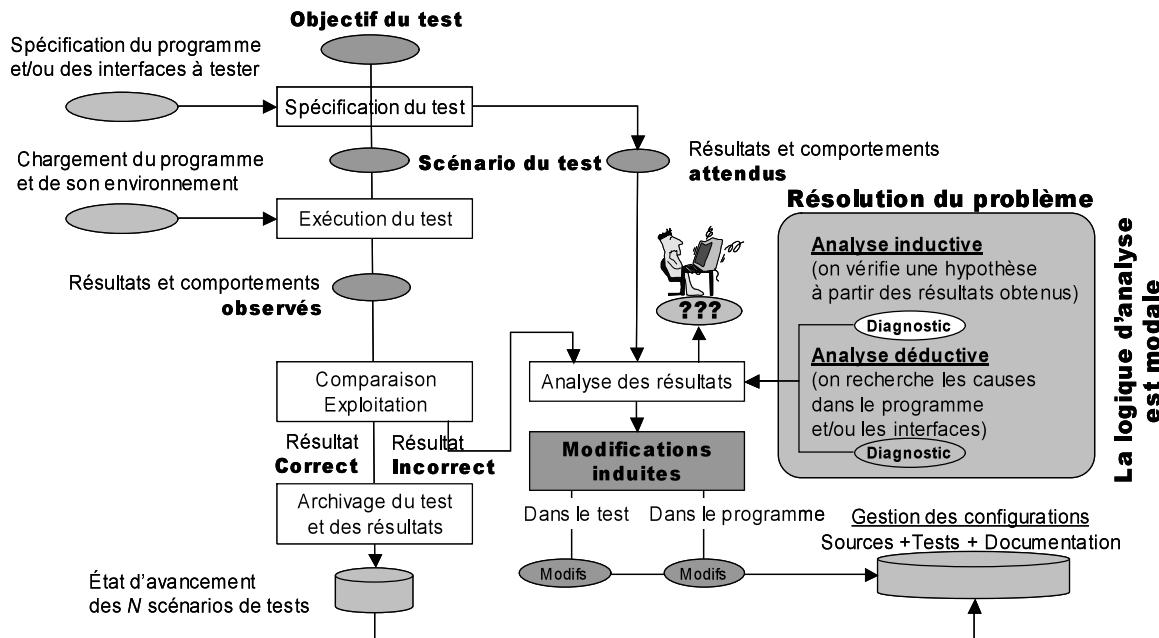


Fig. 3.1 Processus de tests dynamiques

Le processus de tests dynamiques (illustré figure 3.1) utilise les spécifications à différents niveaux : obligatoirement lors de l'analyse des résultats mais aussi au niveau de la génération des données de tests car seules les spécifications peuvent définir le comportement attendu du logiciel dans un cas donné.

Examinons maintenant quelles techniques différentes peuvent être employées pour générer un ensemble de données de tests.

3.1.1. Techniques aléatoires

La première approche, sans doute la plus naïve, consiste à se dire que plus on essaiera de valeurs distinctes plus on aura de chance de découvrir une erreur.

Le nombre l'emportant sur la qualité, il suffit alors de tirer au hasard un grand nombre de valeurs. Le seul critère de choix étant donc le hasard, on nomme cette technique « aléatoire » (*Random testing*). Il est d'ailleurs amusant de constater que le résultat de cette stratégie de test est lui aussi aléatoire : différentes analyses [Ntafos 01, Chen 94, Th 91] montrent, sans donner de réelle explication, que dans certains cas cette stratégie permet de détecter de nombreuses erreurs alors que dans d'autres cas, elle s'avère peu efficiente. On peut supposer que cette stratégie marche bien lorsque le logiciel est de bonne qualité vis-à-vis de sa testabilité, et fonctionne mal dans le cas contraire. Si l'on veut améliorer l'efficacité de cette stratégie, on peut contraindre le choix des valeurs par différents critères statistiques comme le respect de la distribution des données en entrées ou de la fréquence relative des cas d'utilisation concernés. Dans ces derniers cas, on utilise quelques fois le terme « tests statistiques ».

En tout état de cause, l'intérêt majeur de cette stratégie est sa grande simplicité de mise en œuvre concernant la génération des jeux de valeurs qui peut être complètement automatisée ; c'est moins vrai en ce qui concerne la génération de l'oracle sauf dans des cas rares en pratique où les spécifications sont suffisamment précises pour permettre une utilisation automatique (toute exploitation manuelle des résultats ruinerait l'intérêt de cette stratégie).

Enfin, il est bon de compléter l'ensemble des valeurs générées automatiquement par quelques jeux de valeurs particulières ou atypiques, comme des valeurs aux limites ou mettant en avant un comportement particulier.

L'aspect aléatoire peut également être utilisé dans un objectif différent : celui de tester les tests afin d'en mesurer la qualité. Le principe est fort simple : supposons qu'un ensemble de jeux de valeurs E soit utilisé pour tester un composant et supposons que cet ensemble ne mette en avant aucune erreur. Sans autre information, le testeur peut se demander s'il ne trouve pas d'erreur parce qu'il n'y en a pas ou plus ou parce que son ensemble de jeux de valeurs est sans intérêt. Une technique de tests aléatoires, nommée « test par mutation » consiste à modifier aléatoirement le composant, et donc, sous l'hypothèse que le composant est sans erreur, à introduire un ou plusieurs défauts. Ce composant modifié est appelé un mutant. On teste alors le mutant avec les valeurs de l'ensemble E. Si aucune erreur n'est découverte, on est en

droit de se poser la question de la qualité de cet ensemble de valeurs de test. Si par contre, une erreur est détectée, on dit que le mutant est tué, et ceci renforce la confiance dans la qualité de cet ensemble de valeurs E. La difficulté de ce genre de technique est de pouvoir différencier un mutant correct (c'est-à-dire qui réalise ce que l'on attend du composant) d'un mutant incorrect (c'est-à-dire qui ne réalise plus ce que l'on attend du composant). En effet, pour un même problème, il existe plusieurs solutions pouvant être syntaxiquement proches ; c'est une des raisons pour lesquelles, on se limite généralement, dans ce genre de test, à n'opérer qu'une modification par mutation.

Par ailleurs, tout ceci étant coûteux, on réservera le test par mutation à des cas précis où des doutes réels existent sur la qualité des jeux de tests utilisés.

3.1.2. Techniques Boîte noire

Lorsque l'on ne souhaite pas s'en remettre au hasard pour produire des jeux de valeurs de tests, le testeur pourra adopter une stratégie plus construite comme, par exemple, une des techniques de test Boîte noire.

Sous le terme « test Boîte noire » (*black box testing*) on désigne un ensemble de techniques dont les caractéristiques communes sont de se baser uniquement sur les spécifications lors de l'élaboration des jeux de valeurs de test et non sur l'utilisation des détails de réalisation.



Fig. 3.2 Tests Boîte noire

Cette restriction est souvent volontaire, c'est-à-dire que le testeur dispose des sources, les comprend, mais s'en interdit l'utilisation afin de ne pas se laisser influencer lors du choix des valeurs de test par la façon dont le logiciel a été réalisé. En effet, le code peut être vu comme une écriture possible du

raisonnement mené par le programmeur ou par l'équipe de spécification pour répondre au problème posé. Si le raisonnement est incorrect mais convaincant, le testeur peut se trouver à son tour dans l'erreur. En s'interdisant l'utilisation des sources, le testeur travaille moins sous influence de ces raisonnements initiaux et pourra découvrir des erreurs subtiles de conception.

Dans ce contexte, toute spécification peut être utilisée : spécifications informelles, comme un document en langue naturelle, spécification formelle, comme un automate ou un réseau de Petri, spécifications semi-formelles, comme un schéma UML ou un « *use case* ». Il est évident que plus les spécifications seront précises, plus elles pourront être utilisées avec profit. Ces spécifications peuvent décrire le format des interfaces (types et nombre des paramètres, type du ou des résultats), les résultats et comportements attendus pour des valeurs conformes aux spécifications des interfaces ou, mais cela est malheureusement non systématique, les résultats et comportements attendus pour des valeurs non conformes aux spécifications des interfaces.

Nous détaillerons au chapitre suivant les techniques Boîte noire les plus utilisées :

- Tests par partitionnement des domaines
- Tests par utilisation de graphe causes-effets
- Tests aux limites
- Tests à l'aide de diagrammes états/transitions

Mais avant cela, examinons brièvement, les tests Boîte blanche, techniques complémentaires des tests boîte noire.

3.1.3. Techniques Boîte blanche

À l'opposé des techniques Boîte noire, les techniques de test dites Boîte blanche, appelées aussi techniques de « test structurel » se basent sur les détails de réalisation du logiciel (ou sa structure) pour générer les valeurs de tests.

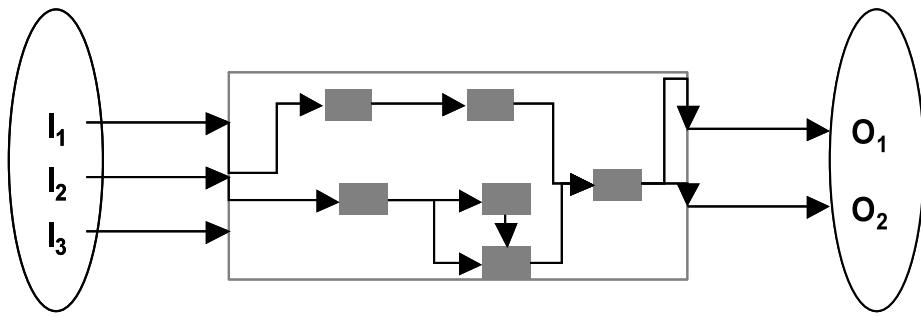


Fig. 3.3 Tests Boîte blanche

Ces techniques, détaillées au chapitre 5, reposent sur l'exploitation de graphes : le graphe de contrôle, qui reflète la partie « contrôle » (instructions conditionnelles, instructions de répétitions, instructions de branchements) ou le graphe flots de données qui synthétise la partie « flots de données » (*i.e.* le suivi de l'accès aux données) du logiciel. D'une certaine façon, ces graphes décrivent les chemins menant d'un point d'entrée du logiciel à une de ses sorties. Les « aiguillages » le long de ces chemins sont les points de contrôle dont l'effet dépend de la valeur d'expressions conditionnelles, et donc de la valeur des données en entrée du logiciel.

On peut ainsi sélectionner des données d'entrées conduisant à exercer une partie spécifique du logiciel, ce qui permet de rechercher des erreurs de réalisation peu visibles sans la connaissance des détails de réalisation présents dans ces sources.

Prenons l'exemple d'un composant calculant le produit de deux nombres entiers x , y et supposons que le programmeur ait voulu différencier, par souci d'efficacité (pas nécessairement heureux), le cas où l'un des deux paramètres est nul mais que suite à une erreur de frappe il ait écrit « if ($x==10$) » au lieu de « if ($x==0$) » ; la génération de jeux de tests à l'aide des seules spécifications (tests Boîte noire) ne permettra pas *a priori* de distinguer le cas où x vaut 10 des autres cas. *A contrario*, l'analyse des sources, et en particulier l'exploitation du graphe de contrôle mettra en avant ce détail (branchement lié à la valeur 10) et conduira à distinguer cette valeur spécifique. On peut penser que ces valeurs de test permettront de découvrir l'erreur de codage.

La structure du logiciel à tester peut servir également à définir des critères de couvertures qui seront utilisés pour évaluer la pertinence des tests menés. Les principales couvertures sont :

- **Couverture des instructions** : A-t-on exercé chaque instruction du logiciel lors des tests ?
- **Couverture des branches** (ou des décisions) : A-t-on, pour chaque décision (*i.e.* expression conditionnelle), exercé la partie VRAI (l'expression est vraie) et la partie FAUX (l'expression est fausse) ?

D'autres types de critères de couverture peuvent être définis, en particulier liés à l'utilisation des variables (par exemple concernant les chemins allant de l'affectation d'une variable à son utilisation dans une expression). Ces différents critères de couverture sont développés dans le chapitre 5.

3.2. Aperçu des stratégies de tests statiques

Des erreurs de conception ou de réalisation peuvent être soupçonnées alors qu'il n'est pas encore possible de faire fonctionner le logiciel. L'activité de test n'en est pour autant pas nécessairement rendue impossible : des tests « statiques » vont pouvoir être menés. En effet, dans le cadre des tests statiques, le testeur cherchera à trouver des erreurs non pas en exécutant le logiciel, mais en analysant les documents décrivant sa conception et sa réalisation, et en particulier, le code source. On distinguera les procédés de revues, connus sous le nom de « revue de code », « revue par pairs », « revue technique » ou « inspection » et les procédés d'analyse statique.

3.2.1. L'analyse statique de code

Dans le cas de l'analyse statique de code, on recherchera à mettre en relief (à l'aide généralement d'un outil proche d'un compilateur) un certain nombre d'aspects élémentaires du logiciel. Ces aspects peuvent être le respect de règles de codages :

- Utilisation de constructions syntaxiques non ambiguës : on évitera par exemple l'appel d'une fonction comportant un effet de bord lors du

passage des paramètres

$f(a, a=b)$ qui peut conduire, selon le compilateur à deux évaluations distinctes $f(b,b)$ ou $f(a,b)$.

- Utilisation de règle de dénomination commune à l'entreprise ou au projet.
- Utilisation d'en-têtes prédéfinis, etc.

On peut également chercher à vérifier que la complexité du code est acceptable en mesurant (moyenne et valeurs extrêmes) :

- Le niveau d'imbrication des boucles
- Le nombre d'instructions par méthode ou sous-programme
- Le nombre paramètres par méthode ou sous-programme

On peut également tenter de vérifier que le programme ne contient pas d'instruction qui le conduiront à se comporter de façon erronée comme :

- L'utilisation de variables non initialisées.
- L'utilisation de boucles conduisant à « sortir » d'un tableau.
- La réalisation d'opérations arithmétiques pouvant entraîner un dépassement de capacités de représentation (par exemple, la division par un nombre trop petit).

On peut enfin rechercher des informations sur la structure du logiciel en comptabilisant :

- La présence de code mort (c'est-à-dire des parties du logiciel qui, structurellement, ne peuvent jamais être exécutées).
- Le nombre de méthodes par classe ou le nombre de sous-programme du logiciel.
- Le nombre de classes dérivées par classe.
- Le nombre de classes utilisant une classe donnée, etc.

Récapitulatif des métriques sur les classes (938)						
Métriques	Nom	Moyenne	Valeur min.	Valeur max.	% OK	% non calculé
Ratio des classes qui utilisent une classe	R_CDUSERS	0,73	0	1	82,42	4,8
Ratio de classes filles	R_IN_NOC	0,97	0,01	1	98,43	4,8
Nombre de lignes de la classe	TAILLEC	79,89	0	953	89,92	4,8
Nombre de classes utilisant directement la classe	USABLE	9,93	0	259	74,36	4,8
Ratio du nombre de classes héritées	R_IN_BASES	0,78	0,17	1	97,87	4,8
Nombre cyclomatique moyen des méthodes de la classe	CVG_VG	2,18	0	20,5	93,84	4,8
Profondeur maximum d'héritage	in_depth	0,93	0	3	97,76	4,8
Nombre de classes utilisées directement par la classe	CDUSED	12,59	0	128	35,61	4,8
Présence d'attributs publics	CPUBLIC	0,27	0	1	72,9	4,8
Nombre de classes qui utilisent directement la classe	CDUSERS	4,11	0	202	82,42	4,8
Pourcentage de commentaires par classe	PCOM	46,49	0	550	87,35	4,8

Fig. 3.4 Exemple de mesures élémentaires

Prises individuellement, ces valeurs élémentaires n'ont pas forcément un grand sens. Aussi, on essaye ensuite de les agglomérer puis de les comparer à des valeurs seuils afin de les mettre en rapport avec des critères qualitatifs (au sens de la norme ISO 9126, qui définit six facteurs principaux et vingt et un facteurs secondaires de qualité) comme la complexité, la portabilité, la maintenabilité ou la testabilité.



Fig. 3.5 Facteurs principaux et facteurs secondaires de la norme ISO 9126

Ces données agrégées peuvent ensuite être synthétisées à l'aide de graphiques permettant une analyse rapide des points forts ou des points faibles d'un logiciel ou encore permettant de mesurer son évolution vis-à-vis de ces critères au fil des différentes versions.

Il est possible de systématiser ce genre d'approche et d'essayer ainsi d'adapter l'effort de test en fonction de la difficulté ou de la nécessité pressentie en définissant des « mesures » ou « métriques » qui vont permettre de chiffrer quantitativement les différents critères analysés par l'analyse statique. Passant brièvement en revue ces différentes métriques.

3.2.2. L'utilisation de métriques

Le pan de l'analyse statique connu sous le terme de « qualimétrie » ou de « mesure » du logiciel est un domaine d'analyse dans lequel on cherchera à mesurer la qualité du logiciel grâce à une métrique donnée.

Les métriques les plus connues sont :

- **Le nombre de lignes de code compté en KLOC (Kilo Lines Of Code)** peut distinguer (ou pas) le nombre de lignes de commentaire, le nombre de lignes vides, le nombre de lignes d'un module, d'une classe, etc.

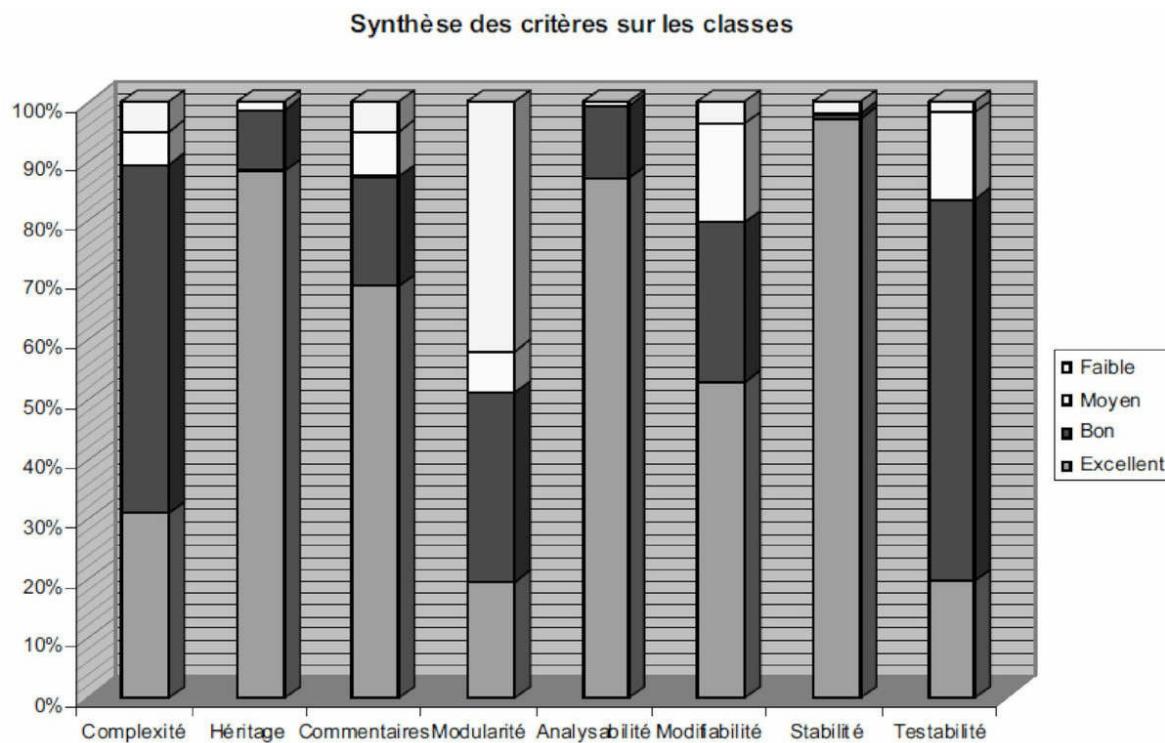


Fig. 3.6 Exemple de synthèse de mesures élémentaires

- **Le nombre cyclomatique de McCabe** : introduit en 1976 par McCabe^[1] analyse le graphe de contrôle du programme et calcule le nombre de chemins linéaires indépendants appelés nombre ou complexité cyclomatique. Plus le nombre cyclomatique est grand, plus il y aura de chemins d'exécution différents et plus le code sera difficile à comprendre ou à tester. On admet que le nombre de test d'un composant doit être du même ordre que son nombre cyclomatique et que ce nombre devrait rester inférieur à 15 ;
- **Les métriques d'Halstead (77)** : cet ensemble de métriques^[2] évalue la taille du programme en comptant le nombre d'opérateurs, d'opérandes et leur nombre d'apparition ; il s'agit plutôt d'une métrique de maintenance et, on pourra se contenter de retenir la formule permettant de calculer l'indice de maintenabilité :

$$MI_{code} = 171 - 5,2 * \ln(aveV) - 0,23 aveCCe - 16,2 * \ln(aveLOC)$$

$$MI_{comment} = 50 * \sin(\text{Sqrt}(2,4 * perCM))$$

où, *aveV* est le volume moyen par module, *aveCCe* est la complexité cyclomatique moyenne par module, *aveLOC* est le nombre moyen de lignes par module et *perCM* est le pourcentage moyen de lignes de commentaires par module

Plus la valeur est élevée plus le logiciel est facilement maintenable. Les seuils admis sont :

- 85 : maintenance facile
- 65-85 : maintenance correcte
- < 65 : difficulté à maintenir

Selon le contexte, on intègre ou non la part relative aux commentaires ($MI_{comment}$). Les constantes doivent être ajustées à chaque projet en particulier celles relatives aux commentaires :

- L'indice de maintenabilité peut être utilisé pour
- L'analyse de l'évolution des versions d'un logiciel dans le temps
- L'analyse de l'uniformité du codage au sein d'un même projet
- Mettre en rapport différents projets réalisés dans des contextes comparables

Exemple : l'indice de maintenabilité du noyau et des programmes utilisateurs de FreeBSD diminuent avec les années (mais faiblement)^[3].

On remarquera que l'on peut artificiellement augmenter la valeur de l'indice de maintenabilité sans pour autant améliorer la qualité du logiciel. Il suffit, par exemple, de découper les gros modules en plus petits ou d'ajouter artificiellement des commentaires

- **Les métriques d'Henry et Kafura (81)** : ces métriques, adaptées au contexte objet, calculent la complexité d'un module par rapport à son contexte en intégrant le nombre d'informations « entrantes » et « sortantes ».
- **Les métriques de Chidamber et Kemerer (94)** : ces métriques, au nombre de six mettent l'accent sur la conception des classes^[4] :
 - WMC (*Weighted Method per Class*) : compte le nombre de méthodes d'une classe pondéré potentiellement par le nombre cyclomatique de la méthode, le nombre d'attribut des méthodes (phase de conception). Cette métrique reflète la complexité d'une classe et tente de prédire l'effort de développement ou de maintenance. Une valeur élevée traduit un grand effort de développement et de maintenance ainsi qu'une faible possibilité de réutilisation (classe trop spécifique).
 - DIT (*Depth of Inheritance Tree*) : compte le nombre maximal de classes ancêtres pour atteindre la racine. Une valeur élevée traduit une plus grande complexité de compréhension de la classe, une plus faible possibilité de réutilisation du fait d'une moins grande généricité, une complexité globale plus grande en nombre de développement. A contrario, une valeur faible traduit une mauvaise utilisation du concept objet.
 - NOC (*Number Of Children*) : compte le nombre de descendants directs d'une classe. Cette métrique reflète l'impact d'une classe sur les autres classes. Une valeur élevée traduit une bonne possibilité de réutilisation (car générique) mais probablement une mauvaise abstraction et aussi l'importance de la classe dans le système et donc un effort de test important. Une valeur moyenne proche de zéro traduit une mauvaise utilisation du concept objet.

- CBO (*Coupling Between Object Classes*) : compte le nombre de couplages (invocation de méthode, utilisation d’attributs, traitement d’exception...) entre une classe et le reste du système. Cette métrique reflète l’interdépendance entre les constituants du système. Une valeur élevée traduit des difficultés de maintenance, des difficultés de test, des difficultés de réutilisation. On cherchera à assurer le paradigme « forte cohésion et faible couplage ».
- RFC (*Response For a Class*) : compte le nombre de méthodes qui peuvent être appelées (directement) à la réception d’un message par un objet de la classe. Reflète la complexité interne d’une classe. Une valeur élevée traduit des difficultés de compréhension ainsi qu’un plus grand effort lors des tests.
- LCOM (*Lack of Cohesion in Methods*) : compte le nombre de méthodes (prises 2 à 2) qui ne partagent pas un accès à un attribut commun moins le nombre de méthodes (prises 2 à 2) qui le font ; la valeur est bornée à 0. Reflète l’absence de cohésion interne d’une classe. Une valeur élevée traduit une faible cohésion de la classe, une possibilité de découpage entre plusieurs classes et une complexité due à une probable forte disparité de la classe.
- **Les métriques de Li & Henry (93)** : celles-ci reprennent en partie les métriques de Chidamber et Kemerer (WMC, DIT, NOC, RFC, LCOM) et introduisent cinq nouvelles métriques pour définir une mesure complète sur l’effort prévisible de maintenance et deux pour mesurer le couplage entre classes :
 - MPC : *Message Passing Coupling (fan out)*
 - DAC : *Data Abstraction Coupling* (ambiguë dans sa définition)
 - et trois pour mesurer la complexité des classes
 - NOM : *Number Of local Methods*
 - SIZE1 : nombre de points virgules dans une classe
 - SIZE2 : nombre d’attributs + NOM

L’ensemble de ces métriques peuvent être utilisées pour comparer différentes parties ou différentes versions d’un logiciel et essayer d’estimer où l’effort de test devra être porté en priorité.

Cependant, si ces mesures ont le grand avantage d'être pleinement automatisables, elles ne donnent pas nécessairement une vision claire de la qualité d'un logiciel et sont de peu d'utilité pour détecter un éventuel défaut. C'est par contre le rôle des revues de code ou inspections.

3.3. Revue de code, revue technique, inspection

Une revue ou une inspection est l'examen d'un produit par un être humain ; selon la norme « ANSI/IEEE Std. 1028-1997 » il s'agit d'une « technique d'évaluation en détail, d'un document, d'un module de code source, d'une correction, etc. par un groupe de personnes pour détecter les défauts, les violations des standards et des conventions, ainsi que tout autre problème ». Ce type de technique est démontré comme très efficace pour améliorer la qualité du logiciel et permet de trouver de nombreux défauts de spécification de conception ou de réalisation. Plusieurs types de revues peuvent être mis en place :

- *buddy check* ou *desk check* ;
- *walkthrough* ;
- *inspection*.

Les inspections sont les revues les plus formelles, mais, toute forme de revue bénéficiera d'un peu de formalisation, en particulier, on prendra garde à :

- Définir des critères précis pour le choix des personnes impliquées
- Définir des règles précises pour la conduite des réunions
- Terminer la revue sur un consensus
- Garder trace du processus mis en place et chercher à l'améliorer sans cesse

Chaque type de revue vise une série d'objectifs comme :

- Trouver des défauts
- Vérifier la conformité par rapport aux spécifications, aux standards
- Vérifier l'exhaustivité des fonctions
- Viser à une approbation du produit

Afin de mener à bien une revue il est nécessaire d'obtenir une certaine exhaustivité des critères étudiés et il faut éviter de changer d'objectifs en cours de revue. Pour cela, il sera bon, ou tout du moins fortement conseillé, d'utiliser une « *check list* ». Cette liste dépendra à la fois de l'organisation (entreprise, service) et du produit étudié (exigence, documentation, code, tests). On prendra garde à capitaliser l'expérience des revues faites en enrichissant systématiquement cette liste avec le temps.

Analysons brièvement les différentes revues pouvant être menées.

3.3.1. Revue de type « *buddy check* »

Une revue de type *buddy check* ^[5] est une revue informelle du produit qui est conduite par l'auteur et à laquelle sont associées une ou plusieurs personnes (les copains). Ce type de revue vise à mettre à jour des problèmes de compréhension dans les choix de mise en œuvre, des problèmes de présentation, de style ou encore d'imaginer des solutions alternatives. Une étude IBM/TRW/MITRE menée en 1999 montre que ce type de revue, peu onéreuse et implicitement utilisée dans le cadre des développements agiles, ne permet de détecter que quelques défauts, par « effet de bord », ce qui est normal puisque cela n'est pas forcément l'objectif principal de ces revues. On remarquera que le caractère « léger » de ce type de revue va à l'encontre de la production de document de synthèse ou de recommandations, ce qui peut être pénalisant sur des projets complexes.

3.3.2. Revue de type « *walkthrough* »

Une revue de type *walkthrough*^[6] apporte légèrement plus de formalisation. Elle sera conduite par l'auteur qui « parcourt » le document (code, documentation, exigences, etc.) et l'expose lors d'une réunion à un groupe de personnes qui interagissent avec l'auteur et apportent des remarques. Le caractère légèrement formel provient du fait que, contrairement au type de revues précédentes, l'auteur doit préparer son exposé l'obligeant ainsi à apporter plus de rigueur à sa présentation. On peut ainsi chercher à vérifier le caractère exhaustif d'une solution ou imaginer une solution alternative. Ce type de revue permet également de former de nouveaux arrivants dans le projet. On trouvera, par contre, généralement encore peu de défauts et la

documentation produite sera fréquemment réduite aux supports utilisés pour la présentation. Le principal souci est de définir un critère d'arrêt plus pertinent que « c'est l'heure ».

3.3.3. Revue de type « inspection »

Introduit par M.E. Fagan en 1976 chez IBM^[7], le processus d'inspection privilégie l'aspect formel et se différencie des autres revues par le fait qu'un agenda est clairement établi, qu'il y a nécessairement préparation de l'inspection de la part des acteurs (avec par exemple distribution des documents suffisamment à l'avance) et qu'un rapport est obligatoirement produit. Une inspection définit six rôles bien précis dans le groupe :

- **Le modérateur** coordonne l'inspection :
 - Il vérifie que le produit est prêt pour l'inspection.
 - Il rassemble une équipe d'inspection.
 - Il assure que la procédure d'inspection est bien suivie.
 - Il vérifie le respect des échéances.
 - Il vérifie que le critère de sortie est atteint.
- **Le secrétaire ou scribe** prend les notes durant la réunion d'inspection. À ce titre, il doit fournir toute l'information pour que le rapport d'inspection soit suffisamment précis et non ambigu (ce qui implique une bonne compréhension du problème et un bon esprit de synthèse).
- **Les inspecteurs (Reviewers)** évaluent le produit, donnent leur point de vue et participent à la rédaction du rapport d'inspection.
- **Le lecteur** guide les inspecteurs dans l'évaluation. À ce titre :
 - il synthétise le contenu du produit,
 - il fait ressortir la logique du produit,
 - il focalise les inspecteurs sur les parties essentielles du produit.
- **Le vérificateur** apporte un soutien logistique en coordonnant la rédaction technique du rapport d'inspection. Par la suite il sera chargé de veiller à ce que les modifications demandées aient bien été apportées.
- **L'auteur** apporte sa connaissance concernant la réalisation. Sa présence est indispensable car il peut fournir oralement des informations qui ne

sont pas exprimées dans le produit.

La réussite d'une inspection repose à la fois sur la qualité de l'équipe d'inspection mais aussi sur l'implication et le soutien de la direction. L'équipe d'inspection devra veiller à rester positive dans la communication (éviter les sarcasmes ou remarques blessantes) et ne pas chercher à sanctionner les auteurs ; sans cette attention, l'équipe d'inspection risque de mener un procès à charge contre l'équipe projet et perdre la confiance et l'adhésion de celle-ci. De même, une inspection se focalisera sur la détection des défauts, mais ne cherchera pas à les résoudre pendant le temps de l'inspection, afin de ne pas prolonger celle-ci de façon non contrôlée (le temps de détection d'un défaut n'est pas proportionnel au temps nécessaire à sa correction qui peut être singulièrement plus long). Afin d'aboutir rapidement et d'éviter une dérive de la revue, il sera important de respecter les procédures mises en place au démarrage. Enfin, le rapport devra être rédigé le plus rapidement possible.

Au sein de la direction, il faudra veiller à rester impliqué tout au long du processus d'inspection sans pour autant participer aux réunions, la présence de responsable hiérarchique pouvant induire des comportements (parade ou soumission) néfaste au bon déroulement de l'inspection. L'implication de la direction se fera sentir en particulier sur l'allocation du budget, du temps et des ressources humaines nécessaires pour mener à bien l'inspection. Enfin, la direction prendra garde de sanctionner les inspections mal conduites et non les mauvais produits et donc de féliciter les bonnes inspections de mauvais produits.

3.4. Le processus d'inspection en six étapes

Une inspection se conduit normalement en respectant six étapes, chacune ayant un rôle bien précis dans le processus complet :

- La première étape est fondamentale car c'est lors de cette étape que la base de l'inspection est mise en place. En tout premier lieu, cette étape aura pour objectif d'élaborer l'agenda de l'inspection et de formaliser la motivation de l'inspection. On essaiera également de mettre en place la

logistique en vérifiant l'implication de l'encadrement et en évaluant les moyens réels dont disposera l'équipe (ressources humaines, finances, support logistique interne). Une fois ces points réalisés, on constituera l'équipe d'inspection proprement dite en définissant le plus précisément possible, et en se référant aux rôles « classiques » présents dans une inspection, les qualifications requises, la mission et les responsabilités de chaque membre de l'équipe. Si nécessaire, on listera également l'ensemble des personnes « étrangères » au projet inspecté auxquelles il faudra faire appel au cours de l'inspection. Enfin, c'est dans cette première étape que l'on définira le contenu du rapport d'inspection (rappelons qu'il y a obligation de remettre un rapport à la fin d'une inspection) et de la « check list » qui servira de « trame » aux réflexions menées lors de cette revue.

- La deuxième étape est la réunion de démarrage où se rencontrent (peut-être pour la première fois) l'ensemble des membres de l'équipe d'inspection. Lors de cette réunion, le modérateur présentera le produit à inspecter (sans entrer dans les détails), les objectifs et les moyens de l'inspection, le rapport à produire, la « check list » les membres de l'équipe, leur mission respective ainsi que le temps estimé de travail pour chacun.
- La troisième étape peut alors démarrer. Cette étape, sans doute la plus prenante, est celle où les inspecteurs analysent le produit selon la technique choisie en suivant la « check list », rencontrent les experts et les personnes désignés pendant la phase de préparation ; s'ensuivent une liste d'avis, de commentaires et de recommandations rédigés par les inspecteurs.
- Lors de la quatrième étape, une nouvelle réunion est provoquée au cours de laquelle le modérateur collecte les données résultantes des inspections individuelles ; celles-ci serviront à rédiger le rapport final qui est ébauché à cette étape.
- La cinquième étape consiste en la rédaction du rapport d'inspection. Des recommandations, des demandes d'ajout ou de modifications à apporter au produit sont formulées par l'équipe et des objectifs en termes de délais de réalisation doivent également être formulés.
- La sixième et dernière étape concerne les suites à donner à l'inspection. Sur le plan du produit, on cherchera à vérifier que les modifications

demandées sont bien en cours de réalisation ou du moins, que leur planification a bien été établie. Sur le plan de l'inspection, c'est le moment du bilan, où l'on essaiera de capitaliser et où l'on réfléchira aux axes d'améliorations possibles du procédé.

Avant de conclure sur ce type de revue, étudions rapidement le format attendu d'un rapport d'inspection.

3.4.1. Plan type du rapport d'inspection

Le rapport d'inspection comprendra une page d'en-tête constituée :

- D'un identifiant unique ;
- De la date du rapport ;
- D'une identification du produit ainsi que ses auteurs ;
- La date, si nécessaire les heures et les lieux de l'inspection ;
- Une référence des documents utilisés pour l'inspection ;
- Les noms et les signatures de l'équipe d'inspection ; les signatures sont importantes car elles responsabilisent l'ensemble du groupe et montrent leur adhésion au rapport et à ses conclusions.

On pourra également mentionner le temps passé et le coût de l'inspection, une indication sur l'historique des inspections successives du produit, le niveau d'approbation du produit inspecté (accepté dans l'état, demande de modifications mineures, demandes de révision majeure, rejeté ou à refaire). On fera également figurer la liste des autres documents produits (comme la liste des recommandations).

Viendront ensuite le rapport lui-même, ses conclusions motivées sur le produit, une liste des recommandations étayées et, si possible, une proposition d'agenda concernant les actions d'améliorations. Il est important que l'ensemble du groupe d'inspection adhère au rapport et à ses recommandations afin que les demandes d'amélioration puissent être concrètement prises en compte dans un futur proche et ne restent pas lettre morte.

En annexe à ce rapport, figureront la « check list » utilisée, un résumé du déroulement de l'inspection et, dans le cas d'une série d'inspection sur le

même produit ou des mêmes équipes, une analyse et un bilan des erreurs le plus souvent rencontrées, des recommandations pour de possibles inspections futures.

De nombreuses études ont démontré l'intérêt économique d'une inspection ; c'est un type de revue qui coûte cher, car elle mobilise au moins six personnes sur une période assez longue, mais elle permet de découvrir tôt des erreurs difficiles à mettre à jour avec des techniques de tests dynamiques. C'est le cas en particulier pour les erreurs concernant le choix de l'architecture du logiciel, ou pour des erreurs graves mais difficiles à reproduire : interblocage, accès mal contrôlé à des ressources partagées conduisant à des incohérences, problème de famine ou d'équité, non-respect de contraintes temps réelles, etc.

Le procédé d'inspection peut également être appliqué avec profit aux tests eux-mêmes avec pour objectif d'améliorer les plans de tests, la qualité des scénarios, ou encore, la pertinence des jeux de valeurs choisies pour un cas de test donné.

3.4.2. Choisir un type de revue

Si l'objectif est de réduire au maximum les défauts ou de gagner en confiance sur un produit complexe ou critique, on choisira de mener une inspection ; si l'objectif est d'obtenir un consensus ou une familiarisation de l'équipe à un produit, on mettra en place un simple *walkthrough*. Si par contre on vise à obtenir un retour rapide sur un produit simple ou de faible niveau de risque, on se contentera d'un *buddy check*.

3.4.3. Processus de revue : conclusion

Les processus de revues apportent une amélioration significative de la qualité à la fois sur le logiciel réalisé mais aussi sur les documents ou les corrections produits. Cette amélioration concerne aussi la qualité des tests mis en place.

On constate également que la mise en place de revues permet d'améliorer les processus de développement en augmentant la visibilité de l'état d'avancement du produit. Une fois passées les phases initiales de méfiance, une meilleure communication au sein des équipes va se développer

permettant un partage d'expérience ainsi qu'une plus grande motivation et stimulation sur la qualité des logiciels et documents produits.

D'une certaine façon, la mise en place de revues permet aux équipes de passer d'un processus de « développement-correction » à un processus plus réfléchi basé sur la prévention des erreurs. Par effet de bord mécanique, l'effort de test à fournir par la suite pourra être réduit de façon très significative.

3.5. Tests dynamiques versus tests statiques : synthèse

Le testeur dispose d'une grande panoplie de méthodes et d'outils pour mener à bien sa mission de traque des erreurs. Son choix sera guidé par un certain nombre de données comme la possibilité ou non d'exécuter le logiciel (chose impossible dans les phases initiales de conception) la possibilité d'accéder au code source du logiciel, ou encore le type des erreurs recherchées (fonctionnelles, liées aux performances, conformité à des standards, etc.).

3.5.1. Choisir une technique de tests dynamiques

Le choix d'une technique de tests dynamiques nécessite avant tout que le logiciel à tester corresponde à un code exécutable, moyennant généralement, une mise sur banc de test avec « bouchonnage » des parties non réalisées. Le choix entre Boîte noire ou Boîte blanche sera lui conditionné par la possibilité d'accéder au code source. Néanmoins, même en cas d'accès au code, on préfère souvent utiliser des tests Boîte noire basés sur l'utilisation des spécifications, réservant l'accès au code pour mesurer la couverture des tests menés et donc, d'une certaine façon, la complétude de ceux-ci vis-à-vis du code potentiellement exécutable. On notera que certains types de tests, comme les tests d'acceptation, seront nécessairement menés à l'aide de techniques dynamiques.

3.5.2. Choisir une technique de tests statiques

Les techniques statiques n'exécutent pas le logiciel à tester et, de ce point de vue, ne sont pas comparables aux techniques dynamiques. Ainsi, elles

peuvent être employées à toutes les phases du cycle de vie du logiciel.

L'analyse statique va permettre, à l'aide d'un outil, de vérifier automatiquement certaines propriétés d'un logiciel fini ou partiellement fini et de découvrir ainsi certaines erreurs de conception.

Un processus de revue sera non automatisé mais permettra de découvrir des erreurs de différents types et à tous les niveaux de conception d'un logiciel, chose que ne permet ni un test dynamique ni l'emploi d'une technique statique. Par ailleurs, de nombreuses études ont montré que les revues sont un moyen très efficace pour réduire de façon importante le nombre d'erreurs ; les chiffres fréquemment avancés donnent de 60 à 95 % d'erreurs trouvées parmi les erreurs présentes à tous niveaux de la conception d'un logiciel. Appliquées tôt les revues permettent de réduire la durée totale de développement le coût et le temps des tests dynamiques ainsi que les risques de défauts lors du déploiement.

3.6. Conclusion

Différentes stratégies complémentaires peuvent être mises en œuvre pour tester un logiciel. Chacune se concentre plus particulièrement sur certains objectifs comme la mise en évidence de défauts dans la réalisation, la vérification de la cohérence ou du caractère complet des spécifications, le respect de standards, etc. Chacune déployera ses propres méthodes et sera supportée par des outils dédiés comme l'aide à l'exécution, l'aide à la collecte de données, la vérification structurelle, etc. Cependant, toutes ces stratégies partagent un certain nombre de points communs comme l'impossible exhaustivité, la qualité de la réflexion et l'importance des relations humaines dans tous les processus liés au test. En cela, la formation et l'expérience du testeur sont irremplaçables.

Notes

- [1] T. J. McCabe, « A Complexity Measure », *IEEE Transactions on Software Engineering*, vol. 2, N° 4, pp. 308-320, July 1976.
- [2] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*, Elsevier Science Inc. NY, 1977.
- [3] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman, « Using metrics to evaluate software system maintainability », *Computer*, 27(8) :44–49, 1994.
- [4] S.R. Chidamber and C.F. Kemerer, « A Metrics Suite for Object Oriented Design », *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, 1994.
- [5] Littéralement : « contrôle de copain ».
- [6] Littéralement : « passage à travers ».
- [7] Fagan, Michael E : « Design and Code Inspections to Reduce Errors in Program Development », *IBM Systems Journal*, Vol. 15, No. 3, 1976.

Concevoir efficacement des jeux de tests grâce aux spécifications

Un logiciel ou un composant logiciel peut être vu comme un automate qui agit en fonction de valeurs qu'il obtient au début ou en cours d'exécution. Son fonctionnement exhibe un ensemble de comportements distincts en nombre important. Parmi cette grande quantité de comportements différents, un grand nombre d'entre eux sont corrects, alors que certains autres sont incorrects ; le rôle du testeur est de trouver le plus efficacement possible les jeux de valeurs qui permettront de faire apparaître ces comportements incorrects lors de la phase des tests. Dans le cadre des tests fonctionnels ou Boîte noire, le testeur va choisir ces jeux de tests en utilisant les spécifications du composant à tester et en premier lieu celles qui concernent les paramètres en entrée du composant. L'enjeu est de combattre la combinatoire tout en maintenant une qualité suffisante des jeux de valeurs de test vis-à-vis de la détection de défauts.

Nous nous intéressons dans ce chapitre aux techniques usuelles employées dans le cadre des tests fonctionnels. Elles sont relativement anciennes et toutes ont fait leurs preuves. Elles sont par ailleurs pour la plupart complémentaires c'est-à-dire que l'on peut les combiner entre elles pour tester plus efficacement un composant, comme nous le verrons à l'aide des différents exemples qui illustreront ces techniques.

4.1. Réduire le combinatoire avec les techniques All Singles et All Pairs

4.1.1. Principes

Dans un certain nombre de cas, les données qui jouent un rôle sur le comportement du composant à tester sont en petit nombre (par exemple 3 ou 4) et prennent des valeurs dans de petits ensembles facilement énumérables (contenant par exemple 5 ou 6 valeurs distinctes). Dans ce cas, le problème est légèrement plus simple que dans le cas général car il est *a priori* possible d'énumérer toutes les combinaisons possibles des valeurs des paramètres (ce nombre de combinaisons est égal au produit des cardinaux des valeurs possibles en entrée). Il est cependant possible de réduire la combinatoire tout en gardant une qualité vis-à-vis des cas couverts comme nous allons le voir. Pour cela on va mettre en place une stratégie qui permet de générer des jeux de tests utilisant soit au moins une fois chaque valeur de chaque paramètre (technique *all singles*) soit au moins chaque valeur de paire de paramètres (technique *all pairs* ou *pairwise*). Nous présentons à l'aide d'un cas concret une façon simple et efficace de mettre en œuvre cette stratégie.

4.1.2. Illustration

Considérons le problème suivant : « *Comment tester efficacement un composant logiciel qui génère des scripts utilisés pour valider le comportement de serveurs web* ». Ce composant prend quatre paramètres en entrée et génère un script adapté à ces valeurs de paramètres. Ces paramètres sont :

- Le type du poste client à partir duquel le script sera exécuté ; les valeurs possibles sont « Linux Type1 », « Solaris », « Windows Me », « Vista », « Mac Os X », soit cinq valeurs distinctes.
- Le type du navigateur client ; les valeurs possibles sont « FireFox », « Internet Explorer » et « Netscape ».
- Le type de données téléchargées ; les valeurs possibles sont « jpeg », « avi », « mp3 » et « wave ».
- Le fait d'utiliser une connexion sécurisée ou non.

Dans cet exemple, il est possible d'énumérer toutes les combinaisons possibles des valeurs de paramètres en entrée et de réaliser ainsi un test exhaustif du composant. Cela a cependant un coût. Dans notre exemple, il faudrait réaliser $5*3*4*2 = 120$ tests différents. Si le temps passé pour faire

fonctionner le test prend une demi-heure (temps tout à fait réaliste au vu des contraintes techniques d'exécution ou d'analyse des scripts générés), il nous faudra passer 60 heures pour tester ce composant !

Afin de réduire ce coût, une première solution est d'adopter la stratégie dite *all-singles*. Celle-ci consiste à construire des jeux de tests couvrant au moins chaque possibilité pour chaque paramètre ; ici, il faudra que les jeux de tests couvrent au moins une fois chacun des systèmes client, chacun des navigateurs, chacun des types de fichiers multimédias, et chaque type de connexion. Par exemple, les deux configurations :

- (« Windows Me », « Netscape », « avi », « sécurisée »)
- (« Mac Os X », « Netscape », « mp3 », « non sécurisée »)

couvrent deux types de systèmes, un type de navigateurs, deux types de fichiers et tous les modes de connexion possible.

De façon générale, pour couvrir au moins une fois chaque valeur de chaque paramètre, il nous faut au moins N_1 jeux de valeurs où N_1 est le cardinal du plus grand ensemble de valeurs ; ici N_1 vaut 5 et correspond au cardinal de l'ensemble des systèmes possibles. Pour couvrir les valeurs des autres paramètres il suffit d'énumérer ces valeurs dans les jeux de tests construits. Ainsi, dans notre exemple, on construit cinq jeux de tests :

- (« Windows Me », ?, ?, ?)
- (« Vista », ?, ?, ?)
- (« Linux », ?, ?, ?)
- (« Solaris », ?, ?, ?)
- (« Mac Os X », ?, ?, ?)

puis, on complète en énumérant les valeurs possibles du second paramètre (en répétant si nécessaire plusieurs fois la même valeur) :

- (« Windows Me », « Internet Explorer », ?, ?)
- (« Vista », « Netscape », ?, ?)
- (« Linux », « FireFox », ?, ?)
- (« Solaris », « Netscape », ?, ?)
- (« Mac Os X », « FireFox », ?, ?)

et ainsi de suite jusqu'à obtenir un ensemble de jeux de tests respectant le critère *all singles* ; lorsque le testeur a le choix entre deux valeurs, il prend celle qui lui semble la plus représentative des cas d'utilisation réels du composant. Dans notre exemple, il est plus probable d'associer le navigateur « Internet Explorer » à un système « Windows » plutôt qu'à un système Unix. Les cinq jeux de valeurs suivants satisfont au critère « *all single* ».

- (« Windows Me », « Internet Explorer », « jpeg », « sécurisée »)
- (« Vista », « Netscape », « avi », « non sécurisée »)
- (« Linux », « FireFox », « mp3 », « sécurisée »)
- (« Solaris », « Netscape », « mp3 », « sécurisée »)
- (« Mac Os X », « FireFox », « avi », « non sécurisée »)

Cette technique permet de réduire très fortement le nombre de jeux de tests à considérer tout en maintenant une certaine qualité : toute valeur d'un paramètre est testée au moins une fois ; un oubli de réalisation ou une grosse erreur seront ainsi facilement détectés.

Néanmoins, dès que les erreurs dépendront d'une association particulière de paramètres, ce genre de stratégie ne marchera pas mieux qu'une stratégie purement basée sur le hasard (avec des efforts de réflexion et de construction en plus !). Dans notre exemple, il est clair que les erreurs subtiles concerneront plutôt l'utilisation d'un navigateur donné avec un type de fichier ou l'association d'un type de données avec un système ou encore l'association d'un navigateur avec un mode de connexion donné.

Si l'on veut obtenir la garantie de tester toutes ces possibilités sans retomber dans l'énumération totale des valeurs possibles en entrée, il nous faut utiliser une stratégie intermédiaire, comme la stratégie dite « *all pairs* ».

Cette stratégie vise à construire des jeux de tests de telle sorte que toutes les paires de possibilités soient couvertes au moins par un jeu de tests. Dans notre exemple, il faudra que tous les couples (système/navigateurs), (système/type de fichiers) et (navigateur/type de fichiers) apparaissent au moins une fois dans les jeux de tests.

Différentes approches sont possibles pour construire ces jeux de tests. Nous en proposons une ici fort simple. Pour cela, on commence par ordonner en ordre décroissant les variables par rapport à leur nombre de valeurs possibles.

Notons V1 et V2 le nombre respectif de valeurs possibles pour les deux premières variables. Dans notre exemple, cela donne Système (5 valeurs), Données (4 valeurs), Navigateur (3 valeurs) et enfin Connexion (2 valeurs).

On construit alors une table avec une colonne par variable et au moins $V1 \times V2$ lignes (ici cela donne $5 \times 4 = 20$ lignes).

Sur les V2 premières lignes, on met la première valeur de la première variable, puis une ligne blanche (qui sert uniquement à aérer le tableau), puis la seconde valeur de la première variable sur V2 lignes, une ligne blanche et ainsi de suite. On a ainsi rempli la première colonne.

Pour la seconde colonne, on alterne successivement les V2 valeurs de la seconde variable (en passant la ligne blanche). On a ainsi, par ligne, toutes les paires des deux premières variables.

À la fin de cette étape, l'ensemble des lignes correspond aux jeux de valeurs de test nécessaires pour tester tous les couples des deux premiers paramètres.

Pour les autres colonnes, on alterne les différentes valeurs des autres variables en faisant en sorte de construire des paires avec les autres colonnes. La seule précaution à prendre est de rompre des cycles lorsque le cardinal d'une colonne est multiple du cardinal d'une autre colonne. Par exemple, le nombre de connexions étant de deux, un cycle peut se produire avec la colonne correspondant au type de fichiers et, sans précaution, on risquerait de ne pas tester tous les couples (fichier, connexion).

Une fois la table construite, l'ensemble des lignes définit un ensemble de jeux de tests couvrant toutes les associations par paires possibles. On obtient donc une très bonne qualité de jeux de tests avec une combinatoire moindre. Dans notre exemple, le nombre de jeux de tests a été divisé par 6 ! On remarquera que l'ajout d'un paramètre supplémentaire (dont le nombre de valeurs possibles reste plus petit que le cardinal des deux premiers paramètres) ne nécessite que l'ajout d'une colonne et ne remet pas en cause le travail déjà effectué. Ceci permet de construire des stratégies de test incrémentales, en ne considérant pas d'emblée l'ensemble des paramètres, et de suivre ainsi le développement incrémental d'un composant que l'on construit petit à petit en intégrant de plus en plus de détails.

Tab. 4.1 Exemple de table all pairs

Système	Type de fichiers	Navigateur	Connexion
Windows Me	Jpeg	Internet Explorer	sécurisée
Windows Me	Avi	Netscape	non sécurisée
Windows Me	Wave	FireFox	sécurisée
Windows Me	Mp3	Internet Explorer	non sécurisée
VISTA	Jpeg	Netscape	non sécurisée
VISTA	Avi	FireFox	sécurisée
VISTA	Wave	Internet Explorer	non sécurisée
VISTA	Mp3	Netscape	sécurisée
Linux	Jpeg	Netscape	non sécurisée
Linux	Avi	FireFox	sécurisée
Linux	Wave	Internet Explorer	non sécurisée
Linux	Mp3	Netscape	sécurisée
Mac Os X	Jpeg	FireFox	sécurisée
Mac Os X	Avi	Internet Explorer	non sécurisée
Mac Os X	Wave	Netscape	sécurisée
Mac Os X	Mp3	FireFox	non sécurisée
Solaris	Jpeg	Internet Explorer	non sécurisée
Solaris	Avi	Netscape	sécurisée

Solaris	Wave	FireFox	non sécurisée
Solaris	Mp3	Internet Explorer	sécurisée

Une fois la table construite, le testeur peut supprimer des cas impossibles (sauf s'il souhaite vérifier le comportement du composant dans le traitement des cas impossibles). Il peut, bien entendu, également ajouter des tests qui lui sembleraient manquer : la méthode ne se substitue jamais à l'intelligence du testeur qui doit se souvenir que le problème est difficile et que son expertise et son expérience peuvent permettre, mieux que toute méthode, de trouver des jeux de tests plus pertinents.

4.1.3. Commentaires

Cette technique est très efficace, simple à mettre en œuvre et produit des documents de test facilement réutilisables. De plus, elle peut être utilisée à travers d'autres méthodes pour générer des jeux de valeurs à partir de cas de test. Malheureusement elle ne peut pas s'appliquer à tout type de composant logiciel. En effet, la plupart des paramètres prennent leurs valeurs dans des ensembles de cardinal très élevé (de l'ordre de 2^n , où n est la taille en bits de la représentation binaire des valeurs). Le nombre de cas de test étant dépendant de ces cardinaux, cette méthode ne pourra pas être appliquée directement. Il faut donc trouver d'autres stratégies ; c'est l'objet des sections suivantes.

4.2. Tester grâce aux classes d'équivalence

4.2.1. Principes

Lorsqu'il est physiquement impossible d'énumérer l'ensemble des combinaisons possibles des valeurs des paramètres du composant à tester, cas le plus fréquent, une façon de procéder est alors de se fier au hasard et de générer aléatoirement un ensemble de valeurs compatibles avec les valeurs attendues. Par exemple, si le composant à tester prend deux entiers positifs en paramètres, on générera un ensemble de couples de valeurs positives à l'aide

d'un générateur aléatoire. Cette approche, simple à mettre en œuvre, peut se révéler être peu efficace dans le sens où il est fort probable que des valeurs, bien que différentes, exercent le composant de la même façon sur toute une plage de données.

Cependant, afin d'optimiser la qualité des jeux de données des tests il est pertinent d'analyser plus en détail les spécifications du logiciel et d'essayer de déterminer ces plages de valeurs sur lesquelles le composant se comporte identiquement. On se ramène alors, de façon implicite, à des paramètres prenant des valeurs dans des ensembles petits (les classes d'équivalence).

Par définition, dans le contexte des tests Boîte noire ou « fonctionnel », une classe d'équivalence est un ensemble de valeurs pour lesquelles on ne peut distinguer le comportement du logiciel. En d'autres termes, quelle que soit la valeur choisie dans une classe d'équivalence, le logiciel aura le même comportement. Ce comportement sera par ailleurs soit correct soit incorrect.

Cette hypothèse de comportement identique pour toutes les valeurs d'une classe d'équivalence permet de réduire drastiquement le nombre de valeurs à utiliser : une seule valeur est suffisante par classe d'équivalence. Si cette valeur ne conduit pas à une exécution incorrecte, alors il en est de même pour toutes les valeurs de la classe. Si au contraire, une erreur de comportement doit survenir, n'importe quelle valeur permettra de la mettre à jour.

Il faut prendre garde à ne pas confondre comportement incorrect et valeur incorrecte. En effet, le logiciel peut refuser certaines valeurs qui seront des entrées invalides et avoir un comportement correct (généralement lever une exception ou renvoyer une valeur prédéterminée en erreur) ou avoir un comportement incorrect (ne pas se rendre compte que la valeur est invalide). De même, pour une valeur valide le composant peut avoir un comportement correct (il fait ce que l'on attend de lui) ou incorrect (il contient un défaut).

Afin de mettre en place cette stratégie de test, on essaie tout d'abord de reconnaître à l'aide des spécifications les classes d'équivalence valides, et les classes d'équivalence invalides. Quand les données en entrées ne sont pas liées entre elles, on peut ainsi facilement partitionner les domaines des valeurs en classes d'équivalence valides et invalides. C'est le cas par exemple si l'on doit tester une fonction qui calcule le nombre d'habitants d'un département français de métropole. Rien ne permet *a priori* de supposer que

le logiciel ait un comportement différent pour une valeur de département donnée. Il y a donc une seule classe d'équivalence valide : les numéros de département compris entre 1 et 95. Par contre on aura deux classes de données invalides : les numéros plus petits que 1 et les numéros plus grands que 95. Il est important de distinguer ces deux classes car le logiciel peut avoir été codé de telle sorte que les deux cas ne soient pas traités de la même façon (oubli par exemple de traiter un cas d'invalidité, mais prise en compte correcte des autres cas d'invalidité).

Le graphisme suivant illustre le partitionnement d'un domaine en classes d'équivalence, avec les points marqués d'une croix représentant des données invalides.

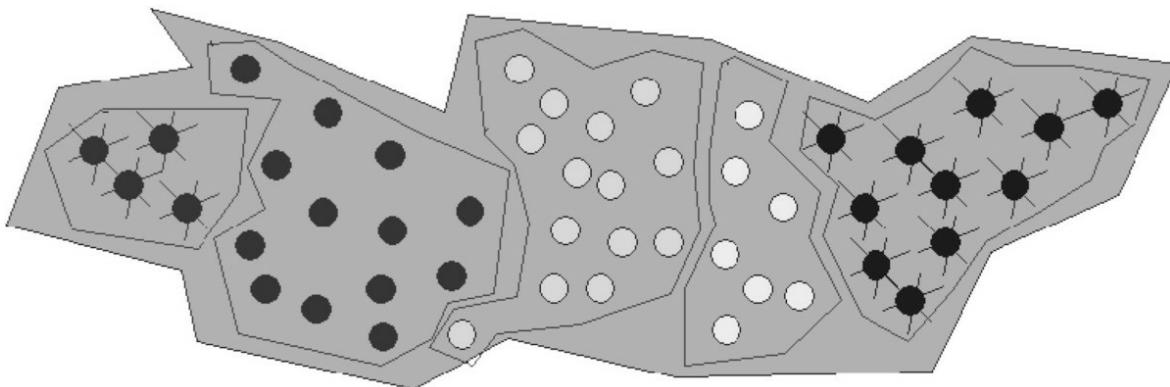


Fig. 4.1 Partitionnement d'un domaine de valeurs

Dans le cas où les variables d'entrées sont liées entre elles, les classes d'équivalence peuvent également être construites, mais il sera plus difficile d'obtenir une réelle partition du domaine d'entrée ; certaines valeurs pourront appartenir à plusieurs classes d'équivalences. Cela n'a pas de réelles conséquences si ce n'est peut-être de générer plus de cas de tests que nécessaire.

4.2.2. Utilisation des classes d'équivalence

Une fois le domaine partitionné en classes d'équivalence, le testeur va choisir un ensemble de valeurs de telle sorte que les jeux de tests couvrent toutes les classes valides (une valeur peut couvrir plusieurs classes) et qu'il y ait une

valeur par classe invalide (une valeur ne couvre qu'une seule classe de sorte qu'on distingue bien ces classes). Si la partition n'est pas exacte, certaines valeurs couvriront plusieurs classes comme sur la figure suivante (figure 4.2) ; comme dit plus haut, c'est fréquemment le cas lorsque les données en entrées sont liées entre elles (on verra plus tard un exemple de fonction manipulant des dates).

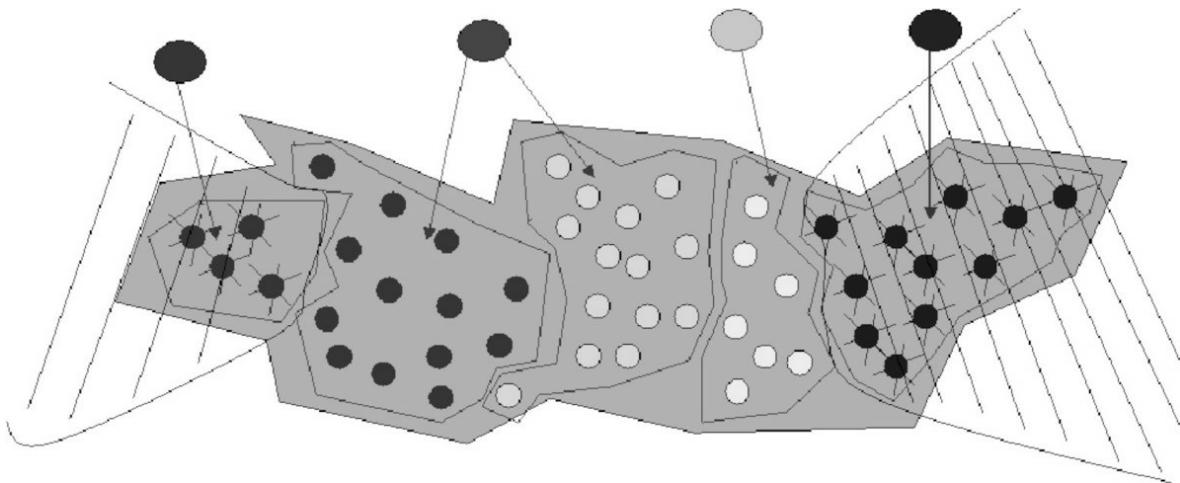


Fig. 4.2 Couvertures des classes d'équivalence

Une fois les classes d'équivalence construites, on est ramené au problème précédent : trouver des jeux de valeurs pertinents lorsque les paramètres prennent un petit nombre de valeurs différentes (ici les valeurs sont assimilées aux classes d'équivalence) ; la stratégie *all pairs* peut donc de nouveau être utilisée avec profit. Dans le cas fréquent où le domaine valide est constitué d'une seule classe, le problème est même trivial puisqu'il n'y a qu'un choix possible ! C'est ainsi que l'utilisation des classes d'équivalence pour notre exemple de composant manipulant un numéro de département conduira à ne générer que trois jeux de tests : un pour l'entrée valide et un par entrée invalide (numéro plus petit que 1 et numéro plus grand que 95). Les valeurs prises dans les classes d'équivalence peuvent être quelconques. On préfère néanmoins prendre, soit des valeurs extrêmes (voir la section consacrée aux tests aux limites) soit, comme ici, des valeurs médianes.

Tab. 4.2 Partitions et choix de valeurs

Validité des entrées	Classes d'équivalence	Données de test
Entrées valides	[1 – 95]	11
Entrées invalides	[minInt – 1 [-30
Entrées invalides] 95 – MaxInt]	100

La validité de cette stratégie de génération de jeux de tests repose sur l'hypothèse initiale : toutes les valeurs d'une classe donnée conduisent au même comportement du logiciel. Il suffit donc d'une seule valeur pour tester toutes les valeurs de la classe. Bien entendu, la pertinence de cette méthode dépend principalement du choix des classes d'équivalence et donc, implicitement, de la qualité des spécifications qui permettent ou non de dégager clairement un ensemble de valeurs pour lesquelles le logiciel doit exhiber un comportement prévisible et identique sur toute la plage. Pour le cas des classes valides, il n'y a généralement pas trop de problème. Les problèmes concernent plutôt le manque fréquent de définition du comportement sur les classes invalides. Le testeur peut donc avoir, par l'utilisation de cette méthode, un rôle d'amélioration des spécifications en faisant préciser des comportements.

Cette méthode est ancienne et le temps a dégagé des bonnes pratiques concernant la construction de ces classes d'équivalence. Nous donnons ici les heuristiques communément employées pour construire les classes d'équivalence.

4.2.3. Construction des classes d'équivalence

En pratique, la construction des classes d'équivalence est uniquement basée sur des heuristiques que nous rappelons ici :

- Si une condition d'entrée ou une condition de sortie définit un intervalle de valeurs (par exemple le numéro de département est compris entre 1 et 95) prendre :
 - une classe d'équivalence valide (la valeur est dans l'intervalle) ;
 - deux classes d'équivalence invalides (une à chaque bout de l'intervalle).

Si une condition d'entrée ou une condition de sortie définit N valeurs

2. (comme par exemple un tableau) prendre :

- une classe d'équivalence valide représentant les éléments à exactement N valeurs ;
- deux classes d'équivalence invalides : un représentant les éléments à moins de N valeurs et un représentant les éléments à plus de N valeurs.

3. Si une condition d'entrée ou de sortie définit un ensemble de valeurs (par exemple une énumération de valeurs) prendre :

- une classe d'équivalence valide représentant les valeurs dans l'ensemble prédéfini ou bien une classe d'équivalence par valeur si le programme semble les différencier ;
- une classe d'équivalence invalide représentant les valeurs hors de l'ensemble (lorsque cela n'est pas rendu impossible par un langage fortement typé).

4. Si une condition d'entrée ou de sortie définit une contrainte devant être vérifiée (par exemple le premier caractère de l'identifiant doit être une lettre) prendre :

- une classe d'équivalence valide (la condition est vérifiée) ;
- une classe d'équivalence invalide (la condition n'est pas vérifiée).

5. Si une classe d'équivalence semble trop complexe ou si la classe d'équivalence semble posséder des éléments traités différemment par le programme :

- Décomposer cette classe d'équivalence complexe en deux ou plusieurs classes d'équivalence moins complexes.

On notera que les règles 4) et 5) sont des règles très générales et peuvent se substituer à toutes les autres. On notera également que le fort typage des langages modernes peut interdire certains choix ; par exemple, une fois un type énuméré défini, il est impossible en Ada ou en Java de prendre une valeur autre que celles permises par la définition du type.

4.2.4. Illustration

Supposons que l'on ait à tester une fonction *Lendemain* qui calcule le lendemain d'une date passée en paramètre et définie par trois entiers : Jour, Mois, et Année. On considère (arbitrairement) que l'année doit être plus grande que 1582 (année de mise en place du calendrier Grégorien en France) et plus petite que 3000. On doit prendre en compte les années bissextiles. La définition d'une année bissextille est « Toutes les années dont le millésime est divisible par quatre sont des années bissextiles, sauf les années séculaires dont le millésime n'est pas divisible par quatre cents. Le nombre de jours du mois de février des années bissextiles compte vingt-neuf jours »^[1]. L'année 1700, qui est bissextile en France, ne l'est pas forcément partout car le calendrier Grégorien ne s'est mis que progressivement en place à travers le monde (par exemple en 1752 au Royaume-Uni et dans ses colonies sur la côte est de l'Amérique du Nord, et de l'actuel nord-ouest des États-Unis).

Construisons maintenant des jeux de tests pour la fonction *Lendemain* en utilisant la technique des classes d'équivalence. Pour cela on commence par prendre en compte les contraintes élémentaires sur les paramètres en entrées. Puis on affinera ces contraintes pour enfin prendre en compte les contraintes liant les entrées et les sorties.

Tab. 4.3 Premier partitionnement

Classes d'équivalences valides	Classes d'équivalences invalides
Jour $\in [1, 31]$	Jour < 1
	Jour > 31
Mois $\in [1, 12]$	Mois < 1
	Mois > 12
Année $\in [1582, 3000]$	Année < 1582
	Année > 3000

Après ce premier partitionnement élémentaire, on applique la règle générale qui stipule qu'une classe complexe pouvant faire apparaître des comportements distincts doit être fractionnée en classes plus petites. Ainsi, il paraît utile de distinguer les mois à 30 jours des mois à 31 jours et mettre à part le mois de février. De même, nous pouvons penser que le traitement des années bissextiles n'est pas le même que celui des années non bissextiles et

que l'an 2000 est un cas particulier intéressant à différencier. La séparation des classes n'entraîne pas nécessairement la création de nouvelles classes invalides car celles-ci coïncident avec des classes existantes (par exemple, un mois qui n'est pas un mois à 30 jours, sera soit un mois à 31 jours soit le mois de février soit un mois invalide déjà pris en compte).

Tab. 4.4 *On affine les partitions*

Classes d'équivalences valides	
Jour $\in [1, 31]$	
Mois $\in [1, 12]$	Mois $\in \{1, 3, 5, 7, 8, 10, 12\}$
	Mois $\in \{4, 6, 9, 11\}$
	Mois = 2
Année $\in [1582, 3000]$	Année bissextile dans l'intervalle [1582, 3000]
	Année non bissextile dans l'intervalle [1582, 3000]
	Année = 2000

On remarque sur cet exemple qu'à moins de compliquer la définition des classes, certaines valeurs peuvent apparaître dans deux classes ; par exemple la valeur d'année 2000 appartient à deux classes.

Si l'on s'arrête ici, nous avons bien pris en compte les contraintes sur les paramètres en entrée, mais nous n'avons pas inclus les contraintes liant les entrées et les sorties (et donc l'aspect fonctionnel du composant). Il s'agit en effet ici de tester une fonction qui calcule la date du lendemain et donc, les valeurs d'entrée caractérisant une fin de mois ou une fin d'année donneront des valeurs de sortie particulières (début de mois ou début d'année). Il est donc intéressant d'intégrer ces contraintes et, en partitionnant des classes existantes, de faire apparaître de nouvelles classes d'équivalence valides (valeur de Jour correspondant à une fin de mois, ou valeur de Mois correspondant à une fin d'année).

Tab. 4.5 *On affine de nouveau les partitions*

Classes d'équivalences valides

Jour \in [1, 31]	Jour \in [1, 28]	
	Jour = 29	
	Jour = 30	
	Jour = 31	
Mois \in [1, 12]	Mois \in {1, 3, 5, 7, 8, 10, 12}	Mois \in {1, 3, 5, 7, 8, 10} Mois = 12
	Mois \in {4, 6, 9, 11}	
	Mois = 2	
	Année \in [1582, 3000]	
		Année bissextile dans l'intervalle [1582, 3000]
		Année non bissextile dans l'intervalle [1582, 3000]
		Année = 2000

Nous avons ainsi quatre classes d'équivalence valides pour les jours, quatre pour les mois et trois pour les années. Nous avons deux classes d'équivalence invalides pour les jours, deux pour les mois et deux également pour les années.

Du fait que les paramètres en entrée sont liés, certaines combinaisons de valeurs valides conduisent à des entrées invalides ; par exemple, le 29 février 2004 ou le 31 avril 1997 sont deux dates invalides alors que chaque valeur composant la date prise individuellement est valide. Les spécifications permettent ici de détecter ce problème très facilement ; ce n'est pas toujours le cas.

Nous pouvons maintenant générer les jeux de tests. Si l'on se contente du critère « *All Singles* », quatre valeurs de test permettent de tester toutes les classes valides (tableau 4.6).

Tab. 4.6 Premiers jeux de tests pour la fonction *Lendemain*

Jeux de valeurs	Résultat attendu	Classes d'équivalences couvertes
(14, 7, 2008)	(15, 7, 2008)	Jour \in [1, 28] Mois \in {1, 3, 5, 7, 8, 10} Année bissextile

(29, 12, 1997)	(30, 12, 1997)	Jour = 29 Mois = 12, Année non bissextile
(30, 4, 2000)	(31, 4, 2000)	Jour = 30 Mois $\in \{4, 6, 9, 11\}$ Année = 2000
(31, 12, 1999)	(1, 1, 2000)	Jour = 30 Mois = 12 Année non bissextile

À cela, il faut ajouter des jeux de tests correspondant aux classes invalides ; on prend garde ici à ne couvrir qu'une seule classe invalide à la fois afin de bien tester individuellement l'effet de l'entrée invalide.

Tab. 4.7 Prise en compte des classes invalides

Jeux de valeurs	Classes d'équivalences couvertes
(-10, 2, 2000)	Jour < 1
(33, 12, 2001)	Jour > 31
(2, 0, 1999)	Mois < 1
(3, 14, 1997)	Mois > 12
(1, 1, 1500)	Année < 1582
(1, 1, 4000)	Année > 3000

Si l'on veut tester plus à fond la fonction *Lendemain*, le testeur peut choisir la stratégie *all pairs* pour générer les jeux de valeurs de test. Cela conduit à générer $4*4 = 16$ jeux de valeurs distincts (tableau 4.8).

Tab. 4.8 Jeux de tests selon le critère *all pairs* pour la fonction *Lendemain*

Classes d'équivalence « Jour »	Classes d'équivalence « Mois »	Classes d'équivalence « Année »	Jeux de valeur

Jour $\in [1, 28]$	Mois $\in \{1, 3, 5, 7, 8, 10\}$	Année bissextile	(14, 5, 2004)
Jour = 29	Mois = 12	Année non bissextile	(29, 12, 1999)
Jour = 30	Mois $\in \{4, 6, 9, 11\}$	Année = 2000	(30, 6, 2000)
Jour = 31	Mois = 2	Année bissextile	(31, 2, 1916)
Jour $\in [1, 28]$	Mois = 2	Année non bissextile	(15, 2, 1789)
Jour = 29	Mois $\in \{1, 3, 5, 7, 8, 10\}$	Année = 2000	(29, 12, 2000)
Jour = 30	Mois = 12	Année bissextile	(30, 12, 1992)
Jour = 31	Mois $\in \{4, 6, 9, 11\}$	Année non bissextile	(31, 9, 1995)
Jour $\in [1, 28]$	Mois $\in \{4, 6, 9, 11\}$	Année = 2000	(7, 11, 2000)
Jour = 29	Mois = 2	Année bissextile	(29, 2, 1964)
Jour = 30	Mois $\in \{1, 3, 5, 7, 8, 10\}$	Année non bissextile	(30, 7, 1903)
Jour = 31	Mois = 12	Année 2000	(31, 12, 2000)
Jour $\in [1, 28]$	Mois = 12	Année bissextile	(8, 12, 1888)
Jour = 29	Mois $\in \{4, 6, 9, 11\}$	Année non bissextile	(29, 9, 1805)
Jour = 30	Mois = 2	Année = 2000	(30, 2, 2000)
Jour = 31	Mois $\in \{1, 3, 5, 7, 8, 10\}$	Année bissextile	(31, 3, 2012)

À ces jeux de tests, il convient d'ajouter ceux du tableau concernant la prise en compte des classes invalides. On obtient alors une qualité des jeux de tests qui laisse peu d'espoir à une erreur de rester inaperçue ! On remarquera que le jeu de tests correspondant au calcul du lendemain du 31 décembre 2000 est bien construit par cette méthode, mais qu'il faudrait rajouter manuellement le test du lendemain des 28 et 29 février 2000.

4.2.5. Commentaires

Cette stratégie de construction de jeux de tests est un des piliers de toute méthode visant à rendre praticable le test de logiciels réels. L'effort à fournir est un effort d'abstraction et d'analyse des spécifications. Cet effort permet donc à la fois de rendre plus efficaces les tests, dans le sens où il permettra de considérer un seul cas là où, sans cette abstraction, plusieurs cas équivalents auraient été produits mais, de plus, il permet de détecter un certain nombre d'incohérences ou d'oublis au niveau des spécifications en portant sur celles-ci un regard différent de celui du programmeur ou de l'utilisateur.

4.3. Tester aux limites

4.3.1. Principes

Le fonctionnement d'un logiciel fait apparaître un mode nominal pour lequel le risque d'erreur est faible. D'expérience, lorsque l'on demande à des élèves de coder la fonction *Lendemain*, très peu commettront des erreurs sur le calcul concernant les dates « nominales » comme le 3 mars 2007 ou le 10 janvier 1876. Par contre, sur des dates plus « atypiques » comme le 28 février 2007 ou le 31 janvier 1775 ou encore le 31 décembre 2004, les erreurs de programmation ou d'interprétation des spécifications seront plus nombreuses : oubli ou mauvaise prise en compte des fins de mois, des fins d'année ou encore erreur ou oubli sur le calcul des années bissextilles. À chaque fois il s'agit de valeurs qui correspondent à des limites sur les valeurs possibles : dernier jour possible d'un mois, dernier numéro de mois, année bissextille ou « presque » bissextille (année 2000).

Ce type d'erreur est fréquent et ce qui vaut pour des élèves est également valable pour des programmeurs plus confirmés mais qui auront mal compris les spécifications du problème ou sous-estimé ses difficultés potentielles de réalisation.

Le test aux limites consiste à choisir des valeurs qui sont aux frontières des domaines de fonctionnement du logiciel. On s'appuiera généralement sur le partitionnement des domaines de définitions des valeurs en entrée et en sortie, mais au lieu de choisir uniquement des valeurs aléatoires au milieu des domaines, on ajoutera systématiquement des valeurs au bord de ces domaines. On prendra garde par contre, à rester dans le domaine afin de pouvoir prédire le résultat attendu.

Comme pour le test basé sur les partitions d'équivalence, le test aux limites peut s'appliquer à différents niveaux dans le cycle de vie du logiciel :

- tests unitaires ;
- test système (contraintes de performance) ;
- tests d'acceptation.

Le plus classique est néanmoins d'utiliser la technique des tests aux limites lors des tests unitaires en Boîte noire comme complément du test par partitionnement. Dans ce cas, les frontières se définissent grâce aux domaines obtenus lors du calcul des classes d'équivalence :

- par la spécification des variables d'entrée,
- par la spécification des résultats.

Certaines données se prêtent naturellement à la définition de limites ; ce sont les valeurs numériques ou les valeurs de types énumérées. Pour d'autres types de données il faut par contre se procurer une « mesure » numérique :

Pour les tableaux, les listes ou plus généralement les collections d'éléments, la mesure sera le cardinal de cet ensemble ; les extrêmes seront alors « tableau vide », « tableau plein », « liste vide », etc.

Pour les valeurs alphanumériques, on pourra utiliser une mesure basée sur l'ordre lexicographique : ainsi, une valeur très proche de « oui » pourra être « ouii » ou « poui » ou encore « Oui ».

L'utilisation de la technique des tests aux limites peut donc se résumer à : 1) calculer les classes d'équivalence comme précédemment puis, 2) pour chaque classe d'équivalence générer une valeur médiane et une ou plusieurs valeurs aux bornes (en utilisant la mesure associée à la donnée si nécessaire).

Reprenons l'exemple d'une fonction qui attend en entrée un numéro de département (en métropole), la donnée d'entrée doit être comprise entre 1 et 95. Le partitionnement conduit de nouveau à une classe valide et deux classes invalides ; le test aux limites va sélectionner des valeurs au centre de l'intervalle mais aussi au bord de l'intervalle (ici 1 et 95 pour la classe valide). Une plus large couverture peut être obtenue en sélectionnant plusieurs valeurs au bord, ce qui revient de façon imagée à « épaisser » le bord de l'intervalle.

Tab. 4.9 *Test aux limites d'une fonction manipulant des numéros de départements*

Validité	Classes d'équivalence	Représentants avec limites	Large couverture
Entrées valides	[1 – 95]	1, 48, 95	1, 2, 48, 94, 95
Entrées invalides	[minInt – 1 [-3000, 0	-3000, -1, 0
Entrées invalides] 95 – MaxInt]	96, 1000	96, 97, 1000

Cette technique peut également être employée sur des composants dont les entrées ne sont pas des valeurs numériques. Par exemple, une fonction qui attend « oui » ou « non » pourra être testée de la façon suivante :

Tab. 4.10 *Test aux limites d'une fonction qui attend « oui » ou « non »*

Validité	Classes d'équivalence	Représentants avec limites
Entrées valides	{« oui »}	« oui »
Entrées valides	{« non »}	« non »
	Autre chaîne	

Entrées invalides	« hello word » « » « ouii » « mon »
-------------------	--

4.3.2. Prise en compte de contraintes liant les paramètres

Il se peut que les variables en entrées soient liées par des contraintes plus sophistiquées que « telle variable est dans tel intervalle ». Par exemple un composant à deux paramètres X et Y peut avoir la contrainte que X et Y vérifient $(Y < 0)$ ET $(X - Y > 0)$ ET $(3X + Y - 15 < 0)$. La figure 4.3 donne une représentation géométrique de ce système de contraintes (avec la zone hachurée représentant la frontière interne au domaine valide) :

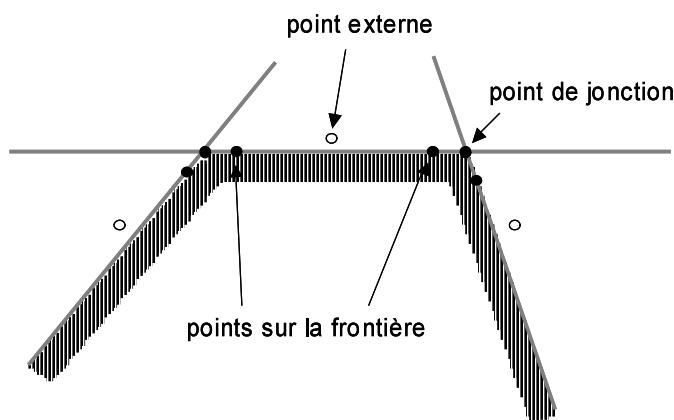


Fig. 4.3 Représentation géométrique d'un système de contraintes linéaires

Ce genre d'information est précieux pour le testeur mais comment l'utiliser astucieusement ? En effet, si l'on considère K contraintes linéaires portant sur N variables entières, le nombre de valeurs aux limites est de l'ordre de 2^N . Un traitement automatique basé sur l'utilisation de librairies mathématiques est possible, mais risque de générer un nombre de cas de tests trop important pour une utilisation réelle.

Une stratégie simple permet d'utiliser ce type d'information pour générer des valeurs aux limites sans tomber dans une énumération laborieuse, il s'agit de la stratégie « Chaque condition - Toutes conditions ».

En pratique on distingue deux formes de contraintes (même si on peut passer d'une forme à l'autre par réécriture) :

- Celles qui sont sous une forme normale disjonctive (disjonction de prédictats élémentaires) ; par exemple $(X > 0)$ OU $(Y > X)$ OU $(Y > 0)$.
- Celles qui sont sous une forme normale conjonctive (conjonction de prédictats élémentaires) ; par exemple $(Y < 0)$ ET $(X-Y > 0)$ ET $(3X + Y-15 < 0)$.

Si les contraintes ne sont pas sous l'une des deux formes, l'on s'y ramène en utilisant la théorie du calcul des propositions qui permet de réécrire une formule telle que $(A$ OU $B)$ ET C en une forme normale disjonctive $(A$ ET $C)$ OU $(B$ et C).

Les valeurs aux limites générées dépendent de la forme de la contrainte :

- Si la condition est une conjonction (ET) de M prédictats booléens :
 - Choisir un cas où tous les prédictats sont « justes » Vrai ; c'est la partie « Toutes les conditions » ; dans ce cas, l'expression vaut Vrai et la valeur attendue correspond à un calcul à partir d'entrées valides ;
 - Choisir M cas avec pour chacun un seul des prédictats « juste » Faux ; c'est la partie « Chaque condition » ; dans ce cas, l'expression vaut Faux et la valeur attendue correspond au traitement d'une entrée invalide.
- Si la condition est une disjonction (OU) de M conditions booléennes :
 - Choisir un cas où tous les prédictats sont « justes » Faux ;
 - Choisir M cas, avec pour chacun, un seul des prédictats « juste » Vrai.

Par exemple, pour la contrainte $(Y < 0)$ ET $(X-Y > 0)$ ET $(3X + Y - 15 < 0)$ qui est sous forme normale conjonctive, on prend :

- Un cas où les trois prédictats sont (juste) Vrai ; i.e. $(X = 0, Y = -1)$
- Trois cas où, un seul des prédictats est (juste) à Faux :
 - $(Y < 0)$ Faux et les autres à Vrai : $(X = 2, Y = 1)$;

- ($X - Y > 0$) Faux et les autres à Vrai : ($X = -2, Y = 1$) ;
- ($3X + Y - 15 = 0$) Faux et les autres à Vrai : ($X = 6, Y = -1$) ;

Le terme « juste » doit se comprendre par le fait qu'une légère variation de la valeur du paramètre inverse la valeur de vérité du prédictat ou de l'expression.

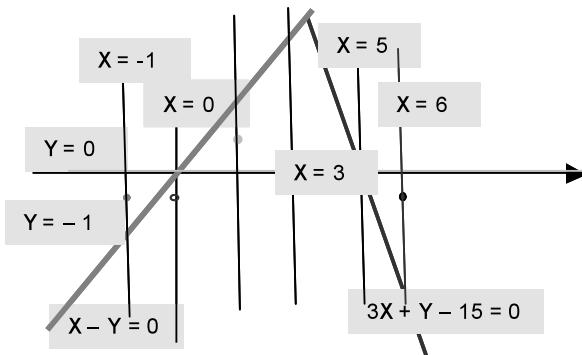


Fig. 4.4 Points extrêmes

Si le testeur souhaite rester dans les domaines valides, il ne se limitera qu'aux valeurs rendant les prédictats vrais. Enfin, une fois les valeurs limites générées, il faut également produire également des valeurs « médianes » afin de couvrir correctement les classes d'équivalence.

4.3.3. Illustration

Reprendons la fonction *Lendemain*. Le tableau 4.11 donne, pour chaque classe d'équivalence du problème, les valeurs limites pouvant être utilisées. Il est clair que lorsque la classe est réduite à un élément, il n'y a guère de choix sur les valeurs possibles.

Tab. 4.11 Valeurs limites possibles pour les classes d'équivalence de la fonction *Lendemain*

Classe d'équivalence	Valeurs limites possibles	Remarque
Jour $\in [1, 28]$	1 et 28	Bords de l'intervalle
	1 ou 10	

Mois $\in \{1, 3, 5, 7, 8, 10\}$		Première ou dernière valeur de l'énumération
Mois $\in \{4, 6, 9, 11\}$	4 ou 6	Même remarque que précédemment
Année bissextile	1600, 2400, 2000	Années « juste » bissextilles (quatre centenaires)
Année non bissextile	1582, 1700, 3000	Années « juste » non bissextilles (centenaires) ou limite de l'intervalle des valeurs possibles (1582 à 3000)

En ne considérant que les cas valides, et en supprimant les lignes correspondant à des valeurs non limites comme le (29,12,XX) ou (29,6,XX) les jeux de valeurs pouvant être générés en complément des jeux de valeurs précédent (tableau 4.8) sont donnés tableau 4.12. Il ne s'agit là que d'un complément, bon nombre de valeurs aux limites ayant déjà été construites à l'étape précédente.

Tab. 4.12 Compléments de valeurs aux limites pour la fonction Lendemain

Classes d'équivalence « Jour »	Classes d'équivalence « Mois »	Classes d'équivalence « Année »	Jeux de valeurs aux limites
Jour $\in [1, 28]$	Mois $\in \{1, 3, 5, 7, 8, 10\}$	Année bissextile	(1, 1, 2400) (28, 10, 1600)
Jour $\in [1, 28]$	Mois = 2	Année non bissextile	(28, 2, 1700)
Jour = 29	Mois = 2	Année bissextile	(29, 2, 1600) (29, 2, 2400) (29, 2, 2000)
Jour = 31	Mois = 12	Année 2000	(31, 12, 2000)
Jour = 31	Mois $\in \{1, 3, 5, 7, 8, 10\}$	Année bissextile	(31, 3, 1600) (31, 12, 3000)

4.3.4. Commentaires

Le test aux limites améliore le test par partitionnement, mais ne le remplace pas ; en effet, la construction des frontières, opération parfois complexe, est généralement déduite d'une analyse préalable des partitions liées aux domaines de définitions des paramètres.

On notera que le test aux limites permet de détecter simplement et efficacement certains types d'erreurs de réalisation :

- Mauvais opérateur de relation : $X > 2$ au lieu de $X \geq 2$
- Erreur de borne : $X + 2 = y$ au lieu de $X + 3 = y$
- Échange de paramètres : $2x + 3y > 4$ au lieu de $3x + 2y > 4$
- Ajout d'un prédicat qui ferme un ensemble : $(2x > 4)$ et $(3x < 6)$
- Frontière manquante : $(x + y > 0)$ ou $(x + y \leq 0)$
- Boucles mal réglées, mauvaise gestion des indices de tableau, etc.

Enfin, on notera également qu'une automatisation poussée peut être atteinte aussi bien dans le cadre de tests structurels que de tests fonctionnels

4.4. Tester grâce à une table de décision

4.4.1. Principes

Le comportement d'un logiciel peut être décrit par les actions qu'il effectue : par exemple, écrire un message à l'écran, provoquer la mise en route d'une alarme, terminer un calcul et lancer une impression, etc. L'occurrence de ces actions est dépendante de plusieurs éléments : l'algorithme mise en œuvre, la qualité de la réalisation et les valeurs d'entrée fournies au logiciel. Selon certaines conditions portant sur ces entrées, certaines actions seront observées ou non. Dans de nombreux cas, l'ordre dans lequel les valeurs sont présentées au composant n'influe pas sur ses actions (le composant peut être alors qualifié de « sans état »). Dans ces cas, une table de décision sera un moyen simple et concis de représenter le lien existant entre une condition portant sur les variables d'entrées et une action visible.

4.4.2. Format d'une table de décision

Une table de décisions comportera une ligne par condition et une ligne par action. Chacune des colonnes de cette table décrira une valeur pour les conditions et une indication pour chacune des actions : action observée ou non observée lorsque les conditions de la colonne correspondante sont vérifiées. Par exemple, sur la figure suivante, l'action « Action2 » devra être observée lorsque la condition « C1 » prend la valeur V1 et la condition « C2 » prend la valeur V2.

		Combinaison de valeurs (variants)						
		Actions sélectionnées						
		Actions						
C1	V1							
C2	V2							
Action2	X							

Fig. 4.5 Format d'une table de décision

Chaque colonne est appelée « variant » et définit un cas de test. Les valeurs des paramètres d'entrée (le jeu de tests correspondant à ce cas de test) sont alors choisies de telle sorte que les conditions prennent les valeurs correspondant au variant sélectionné.

Prenons un exemple, soient trois conditions/variables C1, C2, C3 telles que :

- C1 peut prendre les valeurs 0,1,2.
- C2 peut prendre les valeurs 0,1.
- C3 peut prendre les valeurs 0,1,2,3.

et soient quatre actions A1, A2, A3, A4 à effectuer en fonction des combinaisons des valeurs des trois conditions.

Le nombre de lignes de la table sera sept, correspondant à trois conditions plus quatre actions. Sans autre information, le nombre de colonnes sera de

$3 \times 2 \times 4 = 24$ (3 valeurs pour C1, 2 valeurs pour C2 et 4 valeurs pour C4). Cette table pourrait ressembler à celle de la figure 4.6.

Figure 4.13 Une table de décision

	R	E	D															
C1	1	3	3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	
C2	3	2	6	0	0	0	1	1	1	0	0	0	1	1	1	1	0	0
C3	6	4	24	0	0	0	0	0	0	1	1	1	1	1	1	1	2	2
A1				X	X						X	X						X
A2						X										X		X
A3							X	X	X								X	
A4							X								X			

Ici, pour chaque condition C1, C2, C3, nous faisons figurer trois informations : C.R correspond au coefficient de répétition de la condition, c'est-à-dire le nombre de fois que chaque valeur de la condition devra être répétée au niveau des colonnes de la table. La variable E définit l'étendue de la condition, c'est-à-dire le nombre de valeurs possibles de la condition, et la variable D définit le domaine de la condition, c'est-à-dire le nombre de colonnes qu'occupent les différentes valeurs de la condition (en prenant en compte le coefficient de répétition). Ces informations sont redondantes entre elles, mais elles permettent de remplir ce type de table de façon très systématique et d'exercer un contrôle visuel rapide de la cohérence de la table. Les jeux de tests correspondant aux variants de cette table sont triviaux à générer (chaque condition définit ici une valeur précise du paramètre).

4.4.3. Construction et simplification d'une table de décision

Une table de décision peut être construite de différentes manières, mais une façon simple de procéder consiste à :

1. Identifier les variables et les conditions de décisions.
2. Identifier les actions résultantes.

3. Identifier quelle action doit être produite en réponse à une combinaison de décision particulière.
4. Vérifier que la table est complète (toutes les actions sont au moins exercées une fois) et consistante (pour un même variant, deux actions incompatibles ne sont pas exercées simultanément).

On notera que les étapes 3 et 4 peuvent être remplacées par « Énumérer toutes les combinaisons de décisions (possibles) et associer l'action résultante ».

Avant d'illustrer ces règles sur quelques exemples, il est important de noter que la construction de cette table va également permettre de repérer des erreurs dans les spécifications. En effet, si aucune action ne peut être associée à un ensemble de conditions, deux situations sont possibles : soit il s'agit de l'oubli d'une action possible soit il s'agit d'un manquement de spécification (dans ce cas le testeur met en évidence le fait que les spécifications ne sont pas complètes). Une autre possibilité est que les spécifications indiquent, pour une même combinaison de conditions, deux actions en contradiction ; dans ce cas, le testeur découvre une erreur de consistance dans les spécifications qui peut être soit une véritable contradiction soit un manque de précision sur les conditions de décision. Dans tous les cas, la valeur ajoutée à la construction de cette table est donc importante au regard des efforts à fournir.

Il arrive qu'une table de décision soit de taille importante (du fait d'un grand nombre de conditions et de la combinatoire engendrée). Une façon de simplifier cette table sans risquer une erreur ou un oubli lors de sa construction est de remarquer qu'une condition n'influence pas le résultat pour certains variants. Dans ce cas, on va regrouper ces variants et donner la valeur « *don't care* » à cette condition au niveau de ce variant regroupé. Cette procédure peut être par ailleurs systématisée dans le cadre théorique des « Diagrammes de décision binaires »^[2]. Ce regroupement peut être fait une fois une première version de la table construite ou au contraire, lorsque le cas est simple, avant d'énumérer des valeurs que l'on sait pouvoir regrouper.

	C.R	E	D	Don't care												Un seul variant												
C1	1	3	3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	
C2	3	2	6	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	
C3	6	4	24	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	
A1				x	x					x	x	x				x	x				x			x			x	
A2					x								x	x					x			x		x			x	
A3						x	x						x				x					x			x			
A4						x											x				x			x				

Fig. 4.7 Regroupement par la stratégie 'don't care'

4.4.4. Illustration : le problème des triangles

Prenons l'exemple d'un programme qui lit trois valeurs entières inférieures à 20 et affiche à l'écran un message indiquant si le triangle est « isocèle » ou « équilatéral » ou « scalène » (quelconque). Une spécification plus formelle du problème donne :

- **Données en entrée**
 - Trois entiers (côtés du triangle : a, b, c) tels que chacune des valeurs soit un nombre positif inférieur ou égal à 20.
- **Actions visibles du programme :**
 - A1 : affiche « Équilatéral » si $a = b = c$
 - A2 : affiche « Isocèle » si 2 paires de côtés sont égales
 - A3 : affiche « Scalène » si aucun côté n'est égal à l'autre ou
 - A4 : affiche « Pas Un Triangle » si $a \geq b + c$, $b \geq a + c$, ou $c \geq a + b$
 - A5 : affiche « Erreur d'entrée » ou lève une exception si une des contraintes portant sur les paramètres d'entrée n'est pas vérifiée.
- **Les conditions élémentaires** affectant les actions du programme sont donc :
 - C1 : $a < 20$

- C2 : $b < 20$
- C3 : $c < 20$
- C4 : $a < b + c$
- C5 : $b < a + c$
- C6 : $c < a + b$
- C7 : $a = b$
- C8 : $a = c$
- C9 : $b = c$

À l'aide de ces conditions et actions associées nous pouvons construire la table de décisions (figure 4.8) ; certaines combinaisons sont impossibles et ne figurent pas dans la table. Là encore, il eût été possible de faire apparaître explicitement ces variants puis de leur associer une action spécifique (variant impossible par exemple) avant de la supprimer ou bien de ne pas les construire du tout. Dans un cas, le travail est un peu laborieux mais le risque d'erreur ou d'oubli est faible, dans l'autre cas, la construction de la table est plus légère mais on risque d'oublier certains variants ou de considérer, faussement, que certains sont semblables. Nous avons ici adopté une approche mixte : par exemple, lorsque la condition « $a < 20$ » n'est pas vérifiée le résultat du test est d'afficher un message d'erreur signalant la violation des contraintes portant sur les paramètres d'entrée quelle que soit la valeur des autres conditions ; c'est un cas simple à prendre en compte et l'on peut sans risque d'erreur ne construire qu'un seul variant. De même, dès que l'une des trois contraintes C4, C5 ou C6 n'est pas vérifiée, alors le résultat visible est de signaler que les données en entrée ne définissent pas un triangle (action A4) : il n'est pas nécessaire d'énumérer tous les cas possibles. Par contre, pour des combinaisons de conditions incompatibles entre elles, comme « $a = b$ » et « $a = c$ » et non « $b = c$ » (variant 10), il est préférable de construire ce variant puis de le supprimer (par exemple en barrant la colonne) après vérification ; le risque d'erreur, possible dans ce cas, est réduit tandis que la combinatoire reste faible. Par ailleurs, les conditions étant binaires (valeurs Vrai ou Faux), les notions de « coefficient de répétition », « étendu » ou « domaine » n'ont pas d'intérêt ici et ne figurent donc pas dans la table.

	Variants « impossibles »													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C1 : $a < 20$	F	V	V	V	V	V	V	V	V	V	V	V	V	
C2 : $b < 20$		F	V	V	V	V	V	V	V	V	V	V	V	
C3 : $c < 20$			F	V	V	V	V	V	V	V	V	V	V	
C4 : $a < b + c$				F	V	V	V	V	V	V	V	V	V	
C5 : $b < a + c$					F	V	V	V	V	V	V	V	V	
C6 : $c < a + b$						F	V	V	V	V	V	V	V	
C7 : $a = b$							F	V	F	V	F	V	V	
C8 : $a = c$								F	F	V	F	V	V	
C9 : $b = c$									F	F	F	V	V	
A1 : Affiche « Équilatéral »													X	
A2 : Affiche « Isocèle »										X	X			
A3 : Affiche « Scalène »											X			
A4 : Affiche « Pas Un Triangle »							X	X	X					
A5 : Affiche « Erreur d'entrée » ou lève une exception	X	X	X											

Fig. 4.8 Table de décision pour le problème des triangles

« Nous » avons donc dix cas de tests à générer pour tester ce composant. Si l'on veut réduire ce nombre de cas de tests, il est possible d'appliquer l'heuristique « Toutes conditions-Chaque condition » vue précédemment. Dans le cadre des tables de décisions, ceci revient à supprimer les variants pour lesquels plusieurs conditions, mais pas toutes, sont vraies simultanément. Ici, nous pourrions supprimer selon cette heuristique les variants 7, 8, 9 et 11. En procédant brutalement de la sorte, il est possible de ne plus voir exercer certaines actions (ici les actions A2 et A3) ce qui reviendrait à omettre des cas de tests importants ; toutes les heuristiques doivent être employées avec précaution et, rappelons le encore, le testeur doit garder à l'esprit que son expertise ne pourra jamais être remplacée totalement par une méthode ou une heuristique. En utilisant avec précaution l'heuristique « Toutes conditions-Chaque condition » nous pouvons néanmoins supprimer deux des trois variants 8, 9, 11 (par exemple 8 et 9) et réduire, si cela se justifie par un gain en temps ou en ressource, le nombre de cas de tests à 8.

4.4.5. Illustration : le problème du calcul du lendemain

Le test de la fonction *Lendemain* se réalise facilement grâce à la méthode des tables de décisions. On commence pour cela par établir les conditions et les actions visibles de cette fonction.

- **Actions visibles :**

- A1 : changement d’année, et positionnement au 1^{er} janvier
- A2 : changement de mois et positionnement au 1er jour du mois
- A3 : changement de jour et maintien du mois et de l’année
- A4 : date incorrecte

Les valeurs élémentaires des paramètres influant sur le résultat sont (si l’on ignore pour l’instant la particularité de l’année 2000 et si l’on ne considère que les valeurs d’entrée correspondant aux contraintes des spécifications) :

- **Paramètre Mois :**

- M1 : le mois a 31 jours
- M2 : le mois a 30 jours
- M3 : le mois est le mois de février
- M4 : le mois est le mois de décembre

- **Paramètre Jour :**

- J1 : le jour est entre 1 et 27
- J2 : le jour est le 28
- J3 : le jour est le 29
- J4 : le jour est le 30
- J5 : le jour est le 31

- **Paramètre Année :**

- Y1 : l’année est bissextile
- Y2 : l’année est non bissextile

Si l’on construit alors la table de façon systématique, il nous faut quarante colonnes (cinq valeurs possibles pour le Mois, quatre pour le Jour et deux pour l’Année) et sept lignes (trois paramètres et quatre actions possibles), ce qui reste tout à fait raisonnable à ce stade.

Tab. 4.13 Table de décision initiale pour la fonction Lendemain (en plusieurs parties)

	C.R	E	D	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Année dans	1	2	2	Y1	Y2																		
Jour dans	2	5	10	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5
Mois dans	10	4	20	M1	M2																		
A1 : Année +1																							
A2 : Mois + 1												X	X							X	X		
A3 : Jour +1		X	X	X	X	X	X	X	X				X	X	X	X	X	X					
A4 : Date incorrecte																				X	X		

	C.R	E	D	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Année dans	1	2	2	Y1	Y2																		
Jour dans	2	5	10	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5
Mois dans	10	4	20	M1	M2																		
A1 : Année +1																					X	X	
A2 : Mois + 1						X		X															
A3 : Jour +1		X	X	X									X	X	X	X	X	X	X	X	X		
A4 : Date incorrecte					X		X	X	X	X	X	X											

Cette table peut alors être simplifiée par l'utilisation de la stratégie *don't care* qui va permettre de regrouper un certain nombre de variants équivalents (sur fonds grisés dans le tableau 4.14). Par exemple, lorsque le mois est dans M1, peu importent et le type d'année et la valeur du jour tant que cette dernière n'est pas J5 : d'où un possible regroupement des variants de 1 à 8.

Tab. 4.14 Regroupement de variants par la technique « *don't' care* »

	C.R	E	D	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Année dans	1	2	2	Y1	Y2																		
Jour dans	2	5	10	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5
Mois dans	10	4	20	M1	M2																		
A1 : Année +1																							
A2 : Mois + 1												X	X							X	X		
A3 : Jour +1		X	X	X	X	X	X	X	X					X	X	X	X	X	X				
A4 : Date incorrecte																				X	X		

	C.R	E	D	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Année dans	1	2	2	Y1	Y2																		
Jour dans	2	5	10	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5	J1	J1	J2	J2	J3	J3	J4	J4	J5	J5
Mois dans	10	4	20	M3	M4																		
A1 : Année +1																					X	X	
A2 : Mois + 1						X		X															
A3 : Jour +1		X	X	X										X	X	X	X	X	X	X			
A4 : Date incorrecte					X		X	X	X	X													

En réitérant ce procédé et après simplification nous obtenons la table de décision présentée tableau 4.15 qui ne comporte plus que treize variants.

Tab. 4.15 La table de décision après simplification

	1'	2'	3'	4'	5'	6'	7'	8'	9'	10'	11'	12'	13'	
Année dans	-	-	-	-	-	-	Y1	Y2	Y1	Y2	-	-	-	
Jour dans	J1-J4	J5	J1-J3	J4	J5	J1	J2	J2	J3	J3	J4-J5	J1-J4		J5
Mois dans	M1	M1	M2	M2	M2	M3	M3	M3	M3	M3	M3	M4	M4	
A1 : Année +1														X
A2 : Mois + 1		X		X				X		X				
A3 : Jour +1	X		X			X	X					X		
A4 : Date incorrecte				X				X		X				

La prise en compte de la spécificité de l'année 2000 nécessite de construire une table additionnelle dans laquelle on ne prend en compte que l'année 2000 (valeur Y3) ; ceci est fait dans la table présentée tableau 4.16.

La plupart de ces nouveaux variants peuvent être fusionnés par la stratégie « *don't care* » avec les variants précédents ; par exemple, les variants 1'' à 4'' sont équivalents au variant 1' du tableau précédent ; il s'agit forcément d'une appréciation subjective, mais l'on peut considérer que seuls les variants 12'' et 13'' sont réellement nouveaux par rapport aux variants précédents.

Tab. 4.16 Prise en compte de l'an 2000

C.R	E	D	1"	2"	3"	4"	5"	6"	7"	8"	9"	10"	11"	12"	13"	14"	15"	16"	17"	18"	19"	20"	
Année dans	1	1	1	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3										
Jour dans	1	5	5	J1	J2	J3	J4	J5	J1	J2	J3	J4	J5	J1	J2	J3	J4	J5	J1	J2	J3	J4	J5
Mois dans	5	4	20	M1	M1	M1	M1	M2	M2	M2	M2	M2	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	
A1 : Année +1																						X	
A2 : Mois + 1						X					X												
A3 : Jour +1	X	X	X	X		X	X	X			X	X					X	X	X	X			
A4 : Date incorrecte										X					X	X							

Après fusion de ces deux tables, nous obtenons la table de décision présentée tableau 4.17 ; un tiret pour l'année signifie que l'année peut prendre une valeur quelconque de Y1 (année bissextile) à Y3 (année 2000).

Tab. 4.17 La table de décision complète pour la fonction Lendemain

	1'	2'	3'	4'	5'	6'	7'	8'	9'	10'	11'	12'	13'	12"	13"							
Année dans	-	-	-	-	-	-	Y1	Y2	Y1	Y2	-	-	-	Y3	Y3							
Jour dans	J1-J4	J5	J1-J3	J4	J5	J1	J2	J2	J3	J3	J4-J5	J1-J4	J5	J2	J3							
Mois dans	M1	M1	M2	M2	M2	M3	M3	M3	M3	M3	M3	M4	M4	M3	M3							
A1 : Année +1														X								
A2 : Mois + 1		X		X					X													X
A3 : Jour +1	X		X			X	X			X		X				X			X			
A4 : Date incorrecte											X											

Il y a quinze colonnes et donc quinze jeux de tests à générer (pour les cas nominaux) ; à cela il faut ajouter les six jeux de tests pour les valeurs d'entrées non conformes aux spécifications : deux valeurs de date pour des années non correctes, deux valeurs de date pour des numéros de mois non corrects et deux valeurs de date pour des numéros de jours incorrects.

4.4.6. Commentaires

Cette technique de test donne donc un nombre de cas de tests comparable à la technique précédente ; ce n'est pas aberrant car lors de l'analyse et de la définition des conditions et des actions intervenant dans une table de décisions, nous avons été amenés à faire une analyse partitionnelle des paramètres du composant à tester.

Pour le choix des jeux de tests, nous pouvons prendre aléatoirement une valeur compatible avec le variant et, si nécessaire, nous pouvons réutiliser la technique *all pairs* vue précédemment.

Enfin, on remarquera que cette technique donne en général de « bons » jeux de valeurs ; par exemple, sur le cas de la fonction *Lendemain*, tous les cas « à risque » sont ainsi décrits par un variant.

Pour conclure sur cette technique, nous pouvons dire qu'elle possède d'indéniables qualités :

- Elle est simple à mettre en œuvre car elle n'utilise que des tables qui peuvent être construites avec une simple feuille de papier et un crayon (ou encore un tableur) ;
- Une table de décision complexe peut être construite par partie, en ne prenant en compte que progressivement les différentes valeurs possibles des conditions, puis en fusionnant les tables correspondantes (comme cela a été fait dans l'exemple de la fonction du lendemain) ;
- Les classes d'équivalences utilisées dans la technique du même nom sont ici implicitement construites, sans effort particulier lors de la mise en avant des actions et conditions associées du composant à tester ;
- Le nombre de cas de tests, potentiellement important, peut être facilement réduit par la mise en œuvre de la stratégie *don't care* ;
- Enfin, des erreurs de spécifications peuvent être découvertes lors de la construction des tables de décisions ce qui constitue une valeur ajoutée à cette technique de test.

4.5. Utiliser un diagramme « états transitions »

4.5.1. Principes

Pour certains composants, l'ordre dans lequel les paramètres lui sont passés ou l'histoire de ses différentes stimulations intervient dans son comportement ; on parle alors de composant à état ou encore de « machine à état » (state machine). C'est le cas, par exemple, de tous les composants qui implémentent une partie d'un protocole d'interaction (serveur web, distributeur de billets, composant de chiffrement, etc.). Ce type de

fonctionnement s'oppose à ce que nous avons présupposé jusqu'alors puisque, dans bon nombre de cas, les composants logiciels sont censés être sans état : le premier, le second et tous les autres appels du composant avec les mêmes valeurs de paramètres doivent donner exactement le même résultat. Pour un composant à états, le composant va successivement décrire différents états internes et réagira en fonction des valeurs qui lui sont passées (comme un composant sans état) mais aussi en fonction de son état interne, qui pourra être vu comme un résumé des différentes actions déjà réalisées par le composant. Les changements d'états, qui sont des évolutions discrètes et non continues, sont appelés transitions. La difficulté pour le testeur, dans le cas des tests Boîte noire, est que cet état interne est souvent non observable : on voit les réactions ou valeurs fournies, mais pas nécessairement les changements d'état. Afin de décrire précisément le comportement de ce type de composant, la langue naturelle est trop imprécise et pas assez synthétique ; on préfère utiliser des formalismes adaptés comme les automates^[3], les statecharts ou plus généralement des diagrammes états transitions comme ceux employés dans le formalisme UML^[4]. Le testeur qui doit non seulement trouver les valeurs de tests pertinents mais l'ordre dans lequel il va les soumettre au composant (on parle alors de scénarios de test) a là une source précieuse d'information. Si une telle description n'est pas fournie, c'est à lui de tenter de la reconstituer, au moins partiellement, à l'aide des documents de spécifications. Nous nous plaçons ici dans un contexte plus large que celui d'UML, puisqu'en effet ce type de diagramme existe depuis bien plus longtemps que ce formalisme : tout ou partie des spécifications des protocoles Internet (par exemple IP, TCP, UDP) sont décrits à l'aide de tels diagrammes dans les RFC (*Request For Comment*), documents qui peuvent être vus comme des normes.

4.5.2. Format général d'un diagramme états transitions

Un diagramme état transitions décrit, à l'aide d'un graphique, le comportement du composant ou système qu'il spécifie. Ce type de diagramme est utilisé dans nombreux formalismes de modélisation tels que UML ou SDL, ou de manière informelle comme moyen de description moins

ambiguë que la langue naturelle. De façon générale, les états du composant sont représentés par un cercle (un ovale, ou encore un rectangle aux bords arrondis) auquel est associé, au minimum, un nom : le nom de l'état correspondant. D'autres informations peuvent figurer dans la description de l'état comme la valeur de certaines variables (la valeur de compteurs internes par exemple) ou une description de sous états internes à cet état.

Les transitions entre états sont représentées par des arcs qui relient ces états : une absence d'arc entre un état e1 et un état e2 signifie qu'il n'y a pas de transitions directes possibles entre e1 et e2. Au contraire, un arc reliant e1 et e2 signifiera qu'il est possible, sous certaines conditions, de passer de l'état e1 à l'état e2. Ces conditions sont généralement portées sur l'arc et sont de deux types :

- Quelque chose s'est produit comme la réception d'un message ou la survenue d'un événement (par exemple une temporisation est échue) ;
- Un prédicat qui relie des variables internes et potentiellement des valeurs de données reçues est devenu vrai ; on appelle garde ce type de prédicat.

Les conditions du premier type sont souvent suivies d'un point d'interrogation, comme « Saisie d'un chiffre ? » ou encore « Introduction carte ? » alors que les gardes, présentes également sur les arcs, figurent dans beaucoup de formalismes entre crochets, comme « [Code correct] » ou « [Compteur < 3] ». En plus du changement d'état la transition peut avoir une action visible sur l'environnement : émission d'un message, armement ou réarmement d'un temporisateur, modification de la valeur d'une variable interne. On fait figurer ces actions à l'aide de la description de l'action suivie d'un point d'exclamation comme « Envoie Ack! ».

Une autre représentation commune de ces conditions et actions associées aux transitions, consiste à faire figurer sur les transitions des couples « condition[Garde] / action visible » comme « Introduction Carte [DAB non bloqué] / Afficher 'Saisir Code' ».

Lorsqu'un événement ne figure sur aucun arc partant de cet état, la signification est que si cet événement survient dans cet état alors il sera tout

simplement ignoré et ne provoquera aucune action visible et aucune modification interne du composant vis-à-vis des données observées.

4.5.3. Construction d'un diagramme états transitions

Le cas le plus simple est que le diagramme fait partie des spécifications ; il n'y a alors rien à faire pour cette construction ! Si par contre celui-ci n'existe pas il va falloir procéder par étapes, qui peuvent être vues comme des étapes d'abstraction du comportement du composant à tester. La difficulté est de définir le bon niveau d'abstraction : trop de détails rendent impraticable l'utilisation du modèle, tandis que le manque de détail conduit à l'omission de comportements intéressants pour le testeur. Voici quelques règles simples pouvant être utilisées dans cet objectif :

- Définir les différents états observables du composant et si possible définir le moyen grâce auquel on pourra observer ces états ; cela peut être par la valeur d'une variable accessible, l'appel d'une fonction ou méthode du composant, ou plus couramment par l'observation des réactions du composant vis-à-vis des événements possibles (solution la moins précise) ;
- Définir ce qu'est l'environnement externe accessible au composant (accès à des variables externes par exemple) et quelles sont les variables internes qui interviennent dans le comportement (en particulier au niveau des gardes) ;
- Énumérer les différents événements auxquels le composant va réagir et définir la façon de signaler ces événements au composant (par exemple, appel d'une fonction ou d'une méthode ou modification de la valeur d'une variable) ;
- Définir pour chaque état et chaque événement quelle action doit effectuer le composant : rien, changement d'état simple, changement d'état avec modification de l'environnement externe ou modification interne. Puis définir sous quelle condition (garde) cette action est permise ou possible

Si le composant a été conçu en prenant garde à sa testabilité, les étapes précédentes sont simples à réaliser ; dans le cas contraire, la construction

d'un diagramme états transition peut amener à de nombreux échanges avec les équipes de conception. Néanmoins, cette construction sera toujours fructueuse puisqu'elle améliorera nettement la qualité des scénarios de tests et permettra souvent en mettre en avant des possibles défauts de spécifications.

4.5.4. Construction des séquences de test

Les stratégies d'utilisation d'un diagramme états transitions en vue des tests s'apparentent à des stratégies de couverture de code que nous détaillerons au chapitre suivant. Il s'agit en effet de se fixer des objectifs en terme du parcours du diagramme et de choisir des scénarios de test qui permettent de réaliser ces objectifs. Parmi les objectifs classiques dans ce contexte nous pouvons mentionner :

- Atteindre au moins fois chaque état ;
- Exercer au moins une fois chaque transition ;
- Exercer toutes les séquences d'une longueur donnée ;
- Exercer toutes les séquences permettant d'atteindre un état donné ;
- Évaluer les réactions vis-à-vis de tous les événements dans un état donné.

Les deux premiers objectifs ainsi que le dernier sont relativement simples à atteindre puisqu'il « suffit » de lire le diagramme pour construire les séquences appropriées. Pour les autres, la combinatoire peut rendre délicate l'opération et il peut être nécessaire d'utiliser un outil adapté pour générer automatiquement ces séquences.

Enfin, une fois les séquences construites, il faut trouver les valeurs appropriées à passer au composant, valeurs qui permettront d'exercer la séquence. Cette étape commence nécessairement par l'association systématique, à chaque événement, de la façon concrète de signaler cet événement au composant — ce qui a déjà été fait dans le cas où le diagramme a dû être construit par le testeur du composant. Enfin, il faut que les valeurs choisies puissent satisfaire les gardes des différentes transitions franchies ; il n'existe *a priori* pas de recette « miracle » et seule une étude minutieuse des spécifications et du diagramme états transitions permettra de trouver les

valeurs appropriées même si certains travaux récents offrent l'espoir de pouvoir un jour générer automatiquement des données de tests à partir de tels diagrammes.

4.5.5. Commentaires

L'utilisation de diagrammes états transitions s'avère très utile dans la compréhension des comportements de logiciels basés sur l'interaction de différents composants. Historiquement conçus pour décrire des protocoles de communications (normes OSI ou RFC du monde Internet) ils trouvent un emploi naturel dans la description des logiciels multitâches ou pour décrire une interaction d'un logiciel avec un ou plusieurs utilisateurs. Si la conception d'un diagramme états transitions peut être légèrement technique, cet effort sera doublement récompensé : il fournira tout d'abord un générateur précieux de cas d'utilisation, puisqu'il suffit de parcourir ce diagramme pour générer une exécution possible. Deuxièmement, comme tout effort de modélisation, il permettra de préciser certains détails absents ou ambigus dans les spécifications et apportera ainsi des précisions utiles pour la conception des cas de tests.

4.6. Conclusion

Les techniques de tests Boîte noire constituent l'outil principal de toute activité de tests. Basées sur l'utilisation des spécifications du logiciel, elles permettent de revenir à l'essentiel c'est-à-dire sur ce que doit fournir le logiciel à ses utilisateurs. Différentes techniques sont à la disposition du testeur, mais pour chacune d'entre elles il sera amené implicitement ou explicitement à analyser la complétude et la cohérence des spécifications. Cette analyse permet généralement de mettre à jour des erreurs au niveau de ces spécifications et est donc à intégrer complètement dans l'activité du testeur.

Chacune des méthodes présentées dans ce chapitre privilégie un point de vue. Cependant, toutes essayent de réduire la complexité en regroupant en classes d'équivalence des valeurs a priori distinctes mais conduisant à des comportements similaires. Cette opération d'abstraction peut être menée sans

méthode, mais risque alors de conduire à des oubli ou à des confusions préjudiciables à la qualité des tests. La méthode, si elle ne fait pas tout, constitue un guide appréciable dans une activité plus complexe qu'il n'y paraît aux premiers abords.

Dans tous les cas, une fois les cas de tests définis à l'aide d'une méthode, le testeur gagnera à vérifier qu'aucun cas de tests important ne manque (en particulier des cas de tests basés sur son expérience) ou que, à l'opposé, certaines séries de tests peuvent être regroupées par un nouveau procédé d'abstraction.

Notes

- [1] Définition tirée du *Grand Robert de la langue française*, deuxième édition, 2001, LTV.
- [2] Voir l'article séminal : Sheldon B. Akers « Binary Decision Diagrams », *IEEE Transactions on Computers*, 27(6) pages 509–516, June 1978.
- [3] Voir par exemple une des références sur le sujet : *Introduction to Automata Theory, Languages, and Computation*, 3/E, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Addison-Wesley, 2007.
- [4] On pourra consulter le livre : *UML 2 pour les développeurs – Cours avec exercices corrigés*, [Xavier Blanc](#), [Isabelle Mounier](#) et [Cédric Besse](#), Eyrolles, 2006.

Utiliser les détails d'implémentation dans les tests

L'accès aux détails de réalisation d'un logiciel permet de construire des cas de tests complémentaires à ceux qui sont construits par les techniques dites Boîte noire. Ces détails peuvent être le code ou sa représentation sous forme de graphe de contrôle dans le cas des tests unitaires, un graphe d'appel entre module dans le cas de tests d'intégration ou la connaissance de la valeur de paramètres système (par exemple le nombre maximal de connexions ou la valeur de temporiseurs) dans le cas de tests système.

Les tests de cette famille, appelés tests Boîte blanche ou tests Boîte de verre, vont étudier les détails de la réalisation en compléments des spécifications qui restent nécessaires pour, au minimum, l'analyse du résultat des tests (rôle de l'oracle). Ces détails apportent de nouvelles informations au testeur, mais peuvent aussi l'induire en erreur : la construction de cas de test à partir du code risque de reproduire les mêmes erreurs dans les cas de test que celles présentes dans le logiciel, en particulier concernant l'oubli de certaines fonctionnalités.

C'est pour cette raison que l'on limite fréquemment l'utilisation des détails de réalisation à la construction de mesures sur les jeux de tests utilisés, comme les mesures de couverture ou les mesures des taux de détection d'erreurs dans le cas d'injection volontaire de fautes.

5.1. Définir des objectifs de couvertures par rapport au flot de contrôle

5.1.1. Principes

Un jeu de tests fait fonctionner une certaine partie du logiciel : certaines instructions sont exécutées, d'autres non. Les critères de couverture liés aux instructions vont au minimum mesurer la part d'instructions exécutées. Il s'agit du critère « toutes les instructions ». Ce critère de base est assez faible c'est-à-dire qu'il est assez peu en relation avec le nombre d'erreurs résiduelles : on obtient assez facilement un ensemble de tests qui satisfait ce critère^[1] alors qu'il reste un certain nombre d'erreurs dans le logiciel ; ce critère doit être vu comme une exigence minimale et non une mesure de qualité. C'est pour cela que les standards (par exemple ceux de la norme IEEE 1008, « IEEE Standard for Software Unit Testing ») prévoient des objectifs de couverture plus ambitieux. La difficulté est que plus une couverture est à même de qualifier la qualité des tests menés, plus elle est exigeante et plus elle est difficile, voir impossible à obtenir. C'est pour cela qu'il existe un grand nombre de variantes possibles, chacune cherchant à maximiser le critère efficacité *versus* complexité. Nous présentons ici les plus classiques ; toutes étant accessibles au travers de nombreux outils.

Tout d'abord il nous faut noter que la représentation du code, sous forme textuelle, n'est pas spécialement bien adaptée à sa manipulation en vue de la mesure de ces couvertures : on préfère, pour de nombreuses raisons, le manipuler sous une forme plus compacte et plus universelle appelée « graphe de contrôle ».

5.1.2. Construire et utiliser un graphe de contrôle

Un graphe de contrôle représente, à l'aide de sommets (dessinés comme des cercles, des ovales ou des rectangles) et d'arcs orientés (dessinés à l'aide de flèches), la partie contrôle du code d'un composant (sa séquence d'instructions) : les instructions sont représentées par les sommets et les branchements par les arcs. Lorsqu'un branchement est conditionné par l'évaluation d'une condition (expression conditionnelle ou fin de répétition), la valeur de la condition est portée sur l'arc qui correspond au branchement. Certains sommets ne correspondent pas directement à une instruction mais à une situation particulière : le début ou entrée dans le composant (sommet unique), la fin ou la sortie du composant (sommet unique), la jonction de différentes branches d'exécution (fin d'instruction conditionnelle, fin de

boucle, label associé à un goto). Nous présentons figure 5.1 un exemple de graphe de contrôle et le code associé. On remarquera le point de jonction J1, le point d'entrée E, le point de sortie S. On remarquera également que tout sommet qui correspond à une instruction de fin de la fonction (ici les sommets I3, I4 et I6) conduit au sommet S.

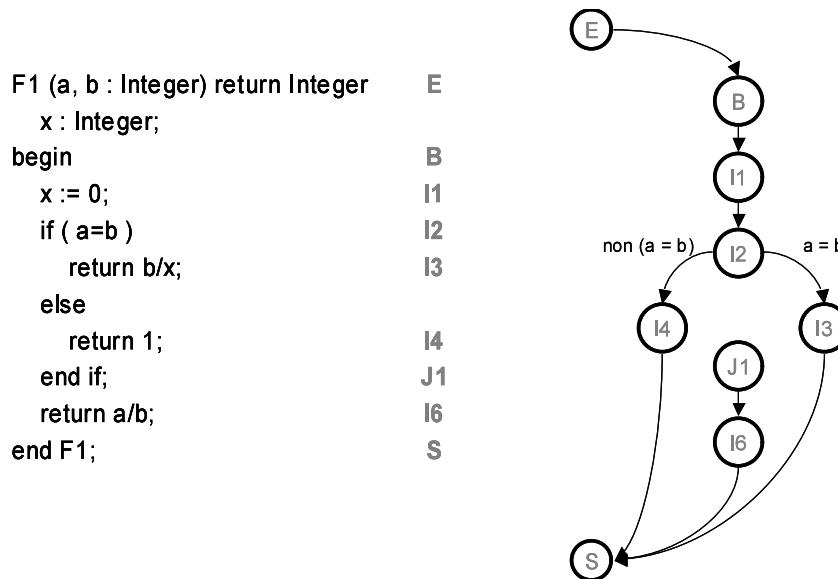


Fig. 5.1 Une fonction et son graphe de contrôle associé

La simple construction de ce graphe de contrôle (généralement faite par un outil spécialisé ou tout simplement par le compilateur utilisé) permet de mettre en évidence une erreur courante : une partie du code, ici l'instruction I6, est inaccessible (c'est ce que l'on appelle communément du « code mort »). Mais ce graphe permet également d'analyser des séries de jeu vis-à-vis de certains critères de couverture.

5.1.3. Critère de couverture « toutes les instructions »

Ce critère est le critère le plus simple : il est atteint dès que les jeux de tests ont parcouru la totalité des sommets correspondant à une instruction du graphe de contrôle. De façon générale, le taux de couverture pour une série de tests vis-à-vis de ce critère sera défini comme le rapport entre le nombre de nœuds couverts par au moins un jeu de tests et le nombre de nœuds total du graphe de contrôle. Ce critère est l'exigence minimale pour la certification

des logiciels embarqués dans le contexte aéronautique (norme RTCA/DO-178B ou EUROCAE ED-12 B) et permet de qualifier les logiciels au niveau C (risques majeurs encourus en cas d'erreurs du logiciel).

Différentes études montrent qu'en pratique, pour des séries de tests fonctionnels « sérieuses », le taux de couverture pour des composants complexes écrit à l'aide de langages complexes (comme le C++) est compris entre 40 % et 60 % alors qu'il est proche de 100 % lorsque le composant est simple. Cette différence s'explique principalement par la difficulté à parcourir la part du code défensif qui correspond au traitement de cas très peu probables. Ce taux peut être mesuré sur le code source, ou sur le code objet, c'est-à-dire après compilation. Ce n'est pas la même chose et B. Beizer^[2] montre qu'une série de tests couvrant 100 % du code objet ne couvre généralement que l'ordre de 75 % du code source^[3] !

Si l'on prend le programme précédent après la correction de l'erreur découverte plus tôt (suppression du code mort) on obtient le graphe de contrôle présenté figure 5.2 avec le code correspondant.

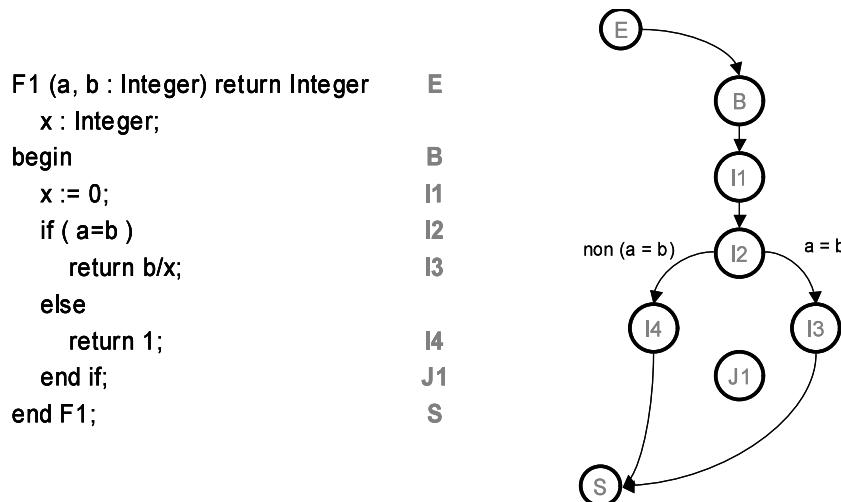


Fig. 5.2 La fonction précédente corrigée (partiellement) et son graphe de contrôle

Les séquences de tests ($a = 1, b = 1$) et ($a = 0, b = 1$) permettent d'obtenir le critère « toutes les instructions » puisque l'on parcourt respectivement les nœuds (E,B,I1,I3,S) et (E,B,I4,S). Cette séquence de test permet de mettre en avant l'erreur de division par zéro au niveau de l'instruction I3. Ne pas satisfaire cette couverture est donc un critère de mauvaise qualité des jeux de

tests ou du programme : certaines instructions ne sont pas exercées et donc des erreurs grossières vont passer à travers les mailles du filet du testeur ou, autre possibilité, le programme contient du code inaccessible, ce qui correspond sûrement à une erreur de réalisation.

Cependant, comme le montre l'exemple présenté ci-après, satisfaire ce critère de couverture n'est pas pour autant une assurance que les tests ont été bien menés : certaines erreurs assez faciles à découvrir peuvent restées cachées alors que toutes les instructions ont été exercées au moins une fois. Ainsi le problème de division par zéro lorsque le paramètre a est égal à b risque de ne pas être détecté. Par exemple, le jeu de tests $(1,0)$, $(\text{MaxInt}, 0)$, $(0, \text{MaxInt})$ semble pertinent (au vu par exemple d'une analyse des partitions) et satisfait le critère de couverture « toutes les instructions ». Cependant, il ne permet pas de découvrir l'erreur de division potentielle par zéro puisque la séquence $(E, B, I1, I2, J1, I4)$ n'est pas exercée.

En effet, cette erreur liée à l'instruction $I4$ n'est pas systématique ; elle dépend de l'histoire de la séquence qui contient cette instruction : si cette séquence passe par $I3$ alors l'exécution de $I4$ est correcte, dans le cas contraire elle conduit à une erreur. Le critère de couverture « toutes les instructions » ne prend pas en compte cette notion de causalité entre les instructions, il faut donc définir d'autres types de couverture afin de mieux intégrer les causalités existantes entre instructions.

```

F2 (a, b : Integer) return Integer      E
  x : Integer;
begin
  x := 0;
  if ( a!=b )
    return x := 2;
  end if;
  return (a+b) / x;
end F1;                                S

```

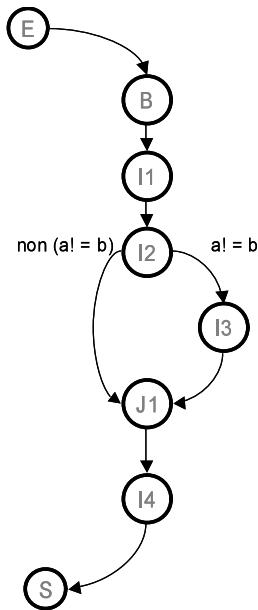


Fig. 5.3 Exemple de la faible pertinence du critère de couverture « toutes les instructions »

5.1.4. Critère de couverture « tous les chemins »

Une façon de remédier au problème précédent est d'imposer que les séquences de test couvrent l'ensemble des chemins possibles : il s'agit du critère « tous les chemins ». Celui-ci n'est atteint que si **tous** les chemins menant d'un point d'entrée à un point de sortie du composant ont été couverts par un test. L'on voit bien que ce critère de couverture ne laisse échapper que peu d'erreurs (on verra cependant que certaines erreurs de codage sont « insensibles » à ce critère) ; il est donc de ce point de vue très satisfaisant. Cependant, le nombre de tests qui sont nécessaires pour satisfaire ce critère est un obstacle majeur à son utilisation en pratique : en effet, d'une part, le nombre de chemins indépendants croît exponentiellement en fonction du nombre d'instructions conditionnelles et, d'autre part, une simple boucle peut rendre quasi infini ce nombre de chemins. On notera que de nombreuses variantes existent sur la façon de traiter les boucles, comme le critère de couverture « tous les *i*-chemins » où l'on impose un cas de test avec zéro itération, un cas avec une itération, un cas avec deux itérations et ainsi de suite jusqu'à *i* itérations. Cela permet de réduire la combinatoire, mais ne l'annule pas complètement.

On notera que, du fait de cette combinatoire, ce critère de couverture n'est pas utilisé en pratique et que l'on vise généralement des critères de couverture plus faibles mais plus réalistes.

5.1.5. Critère de couverture « toutes les branches ou toutes les décisions »

La première façon de procéder pour améliorer le critère de couverture « toutes les instructions » tout en maîtrisant la combinatoire est de chercher à passer par toutes les branches du graphe de contrôle ; ceci revient à imposer qu'au niveau de chaque décision, il y ait un jeu de tests qui rend la décision « vrai » et un jeu de tests qui rend la décision « faux ». Ceci permet d'améliorer la prise en compte des relations de causalité qui existent entre les instructions : I4 dépend causalement de I3 et donc un chemin qui emprunte la branche I2-J1 « masque » cette causalité et conduit à l'erreur. Sur l'exemple précédent (figure 5.3) il faut inclure par exemple dans la série de tests le jeu ($a = 1, b = 2$) pour satisfaire le critère de couverture « toutes les branches » et détecter ainsi l'erreur.

Lorsque ce critère de couverture est atteint (100 % des branches du graphe de contrôle sont couvertes par au moins un jeu de tests) alors le critère de couverture « toutes les instructions » est aussi atteint : le critère de couverture « toutes les branches » est donc strictement plus sévère que le critère « toutes les instructions ». Ce critère permet selon la norme aéronautique D0 178B de qualifier des systèmes embarqués au niveau B (situations dangereuses ou difficiles encourues en cas d'erreurs du logiciel).

Cependant, là encore, certaines erreurs communes risquent de passer inaperçues alors que la couverture « toutes les branches » est atteinte. C'est le cas, fréquent, où les décisions sont des expressions construites à l'aide de plusieurs conditions et que l'évaluation de ces expressions se fait selon un mode « court » ; par exemple, l'évaluation de l'expression « A or B » s'arrête généralement après l'évaluation de A si celui-ci est évalué à « Vrai ». Une erreur liée à l'évaluation de B ne sera pas forcément détectée par une série de tests satisfaisant le critère de couverture « toutes les branches ».

Ainsi sur l'exemple présenté figure 5.4, le critère de couverture « toutes les branches » est atteint avec le jeu de tests ($a = 1, b = 1$) et ($a = 1, b = 2$) ;

pourtant le risque de division par zéro dès que a ou b égale zéro n'est pas détecté par cette série de tests.

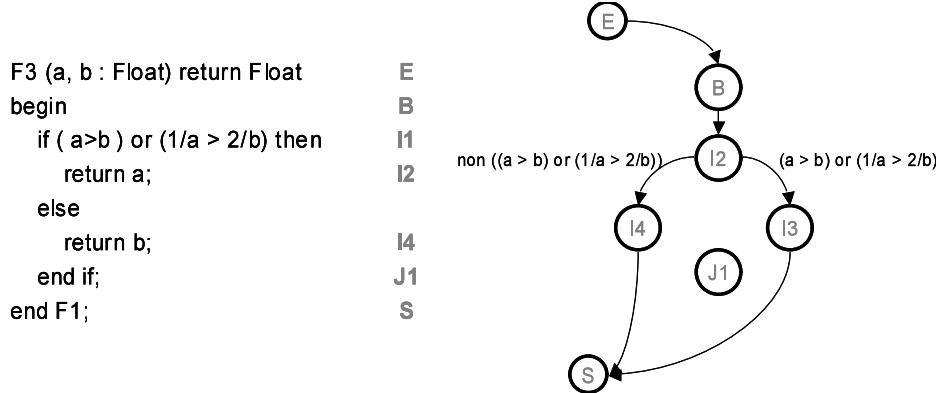


Fig. 5.4 Cas de non-pertinence du critère « toutes les branches »

De façon générale, le critère « toutes les branches » ne permettra pas systématiquement de mettre en évidence une erreur de codage fréquente : l'oubli d'une condition dans une expression conditionnelle composée ou une inversion d'un opérateur. Ainsi, l'expression (A or B) et les expressions (A) ou (A and B) ne seront pas nécessairement distinguées par une série de tests satisfaisant le critère « toutes les branches ». En effet, une série de tests comprenant un jeu de valeurs vérifiant ($A = Vrai, B = Vrai$) et ($A = Faux, B = Vrai$) permettra de couvrir toutes les branches (expression vraie et expression fausse) mais ne permettra pas de découvrir qu'il manque une partie des conditions dans un cas où qu'il y a eu une erreur sur l'opérateur utilisé. En exerçant chaque condition séparément (et donc en ajoutant un jeu de tests vérifiant $B = Faux$), l'erreur pourrait être trouvée.

5.1.6. Critère de couverture « toutes les conditions »

Ce critère se concentre sur les conditions élémentaires des expressions conditionnelles (élémentaires au sens qu'elles ne peuvent être décomposées sous une forme de conjonction ou disjonction de conditions plus simples). Ce critère est satisfait lorsque, pour chaque expression conditionnelle E et pour chaque condition C intervenant dans cette expression, la série de tests couvre les cas $C = Vrai$ et $C = Faux$. C'est donc un critère de couverture orthogonale au critère de couverture « toutes les décisions », qui permet de

trouver des erreurs que ce dernier ne couvre pas. Ainsi l'expression (A or B) et l'expression (A) seront bien distinguées par une série de tests satisfaisant ce critère. Néanmoins, l'absence de contrainte sur les décisions au sens « globale » (et donc sur le parcours des branches) est un handicap à ce critère de couverture. On l'utilise donc généralement combiné avec le critère « toutes les décisions ».

5.1.7. Critère de couverture « toutes les conditions – décisions »

Ce critère est assez sévère et demande à ce que la série de tests permette de couvrir toutes les branches **et** toutes les valeurs des conditions intervenant dans les expressions conditionnelles. Ainsi pour une expression de la forme (A or B) il faudra un jeu de tests où A est à Vrai, un jeu où A est à Faux, un jeu où B est à Vrai, un jeu où B est à Faux, un jeu où l'expression complète vaut Vrai et enfin, un jeu où l'expression vaut Faux. Ces différentes contraintes peuvent être obtenues avec $N + 1$ jeux de tests pour une expression contenant N conditions, ce qui reste raisonnable.

Cependant, bien qu'améliorant le critère de couverture « toutes les branches » (et aussi le critère « toutes les conditions ») ce critère de couverture reste incapable de garantir que les tests permettront de découvrir des erreurs où le programmeur confond un opérateur booléen avec un autre (erreur somme toute fréquente, en particulier lorsqu'une négation apparaît dans l'expression). Ainsi, si l'on considère l'expression (A or B) codée par erreur en l'expression (A and B), la série de tests comprenant un jeu vérifiant ($A = \text{Vrai}, B = \text{Vrai}$) et ($A = \text{Faux}, B = \text{Faux}$) satisfera le critère « toutes les conditions – décisions » mais ne permettra pas de distinguer les deux expressions. Elles valent vrai toutes les deux dans le premier cas et faux toutes les deux dans le second. Pour les distinguer il faudrait ajouter le jeu de tests vérifiant ($A = \text{Vrai}, B = \text{Faux}$) ou le jeu de tests ($A = \text{Faux}, B = \text{Vrai}$). Cette erreur étant relativement fréquente et causant potentiellement des erreurs graves de comportement, d'autres critères ont été développés pour améliorer la pertinence de ce type de critère de couverture.

5.1.8. Critère de couverture « toutes les conditions multiples »

Une possibilité est d'imposer que toutes les combinaisons possibles de valeurs de conditions apparaissent dans la série de tests. Ce critère, très sévère, assure que la série de tests utilisés permet de détecter toute erreur portant sur une décision. Malheureusement, ce type de couverture est inutilisable en pratique puisqu'il nécessite un nombre de jeux de tests exponentiel par rapport aux nombres de conditions intervenants dans une décision. Une étude portant sur le code embarqué dans différents avions^[4] montre que le nombre de conditions dans une expression dépasse fréquemment la valeur de 11 pour atteindre des valeurs supérieures à 36. Dans ce contexte, il faudrait, selon les cas, entre 2 048 et plusieurs milliards de tests pour obtenir ce critère de couverture ! Inutile d'insister sur l'aspect impraticable de cette exigence.

5.1.9. Critère de couverture « toutes les conditions – décisions modifiées (MC/DC) »

Afin d'améliorer les différents critères de couverture basés sur les conditions tout en restant dans une exigence réalisable, un critère de couverture astucieux est employé pour la certification des logiciels embarqués pour l'avionique au niveau A (*i.e.* situations catastrophiques encourues en cas d'erreurs du logiciel) : le critère MC/DC pour « *Modified Condition/Decision Coverage* ». Afin de réduire la combinatoire sur le nombre de jeux de tests nécessaires, on ne s'intéresse à un jeu de tests faisant varier la valeur d'une condition (*i.e.* il existe un autre jeu de tests identique sauf pour la condition concernée) que si la variation de cette condition influe sur le résultat de la décision indépendamment des autres conditions. Par exemple, pour la décision (A or B), la condition A influe sur la décision, il faut donc prendre au moins deux jeux de tests identiques hormis pour A qui vaut Vrai dans l'un et Faux dans l'autre, comme J0 : (A = Faux, B = Faux) et J1 : (A = Vrai, B = Faux) et tels que la décision n'ait pas la même valeur les deux fois. Ici, elle vaut respectivement Faux pour J0 et Vrai pour J1. Donc J0 et J1 permettent d'apprécier l'influence de A sur la décision indépendamment de B.

Cependant B influe aussi sur la condition indépendamment de A : le jeu de tests J2 = (A = Faux, B = Vrai) le prouve car il donne une valeur de la décision différente de celle donnée par J0 (A reste constant dans les deux cas). Par contre il est inutile d'ajouter la quatrième variante possible (A = Vrai, B = Vrai) car ni le changement de A (par rapport à J2) ni le changement de B (par rapport à J1) n'apportera de changement sur la décision qui reste vraie dans tous ces cas. De façon générale, pour une décision contenant N conditions, il faudra au plus $N + 1$ jeux de tests pour satisfaire le critère MC/DC au lieu de 2^N pour le critère « toutes les conditions multiples » avec une qualité des jeux de tests à peu près équivalente.

Il existe différentes façons de construire une série de jeux de tests satisfaisant le critère MC/DC dont une, simple à mettre en œuvre mais longue à exposer, décrite en détail dans un tutorial fait par (et pour) la NASA TM-2001-210876. Nous donnons ici une vieille technique, basée sur l'utilisation de table et adaptée à un usage manuel lorsque le nombre de conditions reste de l'ordre de 4 ou 5 (ce qui est le cas dans la majorité des décisions) ; cette technique s'automatise fort bien et peut alors être appliquée sur des décisions faisant intervenir plus de conditions (disons de l'ordre d'une quinzaine).

Le principe est le suivant : partant d'une décision faisant intervenir un ensemble de conditions, on construit une table, avec en colonne les conditions (et le résultat) de la décision, et en lignes les différentes possibilités pour les conditions. Pour chaque ligne, et chaque condition, on repère (dans la partie droite de la table) la ligne « complémentaire » (*i.e.* celle où seule la valeur de la condition change) et l'on examine si le résultat est le même ou est changé. S'il est changé alors les lignes correspondantes définissent des cas de tests relatifs à l'impact de la condition sur le résultat de la décision. En couvrant toutes les conditions, l'on produit un ensemble de jeux de tests respectant le critère MC/DC.

Considérons la décision :

$$\text{Freiner} := ((\text{Vitesse} > 100) \text{ or } (\text{Pente} > 30)) \text{ and } ((\text{Mode} = 1) \text{ or } (\text{Mode} = 2))$$

avec *Vitesse*, *Pente* et *Mode*, trois variables entières et appliquons cette construction sur cet exemple (voir tableau 5.1).

Nous construisons tout d'abord la table en énumérant les seize cas possibles (il y a quatre conditions, et donc seize combinaisons différentes). Nous supprimons alors les cas impossibles (lignes sombres et texte en italique) car les conditions (Mode = 1) et (Mode = 2) ne pouvant être vraies simultanément.

Tab. 5.1 Table MC/DC pour la décision Freiner : = ((Vitesse > 100) or (Pente > 30)) and ((Mode = 1) or (Mode = 2))

	Vitesse>100	Pente>30	Mode=1	Mode=2	Freiner	Vitesse>100	Pente>30	Mod
1	V	V	V	V				
2	V	V	V	F	V	-	-	
3	V	V	F	V	V	-	-	
4	V	V	F	F	F	-	-	
5	V	F	V	V				
6	V	F	V	F	V	14	-	
7	V	F	F	V	V	15	-	
8	V	F	F	F	F	-	-	
9	F	V	V	V				
10	F	V	V	F	V	-	14	1
11	F	V	F	V	V	-	15	
12	F	V	F	F	F	-	-	1
13	F	F	V	V				
14	F	F	V	F	F	6	10	
15	F	F	F	V	F	7	11	
16	F	F	F	F	F	-	-	

Il faut alors déterminer pour chaque condition les cas de tests (i.e. les lignes) qui montrent une influence de la condition sur la décision indépendamment des autres conditions. Pour cela on recherche des lignes « complémentaires » vis-à-vis d'une condition, c'est-à-dire des lignes telles que le résultat de la décision est différent alors que seule cette condition varie entre ces deux lignes ; c'est ici le cas pour les lignes 2 et 4 vis-à-vis de la condition (Mode = 1) : on indique alors le numéro de la ligne complémentaire dans la colonne

correspondante. On dit dans ce cas que la condition ($\text{Mode} = 1$) et « couverte » par les lignes 2 et 4. En procédant de la sorte pour chaque ligne et chaque condition on remplit la partie droite du tableau.

Une fois cette table remplie, on extrait un ensemble de lignes tel que chaque condition soit couverte par deux lignes complémentaires de cet ensemble. On procède par étapes et par propagation de contraintes dans le but de couvrir toutes les conditions : par exemple, on peut partir de la ligne 6, ce qui nous oblige à inclure la ligne 8 pour couvrir la condition ($\text{Mode} = 1$). Afin de couvrir la condition ($\text{Mode} = 2$) il faut alors insérer la ligne 7. Avec ces trois lignes, nous couvrons les deux conditions ($\text{Mode} = 1$) et ($\text{Mode} = 2$). Il nous faut alors couvrir les autres conditions ; pour cela nous ajoutons, par exemple, la ligne 14, qui nous permet de couvrir la condition ($\text{Vitesse} > 100$). L'ajout de la ligne 10 nous permet alors de couvrir la condition ($\text{Pente} > 30$). À ce point, la totalité des conditions sont couvertes par ces cinq lignes (6,7,8,10,14) qui constituent nos cinq cas de tests satisfaisant le critère MC/DC.

On remarquera que cette technique est très simple à mettre en œuvre et que pour N conditions indépendantes, il nous faut au plus $N + 1$ jeux de tests différents mais qu'*a contrario*, il nous aura fallu construire une table de 2^N lignes pour les obtenir (ce qui limite l'utilisation de cette technique de construction à des décisions avec assez peu de conditions). Le lecteur intéressé pourra se référer au rapport de la NASA mentionné précédemment pour une technique adaptée à des décisions faisant intervenir de nombreuses conditions.

On remarquera également que cette couverture permet de détecter toute erreur de codage portant sur une confusion sur un opérateur logique (un OR est codé AND ou le contraire). Ainsi si la décision Freiner était codée ou aurait dû être codée comme une des variantes F_V1 à F_V4, les jeux de tests correspondant aux lignes 6,7,8,10,14 (en grisé dans la table 5.2) permettent de détecter cette erreur : chacune des variantes présente au moins une différence (en gras dans la table) quand à la valeur de la décision par rapport à celle obtenue avec la décision initiale.

- $\text{Freiner} := ((\text{Vitesse} > 100) \text{ or } (\text{Pente} > 30)) \text{ and } ((\text{Mode} = 1) \text{ or } (\text{Mode} = 2))$

- $F_V1 := ((\text{Vitesse} > 100) \text{ or } (\text{Pente} > 30)) \text{ OR } ((\text{Mode} = 1) \text{ or } (\text{Mode} = 2))$
- $F_V2 := ((\text{Vitesse} > 100) \text{ AND } (\text{Pente} > 30)) \text{ and } ((\text{Mode} = 1) \text{ or } (\text{Mode} = 2))$
- $F_V3 := ((\text{Vitesse} > 100) \text{ or } (\text{Pente} > 30)) \text{ and } ((\text{Mode} = 1) \text{ AND } (\text{Mode} = 2))$
- $F_V4 := ((\text{Vitesse} > 100) \text{ AND } (\text{Pente} > 30)) \text{ and } ((\text{Mode} = 1) \text{ AND } (\text{Mode} = 2))$

Tab. 5.2 Exemple de la pertinence du critère MC/DC

	Vitesse>100	Pente>30	Mode=1	Mode=2	Freiner	F_V1	F_V2	F_V3	F_V4
1	V	V	V	V					
2	V	V	V	F	V				
3	V	V	F	V	V				
4	V	V	F	F	F				
5	V	F	V	V					
6	V	F	V	F	V	V	V	F	I
7	V	F	F	V	V	V	F	F	I
8	V	F	F	F	F	V	F	F	I
9	F	V	V	V					
10	F	V	V	F	V	V	F	F	I
11	F	V	F	V	V				
12	F	V	F	F	F				
13	F	F	V	V					
14	F	F	V	F	F	V	F	F	I
15	F	F	F	V	F				
16	F	F	F	F	F				

5.1.10. Critère de couverture « tous les i-chemins »

Dans cette variante de la couverture « tous les chemins », impraticable du fait de la combinatoire monumentale engendrée par les boucles, on se limite aux

chemins qui passent de zéro à i fois dans les boucles ; par exemple, pour atteindre le critère de couverture « tous les 2-chemins » il faudra passer zéro, une fois et deux fois dans chaque boucle. Ce type de couverture permet de détecter des erreurs non nécessairement détectées par une série de tests satisfaisant le critère « toutes les branches » ou les différents critères de couverture basés sur les conditions intervenant dans les décisions.

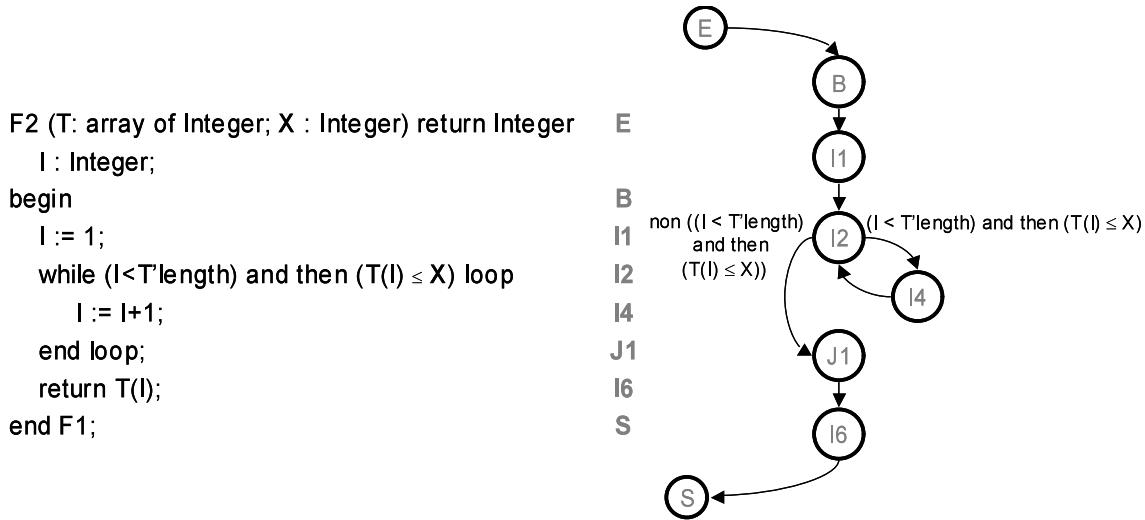


Fig. 5.5 Second cas de non-pertinence du critère « toutes les branches »

Considérons l'exemple proposé figure 5.5 : une fonction qui, étant donné un tableau d'entiers T trié par ordre croissant et un entier X , devrait calculer la plus petite valeur de T strictement supérieure à X . À droite est présenté le graphe de contrôle associé à cette fonction. Le jeu de test ($T = (0,2,4)$, $X = 3$) permet de parcourir le chemin $(B, I1, I2, I4, I2, J1, I6)$. Il satisfait donc le critère de couverture « toutes les branches ». Malheureusement, deux problèmes présents dans cette fonction ne sont pas détectés : lorsque les valeurs sont soit toutes plus grandes que X soit toutes plus petites. Dans les deux cas, la fonction retourne respectivement la première ou la dernière valeur, aucune des deux ne satisfaisant le critère de recherche !

On l'a vu précédemment, le critère de couverture MC/DC permet d'améliorer fréquemment la qualité des jeux de tests produits. Ici, pour l'obtenir, il faut nécessairement ajouter un jeu de tests qui rende Faux la première condition de la décision et Vrai la seconde partie, ce qui oblige à tester le cas où toutes

les valeurs de T sont plus petites que X et donc permet de détecter un des deux problèmes de cette fonction. Cependant, ce critère de couverture n'imposera pas de tester le cas où l'on ne rentre jamais dans la boucle ; en effet tous ces critères basés sur les décisions, ne distinguent pas le cas où l'on sort d'une boucle après y être passé au moins une fois du cas où l'on sort de la boucle sans n'y être jamais entré.

Les critères de couverture « tous les i -chemins » vont imposer de passer de zéro à i fois dans chaque boucle et permettront donc de détecter des erreurs liées aux problèmes des boucles dans lesquelles on ne rentre pas. On n'utilise en pratique des valeurs 1 ou 2 pour i (« 1-chemins » ou « 2-chemins ») afin de rester dans une combinatoire raisonnable tout en améliorant significativement le critère de couverture « toutes les branches ».

5.1.11. Autres critères de couverture basés sur le flot de contrôle

Il existe de nombreux autres critères de couverture basés explicitement ou implicitement sur le flot de contrôle. Nous n'en citons que deux.

Couverture LCSAJ

Ce critère de couverture LCSAJ pour « *Linear Code Sequence and Jump* » (ou PLCS pour « Portion Linéaire de Code et Saut » en français) est une n -ième variation du critère « tous les chemins » ; sa particularité est de pouvoir être mise en œuvre sans la construction du graphe du flot de contrôle associé au code à tester^[5]. Un chemin LCSAJ est une suite d'instructions exécutées en séquence suivie d'un saut. On note généralement un tel chemin par trois numéros de lignes : le numéro de la ligne du début de la séquence linéaire, le numéro de la ligne de fin et enfin le numéro de la ligne atteinte après l'instruction de saut. La notion de séquence linéaire doit être comprise comme séquence de lignes contiguës ($n, n + 1, n + 2\dots$) parcourues lors de l'exécution du test. Cette séquence peut donc contenir des instructions de branchements qui ne donnent, dans cette exécution, pas lieu à un branchement.

L'avantage de cette couverture est qu'elle est plus complète que la couverture « toutes les décisions » sans pour autant demander la prise en compte d'un nombre exponentiel de chemins comme avec la couverture « tous les chemins ».

Son inconvénient, comme celui d'autres mesures basées sur le flot de contrôle est de ne pas prendre en compte les chemins infaisables.

Appel de fonction, module, méthode

Une autre façon de caractériser les jeux de tests est de se pencher sur la manière dont ils se comportent vis-à-vis des fonctions, procédures ou méthodes.

La première exigence possible est de demander à ce que chaque sous-programme soit appelé au moins une fois lors des tests. Il s'agit d'un critère assez faible. De plus, les défauts sont souvent introduits au niveau des interfaces. On peut alors chercher à couvrir, non pas l'exécution de tous les sous-programmes ou méthodes, mais tous les appels à ces sous-programmes. De la sorte, on augmente la probabilité de mettre en évidence un défaut sur une interface. On peut également rechercher tous les enchaînements de deux méthodes ou sous-programmes possibles.

Ces critères de couvertures sont pertinents, mais, de nouveau, le testeur sera confronté à l'efficacité de ses tests dans le temps qui lui est imparti et devra souvent se contenter de rechercher des critères de couvertures plus simples.

5.2. Définir des objectifs de couvertures par rapport au flot de données

5.2.1. Principes

Les différents critères de couverture basés sur le flot de contrôle se focalisent sur les instructions et les séquences d'instruction en termes de chemin d'exécution et ne considèrent pas directement la causalité qui existe entre différentes instructions : typiquement une variable prend une valeur par une instruction d'affectation I1 et cette valeur est utilisée dans un calcul ou un

prédicat par une autre instruction I2. Il peut être intéressant de chercher à couvrir les chemins qui passent par I1 puis par I2, sans pour autant couvrir tous les chemins d'exécutions possibles. Les critères de couverture basés sur le flot de données vont permettre de combler l'écart qui existe entre le critère « toutes les branches » et le critère « tous les chemins » en prenant en compte, non les décisions comme cela est fait pour les différents critères vus précédemment mais l'utilisation des variables faites à travers le programme.

Pour fixer les idées, considérons le graphe de contrôle donné figure 5.6. L'instruction I3 correspond à l'affectation d'une variable X à la valeur 0 et l'instruction I7 à l'utilisation de cette variable dans le calcul $10/X$ pour affectation à une variable Y.

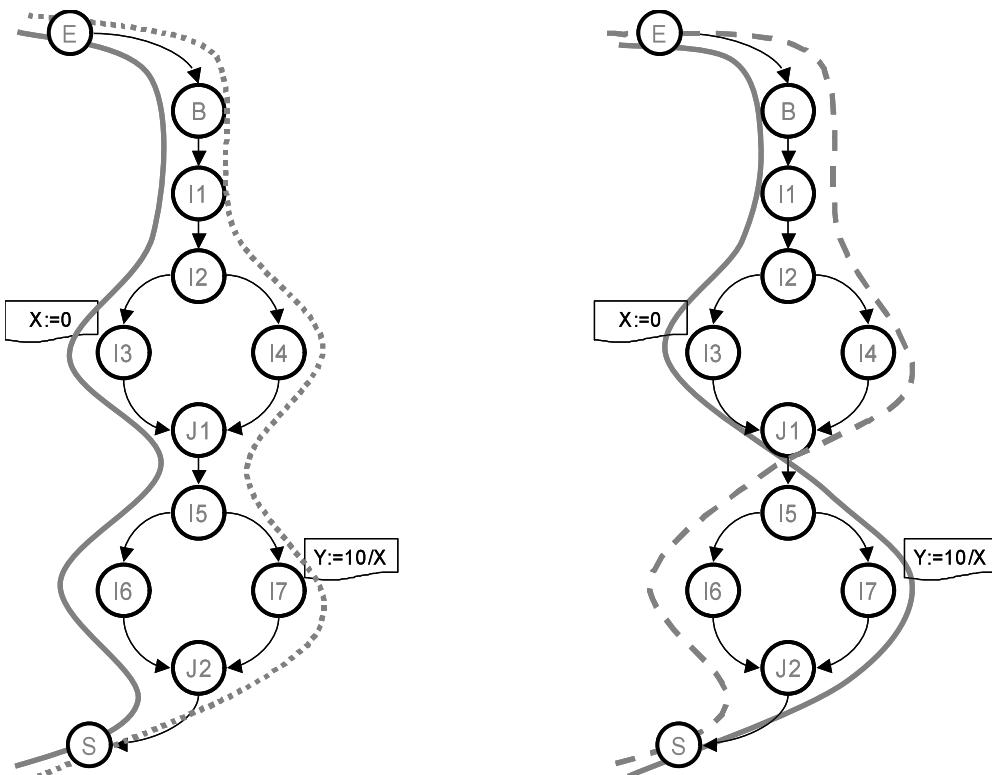


Fig. 5.6 Graphe de contrôle et deux possibilités de couverture « toutes les branches »

À gauche et à droite figurent des chemins d'exécution qui permettent dans les deux cas d'obtenir le critère de couverture « toutes les branches ». Cependant, les chemins sélectionnés sur la gauche (E B I1 I2 I3 J1 I5 I6 J2 S) et (E B I1 I2 I4 J1 I5 I7 J2 S) ne permettent pas de découvrir l'erreur

potentielle de division par zéro en I7. *A contrario*, un des deux chemins sélectionnés sur la droite (E B I1 I2 I3 J1 I5 J7 J2 S), permet de découvrir l'erreur ou de garantir que ce cas est bien pris en compte dans le logiciel (chemin rendu impossible par la conditionnelle I5) ; le critère de couverture « toutes les branches » n'est donc pas pertinent vis-à-vis de ce type d'erreur.

5.2.2. Construire et utiliser un graphe de flots de données

Un graphe flot de données est un graphe flot de contrôle sur lequel on ajoute des informations quant à la vie des variables du composant à tester. Au niveau d'une instruction (et donc au niveau d'un nœud du graphe de contrôle) une variable X peut :

- recevoir une valeur (on dit aussi être définie) comme dans une instruction d'affectation « $X := 5$ » ;
- être utilisée dans un calcul « $Y := X + 5$ » ;
- être utilisée ou un prédictat « $\text{if } (X > 0)$ » ou enfin,
- être détruite (*killed* en anglais) lorsque par exemple elle sort de son domaine de définition.

Dans chacun de ces cas, on ajoute une information (une étiquette), symbolisée à l'aide d'une lettre sur la figure 5.7, au niveau des nœuds du graphe de contrôle. On remarquera qu'une même instruction conditionnelle peut utiliser une variable dans un calcul puis dans un prédictat comme dans « $\text{if } (X + 5) > 0$ » ou qu'une instruction d'affectation peut utiliser la valeur d'une variable dans un calcul avant de lui affecter le résultat de ce calcul comme dans « $X := X + 5$ » ; dans ces deux cas, le nœud correspondant est étiqueté par deux lettres au lieu de une.

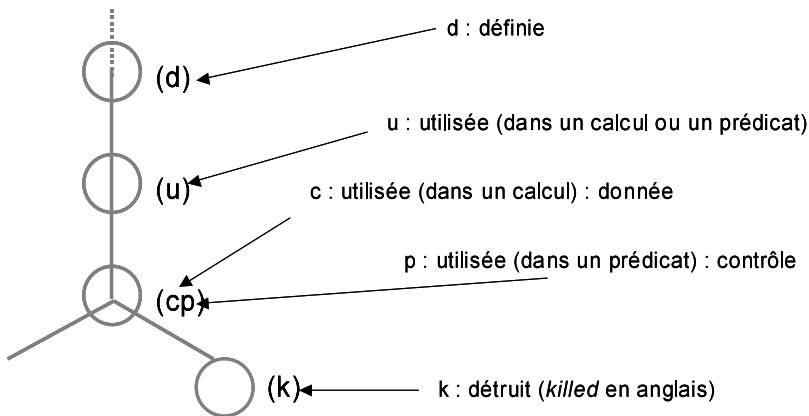


Fig. 5.7 Annotation du graphe de contrôle par rapport à une variable

Une fois le graphe de contrôle construit, il peut être utilisé statiquement par la recherche de séquences particulières mettant en évidence des erreurs de réalisation comme l'oubli de l'affectation d'une variable ou, dynamiquement par la définition de critères de couverture liés à la vie des variables (et donc liés à l'étiquetage des nœuds).

5.2.3. Séquences caractéristiques et utilisation statique de l'étiquetage des nœuds

Chaque exécution d'un logiciel correspond à un parcours du graphe de contrôle associé. Si l'on observe l'utilisation d'une variable, l'annotation du graphe de contrôle va « produire » un mot sur l'alphabet $\{d, u, c, p, k\}$ comme « duddduuuk ». L'ensemble des mots pouvant être engendrés par une exécution du logiciel définit alors un *langage*. Analyser ce langage revient alors à analyser les exécutions possibles du logiciel, cette analyse pouvant être faite *a priori* et de façon statique, c'est-à-dire sans réelle exécution du code. Pour ce faire on décrit le langage pouvant être engendré à l'aide d'expressions régulières composées des lettres de l'alphabet (ici $\{d, u, c, p, k\}$) et de symboles comme le plus ('+') et l'étoile ('*') et si nécessaire avec l'aide de parenthèses. Le symbole '*' sera utilisé pour désigner la répétition d'un mot un nombre quelconque de fois (peut être aucune fois) alors que le symbole '+' désignera la répétition d'un mot un nombre de fois non nul. Les règles dictant la construction de ce langage (présentées figure 5.8) sont au

nombre de trois : la transcription d'une séquence, d'un choix et la prise en compte d'une boucle.

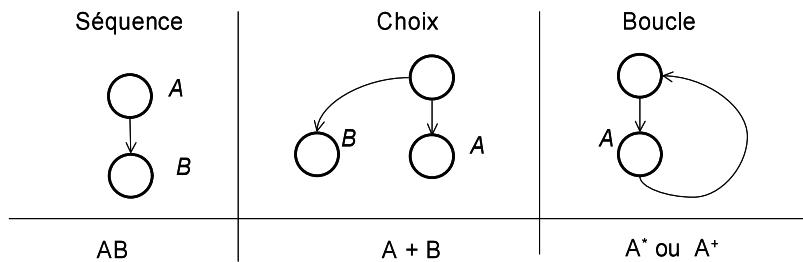


Fig. 5.8 Règles de construction du langage associé à un graphe annoté

```

Function Pgcd(x,y: Integer) return Integer is
  p, q: Integer;
begin
  p := x; q:= y;           B1
  while p /= q do          P1
    if p > q
      then
        p := p-q           P2
      else
        q:= q-p           B2
    end -- if
  end -- while
  return p                  B3
end                         B4

```

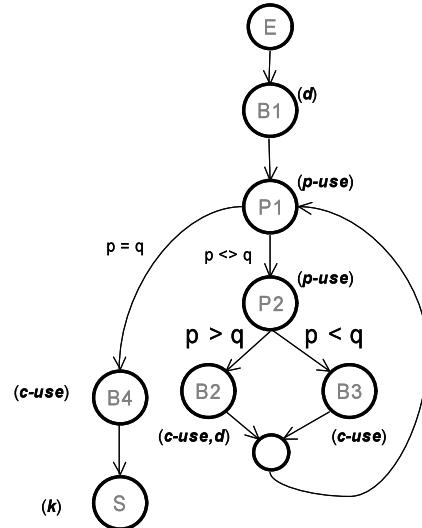


Fig. 5.9 Une fonction et son graphe de contrôle annoté

Considérons l'exemple présenté figure 5.9 qui correspond à une fonction de calcul du Pgcd de deux nombres entiers par successions de différences (algorithmes d'Euclide sans utilisation de la division euclidienne). Le graphe de contrôle associé est annoté par rapport à l'utilisation de la variable p : cette variable est, par exemple, définie en B1 puis utilisée en P1 dans un prédictat (conditionnelle « p > q »). On remarquera qu'en B2, elle est, dans la même instruction, utilisée successivement dans un calcul (« q-p ») puis définie par

une affectation, donnant l'annotation (c-use,d). Avant toute utilisation, une variable est systématiquement considérée comme « killed ». Le langage engendré par les exécutions possibles de cette fonction vis-à-vis de l'utilisation de la variable p peut être défini par l'expression régulière : kd(pp(cd+c))*pck) ce qui correspond au langage composé des mots qui commencent par « kd », qui se terminent par « pck » et qui comprennent en leur milieu un certain nombre (peut-être nul) de mots commençant par « pp » suivis de « cd » ou de « c ». Ainsi, les mots « kd-pck », « kd-ppcd-pck » ou « kd-ppc-ppc-ppcd-pck » font partie des mots de ce langage et correspondent respectivement aux chemins exécutions E-B1-P1-B4-S, E-B1-P1-P2-B2-P1-B4-S, E-B1-P1-P2-B3-P1-P2-B3-P1-P2-B2-P1-B4-S [\[6\]](#).

Les erreurs potentielles d'exécution vont pouvoir être détectées par l'analyse de ce langage et en particulier par la recherche des séquences à deux caractères qui définissent des séquences d'erreur ou d'alarme comme la séquence « ku » qui correspond à l'utilisation d'une variable (u) après qu'elle ait été détruite (k) et avant toute initialisation (absence de d). Le tableau 5.3 détaille la plupart des séquences caractéristiques et précise s'il s'agit d'un cas normal ou d'un cas pathologique (alarme ou erreur).

Tab. 5.3 Séquences caractéristiques et interprétation en termes de comportement

dd	Double définition ; séquence correspondant vraisemblablement à une exécution <i>normale</i> .
du (dc dp)	Définition puis utilisation ; séquence <i>normale</i> .
dk	Définition puis destruction : probablement une <i>erreur</i> (variable inutilisée).
ud	Utilisation avant définition : ne permet pas de conclure directement ; dépend s'il y a eu une définition avant cette séquence ; dans ce cas, il s'agit d'un cas <i>normal</i> ; autrement il s'agit d'une <i>erreur</i> .
uu	Succession d'utilisations ; cas <i>normal</i> .
uk	Utilisation puis destruction ; cas <i>normal</i> .
kd	Définition après une destruction ; cas <i>normal</i> .
ku (kc kp)	Destruction puis utilisation ; probablement une <i>erreur</i> (variable non initialisée).

kk

Succession de destructions ; probablement une *erreur* (cas très particulier).

Afin d'analyser automatiquement ces séquences (lors de la phase de compilation par exemple) il est nécessaire de simplifier les expressions définissant les langages engendrés par l'exécution du code correspondant. En particulier il est nécessaire de simplifier autant que faire se peut les * et les +. Il est par exemple possible d'utiliser le théorème de *Huang* (1979) qui énonce qu'étant donné trois caractères A, B, C, si S, une chaîne de deux caractères est une sous-chaîne des mots AB^nC ($n > 0$), alors S est également une sous-chaîne de AB^2C . Ainsi en utilisant ce résultat il est possible, dans la recherche de séquences d'alarme ou d'erreur (qui sont des chaînes de deux caractères), de substituer X^2 à X^+ et $(1+X^2)$ à X^* . De la sorte, par itération de ce procédé, la recherche de séquences d'erreur ou d'alarme dans une expression régulière quelconque peut se ramener à la recherche de séquences d'erreur ou d'alarme dans une expression sans * et sans +, ce qui est beaucoup plus simple. Par exemple, l'expression régulière concernant l'utilisation de la variable x dans la portion de code donnée ci-dessous est kd^*p . Cette expression se simplifie en $k(1+d^2)p$, qui s'écrit également $kp+kd^2p$, montrant ainsi la séquence d'erreur kp symptôme de l'utilisation d'une variable non initialisée.

Figure 5.10

```
Integer x, y;  
get(z)  
for i:+0 to z loop  
    x:=A(i);  
end loop  
If x = 0 then ...
```

5.2.4. Critères de couverture associés aux flots de données

La démarche précédente s'apparente plus à l'analyse statique qu'au test dynamique. L'annotation d'un graphe de contrôle peut cependant également être utilisée avec profit dans ce cadre en définissant des objectifs de couverture qui cette fois-ci s'appuieront non sur le flux de contrôle du logiciel mais sur son flux de données, c'est-à-dire que l'on cherchera à « suivre » les chemins qui vont de la définition des variables à leur utilisation. Différents critères basés sur ce principe et plus ou moins sévères (*i.e.* difficile à atteindre) peuvent être retenus. Pour les illustrer, nous considérons la portion suivante (figure 5.11) de graphe de contrôle annoté vis-à-vis d'une variable. Les traits entre les nœuds définissent des chemins d'exécution possibles entre deux nœuds modifiant ou utilisant la variable considérée (les autres nœuds ne sont pas représentés dans ce graphe).

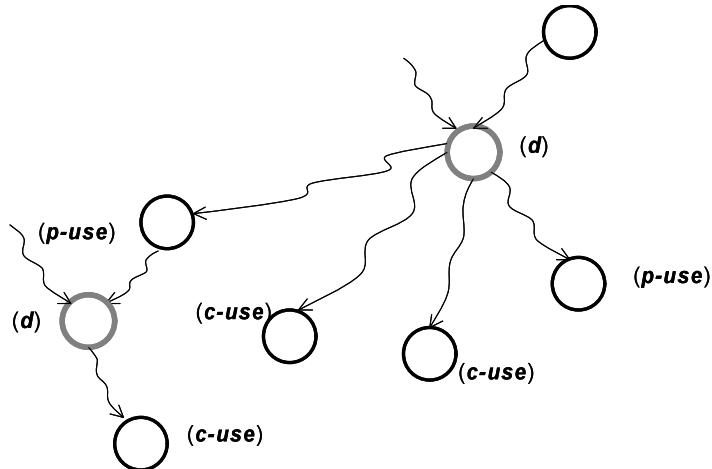


Fig. 5.11 Une partie d'un graphe de contrôle annoté

Un premier objectif peut être de chercher à vérifier que la valeur donnée à une variable est utilisée correctement dans les calculs qui suivent cette définition ; il s'agit du critère « ACU » pour « All Computation Uses » (toutes les utilisations dans un calcul). Pour l'obtenir il faut que, pour chaque variable du logiciel, les jeux de tests parcourent toutes les portions de chemins qui vont d'une définition de cette variable vers une utilisation de celle-ci dans un calcul sans être repassé par une nouvelle définition de la variable (on considère que seule la dernière définition impacte le calcul). Les portions de chemins considérées pour ce critère sont représentées en gris sur la figure 5.12.

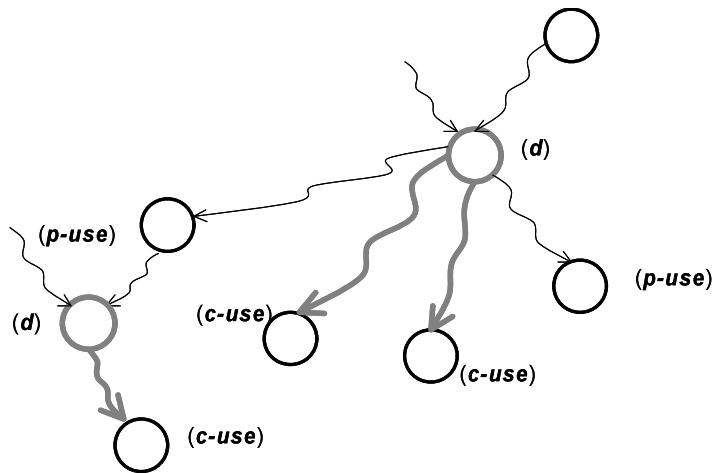


Fig. 5.12 Critère « ACU » (All Computation Uses)

De façon symétrique, on peut chercher à analyser l'impact d'une définition sur les branchements qui suivent cette définition ; il s'agit du critère « APU » pour « All Predicat Uses » (toutes les utilisations dans un prédicat). Pour satisfaire ce critère il faut que, pour chaque variable du logiciel, toutes les portions de chemins qui vont d'une définition de cette variable vers une utilisation de celle-ci dans un branchement soient couvertes par les jeux de tests. Ces portions sont représentées en gras sur la figure 5.13.

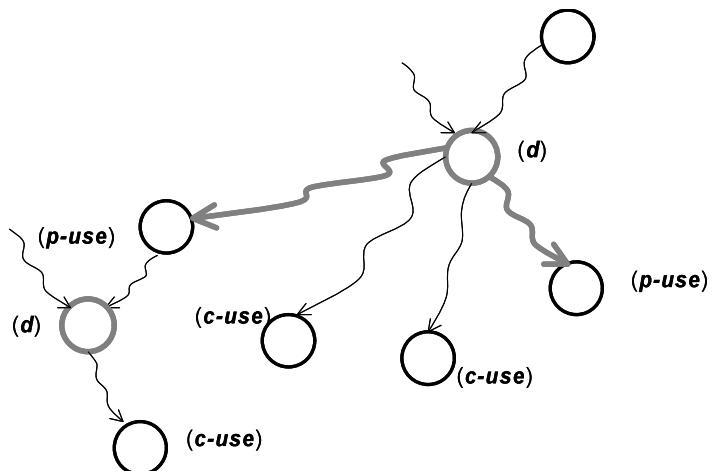
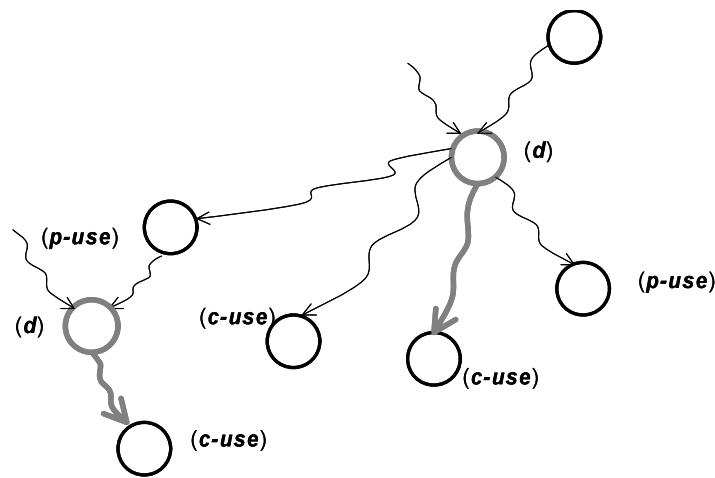


Fig. 5.13 Critère « APU » (All Predicat Uses)

En combinant et en « relâchant » quelque peu ces critères on obtient le critère « AD » pour « *All Definitions* » qui consiste à s’assurer que l’impact de toute définition d’une variable sur une utilisation de celle-ci (calcul ou branchement) est analysé par au moins un jeu de tests. Là encore, toute nouvelle définition de la variable annule les définitions précédentes. Les portions du graphe de contrôle considérées par ce critère sont représentées en gras sur la figure 5.14.



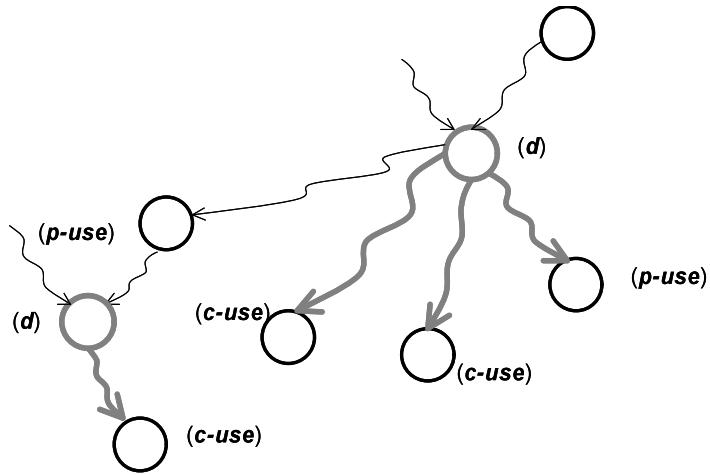


Fig. 5.15 Critère « ACU + P » (All Computation Uses + one Predicat)

Symétriquement, on peut chercher à couvrir toutes les utilisations dans un branchement et au moins une utilisation dans un calcul, critère « APU + C » pour « *All Predicat Uses and one Computation* » (toutes les utilisations dans les prédictats et au moins une utilisation dans un calcul). Ce critère est illustré figure 5.16.

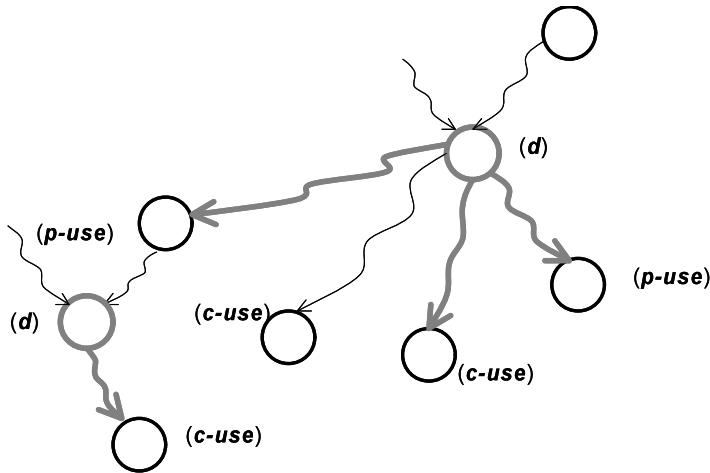


Fig. 5.16 Critère « APU + C » (All Predicat Uses + one Computation)

En poursuivant l’augmentation du degré d’exigence quant à la couverture de l’utilisation des variables après leur définition, on obtient le critère « AU » pour « *All Uses* » (toutes les utilisations) qui permet de garantir que toute

utilisation de toute variable après une définition de celle-ci est couverte par au moins un jeu de tests. C'est un critère relativement exigeant mais qui reste réaliste quant à sa mise en œuvre. Ce critère est illustré par la figure 5.17.

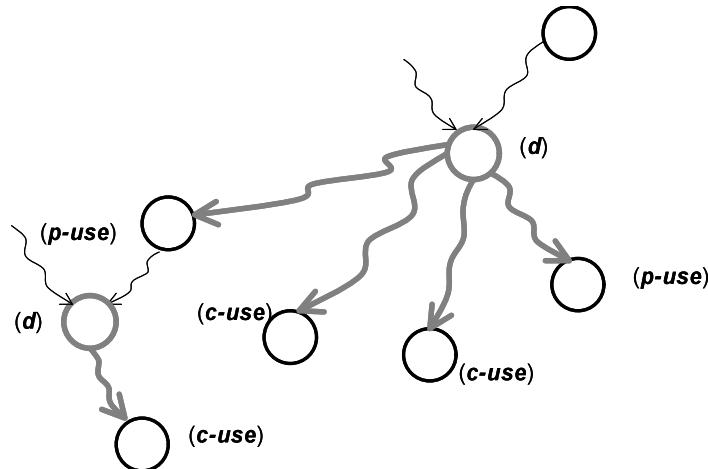


Fig. 5.17 Critère « AU » (All Uses)

Enfin, dans tous les critères de couvertures précédents, on se limite aux chemins ne comprenant pas de redéfinition. Ceci conduit, dans le cas de boucles comprenant une instruction d'affectation, à ne considérer que la dernière instruction d'affectation et donc à ne pas chercher à faire plus d'un tour de boucle. Si l'on souhaite garantir une couverture plus complète des boucles comprenant des instructions d'affectation de variable utilisées après la boucle, il est nécessaire de rechercher à couvrir des chemins allant d'une définition à une utilisation sans se limiter à la dernière définition rencontrée avant l'utilisation de la variable. Cela donne le critère de couverture « ADUP » pour « All Definition Uses Path » (tous les chemins d'utilisation des définitions) qui est illustré dans la figure 5.18.

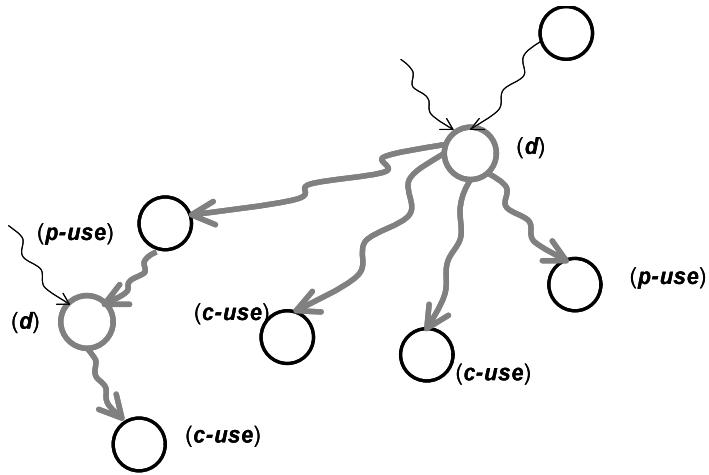


Fig. 5.18 Critères « ADUP » (All Definition Uses Path)

Il faut noter que ce dernier critère est difficile à obtenir du fait de sa définition même qui vise à parcourir la plupart des boucles autant que faire se peut. De la sorte il se rapproche très fortement du critère de couverture « tous les chemins » qui est en général irréaliste. Le schéma de la figure 5.19 compare les différents critères de couvertures évoqués dans ce chapitre en termes de sévérité, c'est-à-dire à la fois du point de vue de la difficulté à l'obtenir mais également de la pertinence de celui-ci vis-à-vis de la détection potentielle d'erreur. On notera que le critère « toutes les instructions » est le critère le moins sévère. On prendra donc garde à ne pas tirer de conclusion trop favorable quant à la réussite d'un passage de jeux de tests satisfaisant ce critère : la couverture « toutes les instructions » est un minimum à obtenir et certainement pas un critère d'arrêt des tests.

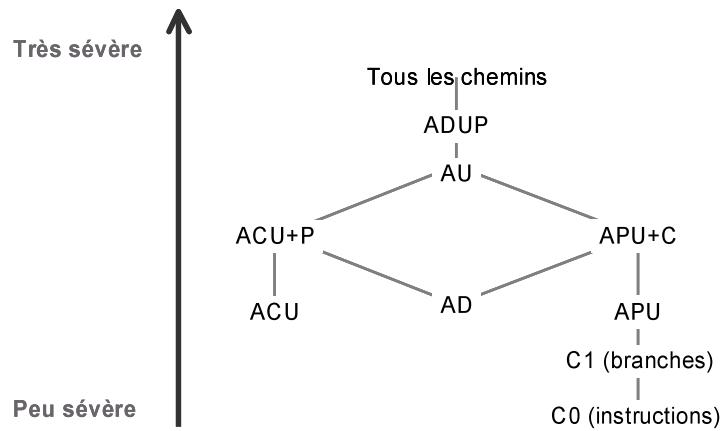


Fig. 5.19 Sévérité des critères de couvertures

5.3. Trouver les jeux de valeurs satisfaisant un critère de couverture

Une fois un objectif de couverture fixé, il faut essayer de l'atteindre le plus efficacement possible. Dans un premier temps, il est préférable de se focaliser sur les spécifications et construire les jeux de tests à l'aide de ces documents. En effet, comme dit précédemment, les défauts présents dans le logiciel peuvent conduire à construire des jeux de tests erronés. Ce qui est fait l'est peut-être correctement mais ce n'est pas forcément ce qui aurait dû être fait !

Par contre, dans le cas où l'objectif de couverture n'est pas atteint, l'accès aux détails de réalisation va permettre d'essayer de construire des jeux de valeurs permettant d'atteindre certaines instructions ou certains branchements. La façon de procéder est forcément empirique car le problème général est très difficile. En pratique, il s'agit de détecter les expressions conditionnelles présentes le long du chemin que l'on souhaite voir explorer, puis d'exprimer les contraintes qui devront être vérifiées au niveau de ces expressions afin d'en déduire un jeu de valeurs correctes. Ce problème est difficile car :

- le nombre d'expressions peut être grand, engendrant des systèmes de contraintes de grande taille ;

- certains chemins sont infaisables du fait de l'impossibilité de satisfaire simultanément plusieurs contraintes portant sur les mêmes variables. Par exemple, le système des trois contraintes $C1 = (X > Y)$, $C2 = (Y = Z)$ et $C3 (Z > X)$ ne peut jamais être vérifié.

Néanmoins, certains travaux récents, qui visent à construire les jeux de tests à partir de modèles du programme ou de ses spécifications, permettent d'envisager prochainement une automatisation efficace du traitement de ce problème.

5.4. Conclusion

L'utilisation des détails d'implémentation permet de compléter les approches de tests « Boîte noire » de trois façons. La première est que ces détails rendent possible la recherche de valeur permettant d'atteindre certaines parties du logiciel jugées importantes mais « difficiles d'accès » car correspondant à la combinaison d'événements ou valeurs particulières plus faciles à caractériser à l'aide du code qu'à l'aide des spécifications. La deuxième permet de définir des critères de couverture qui donnent une certaine garantie quant à une certaine exhaustivité des tests menés : toutes les instructions, toutes les conditions, toutes les branches ont bien été exécutées ou parcourues au moins par un test, ou encore toutes les utilisations d'une variable ont bien au moins été testées une fois. Ces critères de couvertures sont par ailleurs utilisés dans des procédures de certification des logiciels comme à travers la norme DO-178B « Software considerations in airborne systems and equipment certification » éditée par la compagnie américaine RTCA et utilisée pour autoriser l'utilisation des logiciels avioniques. Enfin, la troisième façon d'utiliser les détails de réalisation dans une activité de test est de contrôler le respect de standards par les programmeurs au sein d'une entreprise ou d'une communauté ou de mesurer certains aspects non fonctionnels du logiciel, comme sa maintenabilité, sa robustesse ou sa fiabilité.

Cependant, si ces détails offrent un regard différent au testeur et l'accès à bon nombre d'outils permettant une automatisation poussée des tests, ils peuvent également l'induire en erreur car ces détails définissent ce qui est fait et non

ce qui devrait être fait. Une évolution possible, qui permettrait de combiner les avantages des tests Boîte noire et Boîte blanche, est le développement (et l'utilisation par les programmeurs) de langages d'annotations suffisamment haut niveau pour être le reflet quasi exact des spécifications du logiciel, suffisamment simples d'emploi pour être utilisés sans surcoût, et suffisamment précis pour pouvoir être manipulés de façon automatique par des outils d'aide à la conception de jeux de tests.

Notes

[1] C'est particulièrement vrai pour la partie du logiciel qui traite le mode nominal de fonctionnement et est à nuancer pour la couverture de la partie du logiciel qui se charge du traitement des exceptions ou des cas singuliers.

[2] Beizer, Boris : *Software Testing Techniques*, Second ed., Van Nostrand Reinhold Company, Inc., 1983.

[3] La différence provient cette fois de l'action du compilateur qui, grâce à différents procédés d'optimisation, peut factoriser et éliminer certaines parties inutiles du code source ; le code généré, plus simple, est plus facile à couvrir.

[4] Voir par exemple le rapport de Chilenski, John J. : « An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion ». FAA Tech Center Report DOT/FAA/AR-01/18, April 2001 ou le rapport de la NASA TM-2001-210876 « A Practical Tutorial on Modified Condition / Decision Coverage » par Kelly J. Hayhurst, Dan S. Veerhusen, Rockwell Collins et John J. Chilenski.

[5] Voir par exemple l'article de Woodward, M. R., Hedley, D. et Hennell, M.A., « Experience with Path Analysis and Testing of Programs », *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, pp. 278-286 , May 1980.

[6] Les tirets ne sont utilisés ici que pour améliorer la lisibilité des expressions.

Processus et tests d'intégration

6.1. L'intégration dans le cycle de vie

Dans le cadre normatif du standard ISO/CEI 12207, Software life cycle Processes, qui est également celui de la norme CMM du SEI à très peu de chose près, l'architecture statique des processus d'ingénierie est spécifiée à l'aide de 17 processus de niveau 1 qui eux-mêmes sont raffinés en processus de niveau 2. Cette architecture explicite l'ensemble des activités à effectuer pour mener à bien un projet et ne rien oublier. Elle ne détermine pas l'ordre dans lequel les opérations doivent être effectuées, et encore moins la nature des interactions complexes qui assurent la cohérence entre ces différents processus par rapport à la finalité, *i.e.* un certain niveau de service, dont l'organisation projet sera le siège.

L'intégration intervient dans les processus « primaires », en développement et en maintenance. Elle est un élément essentiel du système qualité mis en œuvre dans les processus « support » pour garantir la qualité des livraisons effectuées par le projet.

N.B. : nous avons conservé les intitulés anglais de la norme, la traduction ne posant pas de problème.

Les activités relevant de l'ingénierie qui forment le cœur de la maîtrise d'œuvre du projet, rassemblent les processus de développement et de maintenance. La norme ISO 12207 décompose le processus de développement en treize activités organisées en quatre couches (voir figure 6.1). Les couches regroupent des ensembles de processus/activités apparentés du point de vue de la nature des activités. La norme distingue la couche métier, la couche système, la couche logiciel et les activités de

codage/tests unitaires qui ne vérifient et valident que ce qui a été programmé est correct. L'architecture constitue le cœur des couches système et logiciel.

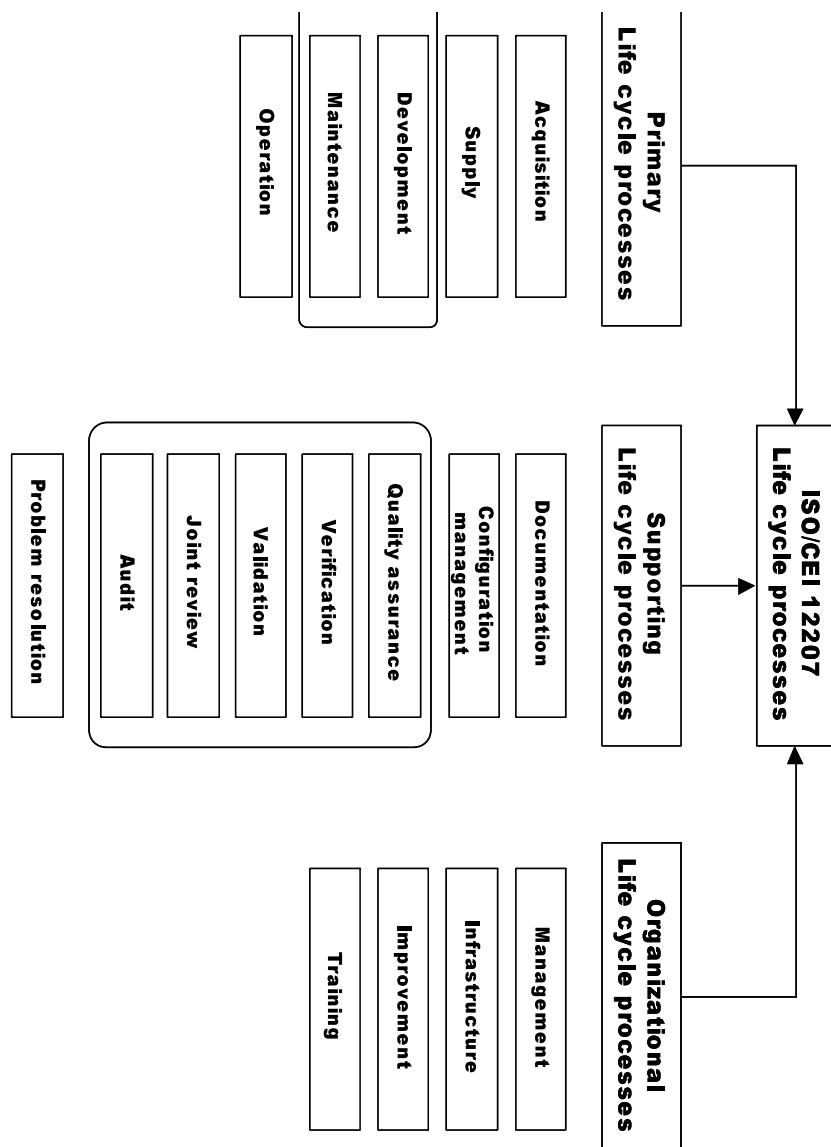


Fig. 6.1 Architecture statique des processus d'ingénierie d'après ISO12207

L'activité « process implementation » consiste à adapter, si nécessaire, le modèle de développement aux spécificités du projet. C'est une méta-activité qui demande une grande expérience car les décisions résultant de ce processus (choix des standards du projet, méthodes spécifiques, outils, langages de programmation) vont engager la réalisation sur le long terme et

l'économie globale du système (TCO – coût de possession totale). Un bon exemple de ce type d'adaptation serait les méthodes de type « *Test Driven Development* » préconisées par les méthodes agiles et/ou l'*eXtreme Programming*^[1].

Les activités « Software installation » et « Software acceptance support » sont la contrepartie, vue du développement, des processus d'exploitation et de support vue de l'organisation cible qui sont des contraintes pour le développement.

L'ensemble du domaine intégration, également dénommé IVVT pour intégration validation vérification test, regroupe les six processus, parmi les treize, encadrés sur le schéma de la figure 6.2.

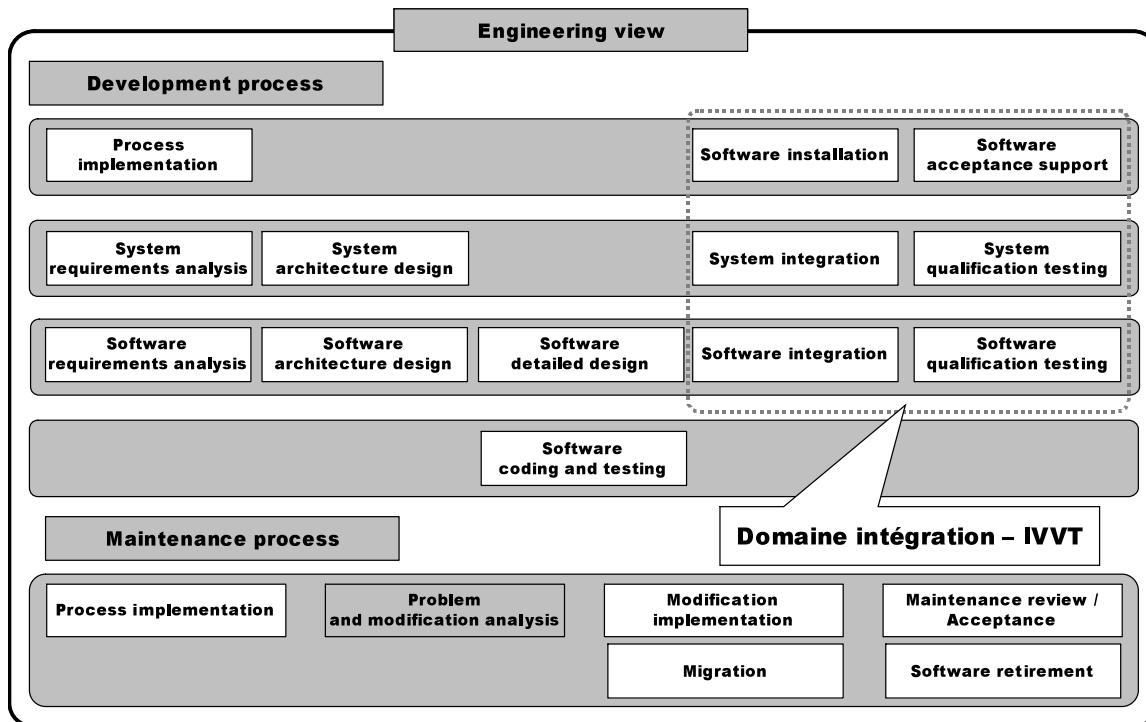


Fig. 6.2 Détail des processus d'ingénierie ISO12207 – Vision en couches

Pour l'étude de la dynamique de ces processus et de leurs interactions nous pouvons faire apparaître les structures de régulation, d'arbitrage et de décision que sont la MOA et la MOE.

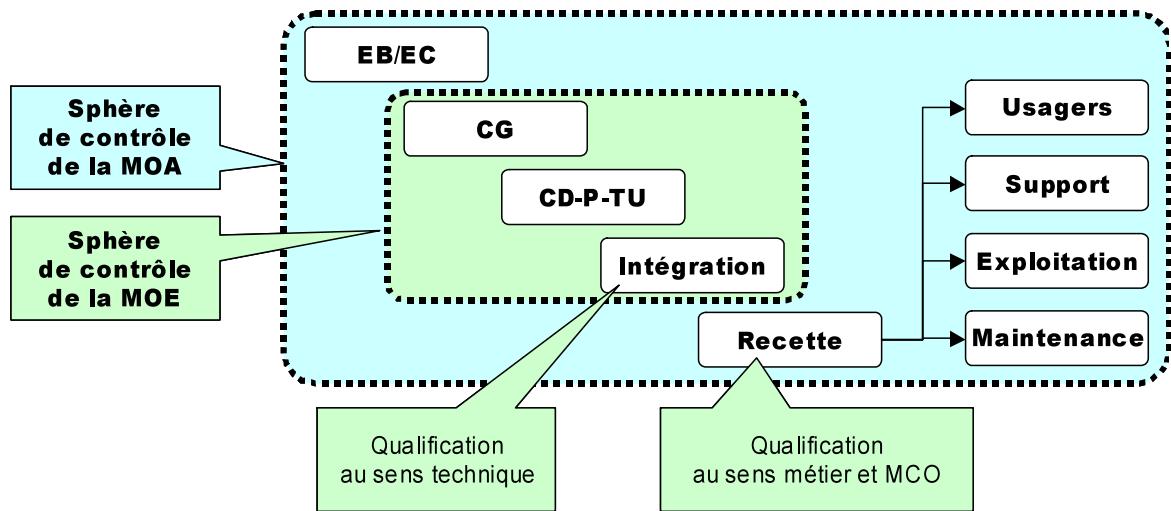


Fig. 6.3 Dynamique de l'enchaînement des processus

Le problème fondamental de la dynamique projet est de pouvoir estimer à quelle vitesse, un besoin exprimé selon les modalités FURPSE de la norme ISO 9126, Caractéristiques qualité des produits logiciel, et validé par la MOA, pourra être livré aux usagers et aux organisations en charge du MCO conformément au contrat de service. Comme il s'agit essentiellement de transformation de l'information, nous devons examiner les conditions de ces transformations successives : fractionnement en « pièces » informationnelles élémentaires, traduction dans le ou les langages de programmation, et surtout vérification et validation à différents niveaux (unitaire, logiciel, système), installation, support et maintenance.

Pour utiliser la terminologie systémique, cela revient à étudier d'une part a) le temps de réponse des processus MOE et plus particulièrement le temps de latence et l'inertie de l'organisation projet MOE, compte tenu des modalités d'allocation et de restitution des ressources affectées aux tâches et aux activités projet, et d'autre part b) de la circulation de l'information entre les différents processus et activités, ce que l'on pourrait appeler la fluidité de ces éléments informationnels.

Pour expliciter les spécificités de cette dynamique, nous allons examiner les aspects suivants :

- Capacité des équipes projet à prendre en compte un nouveau besoin.

- Capacité de l'architecture du système et du logiciel à permettre aux équipes de travailler en parallèle, tout en gardant la maîtrise de la complexité.
- Capacité du procédé de construction du système à ajouter, modifier, supprimer des constituants, sans avoir besoin de repasser tous les tests d'intégration et de recette, et garantir la stabilité de la qualité du service rendu antérieurement. Les deux derniers aspects correspondent aux caractéristiques non fonctionnelles S et E de l'ensemble FURPSE.

Dans le schéma de la figure 6.3, nous avons simplifié la nomenclature des processus pour nous en tenir à l'essentiel que l'on retrouve dans toutes les méthodes. Le problème fondamental n'est pas qu'il y en ait 3, 5, 9 ou 17, mais de savoir en analyser les interactions c'est-à-dire : les messages qu'ils échangent *via* l'organisation projet, les services qu'ils se rendent, les évènements qu'ils émettent et reçoivent, leur nature contrôlable ou observable vis-à-vis des pilotes de processus.

Ce sont les interactions qui vont déterminer dans une large part la fluidité et le bon déroulement des travaux à effectuer par les acteurs projets, et plus particulièrement ceux en charge de l'intégration. Le schéma systémique de ces interactions est celui de la figure 6.4. Ce schéma montre deux niveaux de contrôle :

- La sphère de contrôle de la MOA est optimisée sur l'expression des besoins métier à satisfaire, sur les conditions économiques CQFD de la solution retenue (analyse de la valeur et TCO), sur le niveau de qualité qu'il est raisonnable d'offrir pour garantir l'efficacité des usagers de l'organisation cible.
- La sphère de contrôle de la MOE est optimisée sur la qualité de la livraison et la pérennité de la solution technique proposée. Elle doit être telle que la performance globale de l'ensemble des processus, y compris le support, l'exploitation et la maintenance, soit réellement optimisée.

Chacune des sphères à sa logique propre, liée aux processus et activités dont l'autorité sera confiée soit à l'un soit à l'autre, sachant qu'une règle saine de management stipule que l'autorité ne doit pas se partager. Le système qualité,

via le contrat, joue le rôle de médiateur entre les deux et en assure la cohérence. C'est l'élément essentiel de la régulation globale.

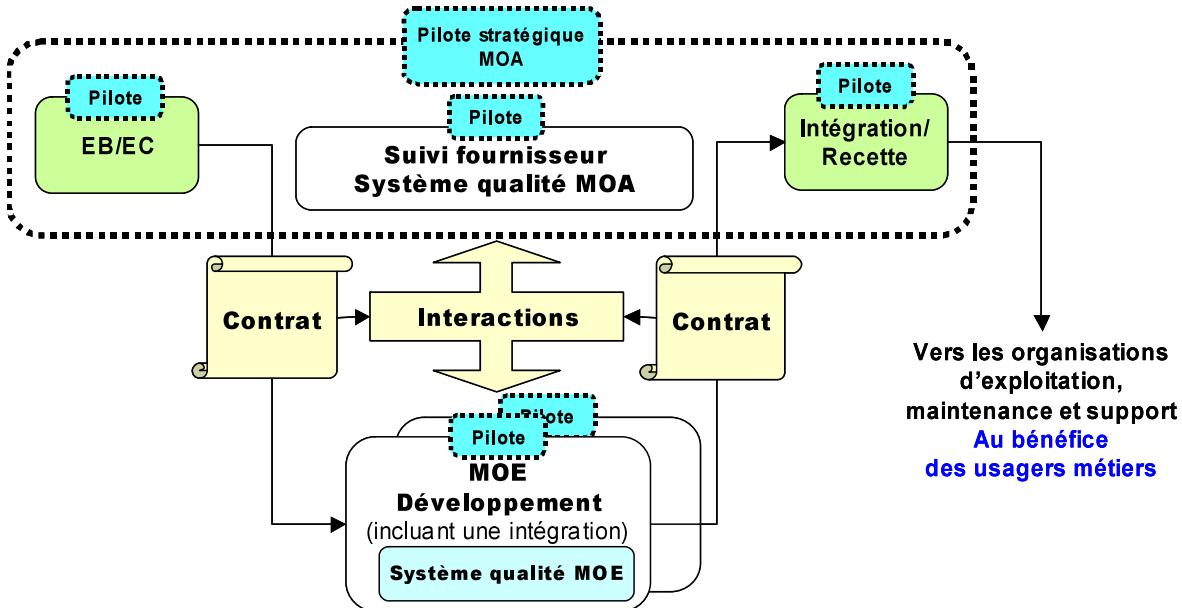


Fig. 6.4 Interaction MOA/MOE

6.2. Intégration dans une architecture client/serveur

Avec le développement des architectures distribuées dites *n*-tiers, dans les années quatre-vingt-dix, de nombreuses applications sont organisées conformément au pattern d'architecture MVC, comme suit :

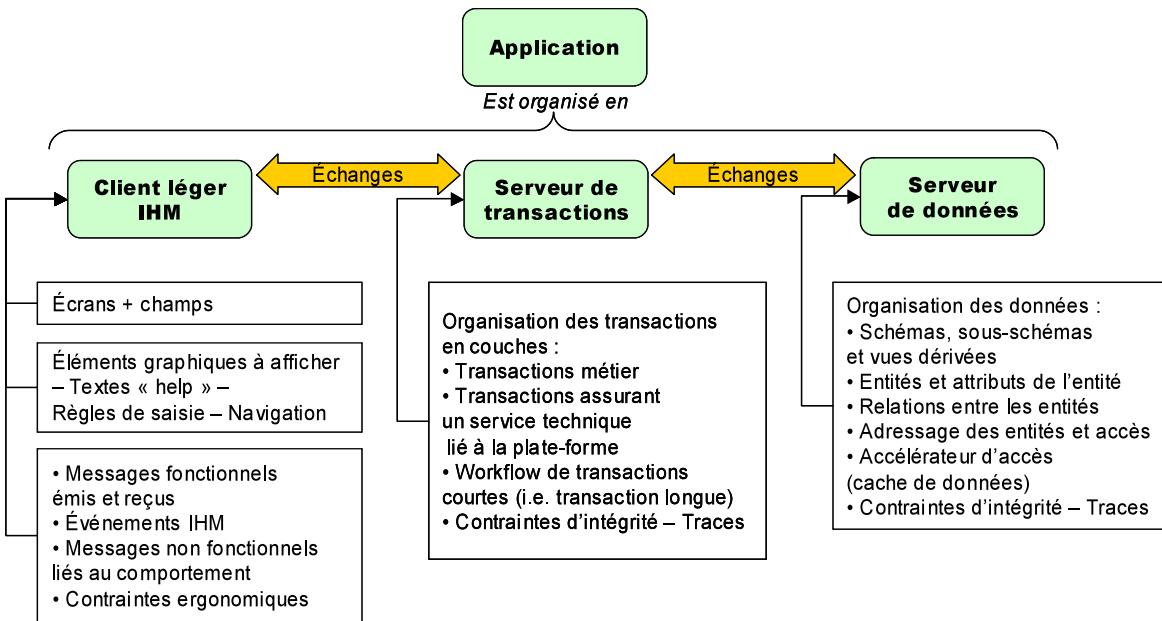


Fig. 6.5 Architecture d'une application conforme au pattern MVC.

En architecture client-serveur, la nomenclature des échanges (messages et événements) sur le « bus » d'intégration est un aspect fondamental de la gestion de configuration. Tout événement émis nécessite une réponse, ne serait-ce qu'un accusé de réception, pour garantir la conservation de l'information et que rien n'a été perdu. En exécution, il faudra gérer une trace des événements émis et reçus, en conformité avec la gestion de configuration.

L'application est articulée autour de trois éléments principaux (composants applicatifs) dont chacun comporte un certain nombre de composants élémentaires qui répondent aux différentes exigences fonctionnelles et non fonctionnelles, conformément aux caractéristiques qualités FURPSE qui ont été négociées pour le projet.

Le niveau application met à disposition de chacun des trois composants applicatifs une bibliothèque générale d'éléments logiciels communs. Le client léger IHM et le serveur de transactions ont en commun une bibliothèque d'interfaces pour les échanges qui leur sont propres. Idem pour les serveurs de transactions et de données.

Chacun des composants dispose de bibliothèques spécifiques pour leurs interfaces internes. La traçabilité des interfaces d'échanges dont peuvent

dépendre certains composants internes de chacun des trois éléments doit être assurée pour gérer la configuration de l'application.

Chacun des composants a une organisation hiérarchique qui lui est propre. En restant dans une logique MVC on peut raffiner les structures internes, et faire apparaître les « pièces » élémentaires qui vont être les intégrats de rang 0 du processus d'intégration, comme suit.

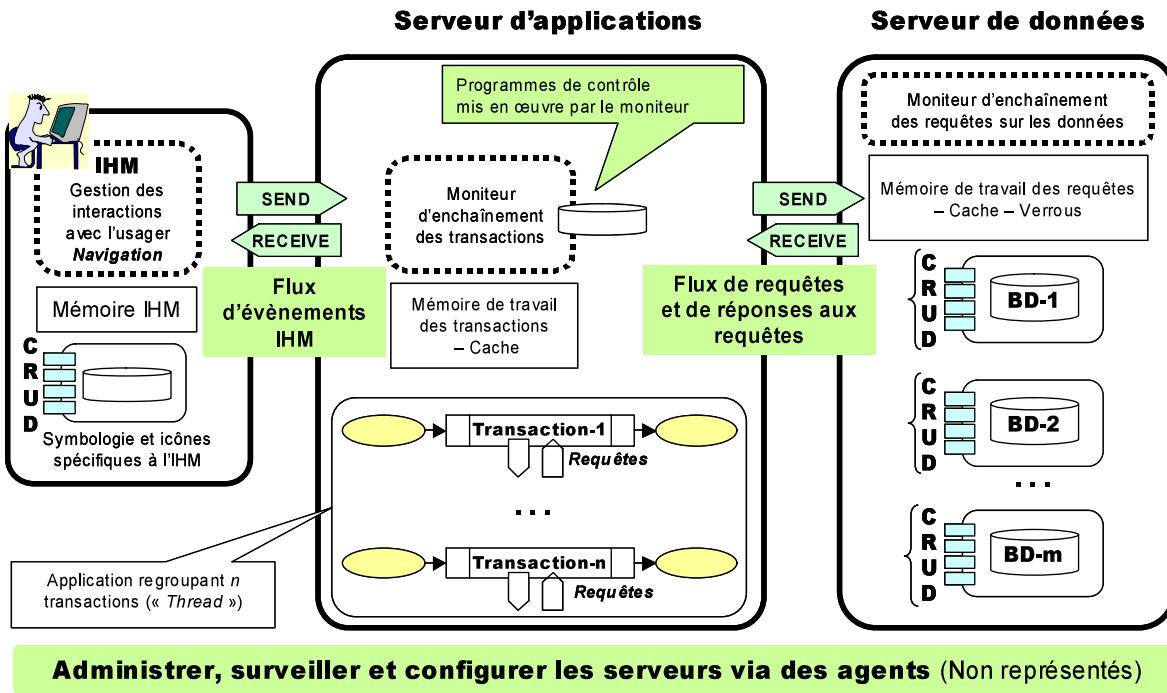


Fig. 6.6 Détails des serveurs d'une application générique conforme au pattern MVC

Chacun des éléments dispose de fonctions et de mémoires qui leur sont propres. Les mémoires sont *a priori* non persistantes, pour ne pas violer les règles élémentaires de saine gestion de l'intégrité des données (non-duplication). Les formats de données utilisées localement peuvent être spécifiques, mais doivent être conformes à une sémantique unique si les données sont communes aux différents éléments.

Chacun des éléments a également une logique d'enchaînement qui lui est propre. Le moniteur d'IHM gère les interactions avec l'usager, conformément aux règles d'ergonomie de l'acteur qui est de l'autre côté de l'interface ; si c'est un acteur non humain, l'ergonomie est imposée par l'appareil ou le

système en interface *via* le « langage » spécifique de cette interface. Les transactions sont enchaînées conformément à la logique transactionnelle bien connue (règles ACID). Les requêtes aux données sont exécutées en conformité avec la logique du SGBD sous-jacent. Si la logique transactionnelle impose des mises à jour simultanées dans plusieurs bases de données, il faut évidemment s’assurer que la logique nécessaire à l’intégrité de ces mises à jour est disponible (gestion de verrous, commit à deux phases, etc.).

Pour assurer une bonne fluidité de la prise en compte de nouveaux besoins, il convient de séparer, de la façon la plus rigoureuse, la logique métier de la logique plate-forme et des COTS utilisés. Cela revient à « encapsuler » les dépendances technologiques dans une couche *ad hoc* comme recommandé dans l’approche MDA/MDE préconisée par l’OMG, mais qui lui est bien antérieure. C’est la mise en pratique du principe de modularité dont l’effet sera de diminuer le nombre de tests à produire et de simplifier le processus d’intégration. C’est ce que montre le schéma de la figure 6.7.

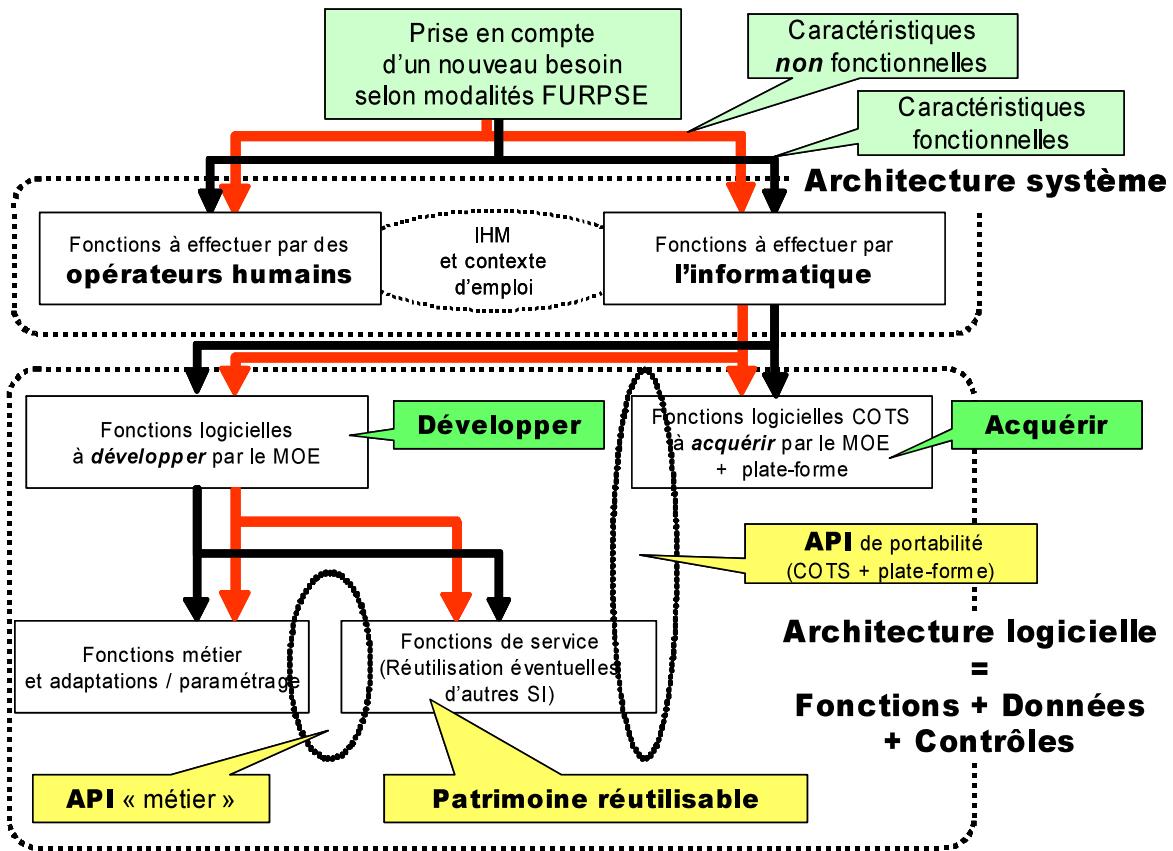


Fig. 6.7 Indépendance métier et plate-forme

Les fonctions métiers doivent être cohérentes avec les fonctions métiers informatisés, *via* une IHM *ad hoc*. Les fonctions métier informatisées peuvent être soit développées, soit acquises *via* des progiciels (COTS) qu'il faudra cependant paramétriser. Idéalement, les fonctions métier informatisées doivent être réalisées de façon indépendante de la technologie sous jacente afin de disposer d'un degré de liberté supplémentaire si le MOA souhaite changer de fournisseur de technologie en minimisant l'impact sur les processus métiers proprement dits (cf. la caractéristique évolutivité de FURPSE). Cette analyse est applicable à chacun des éléments de l'application.

N.B. : sur le schéma, les exigences fonctionnelles et les exigences non fonctionnelles sont clairement distinguées pour signifier que les unes et les autres sont essentielles à

l'expression de besoin et à la spécification des tests systèmes à effectuer en intégration système et en recette.

L'ensemble des entités ainsi identifiées constitue la configuration système qui décrit l'architecture statique du système. Cette configuration doit être conforme au schéma de la figure 6.8.

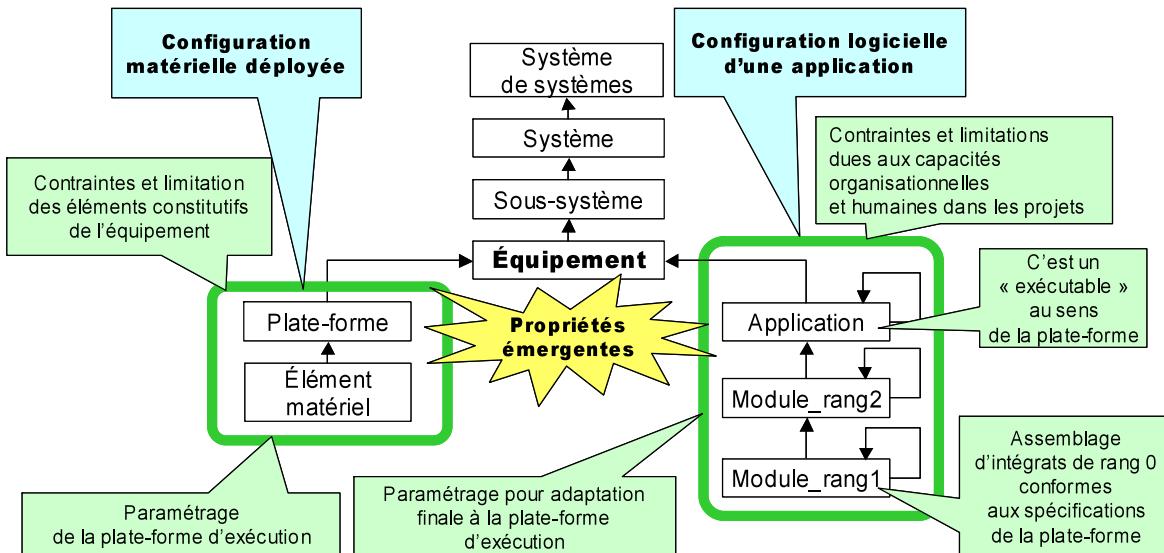


Fig. 6.8 Configuration système résultat du processus de conception

Ce schéma qui suit la norme IEEE 1220, « *Application and management of the systems engineering process* », nécessite quelques précisions terminologiques. L'élément application est un exécutable au sens de la plate-forme, ce qui fait qu'une « application » au sens classique peut être constituée de plusieurs exécutables. Dans une architecture distribuée ce sera toujours le cas car il y aura, au minimum, trois exécutables, mais souvent beaucoup plus : l'IHM sur le serveur client léger, l'application proprement dite sur le serveur de transactions, et les données nécessaires aux transactions sur le serveur de données. Dans l'informatique mainframe, avant les postes de travail « intelligents », le problème ne se posait pas en ces termes car toute la logique et « l'intelligence », i.e. le logiciel, étaient hébergées dans le mainframe ; les terminaux d'alors étant qualifiés de « dumb », i.e. bête ! C'était un monde plus simple, avec des contraintes d'une autre nature. Avec

l'informatique distribuée, la complexité est plus grande, d'où la nécessité d'une gestion de configuration qui décrit l'ensemble des relations entre ces différents éléments.

Dans cette approche, l'application au sens ancien est un sous-système, *i.e.* un élément de quelque chose de plus vaste : un système d'information comme les ressources humaines, la production, la relation clients, etc. Le niveau système de systèmes regroupe tous les systèmes de l'entreprise qui interopèrent entre eux conformément aux objectifs stratégiques de l'entreprise.

On remarquera que la relation entre le logiciel qui est un « texte » exécutable et la plate-forme qui est une ressource capable d'exécuter le texte en question, se matérialise au niveau d'un équipement. Un même équipement peut héberger plusieurs applications qui ont entre elles des interactions privilégiées. C'est à ce niveau que l'on fera la distinction entre les différents environnements utilisés par les acteurs des processus d'ingénierie :

- L'environnement de développement pour les programmeurs, avec tous les outils nécessaires à la programmation.
- L'environnement d'intégration pour l'équipe en charge de l'intégration, avec les outils spécifiques de l'intégration, en particulier la gestion de configuration globale du système, les mesures de performance pour valider le comportement, etc.
- L'environnement d'exploitation, avec les outils d'administration, qui, *in fine*, permettra l'utilisation des applications par les usagers réels des organisations pour lesquelles elles ont été réalisées, conformément au contrat de service.

Pour maîtriser la complexité inhérente au système, il est impératif que l'arborescence correspondante soit une vraie hiérarchie. Toute dépendance latérale doit être exclue et remontée sur le niveau commun aux deux feuilles. Si trop d'entités remontent sur la racine de l'arborescence, cela signifie que les entités sont toutes dépendantes les unes des autres, et que, en conséquence, le système est mal hiérarchisé ; dans ce cas, les couches et/ou les serveurs n'ont pas de réalité véritable, ce qui rendra le processus d'intégration particulièrement coûteux et peu sûr.

Dans une architecture distribuée, notre application générique va engendrer un schéma de déploiement comme présenté sur la figure 6.9.

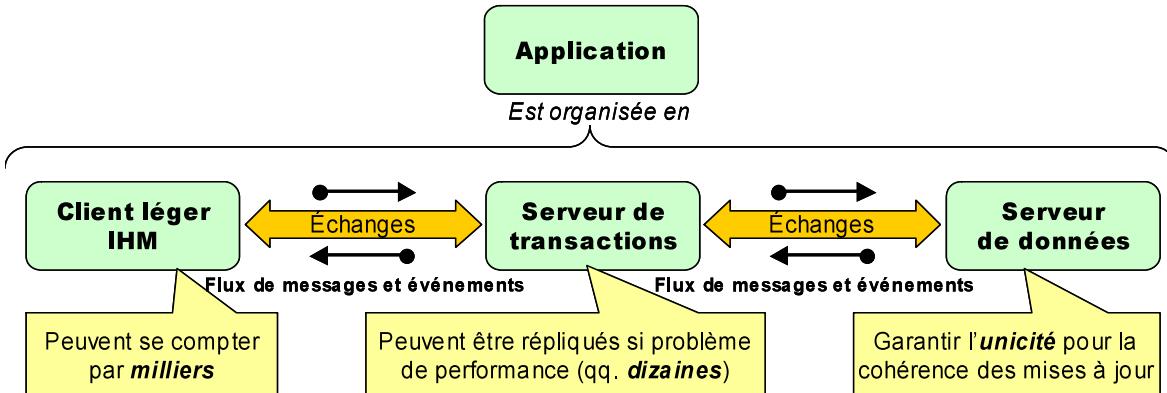


Fig. 6.9 Déploiement d'une application client-serveur

Le logiciel unique va être installé en tout ou partie sur un grand nombre de machines de configurations différentes, voire même avec des systèmes d'exploitation différents, ce qui va occasionner un problème de gestion de la configuration système complexe, surtout s'il est géré « à la main ». Il faut expliciter la relation d'installation entre le logiciel et les machines réceptrices du logiciel pour gérer les commandes de paramétrage. On peut représenter cette relation comme sur la figure 6.10.

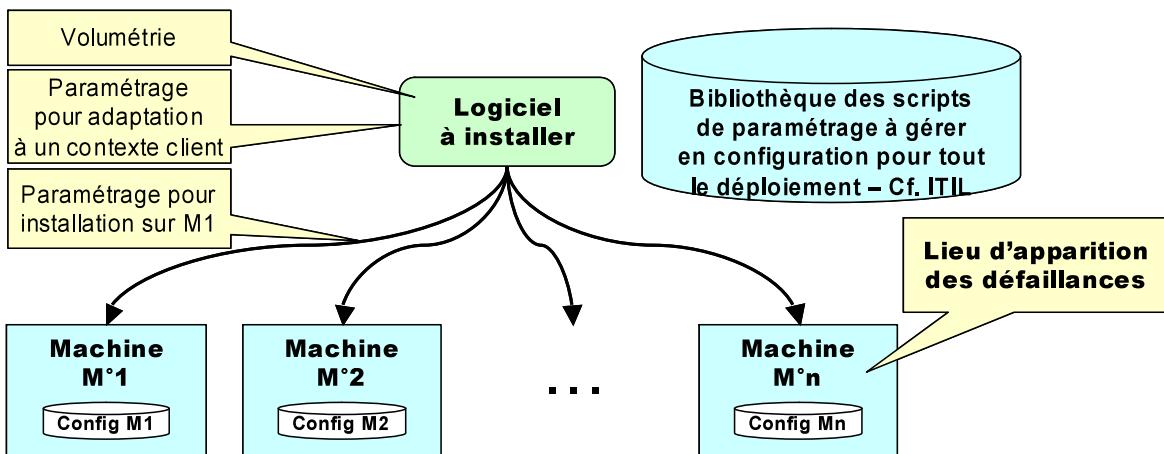


Fig. 6.10 Configuration du déploiement

Il y a autant de scripts d'installation que de types de machines à installer, étant entendu que chacune des machines peut avoir ses propres spécificités et héberger d'autres logiciels que celui qui est installé. Cette nomenclature est fondamentale pour valider les installations (cf. la démarche ITIL^[2]). Du point de vue du logiciel, s'il est correctement structuré en couches, l'installation dans une machine particulière devra faire apparaître les éléments logiciels présentés sur la figure 6.11.

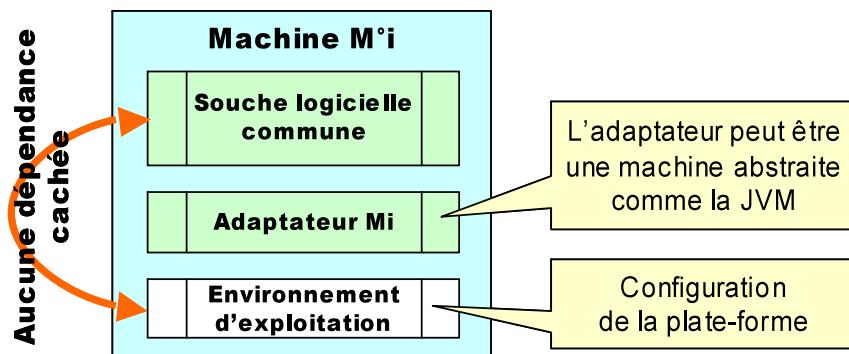


Fig. 6.11 Installation sur une machine particulière

À titre d'exemple, on peut donner des chiffres communiqués par Microsoft^[3] concernant Excel 4.0 dont la souche commune à Windows et à Macintosh totalise 528 287 lignes de code en C, avec 56 522 spécifiques Mac et 59 710 spécifiques Windows. Le SGBD Oracle est également bâti sur ce principe. Cette approche qui presuppose une bonne architecture, simplifie énormément le travail d'intégration d'un environnement à un autre, ou lors des évolutions de la plate-forme et/ou de l'application.

Pour des raisons liées à la complexité, il est recommandé de bien séparer la souche générale, de l'adaptateur à l'environnement d'exploitation, car les paramétrages à effectuer sont de nature très différente, logique pour la souche, physique pour l'adaptateur. Du point de vue gestion de configuration, il faudra distinguer le cas où la souche est un format binaire fabriqué dans un environnement d'intégration standardisé, du cas où la souche est recompilée sur la machine Mi, auquel cas l'environnement de développement de la machine Mi fait partie de la configuration.

On peut facilement imaginer que si toute cette mécanique n'est pas parfaitement gérée et automatisée, ce sera une source d'erreurs et d'incertitudes considérables qui se traduira par des temps de diagnostics importants en cas de défaillances et un mauvais rendement de l'effort associé.

6.3. Notion d'intégrat

Un intégrat logiciel élémentaire regroupe un ensemble de fonctions apparentées de plus ou moins grande taille résultant du travail de quelques programmeurs (2 à 4 pendant 6 à 9 mois, selon la complexité), typiquement 15 000 à 20 000 LS (ou 150 à 300 Points de Fonctions).

Du point de vue de l'intégration, un intégrat dans un projet a les attributs qualité présenté sur la figure 6.12.

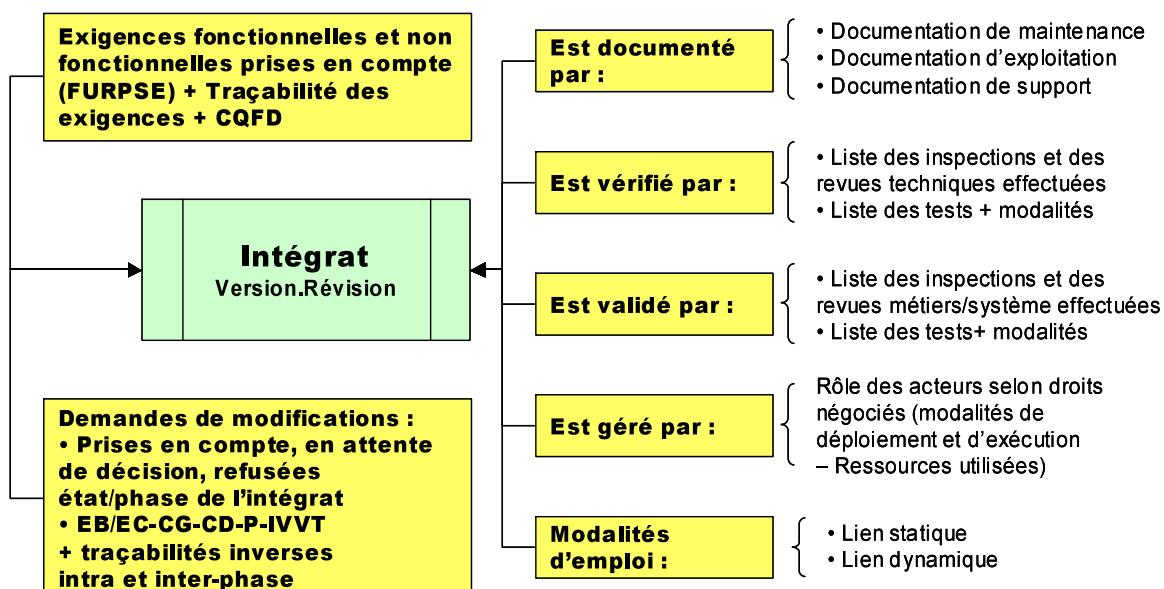


Fig. 6.12 Attributs indispensables d'un intégrat

Tout manquement à la qualité va perturber le travail de l'équipe d'intégration, car elle ne dispose pas de l'information indispensable à l'efficacité de son travail de développement des tests. Elle a le choix entre deux attitudes :

1. elle se débrouille seule, avec le risque de produire des tests de qualité médiocre, ce qui est globalement mauvais ;

- elle va chercher l'information manquante chez les acteurs qui n'ont pas fait leur travail, et dans ce cas c'est la productivité qui se dégrade et le rendement des équipes qui s'effondre ; le coût de non-qualité augmente.

Du point de vue fonctionnel, un intégrat doit saisir les critères de modularité^[4] définis par D. Parnas, précisés par B. Meyer avec les « contrats » d'interfaces et redécouverts par J. Sasseon avec les principes d'urbanisation qui ne sont qu'un autre nom pour l'architecture. Les règles principales, pour résumer, sont l'unicité du point d'entrée et du point de sortie, et la rétention de l'information spécifique du module, *i.e.* l'absence de contexte partagé entre différents modules (*cf.* principe de modularité « *stateless* »).

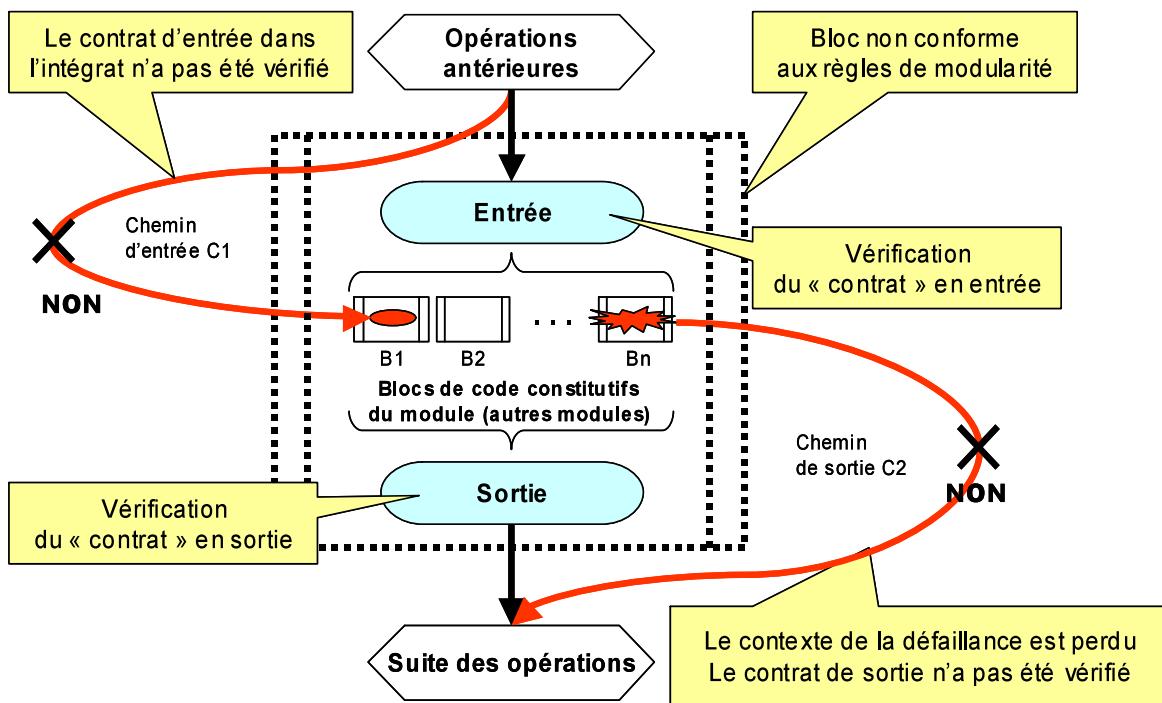


Fig. 6.13 Aspects modulaires des intégrats

L'unicité des points d'entrée et de sortie, et les contrats associés, déterminent des points d'observation qui vont permettre le travail de vérification et de validation. Intégrer un module, c'est s'assurer que les contrats qui le lient à son environnement sont effectivement respectés.

Dans la réalité des vrais projets, il faut comme toujours nuancer la définition en particulier pour ce qui concerne le contexte. Il peut être difficile, par exemple pour des raisons de performance, d'interdire la présence d'un contexte, bien qu'en théorie cela soit toujours possible. Interdire le contexte revient à dire que l'information correspondante transite par les points d'entrée et de sortie *via* des messages et/ou des recopies de données dans la mémoire sous contrôle de l'intégrat. Dans le cas de réseaux, cela peut saturer la bande passante et créer des problèmes de qualité de service (temps de réponse, perte d'information, etc.).

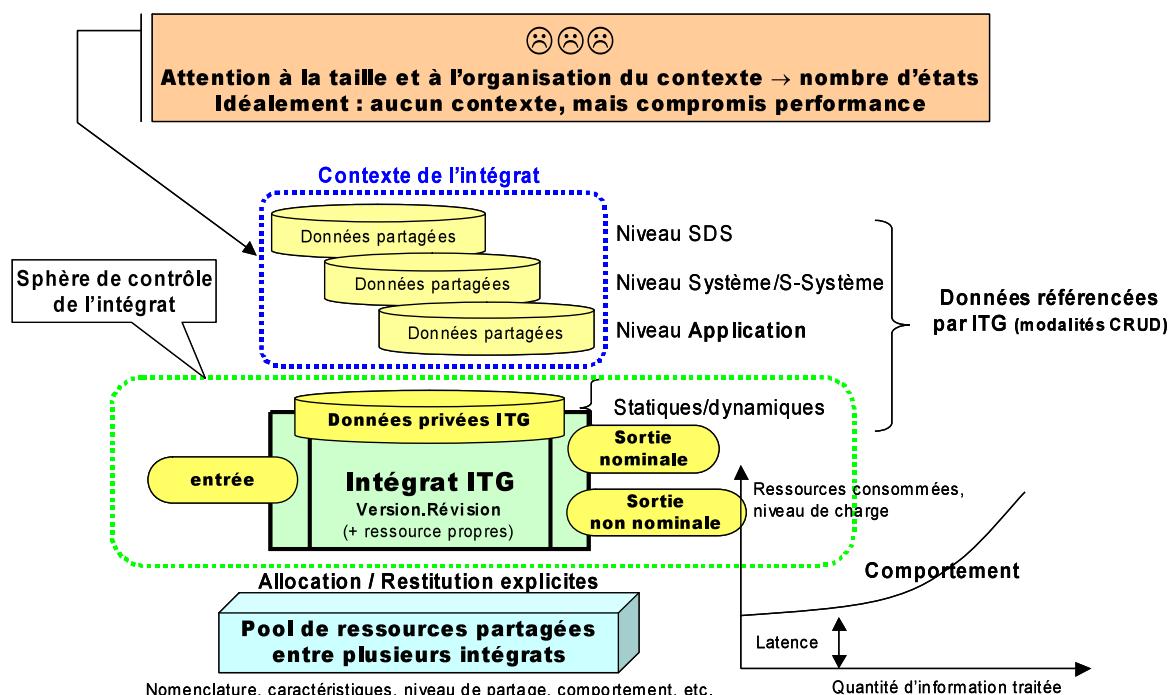


Fig. 6.14 Aspects fonctionnels des intégrats

Dans la pratique, l'intégrat aura l'aspect reproduit sur la figure 6.14. Dans le schéma, nous avons fait apparaître :

- Les données de contexte correspondant aux différents niveaux d'intégration rencontrés couramment dans la pratique, par exemple pour effectuer des paramétrages.
- Les ressources utilisées par l'intégrat pour expliciter les règles d'allocation et de restitution qui font partie du contrat.

- La sortie non nominale correspondant à une défaillance de l'intégrat qui va nécessiter un déroutement vers les mécanismes d'administration qui gèrent les reprises.
- Le comportement prévisible de l'intégrat lorsque le niveau de charge augmente.

Tous ces aspects font partie du travail de vérification et de validation à faire en intégration.

6.4. Difficultés et risques de l'intégration

La première difficulté de l'intégration est qu'elle est tributaire de la qualité des activités et des processus amont. Elle voit le système comme un ensemble de « Boîtes noires » appelés intégrats — ce sont des agrégats d'intégration — qui interagissent entre eux *via* leurs interfaces ou leur contexte commun. À partir des composants élémentaires du système résultat du travail des programmeurs que nous appellerons intégrats de rang 0, l'intégration construit des intégrats de rang 1, qui eux-mêmes permettent de construire des intégrats de rang 2, et ce jusqu'à l'intégrat final qui est LE système intégré. À chaque étape, la complexité de l'assemblage doit être maîtrisée pour que le processus d'intégration ne diverge pas.

Une deuxième difficulté vient de la précision des diagnostics formulés par l'équipe d'intégration. Lorsqu'une anomalie apparaît dans l'exécution d'un test, l'équipe d'intégration doit recueillir le maximum d'information sur le contexte de l'anomalie, *i.e.* des faits techniques (FT) irréfutables et reproductibles, et envoyer le rapport d'anomalie (RA) au bon acteur du côté des processus d'ingénierie. Dans un projet de taille significative, plusieurs dizaines de personnes peuvent à un moment ou à un autre avoir été des contributeurs du projet, y compris dans des sociétés sous-traitantes ; d'où l'importance des attributs de la figure 6.12. Si l'architecture n'a pas été conçue testable, la résolution du problème dont l'anomalie est le révélateur peut s'avérer coûteuse. Toute erreur d'aiguillage sera consommatrice d'effort et de délai, avec le risque additionnel de perdre le rapport d'anomalie (RA) si celui-ci n'est pas géré en configuration.

Une troisième difficulté vient de la façon d'ordonnancer l'exécution des tests, de façon à ce que la stratégie de test soit la plus efficace possible en terme CQFD. L'architecture définit une logique d'intégration, comme le montrent les schémas des figures 6.5, 6.6 et 6.7 de notre architecture générique.

Ces schémas suggèrent deux stratégies d'intégration contrastées S1 et S2.

S1 consiste à développer et à intégrer indépendamment chacun des trois serveurs sur leur plate-forme respective. Une telle approche correspond à une logique projet, simple en apparence, car une fois les interfaces négociées, chacun des maîtres d'œuvre est libre chez soi. Ne reste à résoudre que l'intégration des trois serveurs. On peut qualifier une telle stratégie de verticale ou de silo.

S2 est exactement l'inverse. Elle est fondée sur le fait expérimental que les problèmes les plus difficiles sont précisément ceux de l'interfaçage des trois serveurs et de l'organe de communication qui leur permet d'interopérer. Une des règles de base de l'ingénierie stipule qu'il faut commencer par les problèmes les plus difficiles. Dans notre cas, il faut commencer par la pièce, en fait trois pièces élémentaires, permettant aux serveurs d'échanger des messages. Cette stratégie est l'intégration continue, préconisée par les méthodes agiles et l'approche TDD – « *Test Driven Development* ».

Nous allons détailler ces deux stratégies.

6.4.1. Intégration verticale

On peut représenter les étapes d'intégration de la stratégie S1 comme suit.

Étape E1

Chacun des serveurs est pré-intégré dans un environnement simulé qui matérialise son contrat d'interfaces avec les autres serveurs.

- **Ce qui donne, pour l'IHM :** le serveur « bouchon » donne des réponses conventionnelles à partir des informations fournies par les fichiers de données et les paramètres, de façon à pouvoir tester des scénarios complets. Il est essentiel, pour les tests de non-régression de ce serveur

de disposer de scénarios simulés complets qui permettent d'éviter la présence d'un opérateur humain pour les rejeux.



Fig. 6.15 E1-1 – Environnement d'intégration du serveur client léger

- **Pour le serveur de transactions :** l'environnement de ce serveur nécessite deux pièces additionnelles : un pilote et un bouchon.

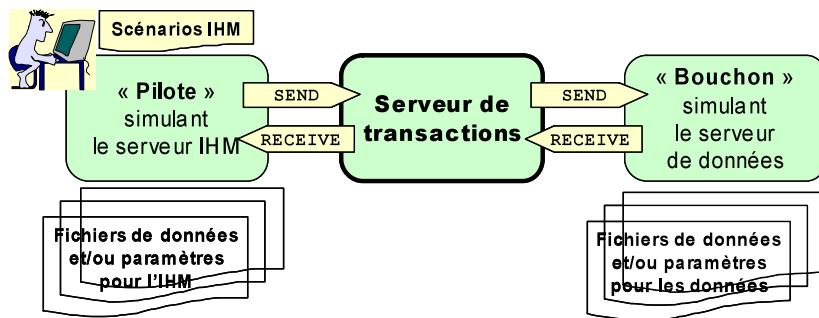


Fig. 6.16 E1-2 – Environnement d'intégration du serveur de transactions

Le « pilote » simule les interactions du serveur IHM. Les scénarios IHM et les fichiers de données et paramètres du pilote n'ont aucune raison d'être les mêmes que pour le client léger car la logique de test est différente. Les scénarios IHM doivent permettre de valider les enchaînements de transactions conformes aux fonctionnalités spécifiées. Les fichiers de données et/ou paramètres du « bouchon » permettent de construire des réponses conventionnelles mais, sémantiquement parlant, représentatives de véritables interrogations (y compris les erreurs imputables à la qualité des données pour la robustesse).

- **Pour le serveur de données :** l'environnement de ce serveur nécessite un pilote et des bases de données d'essais orientées intégration (interfaces, taille, performance, robustesse, etc.).

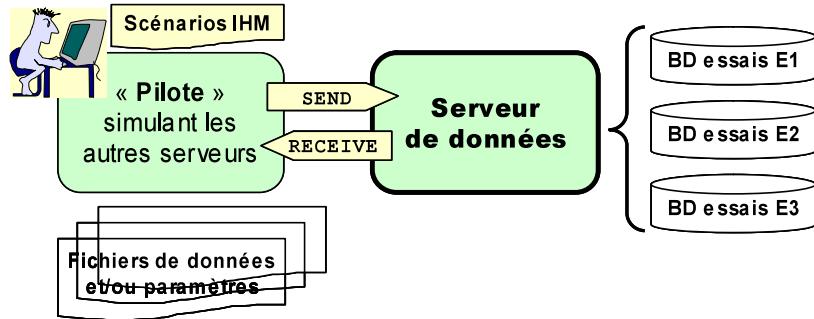


Fig. 6.17 E1-3 – Environnement d'intégration du serveur de données

Le pilote simule cette fois la fusion {IHM + Transactions}, avec des *scénarios ad hoc* orientés gestion des données (logique CRUD – Create, Retrieve, Update, Delete), avec plusieurs bases de données d'essais représentatives de la réalité. Les scénarios IHM doivent permettre de valider tout ce qui concerne les données, les sous-schémas et/ou les vues partielles des données, les enregistrements virtuels résultant de filtrages et/ou de jointures, les conflits d'accès, les performances via les « caches » de données, la robustesse avec la présence volontaire de données erronées et/ou incohérentes dans les BD d'essais, etc.

Étape E2

On valide la combinatoire autorisée par les mécanismes d'échanges. Les combinaisons autorisées sont au nombre de trois, soit :

- **E2-1 – Combinaisons deux à deux** : {Client léger + Serveur de transactions} ; {Serveur de transactions + serveur de données} ; avec, pour chacune des combinaisons, les scénarios et fichiers de données/paramètres *ad hoc*, différents de ceux de l'étape E1. La combinaison {Client léger + Serveur de données} n'est pas autorisée quoique possible (c'est un cas d'erreur) sauf à bouchonner le serveur de transactions si le responsable de l'intégration pense qu'il peut y avoir des dépendances cachées entre ces deux serveurs.
- **E2-2 – Combinaison trois à trois** : une seule combinaison qui correspond au système complet réunissant les trois serveurs. Ce qui donne un environnement comme suit :

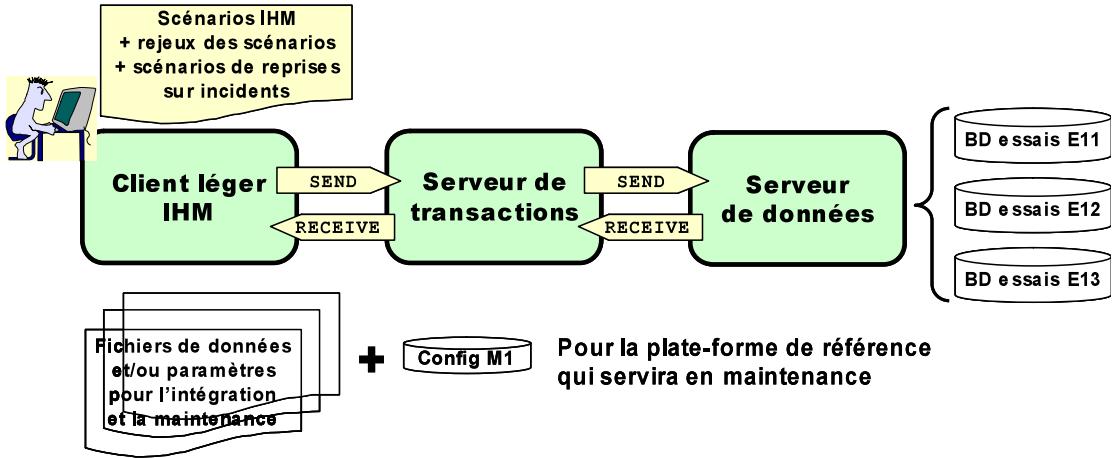


Fig. 6.18 Environnement d'intégration de l'application

Les scénarios de reprises sur incidents sont au minimum ceux de la plate-forme de référence qui servira à l'intégration et à la maintenance. Les bases de données d'essais doivent être représentatives de la réalité pour les tests de charge, la robustesse, les reprises sur incidents, les performances, etc.

Étape E3

On valide simultanément les trois serveurs dans la logique de la recette finale. Idéalement, les bases de données d'essais doivent être des clones des bases de données du système réel. Les différentes configurations doivent être conformes aux configurations rencontrées en exploitation. À ce stade de l'intégration, certains tests peuvent/doivent être effectués sur les plates-formes d'exploitation si le risque est considéré comme acceptable du point de vue des usagers connectés aux systèmes. Il faut valider les installations avec toutes les configurations de plates-formes autorisées.

En termes de coûts et de maîtrise du contrat de service (SLA) on peut mesurer l'intérêt de rationaliser et de simplifier au maximum les plates-formes d'exploitation, car le moindre laisser-aller se traduira par une combinatoire infernale, et un volume de tests à effectuer incompatible avec le budget du projet.

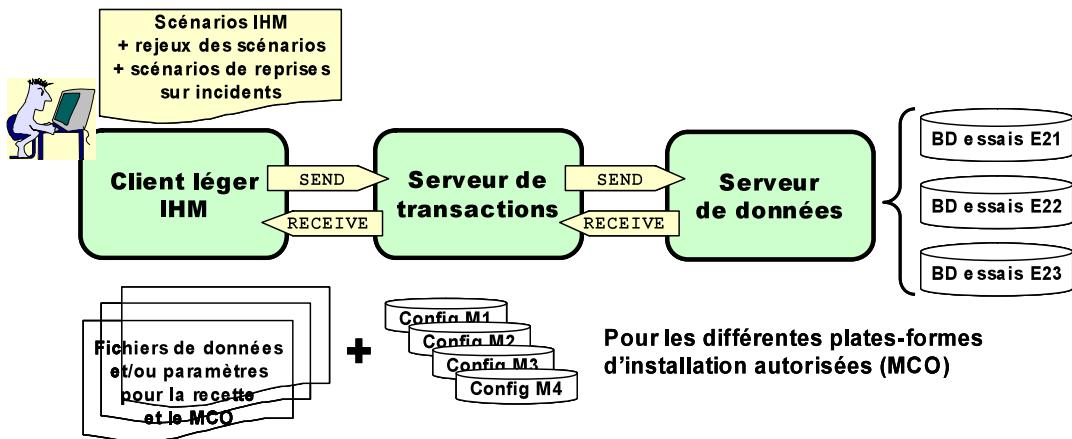


Fig. 6.19 Environnement d'intégration pour la recette finale

Bilan de la stratégie d'intégration verticale

C'est une stratégie prudente, qu'une équipe moyennement expérimentée pourra mettre en œuvre, éventuellement onéreuse en termes de coûts et de délais, mais qui produira un résultat de bonne qualité, du moins avec un petit nombre de serveurs. Le volume de programmation nécessaire pour le développement des pilotes et des bouchons peut devenir important, ce qui est un vrai problème. Il faut que ce code ait un niveau de fiabilité identique à celui des serveurs car il sert à valider les mécanismes d'échanges et l'interopérabilité.

Les risques de cette stratégie résultent de la validation tardive des mécanismes d'échanges et de l'interopérabilité qui ne sera acquise qu'à l'étape E2-2, en fin de projet. Si des incohérences apparaissent dans les interfaces à ce stade, elles risquent d'engendrer des modifications importantes dans les serveurs, ce qui nous ramène à l'étape E1. Si les environnements et les tests n'ont pas été gérés de façon rigoureuse, en gestion de configuration, cela provoquera inéluctablement des retards en délai et des travaux de reprises coûteux. À l'extrême, le volume de travail à fournir peut déstabiliser le projet et conduire à l'échec.

Toutefois, le vrai risque de cette stratégie est lié au nombre de serveurs impliqués dans les échanges et l'interopérabilité. Les trois serveurs de notre exemple sont un cas d'école, car dans la réalité il peut y en avoir beaucoup

plus. Avec N serveurs, le nombre de combinaisons à valider, et conséquemment le nombre de pilotes et/ou de bouchons à développer est de l'ordre de l'ensemble des parties de N , car il faut considérer les configurations 2 à 2, 3 à 3, ..., N à N , soit au total

The image displays a large, bold, black mathematical expression. It consists of a '2' positioned to the left of an '^' symbol, which is positioned above an 'N'. This represents the formula for the number of combinations, illustrating exponential growth.

qui est une exponentielle. La complexité induite implique un nombre de tests incompatible avec le budget du projet, ce qui se traduira par de nombreuses combinaisons non testées, donc une baisse progressive de la qualité de service en fonction du nombre de serveurs fonctionnellement interconnectés.

Si de plus les interactions sont contextuelles, résultant de violation de la règle de modularité dite « stateless », c'est-à-dire que le serveur a mémorisé des états qui résultent des interactions antérieures, il faut alors tenir compte de la taille des chroniques de ces interactions et des permutations possibles, ce qui amène une combinatoire qui de l'ordre d'une factorielle, soit :

$$n! \cong \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

La combinatoire devient une exponentielle d'exponentielle, donc sans espoir du point de vue des tests. Cela montre l'importance de l'architecture et du respect strict des règles de modularité : discipline de programmation et politique d'assurance qualité rigoureuse, comme exigé par le CMM dès le niveau 2 de l'échelle de maturité.

6.4.2. Intégration continue

On peut représenter les étapes d'intégration de la stratégie S2 comme suit.

Étape E1 : validation des interfaces entre les serveurs et des mécanismes d'échanges

Cette stratégie met la priorité sur les mécanismes d'échanges et l'interopérabilité entre les composants applicatifs constitutifs de l'application et/ou du système réalisé. Ces mécanismes constituent le « bus » logiciel du système, par analogie avec les mécanismes hardware. Ce bus peut lui-même intégrer des progiciels EAI (*Enterprise Application Integration*) ou ESB (*Enterprise Software Bus*) qui sont des boîtes à outils neutres facilitant son développement^[5].

Les « squelettes » sont la traduction littérale de la conception détaillée de chacun des serveurs. Les intégrats interfaces matérialisent les contrats négociés entre les différents serveurs, c'est-à-dire entre les différents responsables et architectes des projets, et le responsable de l'architecture d'ensemble qui doit être également le responsable de l'intégration système.

Le code fonctionnel de chacun des serveurs est réduit au minimum indispensable pour les tests de l'interopérabilité des serveurs, tout le reste est

bouchonné. Les bouchons font partie de l'environnement d'intégration et gérés comme tels.

Le but est de construire un environnement minimum, partagé entre les projets, qui va permettre de valider exhaustivement les interfaces entre les serveurs et l'interopérabilité.

La première configuration d'intégration se présente comme sur la figure 6.20.

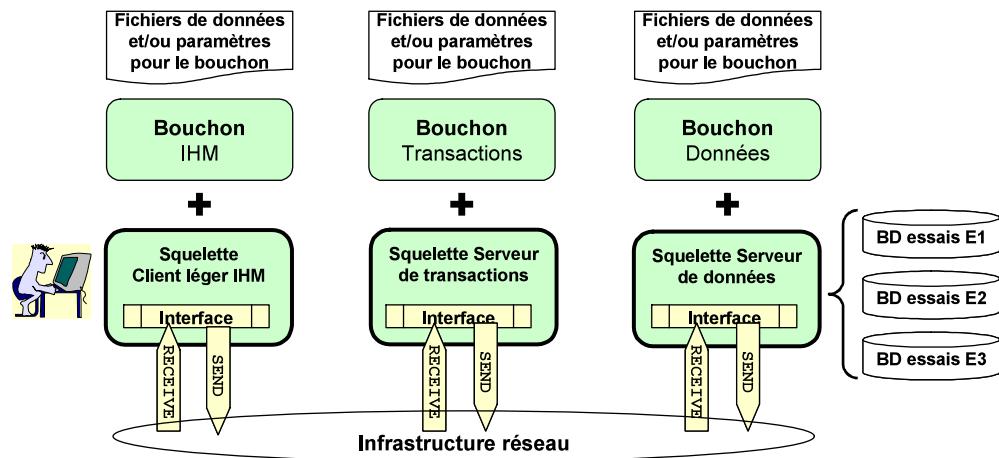


Fig. 6.20 Environnement d'intégration continue

La spécification de cet environnement presuppose un travail fin de synchronisation entre les équipes projets de chacun des serveurs et l'équipe d'intégration, ce qui nécessite une maturité de niveau 4 ou 5 dans l'échelle CMM. C'est du travail d'experts qui ne peut être effectué que par des chefs de projets et/ou architectes dont l'expérience est confirmée par des réalisations antérieures dont la qualité est prouvée (N.B. : c'est le B.A.-BA des méthodes agiles !).

À ce stade de l'intégration, les interfaces et les mécanismes d'échanges pour l'interopérabilité sont validés définitivement. Avec un minimum de code développé, on peut dès cette première étape effectuer des tests de bout en bout et valider les caractéristiques non fonctionnelles comme les performances, les services communs, les mécanismes de reprises qui sont toujours déterminantes pour l'architecture.

La grande différence avec la stratégie silo est que d'entrée de jeu on focalise les acteurs sur tout ce qui est transverse dans le projet. Toutes les incertitudes

sont levées, et la suite des travaux — développement et intégration — va se dérouler sur une architecture stabilisée et validée, ce qui va minimiser le nombre de pilotes et de bouchons.

Étapes E2, E3, ... ex. : croissance progressive et contrôlée de l'application

L’architecture étant stabilisée, les responsables projets vont livrer à l’intégration des blocs de code cohérents, de façon à toujours privilégier les aspects transverses, comme sur la figure 6.21.

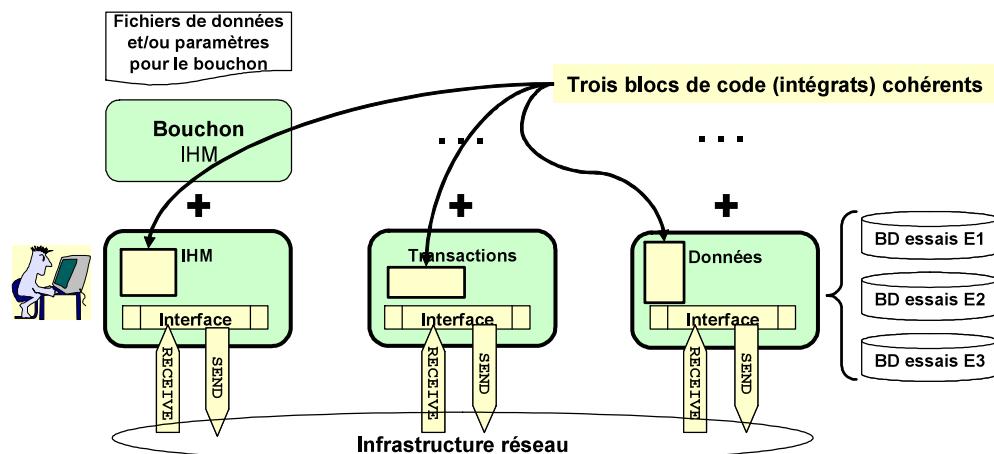


Fig. 6.21 Modèle de croissance en intégration continue

La cohérence des intégrats est de nature sémantique, ce qui signifie que la logique de livraison est fondée sur la logique d’intégration de façon à minimiser la taille des bouchons.

Sur la base d’une logique CRUD de gestion des données, on va enrichir (c’est juste un exemple de stratégie) les serveurs de façon à traiter tout ce qui concerne la création (create) d’entités, puis la suppression (delete) ; à ce stade on peut faire des premiers essais de fiabilité et de robustesse concernant les aspects CD de la gestion des données.

Dans un second temps, on traite les aspects consultation/sélection (retrieve), puis la mise à jour (update) ; ce qui peut se faire en plusieurs étapes si le système est de grande taille, le tout entrecoupé d’essais de fiabilité et de robustesse.

Les bases de données d'essais sont progressivement enrichies jusqu'au stade où elles sont représentatives de la réalité.

Dans la phase UD, on dispose d'un sous-ensemble de système sémantiquement cohérent, ce qui va permettre de faire les premiers essais de recette dans les configurations à installer. Là encore le but est d'anticiper les problèmes et de lever les incertitudes le plus tôt possible dans le déroulement du projet.

Avec une mécanique d'intégration bien rodée (voir ci-dessous), on peut envisager un rythme de livraisons hebdomadaires, voir quotidiennes comme proposé dans les méthodes agiles.

Bilan de la stratégie d'intégration continue

C'est la stratégie optimale, pour autant que les équipes aient la maturité et l'expérience suffisantes. Le processus d'intégration est étroitement couplé aux processus de développement des trois serveurs, ce qui presuppose une architecture effectivement stabilisée et partagée entre les équipes de développement et d'intégration ; dans notre cas d'école, il y a quatre équipes.

Le vrai mérite de cette stratégie est qu'elle rend linéaire le coût du processus d'intégration, en évitant les combinaisons 2 à 2, 3 à 3, etc. Le coût de cette linéarité est le coût de la qualité du travail architectural, en particulier la spécification des interfaces, car ce sont les interfaces qui vont absorber la combinatoire^[6]. La maîtrise de cette mécanique de précision est un bon indicateur de maturité de l'organisation et des équipes d'ingénierie.

Les risques de cette stratégie sont de deux types. Un premier type de risques vient des interactions entre le maître d'ouvrage, le(s) maître(s) d'œuvre, l'intégrateur et le client. Un second type vient du niveau de maturité des équipes exigé pour la mise en œuvre de cette stratégie.

Pour que cela ait une chance de fonctionner, il est indispensable de disposer d'architectes et de chefs de projets expérimentés car tout repose sur la qualité de l'architecture, la stabilité des interfaces et la capacité des équipes projets à travailler en parallèles. On parle, dans ce cas, de management de projet centré sur l'architecture (*Architecture centric project management*, comme

présupposé par les méthodes agiles). Si ce n'est pas le cas, le chaos est inéluctable, et mieux vaut se rabattre sur la stratégie silo.

En termes de diagramme de GANTT, on peut représenter qualitativement chacune des stratégies comme sur la figure 6.22.

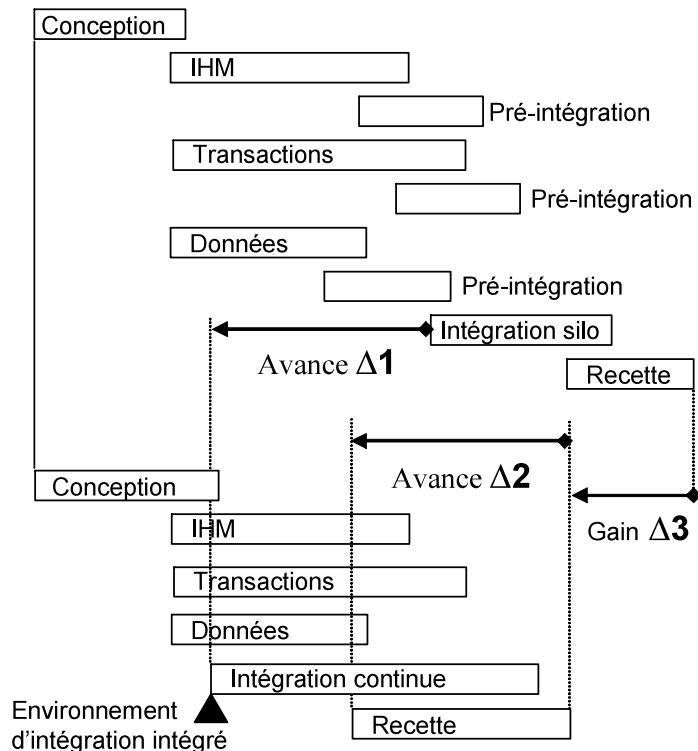


Fig. 6.22 GANTT des stratégies d'intégration silo et continue

On raisonne à coûts constants. Le gain porte sur les délais, ce qui est un avantage concurrentiel certain.

On remarquera que la conception inclut la spécification de l'environnement d'intégration, selon le principe des architectures testables, *i.e.* le « *design to test* » des ingénieurs hardware. Cet environnement se substitue aux pré-intégrations. Ce faisant, l'intégration et la recette démarrent beaucoup plus tôt dans le processus S2, avec des avances respectives $\Delta 1$ et $\Delta 2$. L'avantage concurrentiel se matérialise par le gain en délai $\Delta 3$.

6.5. Mécanique du processus d'intégration

Dans la réalité des projets, l'équipe d'intégration voit l'activité de conception et codage comme un flux d'intégrats qui lui arrivent de façon plus ou moins régulière et qu'elle va devoir assembler pour faire les tests dont elle à la charge. L'ordre d'arrivée des intégrats n'est que très rarement l'ordre logique de l'intégration. Pour passer les tests, l'équipe d'intégration peut être amenée à simuler les intégrats manquant par des intégrats « bouchonnés » qui, via les interfaces, vont fournir des réponses conventionnelles lors de l'exécution des tests. Ceci compliquera d'autant l'exécution des tests. Ce mécanisme vaut également pour la qualification et l'acceptation finale, comme le montre la figure 6.23.

Si l'intégrat n'est pas conforme au contrat passé entre le développement et l'intégration, l'intégrat est rejeté, ce qui fait perdre du temps de part et d'autre. Toute anomalie détectée en cours de test ou de revue d'intégration fait l'objet d'un rapport d'anomalie (RA) complété par les faits techniques (FT) justifiant l'anomalie. La précision de l'information remontée via les RA et FT est primordiale pour la performance des processus de développement et d'intégration.

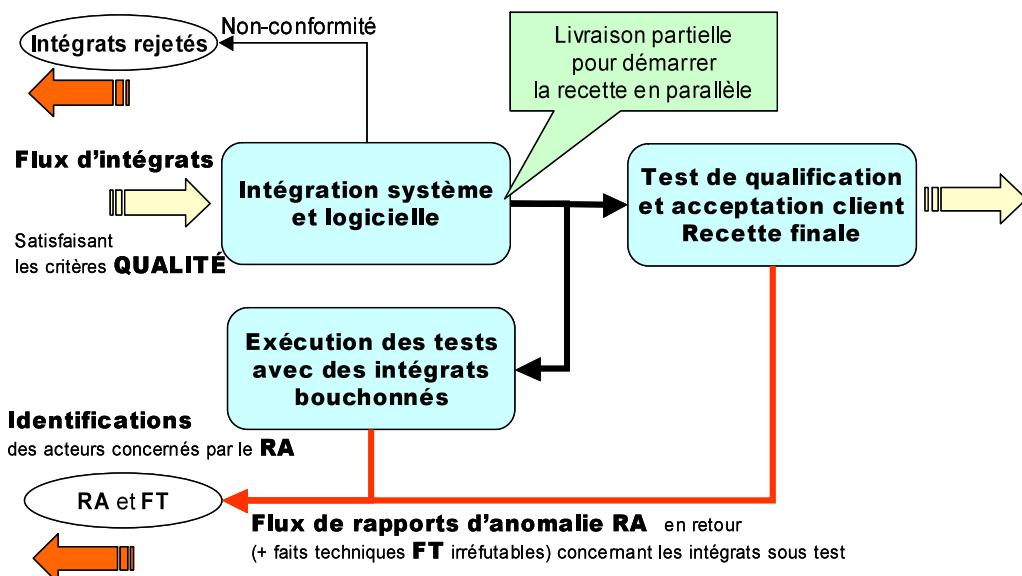


Fig. 6.23 Procédé de construction mis en œuvre par l'intégration.

Au niveau de l'intégration, la stratégie de test est primordiale. L'ordonnancement optimum des intégrats est celui qui minimise le CQFD de

l'intégration : c'est une contrainte à prendre sérieusement en compte dans les activités amont, et plus particulièrement dans la conception du logiciel. Rappelons que, selon le modèle d'estimation COCOMO, l'intégration consomme de 20 à 35 % de l'effort, selon la complexité du projet.

Si les intégrats arrivent en désordre, l'équipe d'intégration a recours à deux attitudes possibles :

- Elle attend que le sous-ensemble d'intégrats à tester soit complet, d'où une quasi-certitude de perte de ressource et chute du rendement (*cf.* la stratégie intégration silo, ci-dessus).
- Elle développe des programmes permettant de « bouchonner » les intégrats manquant afin de fabriquer des intégrats exécutables pour effectuer des tests partiels (*cf.* la stratégie intégration continue, avec l'environnement intégré).
- Elle devient donc une équipe de développement d'un atelier d'intégration, sous le contrôle de l'architecte système, avec tous les risques que cela comporte car ce n'est pas sa finalité initiale. C'est une météo-activité à définir dans le processus d'ingénierie « *Process Implementation* » comme indiqué dans la figure 6.2 présentée en début de chapitre. Par ailleurs le code développé ne fera pas partie de la livraison négociée, mais il faudra quand même le gérer pendant toute ou partie de la durée du processus d'intégration, voire de le conserver pour les versions ultérieures du système et la maintenance. Pour toutes ces raisons, il est impératif de minimiser le volume de ce code, ou alors de le faire prendre en compte par les équipes de conception-programmation au titre de la caractéristique qualité de testabilité, si elles ont la maturité suffisante.

Enfin, une dernière difficulté vient des activités et processus aval. Il est rare que les contraintes du MCO soient prises en compte complètement dès le départ du projet, pour la simple raison que les acteurs correspondant ne sont pas encore identifiés. Dans la mesure où c'est le processus d'intégration qui livre les résultats du projet au MCO, les acteurs MCO et leurs contraintes peuvent être identifiés relativement tardivement. Les plates-formes qui seront effectivement déployées peuvent également n'être disponibles que très tardivement.

Tout concourt donc à ce qu'un dernier train d'exigences apparaissent lors de la transition Intégration → Qualification/Acceptation du système, et bien souvent sous la forme de rapports d'anomalies (RA) concernant les caractéristiques qualité non fonctionnelles. Il est par exemple très fréquent que des problèmes de performance se manifestent sur les plates-formes d'exploitation alors que rien n'avait été anticipé sur les plates-formes de développement et d'intégration. La prise en compte des RA correspondants peut s'avérer préjudiciable à l'équilibre CQFD du projet car ce type de situation est souvent révélateur d'un grave défaut d'architecture qui est à l'origine des mauvaises performances.

Pour pouvoir solutionner des problèmes d'architecture tardifs comme les performances ou la maintenabilité, la conception initiale de l'architecture doit être « ouverte » au maximum, ce qui est plus facile à dire qu'à faire ; par exemple l'architecte peut anticiper la présence d'un cache, sans le développer, mais en prévoyant les interfaces qui conviennent. Seul un architecte expérimenté sera à même d'anticiper les risques et d'installer les interfaces permettant de réagir si nécessaire. Si ce n'est pas prévu, il faudra attendre une seconde version du système, accompagné d'une refonte substantielle de l'architecture initiale. Toutes ces considérations sont au cœur des réflexions de l'OMG sur l'approche MDA (*Model Driven Architecture*) et le MDE (*Model Driven Engineering*). Pour rester dans le domaine des performances, si la conception est suffisamment précise et rigoureuse, il est possible de développer un modèle prédictif de performance permettant de dimensionner les ressources indispensables pour garantir la qualité du service rendu.

6.6. Dynamique du processus d'intégration

Avec la mise en œuvre des architectures client-serveur, le processus d'intégration, déjà complexe à l'époque des grands systèmes centraux, a vu sa complexité encore augmenter. C'est aujourd'hui l'un des processus les plus difficiles à maîtriser car, en plus de la complexité inhérente au système, il est le révélateur de toutes les contradictions latentes, de tous les non-dits, de tous les conflits dont le projet a pu être le siège et qui n'ont pas été correctement arbitrés, ajoutant ainsi leurs lots de complication et

d'incertitude. L'intégration est la minute de vérité des concepteurs, où les limitations des acteurs se manifestent aux yeux de tous, ce qui rend les situations humaines encore plus difficiles à gérer.

Le processus d'intégration, que l'on peut scinder en « *software integration* » et « *system integration* », rétroagit d'une part sur tous les processus amont qu'il va alimenter par un flux de faits techniques et de rapports d'anomalies qu'il a mis en évidence, et d'autre part sur les processus aval en direction du MCO qui doivent être anticipés (*cf.* les activités *software installation* et « *software acceptance support* » dans la figure 6.2. qui donne le détail de l'ingénierie). Tout cet ensemble met en œuvre les activités de soutien explicitées dans les processus de la figure 6.1 ; parmi ces activités, la gestion de configuration joue un rôle central.

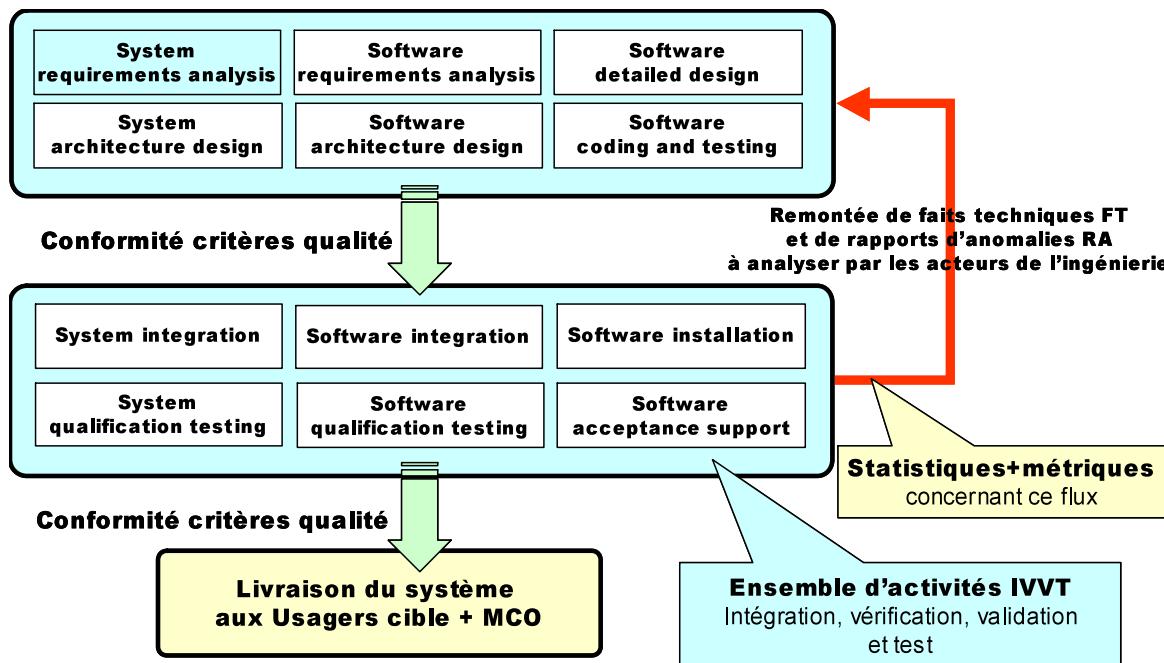


Fig. 6.24 Rétroactions et régulation du processus d'intégration

Il est habituel de regrouper l'ensemble de ces activités sous l'appellation IVVT (Intégration, Validation, Vérification et Test) ou IVVQ (pour Qualification, plus large que le Test). De fait, l'intégration est fondamentalement une activité de test et de planification de test, au sens large du terme. Le déroulement de l'intégration va générer son lot de problèmes

qu'il faudra résoudre jusqu'à satisfaction des critères qualité pour la globalité du système intégré.

Pour toutes les activités d'intégration, la logistique des tests est un facteur dimensionnant de l'efficacité du processus. Examinons les grandes lignes du processus de mise en œuvre des tests qui vaut tout aussi bien pour la vérification et/ou la validation.

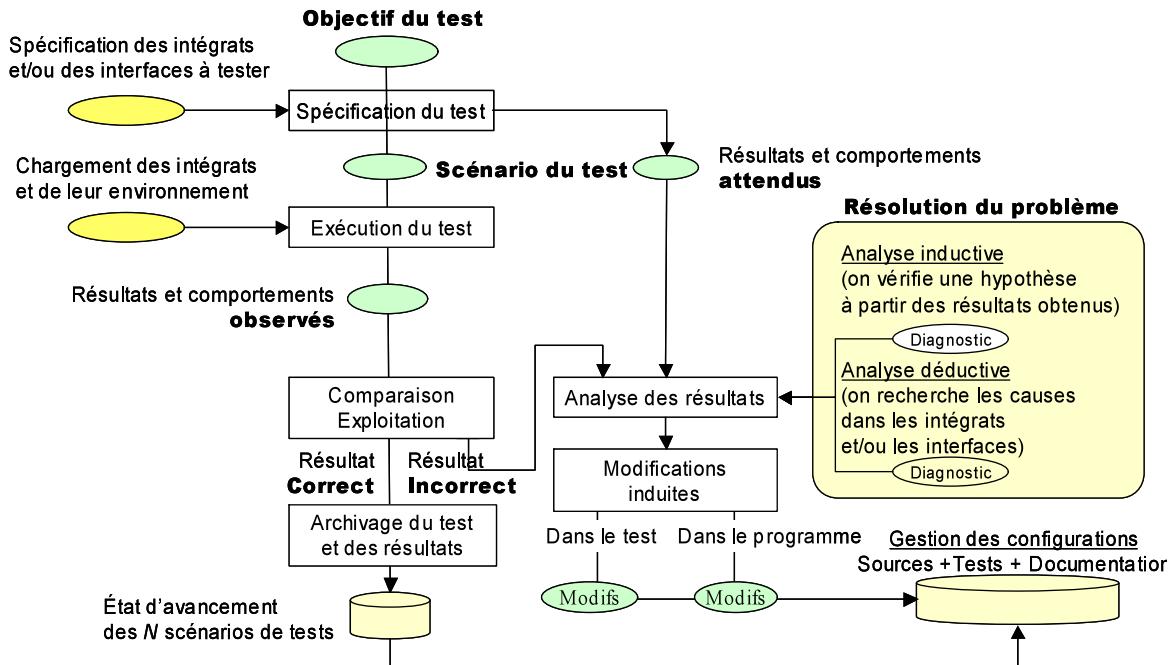


Fig. 6.25 Structure du processus de test

La figure 6.25 montre que si la documentation de l'intégrat est insuffisante, beaucoup de temps et d'efforts seront perdus pour rassembler l'information nécessaire aux tests. La qualité des tests sera médiocre. Si les moyens d'analyse (traces, vérifications des contrats de programmation aux interfaces par des assertions, tests en ligne pour l'autodiagnostic) sont inexistant ou insuffisants, les temps d'analyse et de diagnostic seront très longs.

Le pavé Résolution du problème est une illustration de la mise en œuvre du processus « *Problem Resolution* » des processus de support mentionnés dans la figure 6.1.

Enfin, le schéma montre l'importance de la gestion de configuration pour gérer l'ensemble des informations nécessaires à un processus d'intégration

performant.

Compte tenu du caractère répétitif des tests, l'automatisation du processus d'intégration doit être envisagée dès que le projet dépasse une certaine taille et que les tests sont nombreux. Nous avons montré dans des ouvrages antérieurs^[7] que l'effort consacré aux tests croît comme le carré du nombre de tests, ce qui en soit est une justification pour automatiser tout ou partie du processus (voir règle 6, § 6.8.6).

6.6.1. Rappel sur la mécanique du procédé de construction

Pour bien comprendre la dynamique de l'intégration, il est essentiel d'analyser les mécanismes mis en œuvre par le procédé de construction des codes sources et de fabrication des binaires tel qu'on les trouve sur les plates-formes de développement et dans les environnements intégrés de programmation. C'est cet ensemble de mécanismes qui produit les intégrats de rang 0 ; ces intégrats peuvent résulter de l'assemblage, au sens informatique du terme, de plusieurs unités de compilation pour former un tout sémantiquement cohérent. La taille d'un intégrat de rang 0 est purement conventionnelle ; elle peut varier de quelques KLS (kilo lignes source) à quelques dizaines de KLS, selon la nature des projets.

Les grandes lignes du procédé de construction pratiqué en intégration peuvent être schématisées comme sur la figure 6.26.

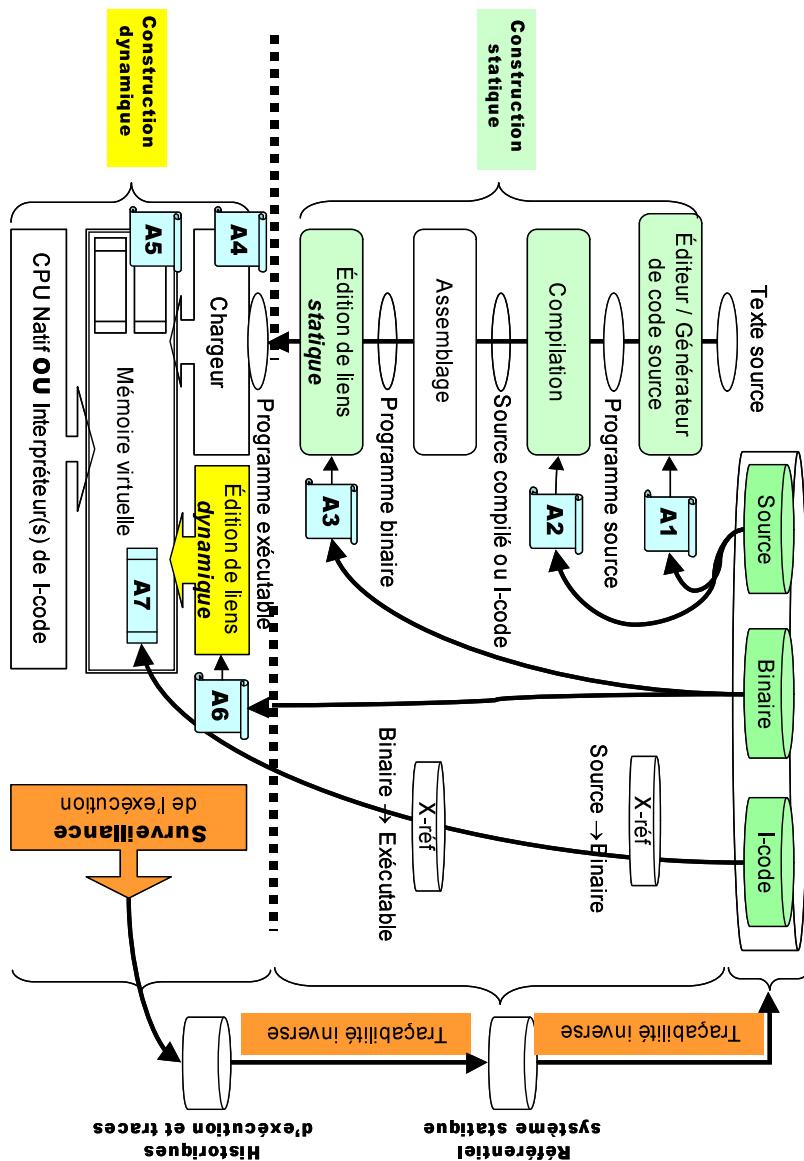


Fig. 6.26 Procédé de construction et d'intégration d'un système

- A1 et A2 sont des adaptations par des textes source que l'on peut incorporer au programme soit avant la compilation à l'aide d'un éditeur de texte, un pré-processeur comme dans C/C++ ou un générateur de programme, soit pendant la compilation avec les fonctions du langage prévue à cet effet (copy, include, etc.).
- A3 est un mécanisme de l'édition de liens statique qui permet d'incorporer au programme des objets binaires au format des unités de

compilation.

- **A4** est une fonction du moniteur de travaux de la plate-forme qui permet d'incorporer au programme, lors de son lancement, un texte (ce sont des données) qui pourra être lu par le programme à l'aide d'API système *ad hoc* disponibles sur la plate-forme.
- **A5** est tout simplement un fichier de paramétrage que l'application pourra consulter chaque fois que nécessaire (par exemple des « constantes » pas assez immuables pour pouvoir être déclarées ; $3,14159\dots$ est une vraie constante, tant qu'on est en géométrie euclidienne ; la TVA à 19,6 % est une « constante » lentement variable).
- Les possibilités A1 à A5 existent depuis plus de 30 ans dans les langages de programmation et les systèmes d'exploitation ; A6 et A7 sont beaucoup plus récentes dans la pratique, bien que connues également depuis plus de 30 ans, car elles nécessitent beaucoup de ressources de performance mémoire + temps CPU.
- **A6** est la possibilité d'incorporer dynamiquement, au moment de l'exécution du programme, du code et/ou des données, lors de leur première référence. C'est le mécanisme d'édition de lien dynamique, apparu pour la première fois avec le système MULTICS du MIT. C'est un mécanisme d'une extrême puissance, très consommateur de ressources plate-forme, qui permet de différer au plus tard des références à des entités qui, faute de quoi, auraient dû être liées statiquement *via* l'éditeur de liens. C'est en fait une façon de mettre à jour dynamiquement l'application, sans avoir besoin de tout recompiler, et surtout de tout ré-intégrer (mais il faut tout de même valider). Seul le protocole d'appel du programme doit être standardisé, son corps viendra au dernier moment.
- Les systèmes d'exploitation disposent généralement de la propriété d'introduire des « patchs^[8] » sur un site particulier. Cette propriété très intéressante, car réversible et locale, permet de corriger les anomalies constatées sur une exploitation particulière sans déstabiliser tous les sites. Le moment venu, on pourra intégrer tous les patchs à l'occasion de la fabrication d'une nouvelle version, ou d'une révision. Les patchs sont la matérialisation de nos imperfections et de nos erreurs ; si l'on était dans un monde parfait, il n'y en aurait pas.

- A7 est la possibilité de masquer la machine réelle hardware par une machine purement logicielle, comme la JVM (*Java Virtual Machine*), ce qui permet de disposer d'autant de types de machines que de besoin, donc une très grande souplesse, au prix :
 - d'une baisse de performance très importante (facteur 100, voire plus !), ce qui, avec les microprocesseurs actuels, ne pose plus de problème dans certains contextes comme les IHM, et
 - d'une augmentation de complexité très importante, car le processus d'IVVT déjà problématique et difficile en mode statique, est cette fois complètement dynamique ; en conséquence, la stratégie d'intégration est intégralement à revoir.

Les machines abstraites comme celles mentionnées ci-dessus, peuvent être réalisées sous la forme de machines virtuelles comme la JVM.

La possibilité de compléter dynamiquement une application avec de nouvelles fonctions mieux adaptées au contexte, soit par édition dynamique soit par ajout d'une machine virtuelle, est une capacité fonctionnelle particulièrement bien adaptée au contexte de l'intégration et de l'évolution des systèmes, mais encore faut-il soigneusement doser les parties dynamiques de l'application pour ne pas rendre l'ingénierie de l'intégration carrément ingérable, voir même infaisable, tant au niveau de l'équipe d'intégration, que de la logistique que cette nouvelle forme d'intégration implique. C'est ce qui figure sur la partie droite du schéma avec les mécanismes de surveillance et les traçabilités inverses. Sans ces mécanismes, sur lesquels repose toute l'intégrité du dispositif d'adaptation dynamique, il est exclu de faire fonctionner correctement une telle architecture et de donner une quelconque garantie de qualité.

Le procédé de construction produit des intégrats primaires, appelés intégrats de rang 0, qui, par agrégation progressive, vont permettre la construction validée et vérifiée du système final. Cette construction doit être réversible.

La structure sémantique de cet intégrat est présentée sur la figure 6.27.

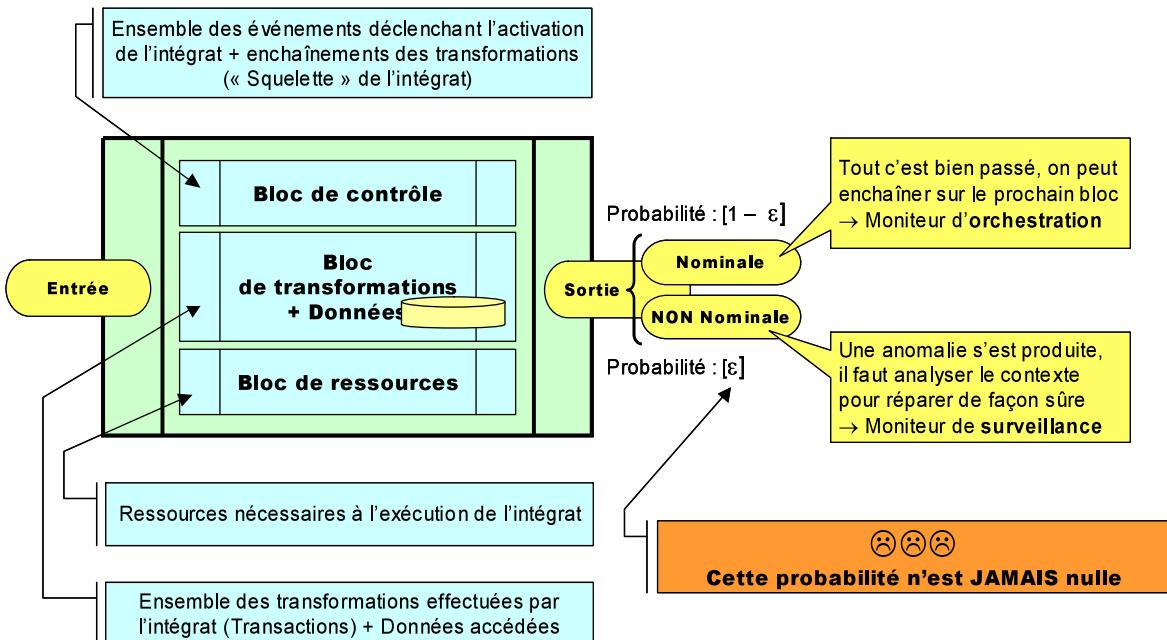


Fig. 6.27 Structure sémantique d'un intégrat

- Le schéma montre les trois blocs sémantiques constitutifs de tout élément logiciel, quel que soit d'ailleurs leur niveau d'intégration.
- Le bloc de contrôle peut être représenté à l'aide des diagrammes d'activité ou des diagrammes état transition tel qu'on les trouve dans le langage UML. C'est le « squelette » de l'intégrat, son épine dorsale, autour de laquelle vont s'articuler les différents éléments, transactions et ressources, nécessaires à l'exécution de l'intégrat. Dans certain cas, en particulier si le contrôle peut être ramené à des automates à états finis ou à piles, on peut représenter le contrôle à l'aide d'une grammaire (*i.e.* la grammaire de l'orchestration), ce qui est le formalisme le plus concis qu'on puisse espérer ; cette notation est utilisée systématiquement dans les compilateurs depuis la fin des années soixante-dix, et beaucoup plus récemment dans les traducteurs de XML.
- Le bloc de transformations est un ensemble de transactions élémentaires qui permettent d'effectuer la transformation ENTRÉE → SORTIE. On peut représenter les transactions par divers descriptifs comme ceux des méthodes objets, soit sous la forme d'une classe, soit sous la forme d'une méthode dans une classe, mais l'essentiel est que les portions de

programmes ainsi modélisées soient de vraies transactions satisfaisant les propriétés ACID^[9].

- Le bloc ressources est constitué des éléments externes dont l'intégrat peut avoir besoin pour s'exécuter. Ce peut être une simple donnée externe du contexte partagé entre différents intégrats dont l'accès nécessite un protocole ou des règles de gestion conventionnelles. Ce peut être une base de données, un service dont les modalités d'appel devront être précisées (séquentiel synchrone, asynchrone avec ou sans messages, accusé de réception, etc.)
- Le schéma distingue la sortie nominale, quand tout se passe bien et que la transformation effectuée a produit ses résultats ; la suite des traitements peut alors être lancée sans risque.
- La sortie anormale est activée lorsque des pannes irréparables ont été rencontrées lors de l'exécution de l'intégrat et que, non seulement la qualité du résultat ne peut pas être garantie, mais que de plus il y a un doute sur l'intégrité de la plate-forme et de la disponibilité des ressources. Dans ce cas, il est indispensable d'activer les mécanismes de surveillance de l'exécution et, si nécessaire, les mécanismes de réparation, conformément à ce qui a été exprimé en tant que besoin dans la caractéristique non fonctionnelle de fiabilité (*i.e.* « *recoverability* » ou capacité de remise en état de marche). Cette distinction est fondamentale car elle évite de considérer comme nominale, une situation à risque qui ne l'est pas ; c'est elle qui permet la tolérance à certaines pannes qui seront jugées réparables, *i.e.* la « *fault tolerance* ». Les mécanismes de surveillance doivent donc être programmés de façon telle qu'ils résistent à des pannes ; pour ce faire, ils doivent disposer d'un vecteur d'état et d'information résumant l'historique de l'exécution effectuée.

On peut schématiser un tel vecteur comme sur la figure 6.28.

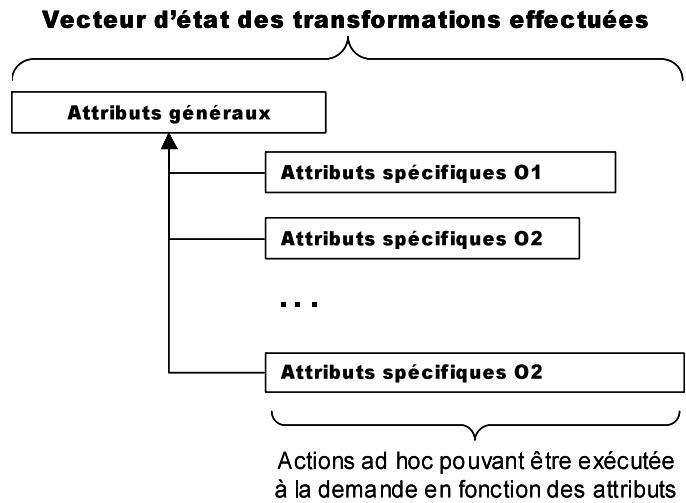


Fig. 6.28 Structure du vecteur d'état d'une transformation

- Les vecteurs d'états peuvent être stockés dans une mémoire conventionnelle, éventuellement gérés par la plate-forme, de façon à garantir une persistance essentielle à la réparation. En intégration, il est indispensable *via* les scénarios et les paramétrages de tests de pouvoir manipuler ces vecteurs de façon à valider les mécanismes de reprises.

6.6.2. La machine à intégrer

Le processus d'intégration peut être assimilé à une machine d'assemblage, analogue à une chaîne de montage automobile, qui construit progressivement les intégrats résultant d'assemblages partiels, jusqu'à obtention de l'intégrat final sur lequel pourront être effectués les tests de recette du point de vue des usagers.

L'organisation de ce travail d'assemblage résulte directement de l'architecture adoptée (*i.e.* l'arbre produit utilisé dans la gestion du projet pour organiser le parallélisme entre les équipes et/ou les acteurs individuels).

En entrée de la chaîne d'assemblage, nous avons les intégrats de rang 0 tels que définis précédemment.

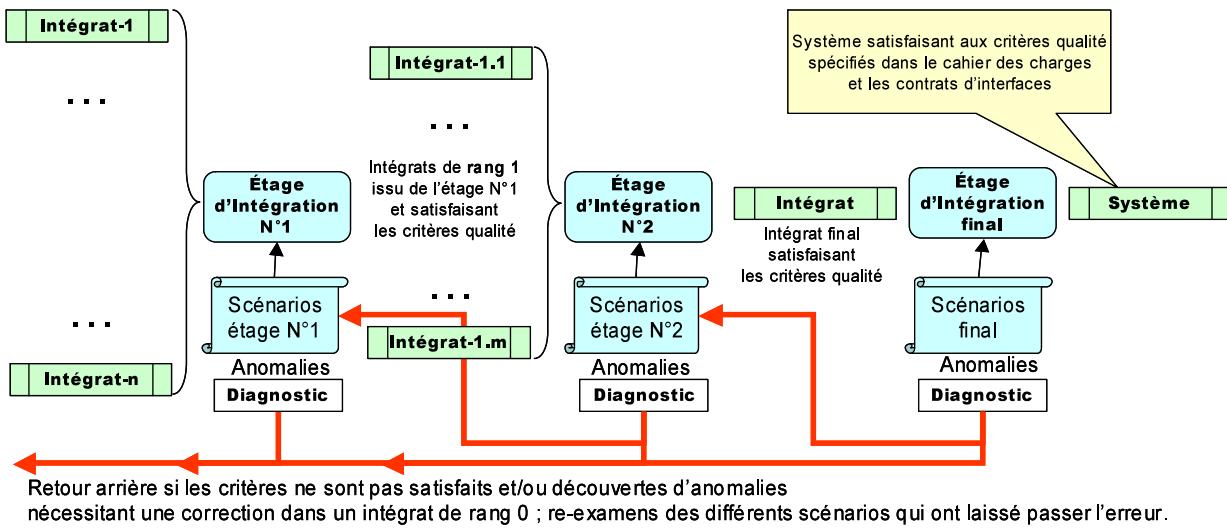


Fig. 6.29 La chaîne d'intégration – Assemblage des intégrants

À chaque étape de l'intégration, les scénarios de tests de cette étape sont mis en œuvre, ce qui permet de découvrir de nouveaux défauts. Chaque anomalie constatée fait l'objet d'un rapport d'anomalie (RA), géré en configuration, redirigé vers l'étape précédente pour diagnostiquer de la façon la plus précise possible l'origine de l'anomalie et la chaîne causale du défaut, et ce jusqu'à l'intégrat de rang 0 qu'il faudra corriger le moment venu.

Pour autant, la mécanique d'intégration n'est pas nécessairement stoppée car des actions correctrices locales peuvent être effectuées permettant aussi d'exécuter d'autres scénarios de tests.

Ceci nous amène à examiner les conditions d'entrée des intégrants de rang 0 dans la chaîne d'intégration. Deux attitudes sont possibles :

- L'intégrat de rang 0 doit être complètement programmé et validé par des tests unitaires ; la documentation doit être disponible.
- L'intégrat de rang 0 peut être incomplet, pour autant que la partie programmée soit validée, et que la partie manquante ait été bouchonnée.

La situation a) est de toute évidence plus sûre et plus simple, mais elle a l'inconvénient majeur d'aligner le processus d'intégration sur la livraison la plus tardive. Il est très difficile de récupérer les marges de planification ainsi

offertes. D'un point de vue management de projet, ce n'est pas une stratégie optimale.

La situation b) permet effectivement de récupérer les marges et de permettre à l'équipe d'intégration de travailler plus efficacement. Pour ce faire, l'équipe d'intégration peut accepter des intégrats incomplets pour autant que la cohérence sémantique de l'assemblage soit garantie. Si ce n'est pas le cas, les scénarios de tests sont inutilisables et il faut réaliser des scénarios adaptés.

La situation b) demande donc une coordination plus précise et plus rigoureuse entre les équipes d'ingénierie et l'équipe d'intégration. Les interactions résultant de ce couplage beaucoup plus étroit devront être soigneusement gérées pour qu'elles ne transforment pas le projet en chaos généralisé. Ceci exige un haut niveau de maturité, en particulier du côté de l'ingénierie qui doit complètement intégrer la contrainte d'intégration dans l'architecture et dans la programmation du système.

Le passage d'un étage d'intégration au suivant peut être détaillé comme sur la figure 6.30.

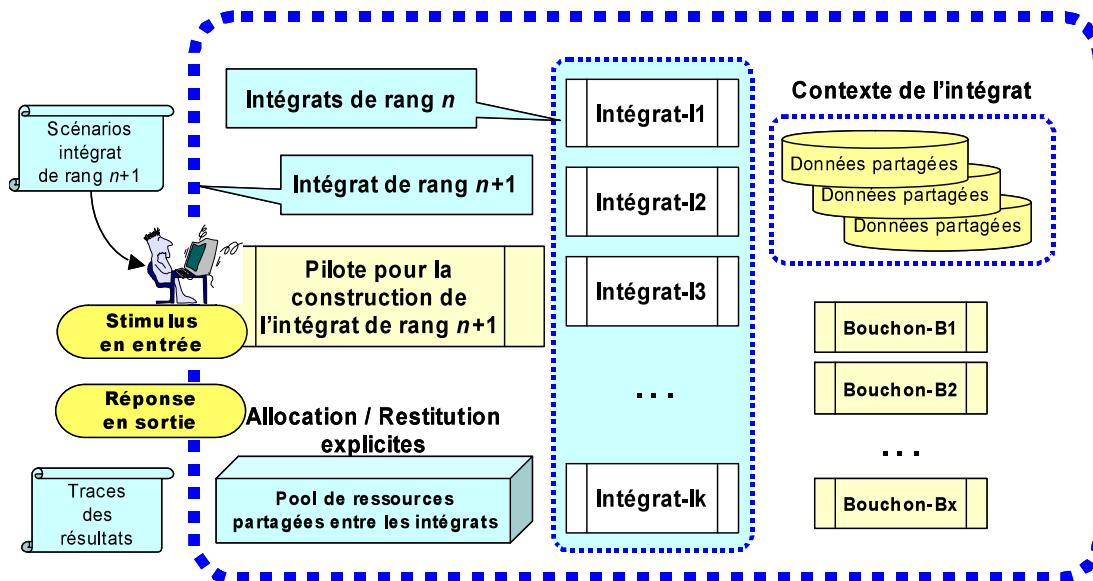


Fig. 6.30 Agrégation des intégrats

Pour que l'intégrat de rang $n + 1$ puisse être testé il faut :

- A. initialiser les données de contexte des intégrats de rang n ;

- B. mettre à disposition des intégrats les ressources indispensables à leur exécution (*N.B.* : elles peuvent être simulées) et valider que les règles d’allocation/restitution sont correctement appliquées ;
- C. définir les « bouchons » pour les éléments des intégrats non encore programmés et/ou livrés mais qui doivent fournir des réponses cohérentes pour construire des enchaînements eux-mêmes cohérents.

Le pilotage est sous contrôle de l’intégrateur qui peut programmer tout ou partie des enchaînements qu’il souhaite valider.

Les entrées et sorties de l’intégrat peuvent nécessiter des fonctions d’édition *ad hoc* qu’il sera utile de standardiser pour l’ensemble du système sous la forme de langage textuel (par exemple en XML).

6.6.3. Retour sur l’intégration continue

L’équipe d’intégration doit avoir défini les tests de façon modulaire de façon à pouvoir les adapter au contexte des livraisons dont il est impossible de prévoir la séquence exacte longtemps à l’avance. Vu sous cet angle, l’intégrat devient une structure lacunaire dont les « trous » correspondent à des blocs de fonctionnalités non implémentées, que l’on peut représenter comme sur la figure 6.31.

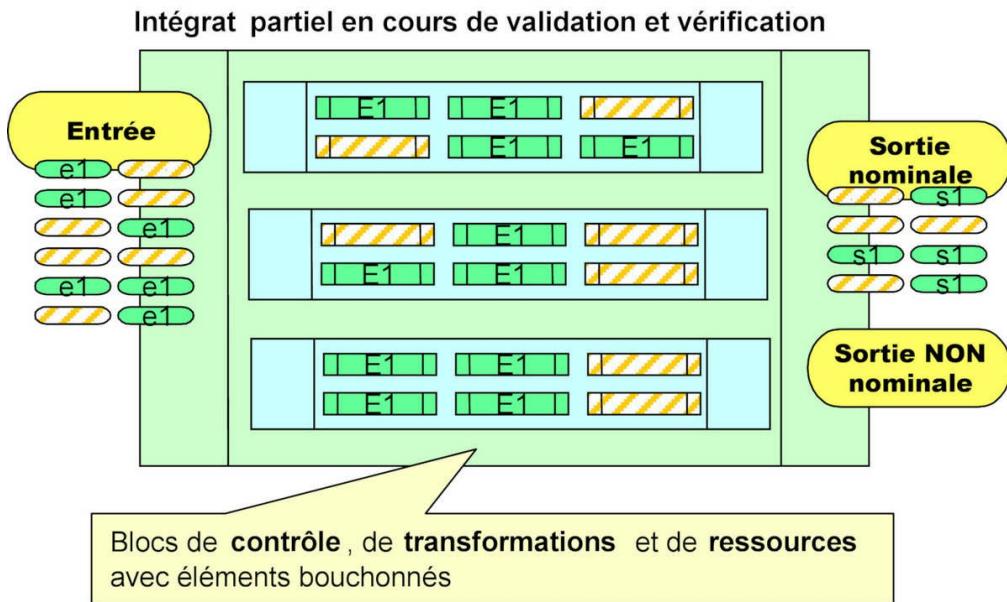


Fig. 6.31 Structure fine d'un intégrat

À l'instant t , seul un sous-ensemble des entrées, des sorties et des blocs peut être utilisé. Les blocs manquants peuvent être bouchonnés ou simulés de façon à rendre la sortie aussi complète que possible.

En procédant ainsi, l'équipe d'intégration va pouvoir assembler un intégrat partiel et commencer l'exécution des scénarios de tests, de façon à garantir que les interfaces entre les intégrats fonctionnent correctement. Un scénario de test valide une séquence particulière d'enchaînement des intégrats, d'où l'intérêt pour l'intégration d'avoir connaissance des diagrammes de séquences élaborés lors de l'expression de besoin et utilisés par les concepteurs pour déterminer la solution architecturale. Si ces diagrammes ont été modifiés, il faut bien sûr que les modifications aient été communiquées à l'intégration.

Une séquence d'intégrats peut être schématisée comme sur la figure 6.32.

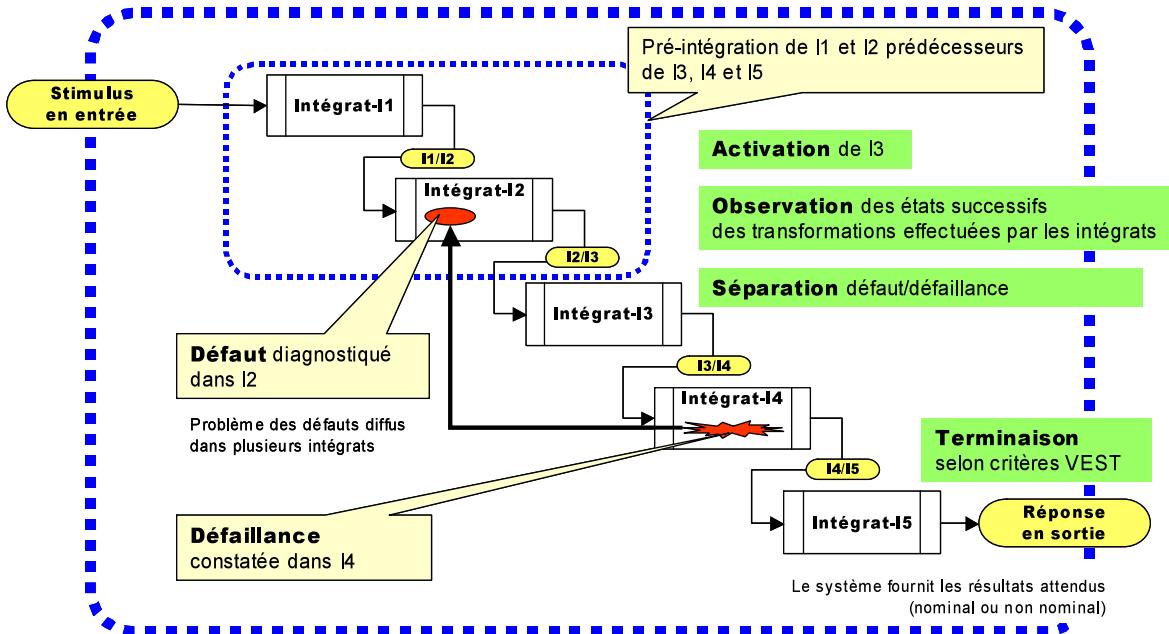


Fig. 6.32 Construction d'une séquence d'intégrats

La figure 6.32 met en évidence quatre principes généraux qui fondent le travail d'intégration.

À partir d'une initialisation des données d'entrée, effectuée éventuellement par simulation *via* le pilote, il faut organiser le travail de façon à parcourir la séquence complète des intégrats le plus rapidement possible, pour obtenir une terminaison satisfaisant les critères qualité, fut-ce au prix d'une simplification drastique des intégrats, en bouchonnant au maximum. Ceci permet de valider les interfaces indépendamment de contenus fonctionnels des intégrats, et de valider la partie du mécanisme de surveillance qui sera utilisé lorsque les intégrats seront plus complets : c'est le principe Initialisation – Terminaison, et obtention rapide de chaînes longues.

L'équipe d'intégration se sert au maximum de la structure existante pour éviter d'avoir à effectuer du travail inutile, et plus particulièrement de réduire la réalisation des bouchons et des simulations au strict minimum : c'est le principe d'activation, c'est-à-dire la capacité à activer tout ou partie d'un intégrat enfoui en se servant des intégrats précédents dans la chaîne d'exécution.

Lorsqu'une défaillance est constatée, le problème est de retrouver le défaut à l'origine de la cause de la défaillance qui est peut-être dans un intégrat antérieur. Il faut donc disposer d'information en quantité et en qualité pour effectuer cette séparation et formuler un diagnostic : c'est le principe de séparation du couple défaut exécuté — défaillance constatée.

Pour ce faire, il est indispensable de savoir observer les transformations effectuées par les intégrats, d'où l'importance de la validation préalable des interfaces et des mécanismes de surveillance : c'est le principe d'observation.

Le mécanisme de surveillance, qui fait partie intégrante de l'architecture, doit pouvoir fonctionner à différents niveaux de finesse, pour ne pas noyer l'intégrateur sous une quantité d'information qui rendrait son travail d'analyse très pénible, et compliquerait la formulation d'un diagnostic.

Comme tout ceci est extrêmement répétitif on voit l'intérêt de programmer ce mécanisme de surveillance à demeure dans le système une bonne fois pour toutes, plutôt que d'avoir à le définir avec des outils externes comme les débogueurs symboliques, et à le reconstruire d'une version à l'autre, car cela fait partie des réalisations effectuées mais non livrées.

Si l'on veut maximiser le rendement des différentes équipes, il faut faire en sorte que tout ce qui est fait par un acteur quelconque puisse servir à un autre acteur qui en aurait le besoin. La programmation systématique des mécanismes de surveillance, plutôt que de façon opportuniste au gré des problèmes rencontrés, est particulièrement rentable puisqu'elle va servir tout au long du processus d'intégration et surtout pendant la phase de maintenance du système. Elle est parfaitement conforme au principe qualité : « Faire bien dès la première fois ».

Une chaîne fonctionnelle d'intégrat ayant ainsi été constituée, le problème de l'équipe d'intégration est d'organiser la croissance fonctionnelle de la chaîne en remplaçant progressivement les blocs bouchonnés par des blocs programmés. C'est ce que montre la figure 6.33.

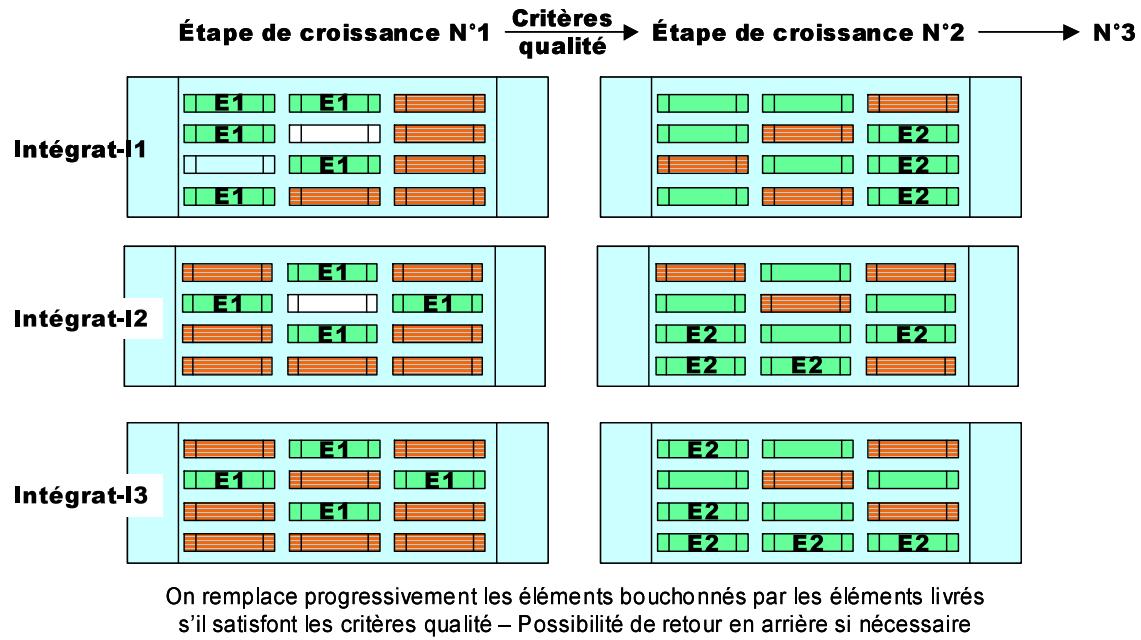


Fig. 6.33 Croissance fonctionnelle d'une séquence d'intégrats

Sur la figure 6.33, on voit qu'un certain nombre de bouchons ont été remplacés par des blocs programmés à l'étape E2.

Cette façon de procéder permet de faciliter la séparation défaut/défaillance. Si un nouveau défaut apparaît, il est raisonnable de penser que cela vient plutôt des nouveaux blocs intégrés. En jouant sur la dichotomie blocs bouchonnés — blocs intégrés, il est plus aisément d'identifier le bloc programmé qui est en cause dans la nouvelle défaillance.

6.6.4. Intégration et gestion de configuration

Cette gymnastique nécessite une gestion de configuration très bien définie, permettant, via un jeu de bibliothèques correctement organisées, de prendre soit le nouveau bloc, soit le bouchon, sous le contrôle de l'intégrateur.

On notera le rôle crucial de l'architecture pour organiser correctement le modèle de croissance, ainsi que celui de la nomenclature du produit (article de configuration).

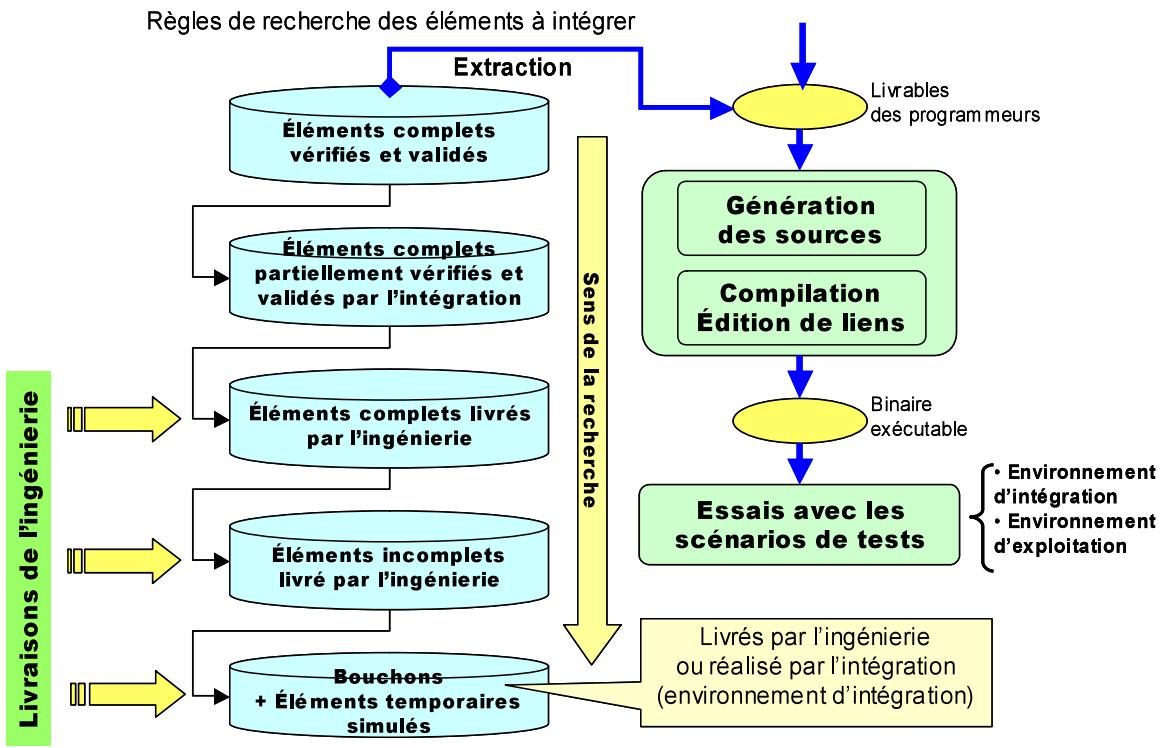
La figure 6.34 montre les principes d'organisation de ces bibliothèques.

Lorsque l'intégrateur déclenche la construction d'un intégrat, *via* les processeurs *ad hoc* de l'environnement de programmation et d'intégration, il doit déterminer l'ordre dans lequel les recherches doivent être effectuées dans les différentes bibliothèques.

Au fur et à mesure de l'avancement du processus d'intégration, les intégrats migrent progressivement vers la bibliothèque qui contient les intégrats de rang 0 qui sont les éléments complets, vérifiés et validés qui seront constitutifs de l'intégrat final.

Cette bibliothèque contient la configuration livrée. Tous les éléments qui ont été utilisés temporairement, à un moment ou à un autre, dans la chaîne d'assemblage sont conservés dans la bibliothèque qui contient les bouchons et les éléments simulés. Le contenu de cette bibliothèque, bien que ne faisant pas partie des livrables fonctionnels, est un investissement important très utile dans le MCO ; il doit donc être géré comme tel, et surtout pas laissé en jachère, car c'est un coût qu'il faudra nécessairement réinvestir tout au long de la vie du système. Un bon moyen de pénaliser la productivité de l'équipe qui sera en charge de la maintenance est de ne pas lui livrer ces éléments que de toute façon elle devra reconstruire, sous une forme ou sous une autre. Notons que le principe d'une telle livraison ne peut être que bénéfique au projet, car elle oblige l'équipe de développement à travailler solidairement avec l'équipe de maintenance avec un contrat explicite.

N.B. : c'est une vraie difficulté, et un risque, dans le cas de projets offshore.



L'organisation de la bibliothèque est en relation duale avec l'architecture et l'organisation du projet

Fig. 6.34 Organisation des bibliothèques de l'intégration.

6.6.5. Intégration dynamique

L'intégrat construit à l'aide des mécanismes de compilation et d'édition de liens statiques est une image figée définie à ce moment de compilation. Néanmoins, nous avons vu, notamment grâce au schéma de la figure 6.26, que la construction peut être différée jusqu'à l'exécution de l'intégrat grâce au mécanisme d'édition de liens dynamique.

Cette possibilité, quand elle existe, peut puissamment aider à améliorer la productivité et le rendement du processus d'intégration, en complétant le procédé d'intégration général, que l'on peut dénommer plus précisément : procédé d'intégration discontinu à gros grains (*i.e.* la stratégie d'intégration silo), par un mécanisme complémentaire, beaucoup plus souple, que l'on peut dénommer procédé d'intégration continue à grains fins.

Pour que ce second mécanisme joue pleinement son rôle, il faut que le mécanisme général garantisse la construction d'un intégrat final dont certains éléments seront intégrés, dynamiquement, à la demande, lorsque l'exécution du système rencontrera une référence non résolue correspondant à un élément qui n'a pas été intégré dans la construction statique. Pour que cela soit possible, il faut que les différentes API du système soient stabilisées de façon à permettre une programmation indépendante et parallèle à ce qui est réalisé pour l'intégrat final intégré statiquement. C'est ce que montre la figure 6.35.

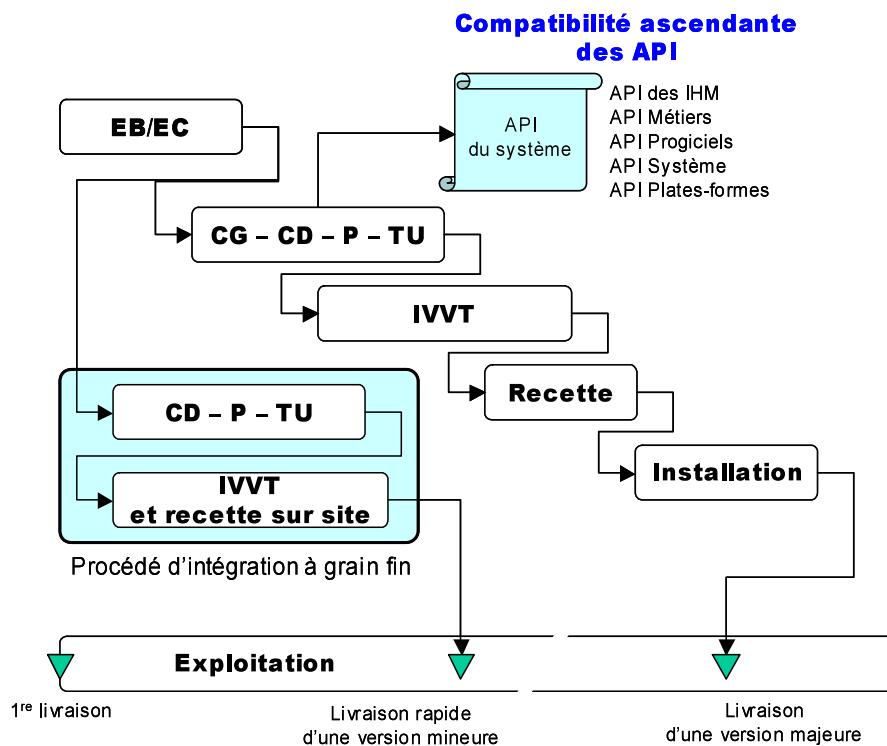


Fig. 6.35 Le procédé d'intégration continu à grains fins

À l'issue du processus de conception générale et détaillée, l'équipe d'architecture a produit une bibliothèque d'API dont les modalités d'emploi sont rigoureusement spécifiées, validées et vérifiées. Cette bibliothèque constitue l'armature logique du système, et d'une certaine façon son axiomatique. C'est sur la stabilité et la cohérence logique de ces API que l'ensemble du système va pouvoir se bâtir. Cette bibliothèque peut être enrichie progressivement par de nouvelles API à la condition expresse qu'elles ne rendent pas le système incohérent (c'est une propriété logique qui

n'est pas évidente à établir), conformément au principe de compatibilité ascendante.

Si l'on fait une analogie mathématique, c'est un peu comme si, sur une base axiomatique, on enrichissait cette base par la définition de nouveaux objets mathématiques et les théorèmes permettant de les manipuler de façon sûre. Par contre, rajouter de nouveaux axiomes nécessite de grandes précautions car il ne faut pas rendre le système contradictoire. C'est ainsi qu'en géométrie, selon l'axiome choisi pour définir le parallélisme, on peut construire différentes géométries qui, chacune, ont des propriétés intéressantes (géométrie sphérique, espace-temps, etc.).

Dans un ordinateur, la programmation par événements nécessite beaucoup de précautions car il faut maîtriser la chaîne de traitement depuis l'entité qui a levé l'exception jusqu'à celle qui traitera l'exception (dépendances fonctionnelles cachées). C'est ce qui rend la programmation en architecture client-serveur sur des plates-formes distribuées particulièrement délicate car, selon le contexte d'emploi, la sémantique peut être différente^[10]. La surveillance et l'intégration dynamique sur de telles plates-formes nécessitent des précautions particulières, et un haut niveau de maturité pour l'équipe qui souhaite mettre ces concepts en œuvre.

L'intérêt de l'intégration continue dynamique est que l'apparition d'un nouveau besoin peut être pris en compte à tout moment, dans la mesure où la traduction informatique de ce nouveau besoin peut s'effectuer à l'aide des API déjà définies, ou de nouvelles API compatibles avec les anciennes. La grosse différence avec l'intégration discontinue est que l'IVVT et la recette vont se faire sur site et non pas dans un environnement d'intégration complètement contrôlé par l'équipe d'intégration.

En termes d'organisation cela revient à dire que l'équipe d'exploitation est également une équipe d'intégration, ce qui n'est évidemment pas la même chose. Une équipe d'exploitation nominale n'a heureusement pas besoin, d'avoir la même compétence que l'équipe d'intégration, car elle ne fait qu'exploiter ce qui a été conçu et préparé par d'autres.

Une intégration continue intempestive peut avoir de graves conséquences sur la disponibilité du système. Si les mécanismes d'édition de liens dynamique ne sont pas parfaitement maîtrisés cela peut se traduire par des

comportements dégénératifs de la disponibilité du système, par des fuites de mémoires, etc., sans parler des problèmes de sécurité.

Ces mécanismes ne doivent donc être mis qu'entre des mains expertes, connaissant bien le fonctionnement des plates-formes sous-jacentes.

Une fois l'intégration réussie, les API nouvellement créées vont enrichir la bibliothèque d'API générales, ce qui permettra une croissance fonctionnelle beaucoup plus souple que par le procédé discontinu.

La condition pour que cela marche est que :

- il existe préalablement une bibliothèque d'API stabilisées,
- l'équipe ait un niveau de maturité suffisant, par exemple un niveau 4-5 dans l'échelle CMM.

Une maturité insuffisante fait courir le risque d'un chaos inextricable, car tout devient dynamique, y compris la configuration.

6.7. Stratégie d'intégration pour la validation, la vérification et l'intégration

Le but du processus d'intégration est d'amener le taux d'erreurs résiduelles observé en sortie du processus de CODAGE-TESTS UNITAIRES (qui produit les intégrats de rang 0) à un niveau tel qu'il permette l'exploitation sans risque, compte tenu du contrat de service passé entre l'ingénierie et l'organisation cible utilisatrice du système.

Les statistiques concernant la découverte des défauts sont connues^[11], comme le rappelle la figure 6.36.

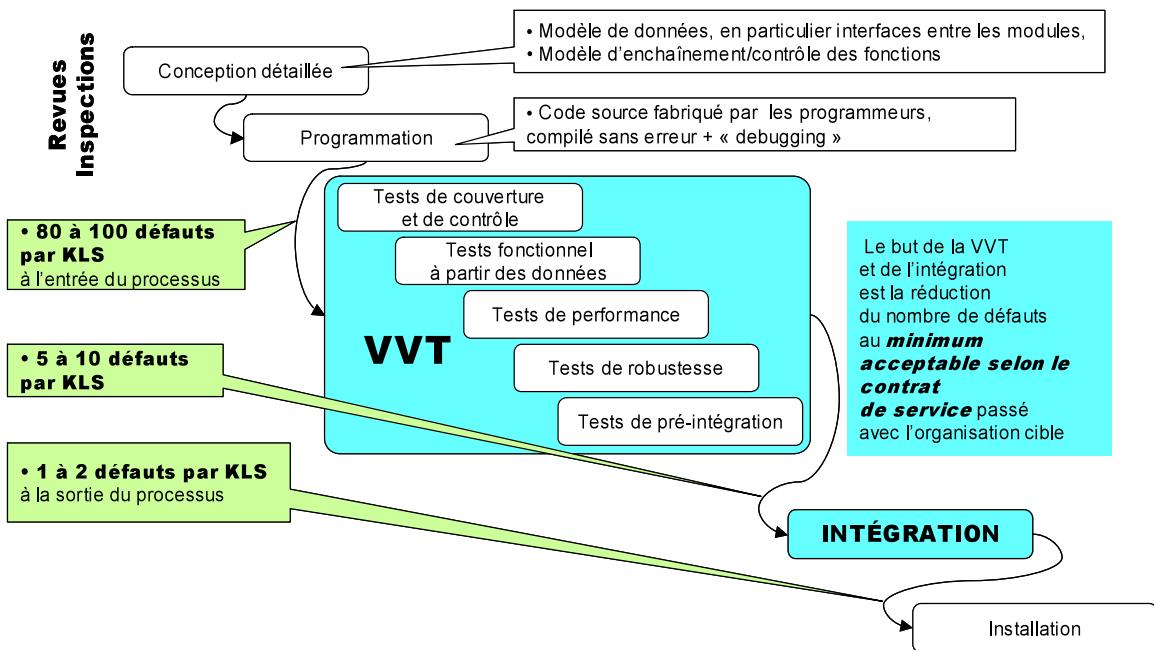


Fig. 6.36 Présence et découverte des défauts

L'effort de test est scindé en trois parties, selon la nature des processus concernés par cet effort. La stratégie de test consiste à trouver l'équilibrage de façon à optimiser la somme des coûts, et non pas les coûts individuels, comme indiqué sur la figure 6.37.

Conformément aux schémas 6.23, 2.29 et 6.32, tout retour arrière à un coût additionnel matérialisé par les coefficients d'amplification α . Pour optimiser la performance des processus, une métrologie est indispensable, conformément aux approches qualité comme celles préconisées par le SEI avec le modèle CMM/CMMI.

En entrée en intégration, selon la statistique, il n'est pas rare d'avoir un taux de défauts de l'ordre de 5 à 10 défauts par millier de lignes source (KLS) nouvelles intégrées. Pour que le système soit exploitable avec un niveau de risque contrôlé, l'intégration doit ramener ce taux à 1-2 défauts par KLS. Les statistiques du SEI montrent que ce taux peut descendre aux alentours de 0,5, à effort constant, pour des organisations ayant un niveau de maturité de 4 ou 5 qui maîtrisent l'intégration continue dynamique.

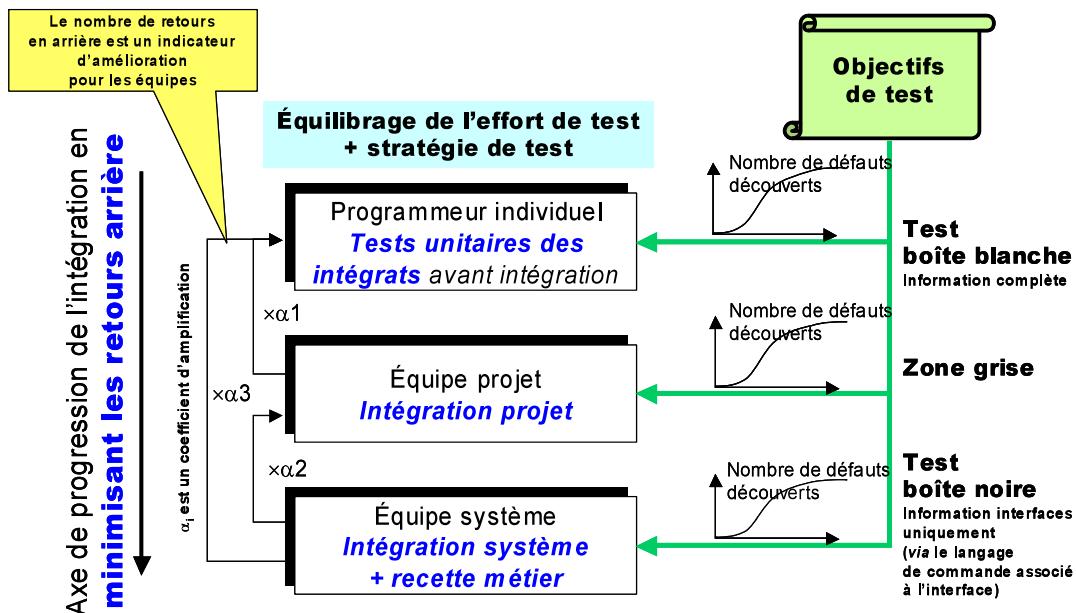


Fig. 6.37 Répartition de l'effort de test

Ces statistiques sont élaborées après coup, et donnent un profil général moyen. Dans un projet réel, on ne connaît évidemment pas le nombre de défauts *a priori*, par contre on peut compter le nombre de défauts découverts à l'occasion des tests. On peut gérer les rapports d'anomalies, jusqu'à un certain point. Dans le cas d'une équipe d'intégration bien organisée, on peut rapprocher le nombre de défauts découverts par l'équipe de l'effort consenti pour les découvrir. On obtient ainsi, avec quelques précautions de normalisation, une courbe de maturité du logiciel sous test.

L'effort de l'équipe d'intégration peut se répartir comme suit :

- Effort pour produire des scénarios de test.
- Effort pour effectuer les essais, analyser les résultats, formuler des diagnostics, produire les rapports d'anomalies et les faits techniques.
- Effort pour gérer la configuration d'intégration et produire la documentation pour le MCO, les RA.
- Effort pour développer des programmes bouchons et/ou des simulations d'éléments (environnement d'intégration).
- Effort perdu (coût de non-qualité, erreurs d'aiguillage, documentation insuffisante, etc.).

Ce qui permet de découvrir les défauts sont les scénarios de tests et le nombre d'essais effectués ; l'effort correspondant est une grandeur fondamentale. La gestion de configuration est clairement un coût qualité.

N.B. : on remarquera que la stratégie d'intégration continue dynamique fiabilise les mécanismes d'échanges et l'interopérabilité qui sont le fondement de l'environnement d'intégration intégré.

L'effort pour développer des programmes bouchons, au-delà d'un certain seuil, peut être le révélateur d'une mauvaise stratégie d'intégration, ou d'une mauvaise architecture.

La corrélation la plus évidente est la courbe de découverte des défauts, rapportée à la somme des efforts scénarios + essais. Plus cet effort est important, plus on a de chance de trouver les défauts, d'où l'allure de la courbe présentée figure 6.38.

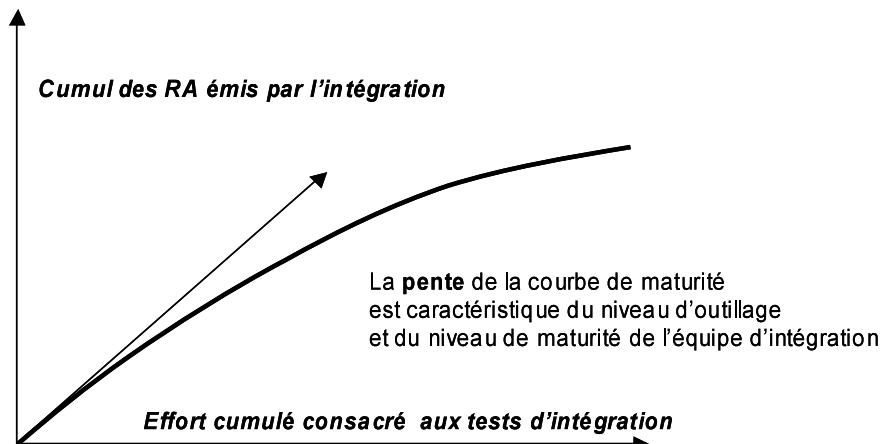


Fig. 6.38 Courbe de découverte des défauts.

Cette courbe est croissante et il est naturel de penser que l'effort pour découvrir le défaut suivant les n défauts déjà découverts va aller en augmentant ; l'analyse devient de plus en plus difficile, la longueur des scénarios s'allonge, etc. d'où la concavité de la courbe.

La pente de la courbe est fonction de l'outillage et du nombre d'essais que l'on peut effectuer. Ceci étant, pour automatiser, il faut investir en outils, ce

qui consomme de l'effort qui ne sera pas disponible pour écrire des scénarios de test. L'équilibre est donc loin d'être évident à trouver ; tout dépend de la taille du système, du nombre de versions installées, de la durée de vie, de la maintenance et des évolutions effectuées.

Une représentation plus fidèle du processus d'intégration serait une courbe en S classique, où l'on retrouverait les trois phases principales du processus, c'est-à-dire :

- **Phase 1** : préparation des scénarios et de la logistique des plates-formes nécessaire à l'intégration (environnement d'intégration).
- **Phase 2** : début des essais et continuation de l'effort pour produire les scénarios d'essais.
- **Phase 3** : essais systématiques de tous les scénarios, avec non-régression, et compléments éventuels avec de nouveaux scénarios.

La courbe de découverte (elle est plus difficile à construire, car la métrologie est plus complexe à établir et à normaliser) devient celle de la figure 6.39.

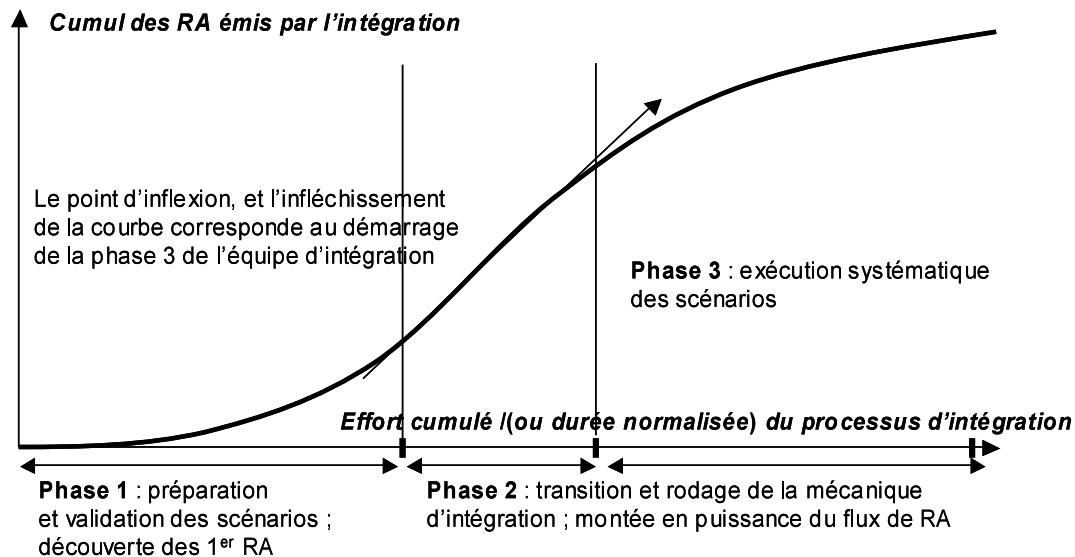


Fig. 6.39 Courbe en S caractéristique de l'effort d'intégration

N.B. : la courbe de découverte correspond aux phases 2 et 3, où l'effort de fabrication des scénarios devient faible.

Le nombre de défauts résiduels est une constante qui dépend uniquement de la qualité du travail fait par l'équipe d'ingénierie. La maturité de l'équipe d'intégration fait que l'on découvrira plus ou moins vite les défauts.

Le nombre d'essais à effectuer est un paramètre dimensionnant de la bonne rentabilité de l'effort d'intégration. Tout essai nécessite un travail de préparation et d'analyse, même avec des automatismes poussés. Il faut donc minimiser le nombre d'essais.

Dans la chaîne d'intégration de la figure 6.29, la logique d'assemblage est calquée sur la structure de l'arbre produit résultant des choix architecturaux du projet.

Par exemple, on peut avoir une architecture qui organise les tests comme sur la figure 6.40.

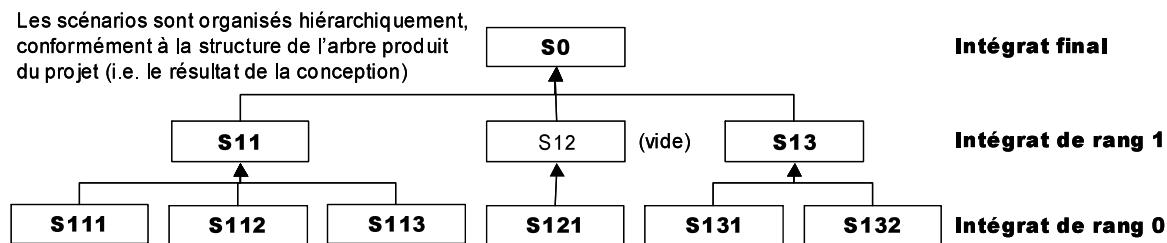


Fig. 6.40 Arborescence des tests basée sur l'arbre produit.

La documentation de conception étant connue on peut produire l'ensemble des scénarios de tests nécessaires à la vérification et à la validation de l'intégrat final.

On peut représenter les essais à effectuer à l'aide du diagramme d'activité de la figure 6.41.

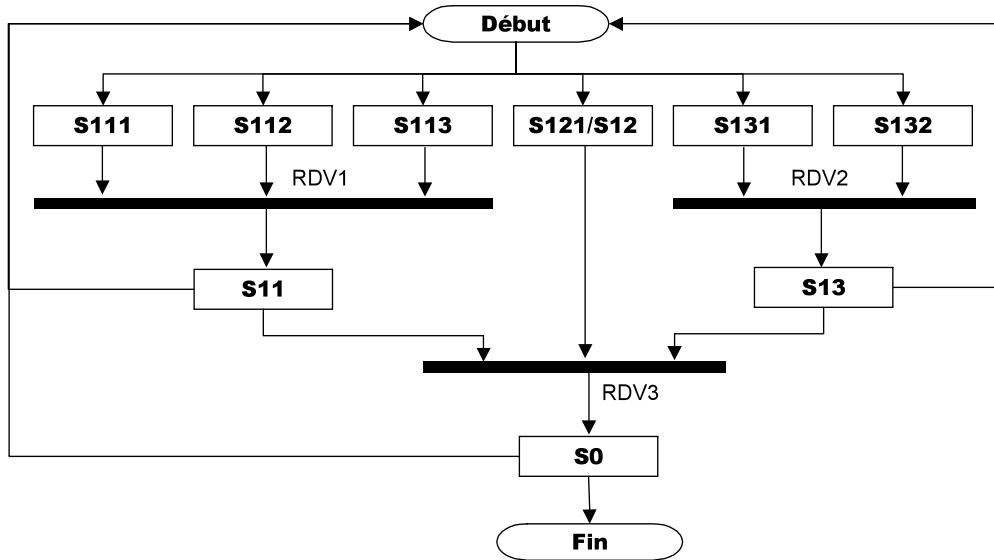


Fig. 6.41 Diagramme d'activité des essais d'une arborescence de test

Tous les essais correspondant aux intégrats de rang 0 peuvent démarrer en parallèle, en fonction des ressources disponibles. Les essais correspondant aux intégrats de rang 1 ne peuvent démarrer que si tous les scénarios des intégrats qui les précèdent sont passés avec succès.

La figure 6.41 montre trois rendez-vous logiques (notés RDV), ce qui n'interdit pas, via des bouchons, de tester des scénarios de bout en bout, jusqu'à S0, dès que l'ensemble des interfaces a été validé.

Il faut remarquer sur la figure 6.41 que tout défaut découvert lors des passages des scénarios de rang 1 et 2 nécessite des corrections dans les intégrats de rang 0 ; il faut donc exécuter à nouveau tous les scénarios correspondant aux intégrats qui ont fait l'objet d'une correction ; c'est la non-régression dont on reparlera dans les règles ci-après.

Sur cette base, on peut dresser le tableau de bord d'avancement de l'intégration comme sur le tableau 6.1.

Tab. 6.1 Tableau de bord de l'intégration

Intégrat de rang 0							
N° RA	État (O/F)	I1	I2	I3	...	In	Coût

RA-1	F						C1
RA-2	F						C2
RA-3	O						
...							
RA-x							
...							
RA-k							
		N-I1	N-I2	N-I3		N-In	C

- Un rapport d'anomalie peut être ouvert (O) ou fermé (F).
- Seuls les intégrats de rang 0 font l'objet de corrections source. À chaque rapport d'anomalie, on note dans la matrice des intégrats ceux qui ont fait l'objet d'une correction. Un rapport d'anomalie peut révéler plusieurs défauts.
- À l'issue du $k^{\text{ième}}$ RA, le nombre de défauts corrigés est donc $N = NI_1 + NI_2 + \dots NI_n$
- Le coût des corrections $C = C_1 + C_2 + \dots C_k$
- Le coût moyen d'une correction est égal à

C



N

- On peut extraire du tableau des statistiques plus fines du type :
 - Nombre de RA ayant nécessité la correction de un intégrat, de deux intégrats, etc.
 - Nombre de défauts découverts par KLS ou par PF.
 - Identité du scénario ayant permis de découvrir l'anomalie.
 - Nombre de défauts découverts en fonction du rang du scénario.

La cassure de la courbe de découverte des défauts est le signe que le potentiel de découverte de défauts par les scénarios est épuisé. Il faut produire de nouveaux tests, ou faire une expérimentation en vraie grandeur sur un site d'exploitation, par exemple des béta-tests.

À titre d'exemple statistique, on peut donner des chiffres comme ceux du tableau 6.2 qui émanent de Hewlett-Packard^[12], collectés sur un très grand nombre de projets.

Tab. 6.2 *Effort moyen de correction des défauts*

Effort moyen de correction des défauts (chiffres H-P)
25 % des défauts sont diagnostiqués et corrigés en 2h
50 % des défauts sont diagnostiqués et corrigés en 5h
20 % des défauts sont diagnostiqués et corrigés en 10h
4 % des défauts sont diagnostiqués et corrigés en 20h
1 % des défauts sont diagnostiqués et corrigés en 50h

Soit un coût moyen par défaut découvert de 6,3 heures, soit environ 1 hj (homme-jour) si on compte les temps physiologiques et la fatigue. En intégration, on est plutôt dans le dernier décile.

Dans des systèmes complexes de grande taille, le coût de fabrication de scénarios d'intégration réalistes peut nécessiter un investissement de quelques hommes mois par scénario. Le chiffrage des scénarios peut se faire sur une base de stimuli et/ou de messages à fournir en entrée et de réponses obtenues et validées. Si le dépouillement des résultats n'est pas automatique, ou assisté, le coût humain peut être considérable (quelques jours ne sont pas rares !) avec en plus un fort risque d'erreur inhérent à l'activité humaine. Le

processus d'intégration est certainement celui où le maximum d'automatisme doit être recherché.

En synthèse, on peut représenter les flux de la dynamique de l'intégration en faisant ressortir la gestion de configuration qui est le processus critique du bon déroulement des opérations, comme sur la figure 6.42.

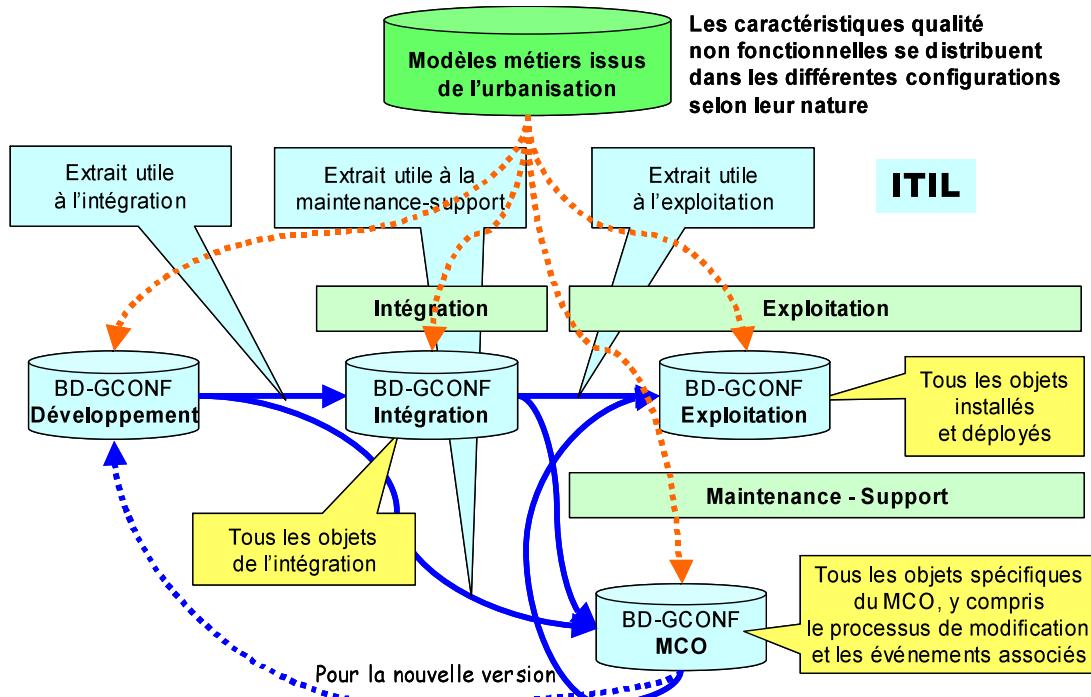


Fig. 6.42 Les flux de gestion de configuration associés à la dynamique d'intégration

Le schéma de la figure 6.42 fait apparaître quatre bases de données de gestion de configuration qui sont interdépendantes. Tout part de la gestion de configuration du développement qui contient toutes les informations nécessaires à la bonne gestion de l'application (*i.e.* les articles de configuration). La gestion de configuration de l'intégration est un extrait de cette BD_Dev avec ce qui est spécifique à l'intégration (tests d'intégration, scripts de paramétrage pour les installations, liste des rapports d'anomalies **RA** et des actions correctrices **AC**, tests de non-régression, etc.). De cette BD_Int est extrait ce qui est nécessaire à l'exploitation, par exemple pour tout ce qui est nécessaire aux diagnostics en cas de défaillance. Les bases BD_Dev et BD_Int alimentent la base nécessaire au MCO qui elle-même se boucle sur la BD_Dev pour la prochaine version. Il est certain que si chaque

responsable de processus adopte une stratégie de gestion de configuration sans concertation avec ses voisins, le résultat sera un grand désordre. En cas de sous-traitance, la complexité s'accroît, surtout si le sous-traitant est laissé libre de ses choix en matière de gestion de configuration. La mise en place d'une telle infrastructure nécessite un haut niveau de maturité, et une très grande discipline de la part de tous les acteurs impliqués dans ce processus.

6.8. Résumé des règles de la dynamique du processus d'intégration

On remarquera que la plupart du temps, une « bonne » architecture induit une structure de coût avec des ordres de grandeur en

$$O[n \log(n)]$$

c'est à dire, par approximation, en

$$O[n^{1+\varepsilon}]$$

avec ε petit, et parfois du

$$O[n^2]$$

. Cependant, très peu de choses peuvent faire complètement déraper la combinatoire vers de l'exponentiel.

6.8.1. Règle N° 1 – Vitesse d’écriture des tests

La vitesse d’écriture des tests (c’est une productivité) est fonction de la taille de la documentation et de la structuration de cette documentation. Si l’architecture de la documentation est hiérarchique, cette vitesse est en

$$O[n \log(n)]$$

, grandeur qui caractérise le temps d'accès moyen dans un arbre comportant n éléments. D'où l'importance de faire une documentation concise et bien structurée, si possible intégrée au logiciel de façon à éviter les manipulations de documents intempestives.

6.8.2. Règle N° 2 – Nombre de tests

Le nombre de tests à produire est fonction de la taille du code (ou du nombre de composants) et de la complexité des composants (relations entre les composants). Une organisation hiérarchique en couches induira une combinatoire en :

$$O[n \log(n)]$$

Une dépendance deux à deux des intégrats induira une combinatoire en

$$O[n^2]$$

correspondant au produit cartésien des intégrats. Une dépendance plus

complexe (2 à 2, 3 à 3...) pourra induire une combinatoire en

$$O[2^n]$$

correspondant à l'ensemble des parties que l'on peut former avec n intégrats. Si les intégrats peuvent former des chaînes de liaisons de longueur n dans lesquelles l'ordre d'occurrence des intégrats est significatif, *i.e.* il y a des dépendances contextuelles, le nombre de chaînes à tester sera en :

$$O[n!] \approx O\left[\left(\frac{n}{e}\right)^n \sqrt{2\pi n}\right]$$

le nombre de tests variera en conséquence et l'effort sera fonction de la longueur des chaînes de liaisons. Dans tous les cas de figure, le nombre de scénarios est supérieur au nombre d'éléments.

La taille du contexte des intégrats est un bon indicateur de ce type de complexité, généralement préjudiciable au projet, voire fatal.

Pour les interfaces entre modules (c'est la complexité explicite rendue visible), le nombre de tests à produire dépend des contraintes (*i.e.* le contrat d'interface) auxquelles l'interface doit satisfaire. La spécification rigoureuse des contraintes, pour chaque interface, optimise l'effort et le rendement des tests.

6.8.3. Règle N° 3 – Effort de vérification des résultats

L'effort d'analyse des résultats de test est fonction de la quantité d'information manipulée pour valider les résultats ; c'est une trace dont la longueur dépend de la taille du graphe de contrôle et des données manipulées. Cette quantité est au moins égale à celle qu'il a été nécessaire pour produire le scénario et les résultats attendus, à laquelle il faut rajouter les traces et tous les résultats intermédiaires permettant de valider le résultat final. Cet effort intègre donc un aspect statique et un aspect dynamique qui dépendent des enchaînements réalisés.

Tous ces efforts doivent être saisis dans la comptabilité analytique du projet, en fonction des types d'activité répertoriés dans la nomenclature du projet. Là encore, la taille de cette trace est un bon indicateur de la qualité de l'architecture.

6.8.4. Règle N° 4 – Effort pour la gestion des configurations

L'effort d'archivage est fonction du volume de tests et de résultats qui doivent être gérés en configuration. La mise en place des liens définissant les relations dépend de la structure des matrices d'interfaces (matrices dites N^2 en ingénierie système^[13]). L'organisation des bases de données de la gestion des configurations (développement, intégration, exploitation et MCO) est un paramètre dimensionnant. Il faut éviter les opérations manuelles redondantes qui sont source d'incohérence, et automatiser tout ce qui peut l'être.

6.8.5. Règle N° 5 – Effort de non-régression

Pour garantir une maturité croissante consécutivement à l'état d'avancement des tests, chaque modification effectuée dans la configuration induit une re-exécution de tous les scénarios précédemment acquis – c'est la garantie de

non-régression – dont l'effort varie en

$$O[n^2]$$

, n étant le nombre de scénarios à prendre en compte dans la non-régression. La valeur de n dépend de l'architecture statique du système, c'est-à-dire de la structure hiérarchique des intégrats. Par exemple dans une architecture client-serveur comme celle de l'application générique des figures 6.5 et suivantes, on aura une arborescence de scénarios comme suit (figure 6.43) :

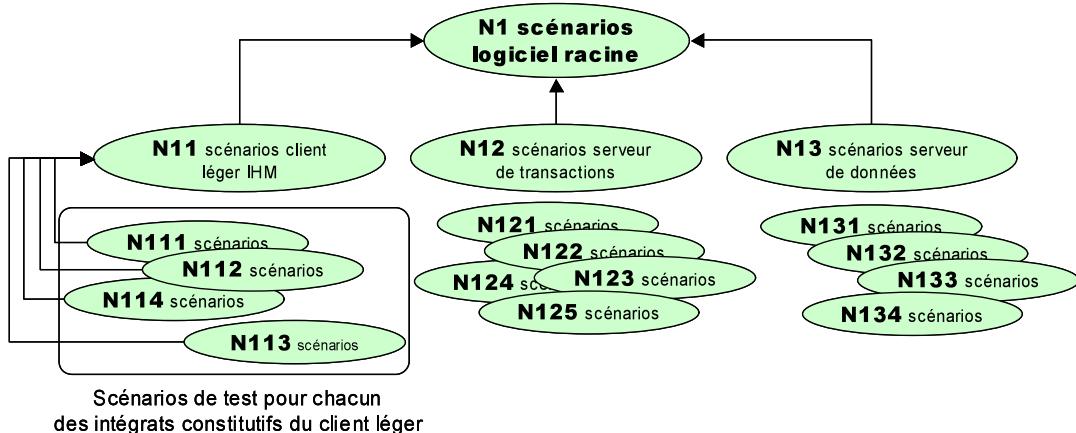


Fig. 6.43 Hiérarchie de scénarios de tests

- N est la somme de tous les scénarios.
- $N1$ est le nombre de scénarios pour l'intégrat logiciel racine.
- $N11$ est le nombre de scénarios pour l'intégrat élément client léger IHM.

$N111 + N112 + \dots$ sont les tests pour les différents composants de l'intégrat client léger.

Si l'on modifie un des composants de l'intégrat de l'élément IHM, il faut, pour effectuer la non-régression du logiciel, repasser un nombre de scénarios de tests = $N1 + N11 + N111 + N112 + N113 + N114$. Une bonne partition de l'arbre produit peut donc notoirement faire baisser le nombre de tests à exécuter lors des non-régressions.

Ceci n'est vrai que si les trois éléments de l'architecture sont parfaitement confinés et qu'il n'existe aucun canal caché entre eux. C'est-à-dire que les seules interfaces utilisées sont celles du bus d'échanges. S'il y a un doute et que la traçabilité n'est pas garantie, il est prudent de re-exécuter l'ensemble des scénarios.

6.8.6. Règle N° 6 – Automatisation des tests

Pour automatiser les tests, les scénarios de tests doivent constituer ce que B. Beizer, dans son ouvrage *Software Testing Techniques*, appelle des langages implicites^[14]. C'est l'architecture qui garantit l'existence de tels langages. Par exemple, vu du client léger IHM, le serveur de transactions est une machine abstraite qui réagit au stimulus émis par l'IHM en fournissant les réponses *ad hoc*. L'existence de canaux cachés qui sont des dépendances implicites résultant de couplages entre les intégrats, soit par le contexte, soit par des événements, interdit l'existence des langages implicites, du fait des relations de dépendances fonctionnelles non explicitées entre les différents éléments du logiciel. L'application rigoureuse des règles de modularité évite la création de ces dépendances néfastes.

N.B. : en théorie des langages, la grammaire d'un tel langage n'est pas « *context-free* », ce qui rend impossible la réalisation d'un traducteur automatique. Un opérateur doit rester dans la boucle pour lever les ambiguïtés.

6.8.7. Règle N° 7 – Simplification et réduction de la complexité du processus d'intégration

La tendance naturelle des concepteurs et des développeurs débutants/novices est de fabriquer des combinatoires qui sont au mieux en

$$O[n^2]$$

, et au pire exponentielles. Une première tâche des architectes sera de s'assurer que les logiques plates-formes ne se combinent pas avec les logiques métiers, y compris dans les situations d'erreurs. C'est l'essence des principes de modularité établis par D. Parnas il y a bien longtemps et remis à la mode par l'OMG avec l'approche dite MDA (*Model Driven Architecture*) MDE (*Model Driven Engineering*). Cette approche préconise une architecture pilotée par des modèles dans laquelle on distingue les modèles métiers indépendants de la plate-forme sous-jacente (PIM, *Platform Independent Model*) et les modèles d'exécution qui sont dépendants de la plate-forme (PSM, *Platform Specific Model*). Entre ces deux modèles, il y a des interfaces qui empêchent la logique plate-forme de remonter dans la logique métier, et réciproquement. Si ce n'est pas le cas, il y a intrication des logiques et explosion combinatoire. Un PSM est une machine abstraite dont l'interface constitue le langage de programmation. Le PIM qui utilise le PSM ne connaît le PSM que par les interfaces qui lui sont communiquées. C'est la systématisation de l'approche des langages cachés de la règle 6, connue et mise en œuvre depuis fort longtemps dans les systèmes d'exploitation, les compilateurs, les SGBD, les protocoles de télécommunications, etc. C'est une technologie éprouvée^[15] qui requiert des architectes expérimentés connaissant bien les domaines sémantiques à modéliser et les technologies de

machines abstraites et des langages associés, à ne pas mettre entre les mains de débutants/novices.

Notes

[1] Cf. J.Printz, *Écosystème des projets informatiques – Agilité et discipline*, Hermès, 2006.

[2] Cf. Th. Chamfrault, C. Durand, *ITIL et la gestion des services*, Dunod, 2006.

[3] Cf. M. Cusumano, R. Selby, *Microsoft secret*, Free Press, 1995.

[4] Voir respectivement : D.Parnas, « On the criteria to be used in decomposing systems into modules », *Communications of the ACM*, Dec. 72, Vol. 15 ; B.Meyer, *Conception et programmation objet*, Eyrolles, 1997 ; J.Sassoon, *Urbanisation des systèmes d'information*, Hermès, 1998.

[5] Pour plus d'informations, voir J. Printz, *Architecture logicielle*, Dunod.

[6] Voir J. Printz, *Architecture logicielle*, Partie 4, Dunod.

[7] Voir bibliographie.

[8] Au sens littéral, c'est un terme de couture qui signifie « pièce », au sens de rapiéçage d'un vêtement.

[9] Pour la notion de transactions, on se rapportera à la littérature technique comme : J. Gray, A. Reuter, *Transaction processing techniques*, ou P. Bernstein, E. Newcomer, *Principles of transaction processing*, tous deux chez Morgan Kaufmann, ou, en plus simple, R. Orfali, D. Harkey, J. Edwards, *Client/server survival guide*, Wiley & Sons, 1999.

[10] L'exemple du « bug » d'ARIANE 501 est encore dans toutes les mémoires.

[11] Voir les ouvrages de B. Boehm sur le modèle COCOMO, ou ceux de B. Beizer dans la bibliographie.

[12] Voir en particulier les deux ouvrages de R. Grady, *Software metrics : establishing a company-wide program* et *Practical software metrics for project management and software improvement*, tous deux chez Prentice Hall.

[13] Cf. *NASA Systems engineering handbook*, la bible de l'ingénierie système.

[14] Le terme anglais utilisé est « *hidden language* », c'est-à-dire caché, au sens de non explicité. La plupart du temps de tels langages sont ambigus et

contextuels ; il est évidemment bien préférable de les expliciter, ce qui permet de les rendre *context-free*, donc beaucoup plus faciles à manipuler. Il est indispensable que l'architecte ait de bonnes connaissances en théorie des langages.

[15] Cf. J. Printz, *Architecture logicielle*, Dunod.

Gérer les tests

Les tests sont un des moyens permettant l'amélioration de la qualité et de la fiabilité des logiciels construits. Schématiquement, il s'agit d'augmenter le temps moyen entre deux défaillances (MTTF) et de diminuer le temps moyen de réparation (MTTR). Une possibilité d'obtenir ce résultat est de réduire de façon aussi complète que possible des défauts résiduels dans le logiciel ; les tests sont le moyen naturel d'y parvenir. Cependant, les tests peuvent aussi être détournés de leur mission première pour permettre la détection « en ligne » d'erreurs prévisibles rendant ainsi possible le retour du logiciel à un état stable et cohérent. On notera qu'il s'agit là plus d'une activité de conception (liée en particulier à la définition d'une architecture testable) que d'une activité de tests à proprement dit, et nous n'aborderons pas ce sujet dans ce chapitre.

Nous allons par contre essayer de définir brièvement quelles sont les bonnes pratiques en matière de gestion des tests. Pour cela, il nous faut définir comment organiser les tests en s'appuyant sur les différents acteurs concernés et en choisissant les types de tests adaptés au contexte. L'étude de la planification des tests va nous permettre de définir la notion de plan de tests ainsi que les points importants dans l'enchaînement des différentes actions liées au processus de test. Les critères d'arrêt des tests sont essentiels et nous y reviendrons tout comme nous rappellerons les éléments fondamentaux de l'estimation du coût et de l'effort associés aux différentes activités induites par les tests. Enfin, nous évoquerons la nécessité de placer les tests sous gestion de configuration comme toute action liée au développement amenée à évoluer au cours du temps de façon contrôlée.

7.1. Organisation des tests et répartition des rôles

Les tests font partie intégrante du processus de développement. Il s'agit d'une activité complexe à forte valeur ajoutée ; rappelons en effet que du point de

vue des coûts, il y a un rapport de 1 à 10 et de 1 à 100 selon qu'une erreur est découverte lors des phases d'expression du besoin ou lors des phases de développement ou lors de la phase d'exploitation.

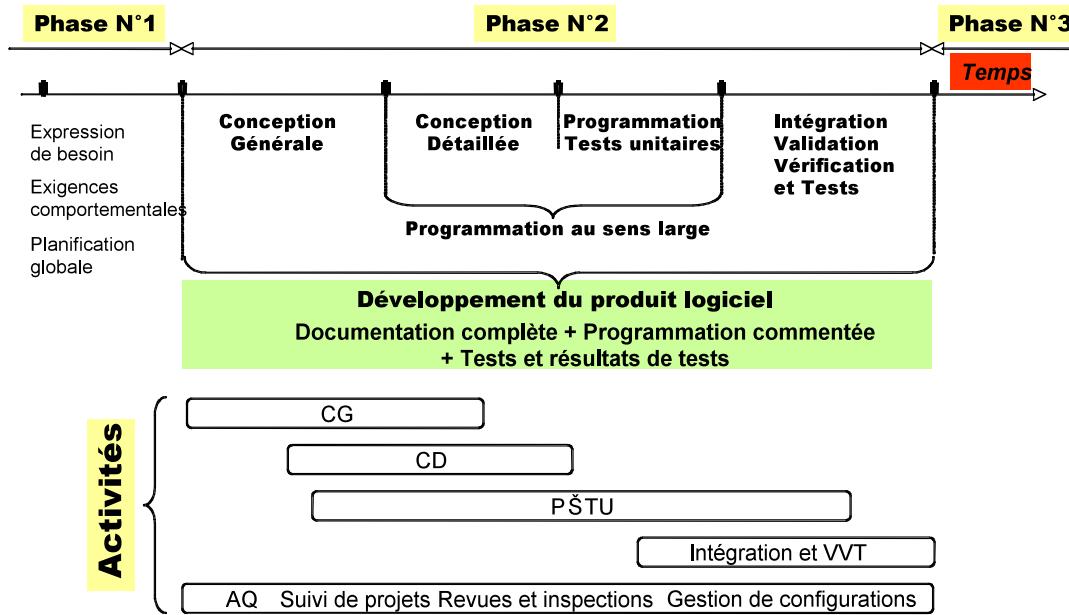


Fig. 7.1 Phases de développement et activités de test

Quel que soit le modèle de développement choisi, il fait apparaître au moins trois phases : l'expression des besoins (phase N° 1 sur le précédent graphique), le développement au sens large (phase N° 2), puis la phase de maintenance (phase N° 3). C'est principalement durant la phase de développement que vont se dérouler les activités liées au test même si, lors de la phase de maintenance, il peut être nécessaire de réaliser des tests de non-régression ou de validation après une demande de correction ou d'adaptation.

7.1.1. Choix des types de tests à effectuer

Différents types de tests peuvent être réalisés à différents stades du processus de développement du logiciel. Plutôt que d'offrir une réponse « clef en main » à la question du choix du type de tests, réponse qui ne pourrait être que partielle et inadaptée à tel ou tel cas particulier, nous préférons fournir les critères de choix qui permettront de prendre la bonne décision.

En tout premier lieu, il convient de définir précisément quels sont les objectifs des tests que l'on souhaite mener : découvrir des défauts, valider une solution ou vérifier empiriquement une propriété non fonctionnelle comme la bonne tenue en charge d'un serveur.

Ce premier axe de choix est évidemment corrélé au stade du développement au cours duquel on souhaite mener ces tests. Aux stades de conception, lorsqu'aucune ligne de code n'est encore produite, il ne peut s'agir que de tests basés sur l'analyse des documents de conception à l'aide de revue ou d'inspection. Lorsque le processus de développement avancera, il sera possible de choisir entre une des techniques de test dynamique et une technique de test statique (incluant les revues et les inspections).

Si l'on dispose d'un logiciel exécutable partiellement ou totalement, on pourra mettre en place des tests dynamiques. Les critères de choix incluront :

- L'accès au code en plus de l'accès aux spécifications ;
- L'accès à des cas d'utilisation avec les spécifications ;
- Un nombre de paramètres (au sens large) faible et avec un nombre de valeurs possibles pour ceux-ci, faible ;
- L'objectif des tests en termes de validation du comportement.

Selon le cas, il faudra ou non, procéder à une première phase d'analyse et d'abstraction afin de réduire la combinatoire et se ramener à un nombre de cas de tests compatible avec l'effort planifié. Cette phase est une étape primordiale dans la qualité des tests produits : un excès d'abstraction conduira à amalgamer des cas distincts et laisser une part du logiciel non testé, un manque d'abstraction fera jouer et rejouer de nombreuses fois le même test. Cette étape est nécessairement manuelle (au vu des connaissances actuelles). Par contre, une fois les cas de tests sélectionnés et les valeurs concrètes d'exécution choisies, on essaiera dans la mesure du possible d'automatiser au maximum le jeu des tests (en prenant garde d'inclure l'analyse des résultats dans le processus d'automatisation) à l'aide d'un des nombreux outils dédiés au sujet.

Ensuite, selon l'objet visé, il faudra sélectionner le type de tests appropriés : tests de performance, test de charge, tests d'intrusion, tests fonctionnels, etc. Deux stratégies s'opposent : tester plusieurs caractéristiques avec un même

test ou spécialiser le plus possible chaque test. Dans le premier cas, la factorisation permet de réduire le nombre de tests à passer, mais peut conduire à masquer des défauts ou rendre difficile leur localisation. On préférera donc la seconde stratégie qui peut augmenter sensiblement l'effort initial, mais qui, s'avérera payante dans la durée en particulier si l'automatisation a bien été menée.

Si par contre le logiciel n'est pas exécutable ou que l'on cherche à tester une propriété qui ne relève pas du comportement dynamique (comme la facilité de maintenance, le respect d'une recommandation officielle) il sera nécessaire de mettre en place une revue ou une inspection.

De façon générale, une revue ou une inspection « coûte » relativement cher car elle mobilise un nombre élevé de personnes (de cinq à dix, selon le degré de formalisation de cette activité) pendant une période allant de quelques heures à plusieurs jours voire plusieurs semaines. En contrepartie, lorsqu'il est possible de choisir entre une inspection et des tests dynamiques, une inspection apportera généralement une meilleure réponse en qualité qu'un procédé de test dynamique.

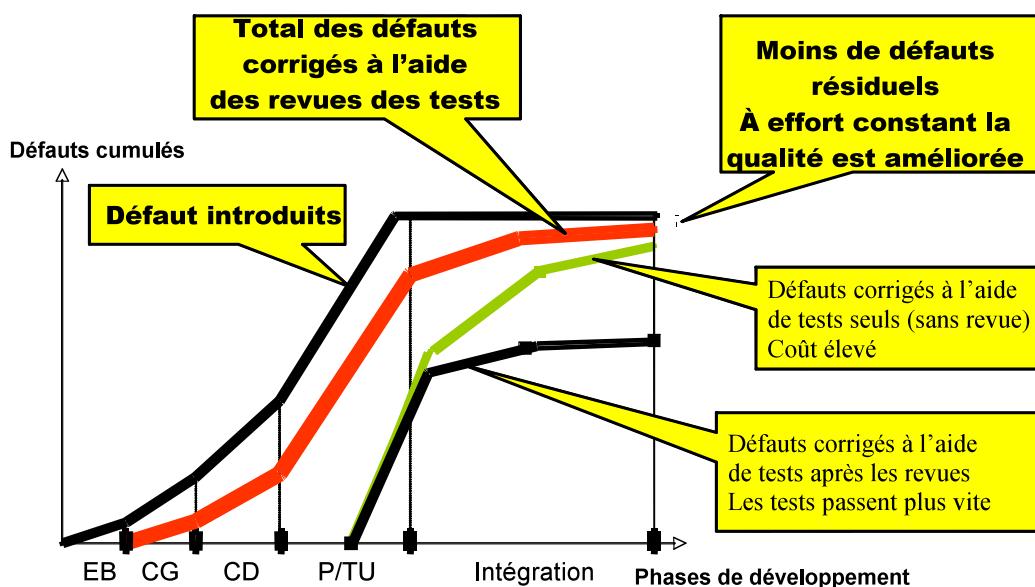


Fig. 7.2 Impact des inspections sur le nombre de défauts découverts

Le graphique précédent (figure 7.2), détaille ainsi l'évolution typique du nombre de défauts cumulés selon les différentes phases d'un projet

(Expression des Besoins, Conception Générale, Conception Détailée, Programmation et Tests Unitaires, Intégration). Le nombre cumulé de défauts introduits croît en fonction de l'avancée du projet pour atteindre un palier lors de l'intégration, phase où l'on ne modifie plus le logiciel. Le nombre de défauts découverts croît également avec le temps pour atteindre une valeur maximale qui dépend de l'efficacité du processus de test. Cette valeur définira le nombre de défauts résiduels par différence avec le nombre total de défauts introduits ; on notera qu'il ne s'agira généralement que d'une valeur estimée car il est impossible de prédire avec exactitude le nombre de défauts qui seront introduits et qui resteront présents lors de la mise en service. L'utilisation de revues lors des différentes phases du projet permet de réduire le nombre de défauts découverts par les tests dynamiques ; la complémentarité des deux approches joue à fond : certains des défauts, plus accessibles à une analyse statique, sont détectés lors d'une revue, d'autres, moins accessibles à la pure analyse, sont trouvés lors d'un test dynamique.

Puisque les revues augmentent le nombre de défauts découverts, elles diminuent mécaniquement le nombre de défauts résiduels, ce qui permet à qualité constante de réduire les tests à passer. Le coût des revues et inspections est alors compensé par la réduction des tests et des coûts induits. Malheureusement, seule l'expérience du chef de projet permet de doser l'effort relatif des deux stratégies.

Si les revues sont efficaces, elles ne sont cependant pas applicables à tous les objectifs de tests. Ainsi, elles seront très difficiles à utiliser seules pour les objectifs de test suivants : tests de charge, tests de performance, tests d'acceptation et de façon générale tous les tests système. De même, l'approche par revue n'est pas spécialement bien adaptée aux tests fonctionnels et est complètement inadaptée aux tests d'acceptation.

Par contre, on privilégiera les revues pour la détection de défauts dans les documents de conception, dans les spécifications (fonctionnelles ou non fonctionnelles), dans les choix d'architecture, dans des choix algorithmiques, de synchronisation ou de protection des données. On pourra également utiliser un processus de revues pour évaluer la capacité d'un logiciel à être maintenu simplement, ou à supporter d'éventuelles évolutions.

7.1.2. Les différents acteurs

De nombreux acteurs participent directement ou indirectement à la bonne tenue des tests à différents niveaux et avec différents impacts.

En amont des phases de développement (voir chapitre 2) la notion de « testabilité » doit être présente lors de la définition des exigences et doit guider les choix d'architecture qui seront faits. Les acteurs concernés sont les architectes du projet au sens large (expression du besoin, spécifications fonctionnelles, conception générale), ce qui inclut le client.

La réalisation des différents tests (unitaires, intégration, système, validation) sera sous la responsabilité d'un chef de projet qui aura pour rôle de planifier les tâches, de répartir les moyens (humains et matériels). Il devra également choisir les stratégies et les méthodes les mieux adaptées aux objectifs visés, qui dépendent des types de tests menés et de l'étape à laquelle ils sont réalisés.

Il faut ensuite citer les développeurs qui sont en charge des phases de conception détaillée et de programmation. Ce sont eux qui mettent en place et réalisent les tests unitaires. Des outils comme JUnit par exemple, pour le langage Java, et des environnements de développement comme Eclipse aident à automatiser au maximum cette tâche et à décharger le développeur d'actions répétitives et consommatrices de temps et d'énergie. Ce qui ne l'exonère pas de concevoir les « bons » cas de tests et les « bons » jeux de valeurs (voir chapitres 3 et 4). La programmation par paire, comme préconisée dans les méthodes agiles, permet d'optimiser la qualité des tests menés et donc des composants développés. Leur rôle est fondamental car les tests d'intégration et les tests systèmes ne pourront être correctement menés que si les tests unitaires ont permis d'éliminer le maximum de défauts présents dans les composants élémentaires du logiciel.

L'équipe d'intégration, qui sera de préférable indépendante, aura pour rôle de vérifier que l'interaction des composants est conforme aux spécifications. Cette tâche ne pourra être menée correctement que si, d'une part, chaque composant a un comportement conforme à ses spécifications, et si, d'autre part, les interfaces logicielles et matérielles ont été précisément définies (voir chapitre 6).

L'équipe de validation sera nécessairement indépendante car les tests de validation seront menés avec une vue « client » en fonction des besoins

métiers. Dans certains cas, ces tests de validation pourront être menés par le client lui-même avec ou sans l'aide de la maîtrise d'œuvre. Dans le cas de tests non fonctionnels, il faudra faire appel à une équipe d'experts du domaine qui rendra compte au chef de projet.

On notera que ces derniers acteurs ne peuvent introduire directement des défauts dans le logiciel, contrairement aux premiers acteurs mentionnés, même si la mise en évidence de « faux » défauts peut induire indirectement une régression en terme de qualité.

Enfin, il faut citer la direction en général qui devra allouer les moyens nécessaires à la mise en œuvre des tests et qui pourra mettre en place une stratégie de qualité visant à reporter une partie de l'effort mis sur le support et la maintenance de premier niveau vers les différents processus de test.

7.2. Planifier les tests

Comme on l'a vu aux premiers chapitres de ce livre, l'effort de test ne doit pas être porté de façon homogène dans le temps et dans l'espace : certaines parties du logiciel devront être testées beaucoup plus en détail que d'autres (règle des 80/20) et tous les types de tests ne pourront être employés à toutes les phases de conception et de développement. De même il sera nécessaire de déterminer précisément quels sont les critères objectifs qui permettront de juger de l'avancement des tests et de déterminer l'arrêt ou la poursuite des tests en cours.

La planification des tests a pour rôle de répondre à ces questions en fixant une stratégie permettant d'atteindre les objectifs fixés. Elle impose en particulier de définir ce qui doit être testé, et à quel niveau de détail, de définir les types de test à mettre en œuvre, de définir des critères d'entrée et de sortie. La planification vise également à aligner les ressources aux différentes tâches définies. Enfin, lors de la planification des tests, on veillera à préciser et mettre en place la structure et les modèles de documentation à produire.

L'ensemble de ces données sera explicité dans un « Plan de tests » qui selon les standards IEEE (IEEE Std 829-1998) contiendra :

- Un identifiant qui permet de référer sans ambiguïté ce document. Il faut garder à l'esprit que ce document est amené à être révisé afin d'évoluer avec les versions du logiciel ; il est donc important qu'il possède un numéro de révision et qu'il soit géré en configuration avec les autres documents relatifs aux tests.
- Les références aux documents mentionnés dans le plan de test comme les documents de spécifications, les normes qualité ou les méthodologies en vigueur. Il ne faut pas dupliquer ici ces documents mais simplement préciser leur référence (avec le numéro de révision) afin de ne pas alourdir l'effort de maintenance.
- Une introduction décrivant l'objet du plan de tests et en faisant référence, si nécessaire, à d'autres plans de tests liés à celui-ci.
- Les éléments devant être testés à travers ce plan de tests au niveau architecture technique (composants, modules, classes, interfaces, etc.).
- Les fonctionnalités devant être testées du point de vue de l'utilisateur ; la différence avec l'item précédent est le point de vue qui est ici utilisateur et non technique : processus métier *versus* processus informatique.
- Les fonctionnalités ne devant pas être testées avec, là encore, un point de vue utilisateur mais incluant les différentes versions possibles. Les raisons de ne pas tester une fonctionnalité peuvent être multiples comme un risque associé faible, l'absence de la fonctionnalité considérée dans la version du logiciel sous tests ou l'assurance par d'autres moyens de la correction de la réalisation de cette fonctionnalité. Comme toujours, le choix de ne pas tester une fonctionnalité doit être évalué par rapport au risque encouru en cas de défaut non détecté.
- La stratégie de test associée à ce plan de tests : en particulier il faudra faire ici le lien avec d'autres plans de tests qui réfèrent celui-ci ou sur lequel il s'appuie. Il faudra également définir les règles et les procédés mis en œuvre à travers ce plan de test, le rythme et la forme des réunions, les métriques utilisées, la façon de gérer en configuration les tests, etc.
- Les critères de sorties, c'est-à-dire les mesures et les conditions associées permettant de décider s'il faut arrêter ou poursuivre les tests.
- Les critères pouvant conduire à une suspension (arrêt temporaire) des tests.
- Les documents produits par ce plan de tests : les cas de tests, les jeux de

valeurs utilisés, les traces d'exécution, les problèmes détectés, en gros tout ce qui peut être utilisé pour analyser les tests et prendre les décisions appropriées ; le seul document qui n'a pas à être produit comme un élément de ce plan de tests est le logiciel à tester lui-même.

- Les activités de tests qui resteront à mener après avoir réalisé ce plan de tests.
- L'environnement nécessaire pour mener à bien ce plan de tests en termes de matériel, logiciels et expertises humaines ; on détaillera ici également non seulement les logiciels ou librairies nécessaires mais aussi les versions de ceux-ci. Il faudra prendre garde de décrire aussi les restrictions d'usage du système hôte lors des tests afin de créer les conditions optimales d'interprétation des résultats.
- La formation nécessaire à l'exécution de ce plan de tests
- Les responsabilités au niveau de ce plan de tests à tous les points de vue : évaluation des risques, décision de formation, clôture des tests, validation de l'environnement de tests, l'action face à un point non traité dans ce plan de tests, la résolution des conflits sur un partage de ressources, etc.
- L'ordonnancement des activités qui planifiera les différentes étapes de tests. Il sera préférable de définir des dates relatives comme : « L'activité de tests d'intégration débutera le jour d'après la livraison du composant X » plutôt que de donner des dates fixes. En effet, de la sorte, le retard d'une activité ne pourra pas être attribué à une tâche qui en dépend. Cela permet de réduire, sans l'éliminer complètement, la probabilité de l'abandon des activités de tests lors d'un retard du développement.
- La planification des risques et des contingences, en particulier sur le manque de ressources humaines ou matérielles pour les phases de tests, sur des retards induits par des livraisons tardives ou des formations décalées, modification sur des exigences. Pour chacun de ces événements possibles (ou du moins les plus probables) il convient de définir les actions à mener comme le décalage des livraisons si cela est possible, le déplacement de ressources, l'augmentation du temps de travail des équipes ou encore la réduction des objectifs du plan de tests.
- La liste des personnes qui peuvent approuver le plan de tests et décider de passer à une étape suivante.

- Un glossaire qui précise les termes et les acronymes utilisés au travers de ce document.

Il ne faudra pas oublier, lors de la conception et de la rédaction de ce plan de tests, de prendre en compte le public visé. En effet, selon que le destinataire est le client ou l'équipe de tests, l'accent sera plutôt mis sur les délais, sur la stratégie utilisée ou sur les détails techniques. En effet, dans le premier cas, l'objectif est un but d'information de la méthode de validation utilisée alors que dans le second cas, il s'agit de permettre une mise en œuvre concrète des tests.

7.3. Définir et évaluer les critères de sorties

Le test d'un logiciel ne peut être exhaustif et les ressources ne sont pas infinies. L'équipe de test est donc conduite à faire des choix : choix des composants à tester, choix du moment pour faire ces tests, choix des techniques à utiliser, choix des scénarios de tests mais également choix, à certains moments, de poursuivre ou d'arrêter les tests. La pertinence de ces choix influence directement la qualité de l'activité de test ; en particulier, décider de poursuivre les tests, alors qu'ils auraient pu être arrêtés, va entraîner un effort inutile. A contrario, arrêter trop tôt les tests conduira à livrer pour l'intégration, pour acceptation ou pire, pour mise en service, un logiciel contenant de nombreux défauts qui auraient pu être éliminés. Cela entraînera par effet mécanique une perte de qualité et un coût au final plus élevé (plus un défaut est découvert tard plus il coûte cher). Il est donc important d'essayer d'arrêter au moment « optimum ». Bien qu'il n'y ait pas dans ce domaine de « recette miracle », nous citons deux de ces critères admis comme pertinents dans ce contexte et permettant de sortir « positivement » de la phase de test.

Le premier de ces critères est le plus simple puisqu'il correspond à l'achèvement du plan de tests : tous les tests prévus et planifiés ont été correctement exécutés. Si la réflexion initiale était de qualité, en s'appuyant par exemple sur une des méthodes évoquées aux chapitres précédents, il est raisonnable de penser que la poursuite des tests n'améliorera pas forcément le nombre total de défauts découverts.

Le second de ces critères correspond à l'accès à un palier prédéfini du nombre de défauts découverts par unités de temps et normalisé vis-à-vis de la taille de l'équipe de tests (pour réutilisation ultérieure de ces données). Ce critère ne peut être utilisé qu'à la condition de maintenir des statistiques précises du nombre de défaut mis en évidence au cours des tests. Pour qu'il soit pertinent, il est nécessaire d'avoir une forte maturité sur ce type de suivi et d'avoir un modèle d'estimation du nombre de défauts présents dans le logiciel. Ce modèle peut être par exemple basé sur l'injection de fautes dans le logiciel : on « injecte » dans le logiciel un certain nombre de défauts (disons N_e), puis on compte le nombre de défauts injectés qui ont été détectés (disons N_d). Une approximation du nombre de défauts présents dans le logiciel est alors obtenue grâce au calcul du produit du rapport (N_e/N_d) par le nombre de défauts déjà découverts ; le rapport (N_e/N_d) permettant d'estimer la pertinence des tests vis-à-vis la détection de défauts. D'autres moyens, basés sur des statistiques menées sur d'anciens projets pourront également être utilisés.

Il existe bien d'autres critères de sortie dont certains correspondent plus à un « abandon » qu'à une fin positive :

- Expiration des délais et obligation de livrer le logiciel ;
- Fin du budget ou déplacement des ressources sur un autre projet ;
- Explosion du nombre de défauts trouvés signifiant un logiciel mal conçu et ne pouvant visiblement pas être amélioré ; dans ce cas, le logiciel est généralement abandonné ;
- Acceptation dans l'état du logiciel par le client.

Dans chacun de ces cas, les tests sont arrêtés car il apparaît que d'une façon ou d'une autre le projet devra évoluer et qu'il faudra passer d'une phase de tests à une phase de maintenance ou d'archivage.

7.4. Estimer l'effort de test

Estimer les ressources à allouer pour la phase de tests est une étape indispensable ; c'est évident si la validation est conduite en interne mais c'est aussi primordial dans le cas où cette activité est externalisée afin de pouvoir

évaluer la pertinence des devis faits et des demandes complémentaires de travaux.

La première chose à faire est d'estimer le coût global du projet, à l'aide d'une méthode adaptée^[1] comme COCOMO (*COnstructive COst MOdel*) de Barry Boehm ou la méthode des Points de Fonctions.

Il est important d'éviter des efforts superflus en matière d'estimation des coûts lorsque des résultats très approximatifs suffisent. Le schéma suivant (figure 7.3) résume ce procédé. Dans celui-ci, le modèle d'estimation produit et utilise des grandeurs « CQFD », dont la signification est rappelée brièvement.

- **Coût** : cette grandeur est généralement fixée par le client lui-même dès le début du projet. Le coût détermine l'effort jugé nécessaire pour réaliser le logiciel ; il s'exprime en « homme années » ou en « homme mois ». Le paramètre coût peut être imposé par le MOA.

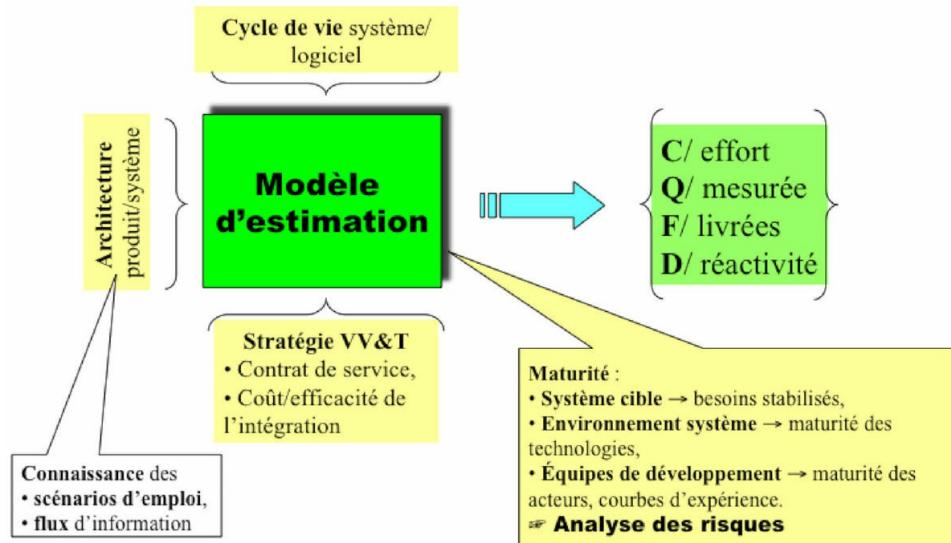


Fig. 7.3 Les paramètres de l'estimation

- **Qualité** : elle dépend des actions du chef de projet MOE, et en particulier de l'effort de vérification, validation et test (VVT) ; en théorie, elle est fixée dès que le plan qualité est approuvé, généralement en début de projet. Il est particulièrement malvenu et maladroit de

réviser la qualité à la baisse en cas de retard ! La VVT est fonction de ce qui est réellement exécuté par la plate-forme (*i.e.* les instructions écrites et celles générées)

- **Fonctionnalités** : cette grandeur caractérise le service rendu (*i.e.* les fonctions offertes) tel que proposé par le maître d'œuvre à son client ; les fonctionnalités peuvent souvent être négociées en contrepartie du coût et du délai ; s'expriment en nombre de points de fonctions (PF) ou en nombre de milliers de lignes source (KLS). On ne compte que ce qui est réellement écrit par les programmeurs.
- **Délai** : cette grandeur est souvent fixée par le client qui en général synchronise le travail avec d'autres projets ; le délai peut varier en cours de projet. Pour tout projet, il existe un délai optimum dit aussi « temps de cuisson ».

Cette première étape d'estimation des coûts conduit à un ordre de grandeur en charge de travail et en délais de réalisation. Partant de cette donnée, il sera nécessaire d'appliquer un ratio sur l'estimation du coût global pour obtenir une première estimation du coût des tests ; les ratios communément admis à ce jour font apparaître une valeur allant de vingt à quarante pour cent selon les études. Il faut, bien entendu, prendre en compte les risques associés aux différents composants ou fonctionnalités et pondérer l'effort de test en fonction de ces risques.

Pour mémoire, dans le modèle initial COCOMO, l'effort en homme par mois (152 heures de travail par mois) associé à un projet est estimé à l'aide d'une formule de la forme :

$$Effort = k \times (KLS)^{1+\alpha}$$

où k , qui est un facteur linéaire de l'estimation, désigne une constante dépendant du type du projet^[2], où KLS désigne le nombre de ligne du code (par milliers, sans les commentaires et les lignes blanches) et où enfin, α désigne le facteur d'intégration, valeur dépendant également du type de projet (de 1,05 pour un projet de type S à 1,2 pour un projet de type E) ; on remarquera que ce dernier terme intervient de façon non linéaire dans le calcul de l'effort. Ce modèle permet également d'estimer le temps de développement $TDev$ (en mois) à l'aide de la formule :

$$TDev = 2,5 * (Effort)^{0,3 \pm \varepsilon \pm \varepsilon}$$

Si ce type de méthode basé sur la modélisation est reconnu, on pourra également utiliser toute autre méthode adaptée à l'estimation des coûts comme les méthodes basées sur l'analogie, sur l'expertise, sur le consensus de groupe (comme la méthode du « planning poker » utilisée dans les développements agiles), ou encore basée sur la loi de Parkinson^[3] (un projet coûtera ce que vous avez décidé de dépenser), sur le prix du client (un projet coûtera ce que le client est disposé à dépenser), etc. Chaque technique a ses avantages et ses inconvénients, mais le plus important, est qu'aucune technique ne peut être appliquée de façon exclusive et que pour les grands projets ; il est préférable d'en appliquer plusieurs en parallèle. Si celles-ci prédisent des coûts radicalement différents, on en déduira qu'il n'y a pas assez d'information.

De façon plus détaillée, le modèle COCOMO retient les valeurs suivantes pour la répartition de l'effort en fonction de la phase du projet et de son type (voir le tableau 7.1).

Tab. 7.1 Répartition de l'effort en fonction des types de projet

Typologie	Phase	Taille du projet (en KLS)		
		32	128	512
S	Expression du besoin et planification	6	6	
	Conception générale	16	16	
	Réalisation	62	59	
	<i>Conception détaillée</i>	24	23	
	<i>Codage et tests unitaires</i>	38	36	
	<i>Tests et intégration</i>	22	25	
P	Expression du besoin et planification	7	7	7
	Conception générale	17	17	17
	Réalisation	58	55	52
	<i>Conception détaillée</i>	25	24	23
	<i>Codage et tests unitaires</i>	33	31	29
	<i>Tests et intégration</i>	25	28	31

E	Expression du besoin et planification	8	8	8
	Conception générale	18	18	18
	Réalisation	54	51	48
	<i>Conception détaillée</i>	26	25	24
	<i>Codage et tests unitaires</i>	28	26	24
	<i>Tests et intégration</i>	28	31	34

Ainsi, selon ce modèle d'estimation, il faudra consacrer au moins 50 % de l'effort total du développement aux tests pour un projet de type S (38 % pour les tests unitaires et 22 % pour les tests d'intégration).

Une fois cette première estimation obtenue, il va falloir l'affiner à partir d'éléments rationnels du projet puis essayer d'évaluer cet effort dynamiquement au cours de l'exécution des tests. On cherchera à déterminer si ceux-ci peuvent être réduits (par exemple, en cas de densité de défauts très faible ou montrant une forte diminution au cours des phases de test) ou au contraire doivent être revus à la hausse (densité de défauts stable par exemple ou découverte de difficultés mal évaluées). On notera que dans ce dernier cas, il peut être astucieux, non d'augmenter l'effort de tests, mais de réduire le nombre de fonctionnalités livrées ou de prévoir une version « contrôlée » du logiciel qui modifie son profil d'utilisation.

7.5. Gérer les tests en configuration

Les tests sont une production qui est amenée à évoluer au fil des versions du logiciel (qu'elles soient évolutives ou correctives). Le logiciel étant géré en configuration, il va falloir faire le même effort pour les tests. En effet, il peut être nécessaire à tout moment de pouvoir reconstituer une version complète et cohérente des tests pour une version donnée du logiciel. Cela implique de pouvoir retrouver quels sont les environnements nécessaires en matériel et en logiciel (avec les bonnes versions !). Seule une gestion sérieuse en configuration des tests permettra d'atteindre cet objectif.

S'il est simple d'en affirmer le besoin, la mise en place de cette gestion pose les mêmes problèmes que pour toute gestion en configuration :

- Nombre d'objets à gérer ;
- Variété des objets : scénarios, cas de tests, jeux de valeurs, rapports d'inspection, paramètres des environnements d'exécution, etc.
- Variété des supports d'archivage et de stockage ;
- Organisation du développement et des tests ;
- Les acquisitions en logiciel et matériel nécessaires à la mise en place des tests.

La première étape va consister à identifier les éléments (articles) à placer sous contrôle ; dans le cadre des tests, ces éléments seront les scénarios, les jeux de valeurs, les objectifs ainsi que les liens vers les spécifications ou les codes sources des tests pour une version donnée du logiciel. Il faudra également préciser, si besoin, les environnements matériels nécessaires pour le déroulement des tests. De plus, lorsqu'un défaut sera mis en évidence par un test, il faudra également créer une fiche d'anomalie qui sera ouverte et qui devra être clôturée par une annonce de modification corrective et un test de confirmation de correction.

Une fois ces éléments définis, il faut préciser quels sont les acteurs possibles vis-à-vis de ceux-ci et quels sont leurs droits.

Schématiquement, les solutions consistent à définir des espaces de travail authentifiés (au niveau des individus ou des groupes), protégeant contre toute évolution non autorisée. Ces solutions, bien outillées à ce jour, permettent une reconstitution des données en cas de besoin et la gestion contrôlée de différentes copies qui sont sous contrôle.

Les modifications des documents placés sous gestion de configuration ne peuvent se faire qu'avec le respect des procédures prédéfinies par le chef de projet. Ces modifications sont alors nommées révisions et possèdent un identifiant unique. On associe presque systématiquement à une révision son auteur, la date de celle-ci ainsi que la description de la cause à l'origine de cette révision.

7.6. Conclusion

Si l'effort de test est variable selon le type de projet et la façon dont il est mené, il semble acquis que la proportion des coûts associés aux tests ne va

pas aller en diminuant dans le futur. En effet, la réutilisation de composants réduit le coût global, mais paradoxalement, conduit à augmenter le taux de tests de non-régression et d'intégration ce qui, mécaniquement, conduit à augmenter la proportion des tests dans le projet global. Le modèle COCOMO2 prend d'ailleurs en compte spécifiquement cette façon de procéder dans son modèle d'estimation.

La planification des tests est un des rôles essentiels du chef de projet en charge des tests. Celle-ci prendra d'autant plus d'importance que le projet reposera sur de nombreux acteurs et fera appel à différents équipements spécifiques. Il faudra prendre garde en particulier à définir des critères précis autres que l'épuisement du temps pour décider de l'arrêt des tests.

Par ailleurs, l'évolution des modèles de développement vers des démarches basées sur l'agilité, qui procède par itération et incrément, fait que la gestion en configuration des tests devient un enjeu crucial. Caricaturalement, on peut affirmer que sans gestion de configuration des tests, il ne peut y avoir d'agilité dans le développement !

Notes

[1] Voir par exemple l'ouvrage *Coûts et durée des projets informatiques : Pratique des modèles d'estimation*, Jacques Printz , Christiane Deh, Bernard Mesdon, Nicolas Trèves, Hermes, 2001.

[2] Pour un projet de type Simple, type S, k vaut 2,4. Pour un projet de type semi détaché, type P, k vaut 3,0 et enfin pour un projet Embarqué, de type E, k vaut 3,6.

[3] La loi de Parkinson fut exprimée en 1957 par Cyril Northcote Parkinson dans son livre *Les Lois de Parkinson* (C. Northcote Parkinson, Parkinson's Law, or The Pursuit of Progress, John Murray, London, 1957) et est basée sur une longue expérience dans l'administration britannique et stipule que « *le travail s'étale de façon à occuper le temps disponible pour son achèvement* ». On trouve assez facilement cet ouvrage sous forme numérique.

Outils pour les tests

Comme pour toute activité humaine, l'utilisation d'un outil dans le cadre du test d'un logiciel permettra de gagner en qualité et en efficacité (les tests seront meilleurs et permettront de détecter plus de défauts) mais permettra aussi de réaliser des gains en efficience (les tests seront menés avec la même efficacité mais avec moins de ressources et en moins de temps).

Par ailleurs, on notera que si pour certains types de tests l'utilisation d'un outil est une option elle devient une quasi-obligation pour toute une frange des tests. C'est le cas d'une grande part des tests dynamiques : tests de charge, de performances, de stress ou de robustesse ; c'est aussi le cas pour la plupart des tests structurels qui sont basés sur une analyse automatique des codes des programmes. C'est aussi le cas si l'on veut réellement mettre en œuvre des tests de non-régression dont l'intérêt est de pouvoir être refait très facilement dès que l'on modifie le logiciel pour corriger un défaut ou pour ajouter une fonctionnalité.

Cependant, comme pour l'utilisation de tout outil, la transformation d'une activité manuelle en une activité partiellement ou complètement automatisée à l'aide d'un outil qui a sa propre logique et ses propres contraintes va induire des transformations plus profondes au sein de l'entreprise que la simple introduction de l'outil, transformations, positives ou négatives qu'il ne faudra pas sous-estimer.

L'objectif de ce court chapitre est de faire le point sur ce domaine en l'abordant sous ses différentes facettes.

8.1. Typologie attendue des outils de tests

Les outils de tests sont nombreux et ne cessent d'évoluer ; il est donc peu utile de tenter d'en dresser une liste exhaustive qui sera, par nature, périmée à peine définie. Pour autant, si les outils évoluent et se renouvellent au fil du

temps, les principes qui les définissent et les objectifs qui les sous-tendent restent à peu près stables. Il s'agit en effet d'automatiser certains processus manuels qui, dans le cadre des tests logiciels, ont relativement peu évolué. C'est donc sous cet angle que nous explorerons ce domaine en essayant de comprendre comment une activité de test peut se prêter à une automatisation ; nous essaierons également de présenter brièvement les solutions qui, lors de l'écriture de cette seconde version du livre, peuvent être vues comme les références du domaine.

Mais avant cela, il nous faut lister les bénéfices que peut apporter la mise en place d'un outil sans négliger les risques qui pourraient provoquer un rejet de l'outil et une régression dans la qualité du travail mené.

Ainsi, les bénéfices attendus incluent :

- La réduction de tâches répétitives en particulier dans le cas des tests de non-régression.
- Une meilleure qualité de tests produits : l'outil utilisé permet de faire mieux car il « embarque » un savoir-faire et une expérience importante et guide les activités par la mise en place de processus métiers ; il permet également de se concentrer sur les tâches les plus importantes en dégageant du temps sur la réalisation de tâches répétitives, techniques mais sans réelle valeur ajoutée.
- Une capitalisation plus simple et plus efficace : par nature un outil peut aisément enregistrer ses actions et les résultats de celles-ci ; il peut également faciliter l'ajout de commentaires sur ces actions de sorte à ne pas commettre les mêmes erreurs lors des prochains tests ou de sorte à conserver les raisons qui ont mené à réaliser tel ou tel test.
- La définition de critères objectifs en particulier grâce au calcul de mesure sur les parties de code testés, sur le nombre de défauts trouvés par unité de code, par unité de temps ou par unité de réalisation ; on peut également mesurer le temps moyen de correction d'un défaut.
- Une meilleure communication entre les différents acteurs par l'aide à la production de documents de synthèse ; différents outils, en particulier ceux d'intégration continue, permettent d'offrir des tableaux de bord complets sur les différentes facettes de l'état d'avancement d'un logiciel, des équipes impliquées, des différentes échéances, etc.

- Enfin, la réalisation de tests (quasi) impossible manuellement comme les tests de charge ou de stress qui nécessitent la mise en œuvre d'actions très techniques et/ou basées sur des notions statistiques avancées.

S'il est donc clair qu'un outil adapté et bien utilisé apporte de nombreux bénéfices à la réalisation des différentes activités liées aux tests, il ne faut pas sous-estimer les risques inhérents à l'introduction d'un outil dans une organisation. Parmi les plus importants, nous pouvons citer :

- La sous-estimation du temps nécessaire à la mise en place de l'outil et à son utilisation de manière efficace ; il s'agit là d'un défaut d'optimisme fréquent, encouragé par ailleurs par les ingénieurs d'affaires ou autres commerciaux, qui peut conduire à un véritable désastre en termes de mobilisation des équipes techniques et des services métiers.
- Le mauvais choix d'outil ; bien que cela semble une erreur grossière et simple à éviter, il peut arriver que du fait d'inadéquations techniques de besoins mal estimés, ou en encore de décalage en terme de connaissance nécessaires et maîtrisées, le choix se porte sur un outil inutile pour l'entreprise ou très difficile à utiliser.
- Une perte en qualité (valeurs générées, processus en place) ou une désorganisation profonde des équipes ; en effet, la plupart des outils sont conçus pour répondre aux besoins du plus grand nombre et supposent un modèle d'organisation donné. Dans le cas où l'entreprise, du fait de son histoire, de son savoir-faire ou de son marché, a mis en place un modèle de fonctionnement spécifique, l'outil choisi peut se révéler être inadaptable à ce modèle ; l'utilisation de l'outil risque alors de se heurter au mode de fonctionnement en place et créer des tensions et des erreurs qui vont faire régresser la qualité des tests.

Enfin, pour la plupart des besoins, l'on pourra se tourner vers une solution « propriétaire » ou vers une solution « Open Source ». Il n'est pas nécessaire de rappeler qu'ouvert ou libre ne veut pas dire gratuit et que payer cher une solution garantit une adéquation aux besoins.

8.2. Les grandes familles d'outils

8.2.1. Outils pour les tests statiques

Les outils de tests statiques sont sans doute les plus anciens ; ils sont nés avec les premiers compilateurs et ont tiré profit des avancées de la formalisation de la sémantique des langages de programmation, en particulier avec l'introduction du typage des données qui a permis d'offrir des moyens simples de vérifier très tôt que le programme ne contient pas trop d'erreur. Par exemple, la confusion sur le nom d'une variable ($x = y$ au lieu de $x = z$) sera plus facilement détectée avec les types qui limitent les « mélanges » entre valeurs de types différents. De même les notions de déclaration et de portée des déclarations aident le compilateur, ou un outil d'analyse de code, à détecter d'éventuelles erreurs de codage, et donc d'éventuels défauts du programme.

Ainsi, pour peu que le programme soit réalisé avec certaines règles qui aident à sa compréhension, et que ces règles^[1] soient conçues dans le but de faciliter la détection des erreurs communes, un outil pourra analyser le code du programme et vérifier que ces règles sont bien respectées et ainsi signaler un certain nombre de problèmes potentiels à l'exécution du logiciel.

De nombreux efforts de recherche ont également permis d'étendre la notion de test statique à la notion de vérification ou de preuve de propriétés du programme dépassant alors les objectifs initiaux des tests en fournissant une preuve effective du respect par le programme d'une propriété ou d'un ensemble de propriétés. Cependant, très vite, la complexité intrinsèque à ces vérifications rend difficilement utilisable en pratique ces outils sur des programmes un peu complexes.

Dans cette grande famille d'outils d'analyse statique, il existe de nombreux logiciels « libres », c'est-à-dire qu'ils peuvent être installés et utilisés librement et qu'en plus il est possible d'en obtenir les sources et, si l'on en a les compétences, d'adapter le logiciel utilisé à son contexte propre en modifiant les sources. Parmi les plus utilisés, ou les plus connus, nous pouvons citer les logiciels Lint, Slint, Blast ou encore Coccinelle pour les langages C et C++, ou FindBugs et Checkstyle pour le langage Java^[2].

Ce créneau est également bien occupé par les grandes sociétés du domaine avec des outils comme Logiscope de TeleLogic (initialement conçu par la

SSII Verilog et maintenant distribué par IBM), DevPartner Code Review de Borland ou encore les outils de la suite Rational d'IBM pour ne citer que les outils les plus populaires.

Le principal souci de ces outils est la présence, parfois en très grand nombre, de « faux positifs » c'est-à-dire d'éléments du programme indiqués par précaution comme des potentielles erreurs. Ainsi parmi les messages d'alerte, certains vont refléter de véritables erreurs alors que de nombreux autres n'indiqueront que des éventuelles erreurs dont la survenue est impossible du fait de contraintes d'exécutions non capturées par l'analyseur. Le testeur doit alors faire le tri manuellement entre les faux et les véritables cas à problème ce qui se révèle être une tâche relativement complexe. Néanmoins, l'expérience montre que l'usage d'un tel outil est profitable dans le temps et permet d'améliorer significativement la qualité du logiciel en réduisant fortement les défauts de comportement^[3].

Dans cette catégorie d'outils d'analyse statique, on peut trouver de très nombreux produits qui permettent de cibler des caractéristiques particulières du code à l'aide du calcul de métriques (voir le chapitre 6). Citons Logiscope de Telelogic, ou la suite Visual Studio de Microsoft qui inclut de nombreuses options de mesure du logiciel ou encore la suite d'outils d'analyse statique « *Klocwork insight* » de la société canadienne Klocwork.

Enfin, les outils d'analyse statique sont également utilisés pour mesurer la couverture du code exercé par des tests dynamiques et vérifier ainsi la pertinence des cas de tests vis-à-vis de ce qui a été codé. Du code non couvert lors des campagnes de test va mettre en lumière soit des parties de code « mort » (i.e. du code qui ne sert pas), soit des parties de code défensif (utilisé pour prendre en compte des comportements exceptionnels), soit encore le manque de complétude des cas de tests imaginés (les tests n'exercent pas certaines parties du code). Bien qu'il soit illusoire, et souvent inefficace, de couvrir 100 % du code, cette notion de couverture permet de suivre l'évolution d'une campagne de tests ou des différentes versions du logiciel.

8.2.2. Outils d'aide à l'exécution des tests dynamiques

Si les tests statiques ont montré leur utilité à travers le temps, ils ne permettent de détecter que certains types d'erreurs pouvant être capturées par

la seule analyse du code. Par ailleurs le problème étant d'une complexité théorique et pratique réelle, ces tests statiques ne peuvent être utilisés seuls dans le cas d'un projet, et ils sont systématiquement complétés par des tests dynamiques. Ces tests, basés sur l'exécution du programme pour certaines valeurs bien choisies, sont plus pragmatiques et ont l'avantage d'ignorer la complexité du code (et la façon dont il est conçu). Ils se concentrent sur la manière dont le logiciel répond concrètement au problème pour lequel il a été imaginé puis développé : répond-il aux exigences fonctionnelles, y répond-il dans un temps raisonnable, est-il facilement utilisable, est-il robuste à la charge ou à des conditions dégradées de fonctionnement, etc. On retrouve là les objectifs des tests présentés au chapitre 2. Parmi ces objectifs, tester la non-régression, c'est-à-dire être capable de vérifier que le comportement du logiciel ne s'est pas dégradé après la correction d'une erreur ou l'ajout d'une fonctionnalité est essentiel puisqu'il s'agit d'une des causes les plus fréquentes de dysfonctionnement des logiciels. Pour ce faire, il est nécessaire que le coût associé à l'exécution des tests soit maîtrisé et relativement faible ; cela n'est pas le cas si l'exécution des tests nécessite une grande part d'intervention humaine. L'utilisation d'un outil qui facilite l'exécution, et la réexécution des tests est dans ce contexte tout à fait pertinent et peut facilement justifier le coût de préparation des campagnes de tests et le potentiel surcoût lié à l'usage d'un outil d'automatisation.

Il existe de nombreux outils adaptés aux activités de tests dynamiques pouvant bénéficier d'une automatisation partielle ou complète : tests des éléments de base lors des phases de développement avec les outils de tests unitaires, tests lors des phases d'intégration en particulier avec les outils d'intégration continue, tests de performance du logiciel en phase de recette ou encore tests de stress ou de disponibilité avec des outils spécialisés.

Tests unitaires

Les tests unitaires se sont considérablement développés avec l'apparition des outils xUnit, ou plus précisément des cadres de développements (*frameworks*) xUnit : jUnit (pour le langage Java), jsUnit (pour le langage JavaScript), cUnit (pour le C), cppUnit (pour le C++), phpUnit (pour le PHP), etc. la lettre x faisant référence au langage ciblé. Ces structures logicielles sont des évolutions de SUnit mis au point par Kent Beck en 1994 pour le

langage objet avant-gardiste Smalltalk et fournissent un canevas pour la production automatisée de tests unitaires, principalement des méthodes dans le cadre des langages objets. Cette automatisation poussée est la clef du succès rencontré par ces technologies puisque toute l'architecture des tests est fournie ou produite automatiquement.

Ainsi, avec un environnement de développement récent (Eclipse par exemple) l'effort de test se résume à l'essentiel, i.e. choix des bons scénarios et des bonnes valeurs pour mener les tests et construction de l'oracle pour ces valeurs de tests à l'aide « d'assertions » qui seront utilisées dans les « tests suites ». Ces assertions permettent de définir que l'on attend une valeur donnée dans l'évaluation d'une expression ou qu'une exception doit être levée dans les conditions de tests choisies. L'exemple suivant (figure 8.1) illustre les différentes possibilités offertes par la version 4 de l'infrastructure jUnit^[4] (exemples tirés de la documentation associée à cette infrastructure) ; on notera l'utilisation d'annotations (@Test) qui permettent de donner de directives ou recommandations au compilateur ce qui simplifie encore la conception des tests unitaires par rapport aux versions qui utilisent uniquement les concepts objets (comme jUnit3). La première méthode permet de tester qu'à la construction, un tableau dynamique est vide. Dans le second test on utilise une annotation qui spécifie que l'exception de la classe IndexOutOfBoundsException doit être levée durant l'exécution de la méthode qui suit (méthode `outOfBounds`). Si l'exception n'est pas levée ou si une autre exception est levée durant ce test, le test échoue. La troisième possibilité illustrée dans cet exemple concerne la durée maximale d'exécution d'une méthode. Au-delà de cette durée (dont la valeur en millisecondes est passée en paramètre) la méthode est arrêtée et le test échoue.

```
public class Example {  
    @Test  
    public void method() {  
        org.junit.Assert.assertTrue( new ArrayList().isEmpty() );  
    }  
    }  
    @Test(expected=IndexOutOfBoundsException.class) public void outOfBounds() {  
        new ArrayList<Object>().get(1);  
    }  
    @Test(timeout=100) public void infinity() {  
        while(true);  
    }  
}
```

Figure 8.1 Exemple de tests unitaires avec jUnit4

Cet exemple montre bien en quoi une infrastructure comme jUnit simplifie la construction des tests unitaires. Couplée à un environnement de développement comme Ecplise^[5], cette infrastructure permet de plus de jouer et rejouer les tests très facilement à l'aide de boîtes de dialogues qui permettent de personnaliser l'environnement d'exécution des tests (par exemple en initialisant un environnement spécifique d'exécution avant de démarrer les tests). Les résultats d'exécutions des tests sont alors présentés par une barre latérale qui permet de suivre la progression des tests comme illustré sur la figure 8.2.

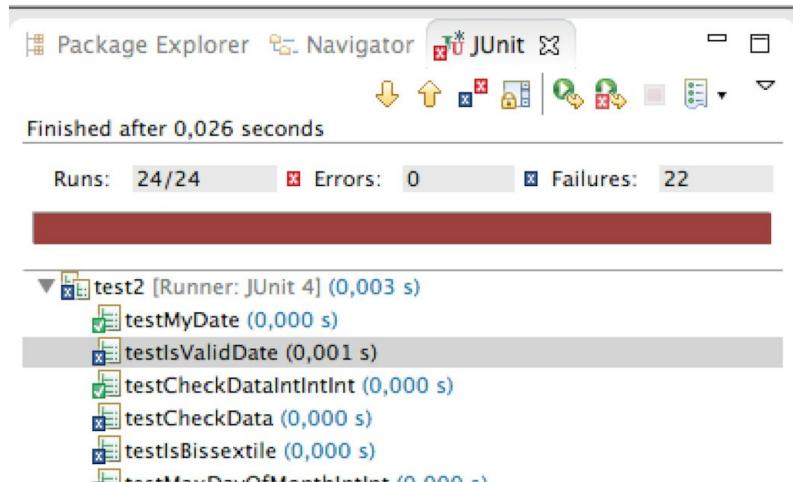


Fig. 8.2 Exemple d'utilisation de jUnit sous Eclipse

Sur la figure 8.2, on peut apercevoir la barre de couleur (rouge) sombre, signifiant que les tests ne sont pas encore réussis et la précision que sur les 24 tests prévus, 24 ont bien été exécutés, 22 donnant lieu à une « *failure* », c'est-à-dire à une violation d'assertion, et 0 conduisant à une erreur d'exécution non prévue (*i.e.* une exception non prévue a été levée). On peut également distinguer les tests qui ont réussi (petit onglet « v ») et ceux qui ont échoué (onglet « x ») ainsi que la durée d'exécution de ces tests.

Tests d'intégration et intégration continue

Une fois les tests unitaires réalisés, la phase d'intégration des différents composants soulève généralement de nouveaux problèmes. Ces problèmes ont pour cause principale une interprétation différente ou fluctuante des spécifications fournies ; les tests unitaires dirigés vers le respect des spécifications, ou plus précisément, sur la façon dont on a compris les spécifications du composant, ne permettent donc pas de détecter facilement ces défauts.

La découverte d'un défaut lors de la phase d'intégration nécessite de reprendre le codage d'un des composants qui n'est pas nécessairement le dernier intégré ; plus le défaut sera découvert tard plus il sera difficile à corriger car il nécessitera de revenir sur du code réalisé il y a des semaines voir des mois avec des intervenants qui ont pu migrer vers d'autres projets. Il

est donc important de vérifier aussi régulièrement que possible que les corrections, évolutions ou compléti ons des modules ne posent pas de problème avec les autres parties déjà réalisées.

Les outils de tests d'intégration vont tenter de répondre à ce problème en facilitant l'intégration des composants au fur et à mesure de leur réalisation et en automatisant le test des interactions entre les composants. Dans une démarche d'intégration continue, dès qu'un module est prêt il est intégré à l'ensemble et testé dans son environnement ; ce point est essentiel si l'on veut mettre en place une démarche agile. On réduit ainsi les risques de réponse tardive à un problème provoqué par la mise à jour d'un composant.

Un outil d'intégration va permettre de suivre l'assemblage progressif du logiciel et de mesurer sa qualité par le calcul de métriques et par le passage de différents tests qui sont réalisés de façon automatique lors de la mise à disposition d'un composant. On ajoute donc à l'aspect technique un aspect communication et collaboratif sur le développement du logiciel par ce suivi régulier de l'état global. Le schéma d'intégration des composants peut suivre une des stratégies évoquées précédemment : de haut en bas, de bas en haut ou plus généralement dans une démarche agile dès qu'un composant est intégrable.

De nombreux outils dédiés aux tests unitaires peuvent être détournés pour aider aux tests d'intégration mais ne seront pas aussi efficaces que les outils dédiés spécifiquement à ce type de tests. En effet, outre la spécificité de se concentrer sur les interfaces et sur l'interaction des modules, les outils de tests d'intégration doivent nécessairement intégrer un module de gestion de configuration. Cette fonctionnalité comprend non seulement des capacités de gestion de version mais également des capacités de gestion des liens entre les différents éléments d'une version. Grâce à cela, un outil d'intégration permet de mener des tests d'intégration mais également de produire une version opérationnelle du système complet par des « *builds* » automatique des versions complètes.

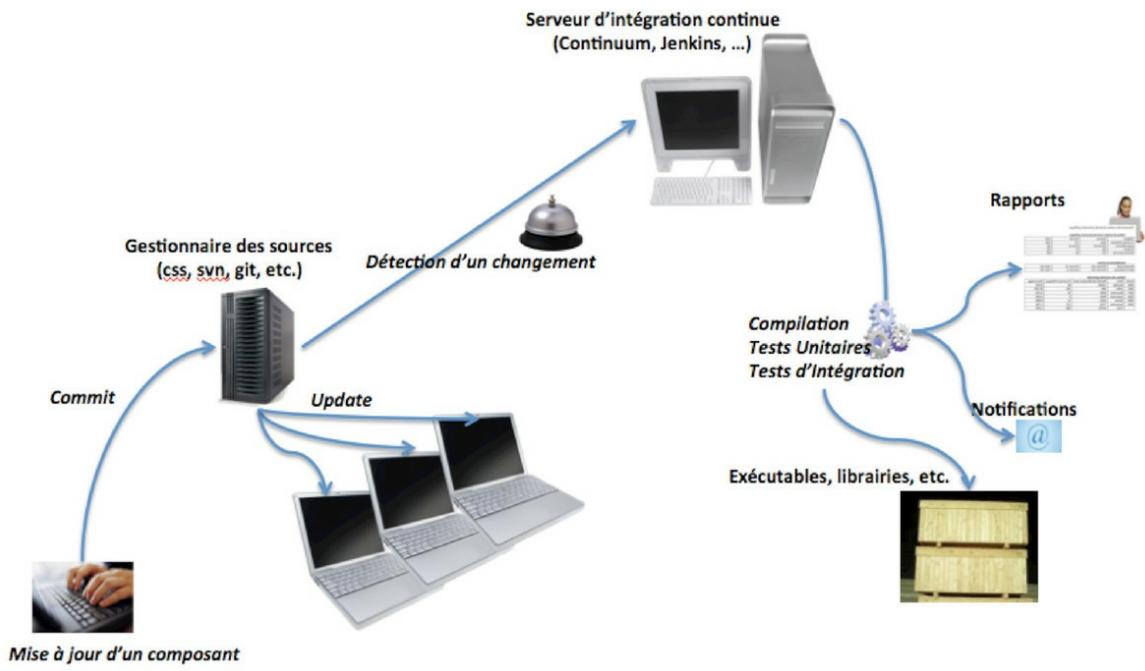


Fig. 8.3 Intégration continue

Le logiciel Selenium, cité précédemment et distribué sous la licence Apache, permet de tester les interfaces d'applications web en définissant des tests au niveau des interfaces. Cependant il n'intègre pas réellement de modules de gestion de configuration ce qui ne le place donc pas exactement dans les outils de test intégration. En effet, si l'on inclut au besoin de test des interfaces et des interactions le besoin de gestion de configuration, il faut se tourner vers un des deux logiciels libres phares de ce domaine : Continuum, distribué sous licence Apache et Jenkins (fork de Hudson) distribué sous licence MIT qui connaît un succès croissant de part sa simplicité d'utilisation et des nombreux modules d'extension (*plugins*) intégrables à ces outils.

Tests de charge, tests de stress, tests de performance

Un des aspects les plus difficiles à prévoir lors de la conception d'un logiciel ou à mesurer par la simple analyse du code réalisé est le comportement d'un logiciel vis-à-vis de ses performances. Les tests, dans ce contexte, sont donc non seulement utiles pour détecter d'éventuels défauts mais nécessaires pour mesurer le comportement réel d'un logiciel dans un environnement donné

avec la prise en compte des contraintes matérielles, des interactions avec d'autres applicatifs, des interactions avec des utilisateurs, etc. Par ailleurs, dans ce type de tests, la préparation de l'environnement de test puis sa remise en état initial peut nécessiter un travail complexe et technique. Sans outils, la réalisation de tests de ce type peut se révéler difficile, voire impossible, d'autant plus si l'on inclut la dimension de non-régression qui implique la possibilité de repasser régulièrement les tests.

Les tests de performance peuvent être vus comme un modèle générique pour un ensemble de tests proches : tests de charge, tests de stress, tests de montée en charge, tests d'endurance. Tous ces tests vont mesurer le comportement du logiciel vis-à-vis de propriétés liées à des notions techniques comme le temps de réponse, le taux d'occupation des CPU, le taux d'occupation de la mémoire, le nombre d'entrées sorties par unité de temps, etc. À la différence des autres tests de cette famille, les tests de performance vont se concentrer sur les caractéristiques intrinsèques de l'application dans un monde qui n'est pas nécessairement totalement réaliste. Ces tests peuvent s'apparenter aux mesures données des performances d'une automobile sur circuit dans des conditions d'essai bien particulières. Ils sont utiles car ils permettent de vérifier que l'architecture et le choix des algorithmes sont en phase avec les attentes vis-à-vis de l'application et qu'il n'y a pas a priori de « ratés » sur le plan des performances. Néanmoins, ces tests de performances ne donnent qu'une image partielle du comportement du logiciel dans son futur domaine d'utilisation ; on les complète généralement par des tests de charge ou de stress ou de montée en charge selon que l'on veut observer le comportement dans des cas d'utilisation « normaux » ou dans des cas dégradés ou encore lors des évolutions des contextes d'utilisations.

Le domaine d'application le plus répandu actuellement pour ce type de tests est la mesure de performance d'une application web de type *n*-tiers qui réagit à des stimuli en provenance d'utilisateurs qui, par leurs actions, vont déclencher des opérations sur des serveurs de base de données ou des serveurs applicatifs. On cherche alors à vérifier que l'application sera capable de « tenir » face à une montée du nombre d'utilisateurs. Les indicateurs mesurés sont alors principalement le temps de réponse vis-à-vis du client. Lorsque ce temps n'est pas bon, ou n'est pas conforme à ce que l'on espérait, on peut chercher à comprendre ces mauvais résultats en mesurant d'autres

indicateurs comme la charge des processeurs, le taux d'occupation mémoire, le volume de données échangées entre serveurs, etc.

Ce domaine d'application, en plein essor depuis quelques années, a rendu populaire ce type d'outils autrefois dédiés à quelques projets techniques. Parmi les très nombreux outils existants, on peut citer les outils de la société HP, « HP Performance Center » et « HP LoadRunner », outils très complets et couramment utilisés, l'outil « NeoLand » d'une jeune société française Neotys ou encore le logiciel libre « JMeter » distribué par la fondation Apache et qui propose de nombreuses possibilités d'analyse d'une application web.

De façon sommaire, ces outils vont simuler une montée en charge des stimuli sur l'application afin de mesurer son comportement à l'aide d'indicateurs simples comme le temps de réponse, le débit mesuré, le nombre d'éléments traités dans une unité de temps, etc. Le principe est de modéliser, graphiquement ou à l'aide d'un langage de script, le comportement d'un utilisateur puis d'écrire des scénarios de tests qui vont créer et faire agir un certain nombre d'utilisateurs. C'est là que l'usage d'un outil et l'automatisation de ces actions apportent réellement une aide par rapport à une approche manuelle.



Fig. 8.4 Exemple de résultat produit par l'outil JMeter

Une fois les scénarios de tests définis, on les exécute puis l'outil recueille les résultats qu'il présente soit sous forme de table, soit sous une forme graphique comme présenté sur la figure 8.4.

Test fonctionnel des applications web

Les applications web sont en plein essor et si l'on s'intéresse à mesurer leur performance et leur tenue à la charge avec des tests spécifiques, on peut également s'intéresser à tester leur fonctionnalité comme une application « normale ». Une des particularités de ces applications est qu'elles sont entièrement conçues pour interagir avec un utilisateur à travers le protocole HTTP : dans le cas le plus courant, le navigateur client de l'utilisateur envoie des « GET » et des « POST » au serveur en fonction des actions de l'utilisateur (clic sur des boutons ou saisie d'information dans des

formulaires). Dans la majorité des cas le serveur répond par l'envoi de données au format HTML après avoir exécuté quelques instructions (décrisées en PHP ou en Java à travers des JSP). Il s'agit donc par essence d'applications réparties avec une interface graphique. Tester ce type d'application nécessite donc de pouvoir jouer et rejouer facilement les actions d'un utilisateur sur l'interface graphique, comme pour un test de performance, mais cette fois-ci non pour mesurer le temps mis à fournir la réponse mais pour vérifier que les données fournies sont bien celles attendues dans ce cas. Il s'agit là d'une technique de test relativement ancienne et qui s'est considérablement améliorée avec la substitution progressive des interfaces graphiques type web aux interfaces graphiques plus complète (comme AWT ou Swing en Java). La relative simplicité de ces interfaces graphiques et le fait que qu'il soit possible de repérer un bouton ou un champ de saisie de texte autrement que par le positionnement de la souris sur une zone de l'écran rendent plus simple la mise en place d'outils qui capturent les actions de l'utilisateur et les reproduisent lors des tests.

Sur ce segment on peut trouver de nombreux outils efficaces. Les deux outils phares du domaine en 2013 sont « Selenium », qui est un logiciel libre et le logiciel « HP QuickTest Professional » (inclus maintenant dans l'outil HP « Unified Functional Testing software »).

Une seconde caractéristique de ces applications web est leur aspect transactionnel obtenu par l'orchestration d'un ensemble de services web. La réservation d'un séjour à l'étranger va faire intervenir une série de transactions : choix puis réservation d'un billet de transport auprès d'une compagnie aérienne ou ferroviaire, choix et réservation d'une chambre dans un hôtel et potentiellement choix et réservation d'activités sur place. Il s'agit alors de réaliser une série de transactions avec des opérateurs distincts, la série de transactions élémentaires étant elle-même une transaction à part entière ; un échec sur une de ces réservations doit entraîner un échec de l'ensemble des transactions. Tester ceci est relativement complexe du fait des différents mécanismes impliqués qui ne sont pas maîtrisés complètement par le site principal et l'on pourra se référer au chapitre 9 pour un exemple complet de test d'intégration de systèmes transactionnels complexes.

8.2.3. Outils d'appui à la gestion des tests

Les activités de tests sont nombreuses, monopolisent différents acteurs et peuvent s'étaler sur une période longue. Par ailleurs les besoins en communications, en interne des équipes projets, ou vis-à-vis de services tiers, ont pris progressivement une grande importance. Pour toutes ces raisons, il est indispensable de gérer les tests comme un autre projet et l'usage d'un outil de gestion de projet est fortement recommandé.

Si tout outil de gestion de projet peut être utilisé, un point essentiel doit être présent : la gestion des exigences qui permettra de tracer les liens entre les exigences fonctionnelles ou techniques du projet et les tests qui ont été conçus pour vérifier la bonne prise en compte de ces exigences. S'il manque des tests en face d'exigences on peut penser que certaines facettes du logiciel n'ont pas été ou ne seront pas testées correctement ; d'un autre côté, des tests prévus qui ne correspondent à aucune exigence particulière peuvent interroger sur l'intérêt de ces tests. Cela peut être également le signe d'une possible surinterprétation des spécifications qui a conduit à la réalisation de fonctionnalités non demandées, ou encore à la prise en compte de contraintes non nécessaires. L'outil de référence pour la gestion des exigences est IBM Rational DOORS et permet, entre autres, de maintenir la traçabilité dans le temps entre les exigences, les éléments de conception et les plans et scénarios de test.

Une autre facette essentielle de la gestion des tests qui pourra être outillée efficacement est la gestion des plans de tests. Les plans de tests décrivent les objectifs des tests, les configurations nécessaires pour les mettre en œuvre, ainsi que les calendriers et les équipes associées. Ces plans regroupent généralement des cas de tests en suites de tests ce qui permet de simplifier la gestion des projets complexes. Par ailleurs, on associe également aux plans de test les résultats des tests ainsi que le calcul de certaines métriques. Tout cela constitue donc une importante documentation qu'il convient de gérer correctement, à la fois dans un objectif de communication à destination des développeurs, des testeurs ou des donneurs d'ordre, mais aussi dans un objectif de suivi du projet. De nombreux outils sont disponibles dans cette catégorie, mais le plus complet (et aussi sans doute le plus onéreux) est l'outil distribué par Hewlett-Packard : HP Quality Center qui intègre l'ancien Test Director de Mercury.

Enfin, différentes solutions peuvent être mises en place en ce qui concerne le suivi et la gestion des anomalies. Les anomalies peuvent être présentées par gravité (mineure, majeure, bloquante) ou par leur état (de signalée à clôturée) ou encore par la ou les personne(s) en charge de résoudre l'anomalie. Par ailleurs, le suivi des anomalies permet de remonter un certain nombre d'indicateurs comme le nombre d'anomalies relevées par lot (qui peut donner une indication quant à la qualité de la réalisation), les délais moyens de prise en charge, de correction et de clôture (qui peuvent donner une indication sur la réactivité de l'équipe en charge des tests).

On peut trouver de nombreux outils répondant à ce besoin, en particulier dans le monde des logiciels libres ; les plus populaires à ce jour sont Bugzilla, proposé par l'organisation Mozilla et utilisé par des sociétés très connues (comme Facebook, Yahoo, la Nasa, etc.), le logiciel RedMine et qui a un spectre un peu plus large que la gestion des anomalies et enfin, le logiciel Mantis.

8.3. Conclusion

Un outil, bien choisi, permet sans conteste un gain dans la qualité et l'efficacité du travail réalisé. Le domaine des tests est doté d'une grande panoplie d'outils couvrant l'ensemble des activités de tests. Cependant avant le choix et l'installation d'un nouvel outil il est important d'auditer les processus et les pratiques existantes afin d'identifier les apports potentiels d'un outil en terme d'amélioration et/ou d'optimisation des pratiques en place.

Une fois les axes d'amélioration identifiés, il faut faire l'évaluation des produits du marché en incluant des critères tels que la qualité de la formation associée, la qualité du support technique, la communauté d'utilisateurs, les retours d'expérience que l'on peut recueillir. L'évaluation des différents logiciels disponibles se fera après avoir défini des critères objectifs et clairs qui permettront l'évaluation des différents produits par rapport aux améliorations et/ou optimisations visées. Assister à des démonstrations d'outils pour mieux appréhender les apports et les difficultés prévisibles et bien entendu efficaces dans cette phase de réflexion.

Une fois une liste restreinte de solutions retenues (dans l'optimal pas plus de trois candidats) il va être intéressant de mettre en place un pilote sur un ou deux projets afin d'avoir un retour concret de la solution envisagée dans le contexte de l'entreprise. Une revue détaillée de ces quelques expérimentations menées permettra de choisir de façon éclairée. Sans que le choix donne lieu à une décision collective, il est important de mettre dans la boucle tous les acteurs potentiellement impliqués dans l'usage au quotidien de cet outil. Une oreille attentive pourra y trouver une confirmation que les bénéfices attendus seront bien au rendez-vous et que les risques encourus ne risquent pas de rendre caduc cet investissement.

Une fois l'outil retenu et installé puis configuré, il faut mettre en place une stratégie d'accompagnement des différents collaborateurs impliqués. Cela inclut de la formation à l'outil mais aussi la mise en place d'une cellule d'appui aux utilisateurs qui saura conseiller et guider ces utilisateurs dans la transformation de leur façon de procéder. À noter que cet accompagnement est nécessaire même si l'outil semble accessible et convivial.

Enfin pour conclure ce chapitre, il ne faut pas oublier qu'automatiser une partie des tests est un projet en soi avec des développements et qu'en partie du fait des tests de non-régression, la durée de vie des tests peut être grande et nécessite de les gérer dans le temps comme tout autre projet.

Notes

[1] Parmi ces règles on peut citer la structuration du programme en sous-programmes ou modules ou classes, la déclaration des variables et des sous-programmes, le typage fort, etc.).

[2] Les liens vers les logiciels cités, ainsi qu'une rapide description de ces logiciels sont disponibles sur www.dunod.com/contenus-complementaires/9782100706082.

[3] On pourra lire à cet effet le blog John Carmack, développeur de Doom et Quakede, fondateur de la société Armadillo Aerospace, à l'url <http://www.altdevblogaday.com/2011/12/24/static-code-analysis/>.

[4] <http://junit.org>

[5] <http://www.eclipse.org>

Génération automatique de jeux de test

Tester revient à confronter par des moyens statiques (analyse de code, revue, etc.) ou par des moyens dynamiques (exécution avec des valeurs particulières) les spécifications du logiciel, c'est-à-dire ce qu'il doit faire et éventuellement sous quelles contraintes (temps, utilisation de la mémoire, etc.) à sa réalisation, c'est-à-dire de quelle façon il répond au besoin exprimé en enchaînant différentes actions élémentaires.

Trouver des jeux de tests pertinents, c'est-à-dire permettant de trouver des défauts dans la réalisation est, comme nous l'avons déjà dit, une activité qui va donc viser à déterminer, à partir des documents de spécifications, quelles valeurs intéressantes peuvent permettre de confronter la réalisation à celle attendue dans l'absolu (la réalisation qui implémente exactement les spécifications, *spécifications qui sont supposées être sans défauts*).

Générer automatiquement des cas de tests nécessite donc une analyse automatique des spécifications pour déterminer quelles valeurs seront pertinentes et lesquelles ne le seront pas. Les spécifications étant, dans la majorité des cas, rédigées en langues naturelles, il est dans l'état actuel des connaissances impossible d'automatiser de façon efficace cette activité sans apports d'informations complémentaires. Nous omettons volontairement les cas où le code est généré automatiquement à partir de spécifications complètement exprimées à l'aide de langages formels (par exemple, dans le cadre de l'utilisation de la méthode B (Abrial, 1996^[1]).

Nous détaillons ici quelques pistes et travaux récents qui apportent une réponse partielle à cette problématique et qui dans certains cas ont donné (et donneront de plus en plus) de véritables résultats en termes d'amélioration de la qualité des logiciels produits ainsi qu'en termes de réductions des coûts engendrés par les tests. On trouvera une étude assez complète faite en 2013

dans Saswat Anand et al.^[2] Chacune de ces stratégies va s'appuyer sur des compléments d'informations provenant par exemple de :

- l'utilisation de modèles plus ou moins formels décrivant le comportement attendu du logiciel ou l'utilisation d'annotations dans le code permettant de préciser des hypothèses sur les entrants ou les sortants d'un composant ou encore des invariants sur l'état attendu du logiciel à un point précis de son exécution ;
- l'utilisation de versions reconnues comme « exactes » qui serviront de référence en termes de spécifications ;
- l'utilisation du code lui-même et de données de tests existantes pour tenter d'améliorer la couverture du code par un complément de données de tests.

9.1. Générer des tests à partir de modèles

9.1.1. Principes

L'utilisation de modèles, stratégie de test connue sous le terme « Model Based Testing » consiste à utiliser une modélisation semi-formelle du comportement et /ou des données manipulées par le logiciel à tester (un composant, un système partiel ou complet) afin de générer des cas de tests qui représentent une partie des séquences possibles d'exécution décrites par le(s) modèle(s). Le logiciel à tester est alors défini comme le « système sous test » (*Système Under Test – SUT*) et est vu comme une boîte noire qui produit des données de sortie en fonction de données en entrée (et éventuellement de son état interne au moment de l'exécution du test). L'objectif dans ce contexte n'est pas de démontrer que le logiciel implémente exactement le modèle mais qu'il se comporte correctement dans des cas précis décrits par le modèle, cas qui sont choisis à partir d'hypothèses justifiant l'adéquation des cas de tests avec le modèle. La sélection peut se faire en amont de l'exécution des tests ou lors de l'exécution des tests par des procédures automatiques, ou semi-automatiques.

La difficulté de cette approche provient de la nécessité de fournir un modèle adapté à la génération automatique de tests pertinents : trop abstrait, il ne

capturera pas certaines exécutions particulières et qui présentent néanmoins un intérêt pour le test du système, trop détaillé, il sera à la fois complexe à définir et offrira trop de précisions sur certains aspects ; précisions qui nuiront à l'efficacité des tests qui seront trop importants en volume et sans doute redondants ; voir le livre de Bruno Legeard et al. *Industrialiser le test fonctionnel*, 2^e édition chez le même éditeur pour aller plus loin dans l'analyse du bon degré de modélisation pour une stratégie effective.

Les types de modèles que l'on peut utiliser sont nombreux et anciens pour certains. Dès les années 70, de nombreux travaux concernant le test de logiciel à l'aide de descriptions formelles voient le jour et le courant français, mené par Marie-Claude Gaudel puis par Gilles Bernot et Bruno Marre, s'appuie sur les spécifications algébriques pour dériver des suites de tests en fonction de certains objectifs de couverture. Ce courant a alimenté bons nombres de recherches et fourni de nombreux résultats avec quelques applications réussies dans le domaine de l'industrie^[3]. Néanmoins, cette stratégie n'a pas été réellement acceptée telle quelle dans l'industrie car jugée trop complexe à mettre en œuvre.

D'autres voies sont également explorées dès les années 70 et ont alimenté bons nombres d'approches et outils modernes. Parmi ceux-ci, les machines à états finis (*Finite State Machine* – FSM), permettent de décrire le SUT à l'aide d'états et d'actions (Chow 1978^[4]) ; afin de faciliter la possibilité de décrire des systèmes réels, les machines à états finis peuvent être étendues (*Extended State Finite Machine* – ESFM) par l'ajout de données sur les états, les entrants et les sortants et par l'utilisation de ces variables dans des gardes qui conditionnent les actions de la machine à états. À partir de cette description du comportement attendu du SUT, il est possible de générer des suites de tests qui permettent d'explorer tous les états, toutes les actions ou autres critères de couverture. Il est également possible de générer l'oracle qui va déterminer si les séquences et les valeurs de sorties observées correspondent bien au comportement attendu (Chen & al. 2015^[5]).

Enfin, une troisième approche théorique, basée sur les systèmes de transitions étiquetées (*Labelled Transition System* – LTS) a servi d'appui à l'approche MBT. Dans cette approche, plutôt utilisée dans le cadre de tests de protocoles, le comportement attendu est modélisé par un LTS. Par différentes techniques de parcours de l'ensemble des états accessibles du LTS ou encore

des séquences d'exécution possibles, des cas de tests sont générés ainsi qu'un oracle (sous forme de LTS) qui permet de confronter, sur les cas de tests générés, le comportement du SUT au comportement du LTS, permettant ainsi de détecter des divergences et donc des défauts d'implémentation (Tretmans 2008^[6]).

Actuellement, le test guidé par les modèles est une voie qui continue son essor et est supportée par de nombreux outils qui mettent en pratique les trois stratégies évoquées précédemment en les combinant ou en les complétant par des descriptions du comportement attendu par le logiciel à tester à l'aide de scénarios (diagrammes d'activités, diagrammes de flots, cas d'utilisation, etc.), de machine à états (le modèle le plus utilisé étant les diagrammes UML états-transitions), ou encore plus généralement des descriptions à l'aide de langage dédiés (*Domain Specific Language*), comme les langages de descriptions de processus métiers (par exemple BPMN).

9.1.2. Outils disponibles

Les outils sont nombreux mais nous pouvons en mentionner quelques uns qui ont été (et sont encore) concrètement utilisés dans des applications industrielles :

TGV

TGV (*Test Generation from transitions systems using Verification techniques*) est un outil qui fait partie de la suite d'outils CADP (*Construction and Analysis of Distributed Processes*) développée par l'INRIA depuis les années 90 et vise à fournir une solution à la génération de suites de tests pour des protocoles ou des systèmes asynchrones embarqués ou répartis. Cet outil se base sur une description du système à l'aide d'une forme de système de transitions étiquetées^[7].

Smartesting CertifyIt

L'outil CertifyIt^[8] de la société française Smartesting^[9] permet de générer des suites de tests avec différents objectifs de couverture (couverture des exigences, des transitions, etc.) à partir de différents types de modèles comme

les diagrammes d'états UML (Statecharts), des scenarios de comportement décrits avec le langage BPMN ou encore des pre et post conditions sur les actions modélisées à l'aide du langage OCL (*Object Constraint Language*). L'outil offre aussi de nombreuses possibilités de suivi et de gestion des campagnes de tests.

Conformiq Designer

Conformiq Designer est un outil permettant la génération automatique de suites de tests à partir de diagrammes UML états-transitions et/ou de description de comportement attendu à l'aide du langage Java. Les suites de tests générées peuvent également être exportées dans différents langages incluant le langage TTCN-3 et analysées à l'aide de diagrammes de séquences de messages (*Message Sequence Chart*). Cet outil est commercialisé par la société Conformiq[\[10\]](#) et est distribué en France par la société VerifySoft[\[11\]](#).

Microsoft Spec Explorer

L'outil Spec Explorer 2010[\[12\]](#) de Microsoft est un outil mature qui est intégré à la suite logicielle de développement Visual Studio (extension gratuite). Le comportement attendu du SUT est décrit ici à l'aide du langage de programmation C# dans l'environnement .NET et des scénarios de tests peuvent être décrits à l'aide du langage de script Cord.

9.1.3. Avantages / Inconvénients

Les avantages à utiliser l'approche des tests dirigés par les modèles (MBT) sont nombreux ; en particulier cette approche permet à la fois d'automatiser la génération de cas de tests et l'oracle associé dans la plupart des cas. L'effort de modélisation demandé permet de découvrir au plus tôt, à la fois des erreurs dans les réalisations, mais également des défauts présents dans les spécifications. De nombreuses études montrent un réel retour sur investissement quant à la qualité des logiciels produits avec cette démarche une fois l'appropriation de la démarche faite par les équipes impliquées.

Les difficultés que l'on peut rencontrer proviennent de l'apprentissage de langages de modélisation et du fait que la modélisation est intentionnelle. Il est ainsi parfois peu simple de trouver le bon niveau de modélisation ; il faut abstraire, mais pas tout et ni trop ni trop peu. Il faut également prendre en compte la transition nécessaire des pratiques des équipes impliquées.

9.2. Générer des tests à partir du code

Le code est une interprétation des spécifications et il contient généralement des défauts. Il est donc souvent moins efficace dans le processus de test d'utiliser le code à la place des spécifications pour générer les jeux de tests. Néanmoins, d'un point de vue pragmatique, le code représente exactement la façon, même imparfaite, dont les fonctionnalités seront ou pas réalisées et prend en compte des détails concrets, utiles pour la réalisation, mais non présents dans les spécifications.

Nous distinguons deux cas qui se présentent assez couramment :

- l'équipe dispose d'une version de référence, par exemple une correction dans le cadre de la formation, ou une version V0 que l'on fait évoluer pour des raisons de supports techniques (évolution de matériel, d'environnement logiciel, etc.),
- l'équipe a une simple version de développement mais a déjà des jeux de tests qu'elle souhaite faire évoluer ou compléter.

9.2.1. Utilisation d'une version de référence

Ce contexte est assez favorable puisqu'une version de référence est connue et est, soit considérée comme « exacte », c'est-à-dire conforme en tout point aux spécifications, soit imparfaite mais dans ce cas, les divergences vis-à-vis des spécifications ont déjà été sans doute identifiées. Il est donc possible d'utiliser cette version comme une description précise et non ambiguë des spécifications, ce qui permet de générer à la fois des jeux de tests et l'oracle ou plus généralement, de confronter de façon systématique les comportements de la version en cours de développement avec les comportements de la version de référence.

Pour ce faire plusieurs stratégies sont possibles mais la façon la plus effective consiste à générer des jeux de tests en s'appuyant sur une combinaison d'exécutions symboliques et concrètes (dites concoliques) de la version de référence visant à assurer une couverture du code donnée (par exemple tous les chemins, ce qui est assez fort, toutes les décisions, etc.).

Le principe est assez simple mais présente quelques subtilités : partant du logiciel on transforme les chemins d'exécutions possibles en ensembles de contraintes à satisfaire pour atteindre un point donné du programme (liées au conditions de branchement), contraintes qui portent sur des variables symboliques représentant les variables concrètes. Par exemple, si le logiciel fait intervenir deux variables entières x et y , et une décision de branchement « $\text{if } (x > y) \text{ then } \dots$ », deux variables symboliques sont introduites X et Y ainsi qu'une contrainte symbolique de chemin « $X > Y$ » pour la branche d'exécution correspondant au cas où la décision est vraie (la variable x est plus grande que la variable y) et une contrainte symbolique « $X \leq Y$ » pour la branche où la décision est fausse. Les contraintes sont ainsi cumulées lors de l'exécution symbolique du programme avec la couverture recherchée. À certains points, l'ensemble de contrainte peut ne pas avoir de solution (par exemple « $X > Y \&& Y > 3 \&& X < 2$ ») ; dans ce cas, la branche est coupée et le chemin symbolique abandonné. Lorsque la couverture est obtenue, il « suffit » de résoudre les systèmes de contraintes pour obtenir des valeurs concrètes des variables qui correspondront donc à des jeux de tests satisfaisant la couverture recherchée. Pour cela on utilise fréquemment des solveurs de contraintes type SMT (*Satisfiability Modulo Théorie*) qui s'appuient eux-mêmes sur des solveurs de contraintes booléennes SAT (Satisfiabilité booléenne) et permettent de décider si un problème admet une solution dans un modèle donné (par exemple une solution au problème $3*X=1$ admet une solution dans les nombres réels mais pas dans les nombres entiers) et, si une solution existe d'en donner une valeur.

Les difficultés de cette approche sont néanmoins multiples et ont longtemps freinées l'utilisation des outils associés sur des problèmes réels (difficultés moins présentes actuellement du fait de l'avancée de la théorie sous-jacente et des capacités de calcul et de mémoire maintenant disponible à faible coût).

Parmi ces difficultés, la première concerne « l'explosion » de la taille de l'ensemble de contraintes et la « complexification » de cet ensemble en

particulier lors de la présence de boucles ou d'appels de sous-programmes. Pour combattre ce problème, on peut remplacer des appels de sous-programmes par des abstractions (forme de bouchonnage symbolique), limiter la profondeur de déroulement des boucles (par exemple 3 tours maximum) et / ou remplacer certaines valeurs symboliques par des valeurs concrètes en s'appuyant sur une exécution concrète du programme.

L'autre problème provient de l'utilisation de librairies ou de code en assembleur qu'il n'est pas facile de traduire automatiquement sous formes de contraintes (puisque l'on ne dispose pas ou de façon peu utile le code source impliqué) ; dans ce cas, un problème de divergence peut apparaître entre l'exécution symbolique et le comportement réel du fait d'imprécisions dans les contraintes. En détectant les zones où l'imprécision a un impact, il est possible de demander au testeur de donner un modèle de la partie impliquée, ce qui nécessite une opération manuelle et réduit un peu l'intérêt de l'approche.

9.2.2. Utilisation d'une version incomplète ou partiellement testée

Ce cas est encore plus fréquent que le précédent puisqu'il apparaît dès que des tests « boîtes noires » ont été générés manuellement mais que la mesure de couverture montre que l'objectif fixé (toutes les décisions, tous les chemins, critère MC / DC) n'est pas atteint. Ce cas apparaît également fréquemment dans le contexte de chaîne de produits logiciels^[13] où l'on développe des logiciels par assemblage et composition de différents composants (ou versions de composants) ou encore dans le test d'une nouvelle version d'un composant dans un développement incrémental.

Ce domaine connaît une activité de recherche importante et s'appuie sur les techniques de propagation et de résolution de contraintes, d'exécution symbolique et d'apprentissage pour combiner les jeux de tests existants, vus comme un ensemble de contraintes initiales que l'on va chercher à compléter, à des données de tests produites par une génération automatique de valeurs dans le but d'obtenir une couverture donnée ou dans le but de trouver des chemins « différenciant » c'est-à-dire présents dans une version et absents dans l'autre ; ceci permet de tester les nouvelles caractéristiques du logiciel

ou au contraire de détecter des exécutions devenues impossibles. Nous pouvons mentionner ici le prototype CAT développé à Paris Nanterre par François Delbot et Valentin Bouquet avec l'appui de Souheib Baarir, Lom Hillah et Sang Dao dans la nouvelle équipe de test et qualité du logiciel dirigée par Pascal Poizat^[14].

9.2.3. Outils disponibles

Là encore les outils sont nombreux mais nous en citons deux qui ont fait leurs preuves aussi bien dans le contexte académique que dans le test et la vérification d'applications industrielles.

PathCrawler

PathCrawler^[15] est un outil ancien et robuste faisant maintenant partie de la plateforme Frama-C et permet de générer des jeux de tests pour des programmes écrits en langage C. Développé et maintenu conjointement par le CEA List et l'INRIA Saclay, cet outil est disponible sous licence GPL ou sous licence commerciale^[16] ; une utilisation en ligne est également disponible permettant de mesurer l'intérêt (ou pas) d'utiliser ce logiciel dans un cas donné.

JavaPathFinder (JPF-SE)

JavaPathFinder est un outil développé par la NASA depuis de nombreuses années dans le but de vérifier des programmes Java à l'aide de différentes stratégies de vérification de modèles (Model Checking). Il a été étendu en 2007^[17] par l'ajout d'un module d'exécution symbolique permettant ainsi de générer des jeux de tests sur des programmes Java, incluant les programmes multitâches. Cet outil est disponible gratuitement et peut être obtenu à partir du site babelfish de la NASA^[18] et peut être intégré dans les environnements de développement Eclipse^[19] ou NetBeans^[20].

9.2.4. Avantages / Inconvénients

Les avantages à utiliser une génération automatique de jeux de tests à partir de code sont nombreux : automatisation quasi complète, réduction des coûts liés au test, amélioration des couvertures, détection rapide d'erreurs subtiles dans la réalisation et peu de changements à introduire dans les procédures de développement. Néanmoins, cela ne peut être la seule stratégie utilisée pour tester un composant ou un sous-système puisque l'on se focalise sur ce qui est fait et non sur ce qu'il aurait fallu faire.

9.3. Conclusion

La génération de jeux de test par des procédures automatiques est une volonté ancienne des équipes projets. Longtemps inaccessible du fait de théories trop complexes et difficiles à mettre en œuvre avec des moyens techniques limités, cette approche est maintenant rendue possible par d'un côté l'accès à des puissances de calculs et de mémoire pour un faible coût et par une grande maturité des approches, en particulier l'approche basée sur les modèles (approche MBT) qui apporte une véritable amélioration de la qualité des logiciels produits moyennant une appropriation pas toujours évidente de l'aspect modélisation.

Notes

- [1] Abrial, J., Hoare, A., & Chapron, P. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511624162
- [2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. 2013. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.* 86, 8 (August 2013), 1978-2001. DOI=<http://dx.doi.org/10.1016/j.jss.2013.02.061>
- [3] P. Dauchy, M.-C. Gaudel, and B. Marre. 1993. Using algebraic specifications in software testing: a case study on the software of an automatic subway. *J. Syst. Softw.* 21, 3 (June 1993), 229-244. DOI=[http://dx.doi.org/10.1016/0164-1212\(93\)90025-S](http://dx.doi.org/10.1016/0164-1212(93)90025-S)
- [4] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.* 4, 3 (May 1978), 178-187. DOI=<http://dx.doi.org/10.1109/TSE.1978.231496>
- [5] Chen, Z., Xu, B., Yang, R., & Zhang, Z. (2015). *EFSM-Based Test Case Generation: Sequence, Data, and Oracle*. *International Journal of Software Engineering and Knowledge Engineering*, 25, 633-668
- [6] Jan Tretmans. 2008. Model based testing with labelled transition systems. In Formal methods and testing, Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.). Lecture Notes In Computer Science, Vol. 4949. Springer-Verlag, Berlin, Heidelberg 1-38.
- [7] <http://cadp.inria.fr/>, TGV
- [8] <http://www.smartesting.com/fr/certifyit/>
- [9] <http://www.smartesting.com/>
- [10] <https://www.conformiq.com/>
- [11] <http://www.verifysoft.com>
- [12] <https://msdn.microsoft.com>
- [13] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up adoption of software product lines: a generic and extensible approach. In Proceedings of the 19th International

Conference on Software Product Line (SPLC '15). ACM, New York, NY, USA, 101-110. DOI=<http://dx.doi.org/10.1145/2791060.2791086>

[14] <http://dep-mathsinfo.parisnanterre.fr/>

[15] <http://frama-c.com/pathcrawler.html>

[16] <http://frama-c.com/download.html/>

[17] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2007. JPF-SE: a symbolic execution extension to Java PathFinder. In Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'07), Orna Grumberg and Michael Huth (Eds.). Springer-Verlag, Berlin, Heidelberg, 134-138.

[18] <https://babelfish.arc.nasa.gov/trac/jpf/wiki/install/start>

[19] <http://www.eclipse.org/>

[20] <https://netbeans.org/>

Tester des systèmes interactifs

Les systèmes embarqués et les services Web sont utilisés dans des contextes différents mais présentent, du point de vue des tests, des caractéristiques communes : ils interagissent avec de nombreux systèmes extérieurs (variés et ayant une grande variabilité), ils sont généralement composés de sous-systèmes qui collaborent pour fournir les services pour lesquels ils ont été conçus, les caractéristiques testées dépassent généralement le cadre fonctionnel et incluent les problèmes de charge, de fiabilité et de sûreté. Plus généralement, on peut aussi constater que c'est au niveau des interactions que les défauts vont fréquemment apparaître et que les problèmes de temps de réponse sont cruciaux dans ces deux univers.

Nous détaillons maintenant quelques aspects saillants des stratégies de test à mettre en œuvre dans ces deux domaines.

10.1. Test des systèmes embarqués

Les systèmes embarqués sont de plus en plus présents dans le monde actuel. Ils ont des missions variées (et souvent critiques), ils doivent être relativement autonomes et ont des exigences fortes en termes de temps de réponse (on les classe dans la catégorie des systèmes « temps-réel » c'est-à-dire qu'ils doivent travailler avec de véritables horloges et non des horloges logiques). Ils sont réactifs (ils réagissent à des stimuli extérieurs), ils sont globalement asynchrones (les horloges des différents éléments ne sont pas forcément les mêmes) mais localement synchrones.

Cette hypothèse synchrone, qui stipule que les temps de calcul peuvent être considérés comme nuls et donc que les valeurs d'entrées des composants ne changent pas durant les calculs, est une hypothèse forte mais qui simplifie grandement le test ou la validation des parties synchrones et a donné de belles réussites, en particulier en France avec le développement du langage Esterel

par Gérard Berry et du langage Lustre par Nicolas Halbwachs, Paul Caspi et al., ainsi que le logiciel SCADE proposé par la société Esterel Technologie^[1] (voir le cours du collège de France donné par Gérard Berry en 2013 et 2014^[2] pour une synthèse éclairée de ces travaux et de la nécessaire collaboration entre le monde académique et le monde industriel sur des sujets complexes pour obtenir des résultats effectifs).

Néanmoins si l'hypothèse synchrone est efficace elle est reste complexe à mettre en œuvre et le test des systèmes embarqués va faire appel aux techniques classiques de test des logiciels vues dans cet ouvrage avec toute fois une attention particulière sur certains aspects : simulation nécessaire et plus complexe de l'environnement lors du test des composants, prise en compte de la différence entre l'environnement de développement et celui de l'exécution du logiciel (système cible), prise en compte des ressources souvent limitées (cadence processeur et capacité mémoire) sur le système cible.

Nous revisitons l'approche classique du test d'un système avec un focus particulier sur les spécificités de systèmes embarqués.

10.1.1. Isolation et test des composants ou des sous-systèmes

La première étape consiste à isoler les unités composant le système et de définir pour chacune de ces unités les points d'observations du comportement ainsi que les paramètres d'entrée auxquels va réagir l'unité testée. Ceci va permettre de procéder aux tests unitaires mais aussi de préparer les tests d'intégration en désignant clairement les interfaces des composants et le format des données échangées sur celles-ci. Les spécificités du monde embarqué vont intervenir principalement sur l'utilisation de bancs de tests mêlant matériels et logiciels avec des parties réelles et des parties simulées ; on parle alors de simulation HIL (*Hardware In the Loop*) approche qui permet de tester efficacement sous différents aspects un composant de contrôle déjà réalisé ou d'approche MIL (*Model In the Loop*) si le composant est encore en phase de spécification et que l'on veut tester son potentiel comportement avant la phase de réalisation ; dans ce cas le composant est

souvent décrit à l'aide d'un langage dédié comme Matlab ou Simulink développés par la société The MathWorks^[3].

Chaque unité devra être testée en isolation la plus complète possible afin de ne pas confondre un défaut du composant avec un défaut d'un autre composant avec lequel le composant sous test interagit.

À ce niveau il est aussi important de préparer les mécanismes à mettre en œuvre pour sauvegarder les différents résultats de tests afin de pouvoir faire une post-analyse des résultats de tests (une analyse en ligne est en effet souvent difficile pour ces systèmes du fait de leurs caractéristiques temps-réel). La difficulté proviendra dans la capacité à sauvegarder le plus d'éléments possibles sans introduire de modifications de comportement de l'unité qui pourraient être induites par des écritures de logs trop coûteuses et qui perturberaient les temporalités des actions de l'unité sous test.

La mesure de couverture sera fondamentale dans ce contexte et la mesure la plus usitée sera sans doute la couverture MC / DC car elle offre un bon compromis entre efficacité et exhaustivité mais, dans les cas moins critiques, on peut se contenter de couvertures plus faibles comme la couverture « toutes les instructions » ou la couverture « toutes les décisions ».

10.1.2. Test des interactions

Une fois les composants ou sous-systèmes testés en isolation, on va pouvoir utiliser les points d'observation sur les interfaces pour tester les échanges sur ces interfaces entre les unités afin de s'assurer que les protocoles d'échange d'informations sont bien respectés. Dans cette étape, des bouchons logiciels ou matériels (réels ou simulés) seront souvent utilisés car ils permettent de tester efficacement les interactions. La réussite de cette phase sera d'autant grande que les unités ont été bien isolées et bien testées en isolation et donc que l'architecture du système a été conçue autant que possible dans cette optique.

Là encore, il sera nécessaire de sauvegarder les données échangées sur les points d'interaction pour permettre une analyse post-exécution. L'utilisation d'architecture basée sur des bus physique ou logiciel simplifiera grandement la démarche puisque les sondes seront assez aisées à mettre en œuvre et

qu'elles ne viendront pas perturber le fonctionnement au niveau des interactions entre les composants sous test.

10.1.3. Test système et de validation

Les sous-systèmes assemblés, il convient de tester le système complet afin de valider son comportement.

Par rapport à un système logiciel classique, les tests système et de validation vont mettre l'accent sur des aspects de robustesse selon la norme IEEE 61012-1990^[4] : tests de charge et de résistance aux pannes, comportement face à des valeurs d'entrée incorrectes ou non attendues. Une étude récente^[5] montre que les tests de robustesse de systèmes embarqués sont globalement bien acceptés dans de nombreuses entreprises mais pratiqués de façons différentes et ad hoc.

Comme pour les tests unitaires des systèmes embarqués, l'utilisation de simulateurs et l'instrumentation de code vont permettre de mettre le système dans des conditions de fonctionnement proches de son fonctionnement réel et permettront de détecter des défauts qui seraient restés non détectés lors des phases précédentes mais qu'il faudrait résoudre avec un coût bien plus important lors de la mise en production.

10.1.4. Normes et standards

Ces systèmes étant sensibles car pouvant potentiellement mettre en danger des personnes ou impacter fortement la société (par exemple, dans les secteurs de l'économie, des transports ou du médical), de nombreux standards existent à travers le monde. Le standard « racine » est le standard EC-61508 qui définit des processus et des recommandations pour réduire les risques provenant de défauts logiciels ou de pannes de matériels.

Quatre niveaux de risques sont définis (*Safety Integrity Level – SIL*) correspondant à un niveau de fiabilité attendu par le système, ou de l'unité concernée, déterminé par l'analyse de sécurité ; SIL1 est le niveau le plus faible et SIL4 le niveau le plus contraignant. La mesure est donnée en probabilité de voir apparaître une défaillance dans l'exécution d'une fonctionnalité ou de probabilité de panne par heure. Les niveaux d'exigences

peuvent aussi être donnés en termes de couverture de code par les tests et plus généralement sur les processus de développement utilisés pour les logiciels embarqués.

Dans certains référentiels, ces niveaux sont codés de A à D, A étant le niveau le moins exigeant et D le niveau correspondant aux risques catastrophiques.

La norme ISO 26262:2011 (bientôt remplacée par la norme ISO/DIS 26262-1) se focalise sur les aspects de sûreté dans les véhicules. La norme DO-178B/C que nous avons déjà mentionnée est utilisée dans le contexte avionique tandis que les standards européens EN 50128 ou EN 61375-3-2 décrivent les exigences en termes de développement et de tests pour assurer la certification de systèmes logiciels et électroniques devant être utilisés dans le cadre ferroviaire ; enfin le standard IEC/EN 62304 décrit les exigences en termes de cycle de vie de développement des logiciels utilisés dans les environnements médicaux.

10.2. Test des services Web

Un service Web peut être défini comme un logiciel (ou ensemble de logiciels) permettant l’interaction de logiciels ou machines hétérogènes à travers un réseau en fournissant un service, par exemple une opération arithmétique sur des nombres ou la réservation d’une chambre d’hôtel ou encore la réservation d’un voyage complet : transport, logement, visites sur place, etc. Le protocole d’échange entre le service Web et ses clients est la plupart du temps le protocole http ou https mais il est aussi possible d’utiliser le protocole ftp ou tout protocole standardisé permettant l’échange de données. On distingue :

- les services Web de type SOA (*Service Oriented Architecture*) qui définissent à la fois une interface d’appel (un URI^[6] – *Uniform Ressource Identifier*, – les URL étant des URI –) ainsi qu’une description du service fourni par le service (avec le langage WSDL – *Web Service Description Language*) combiné avec la description des messages échangés à l’aide du standard SOAP (*Simple Object Access Protocol*) et du format de représentation des données XML.
- les services Web de type REST (*Representational State Transfer*) qui s’appuient sur une architecture Web de type client serveur sans état et

qui, à la différence des services de types SOAP, découpent complètement le client du serveur ; le client doit en effet mémoriser les données utiles au traitement de ses requêtes et les présenter au serveur dans ses demandes. Ceci peut induire un plus grand besoin de bande passante mais permet la mise en place de caches efficaces au niveau du serveur) et cela permet de remplacer plus facilement un service (ou un serveur) par un autre ; lorsque cette façon de découpler le client et le serveur est totalement respectée, on parle alors d'applications RESTful.

Comme pour les systèmes embarqués, les services Web sont donc des applications relativement autonomes qui interagissent avec un monde qui évolue en permanence. Elles vont donc pouvoir être testées par des approches similaires, à la fois sur les aspects fonctionnels mais aussi sur les aspects performance et robustesse.

La spécificité est ici que les services Web sont testés principalement sous la forme de tests d'intégration sans faire appel à des notions de couverture de code qui s'applique mal dans ce contexte.

Pour ce faire on décrira des scénarios que l'on exécutera de façon automatique avec des outils comme :

- Apache JMeter^[7] qui permet de tester le service Web sous un aspect fonctionnel en simulant différents utilisateurs, des tests de charge et de robustesse ;
- SoapUI^[8] distribué sous forme de logiciel libre ou payante par la société SmartBear^[9] qui permet également de tester les différentes facettes d'un service Web, qu'il utilise une architecture SOAP ou une architecture REST ;
- TestingWhiz^[10] qui permet de tester les aspects fonctionnels, de robustesse mais également de réaliser des tests d'intrusion ou de déni de services ;
- et bien d'autres outils qui élargissent leurs champs d'application au test des interfaces de programmation (API) à l'aide de techniques similaires.

10.3. Conclusion

Nous avons vu dans ce court chapitre que le test des systèmes embarqués et des services Web ont de nombreux points communs en particulier dans l'approche méthodologique des tests : prendre grand soin de l'environnement, se focaliser sur les interactions, les notions de sûreté et de robustesse face à la charge et aux fautes. Les différences existent néanmoins puisque les aspects de contraintes techniques pèsent peu sur les services Web alors qu'elles sont prégnantes sur les systèmes embarqués.

On constate actuellement également une grande convergence de ces deux mondes, en particulier dans le domaine de l'automobile où les systèmes embarqués vont de plus en plus communiquer avec des services Web, à la fois sur des questions de confort ou de guidage mais aussi sur des questions de sécurité dans les sous-systèmes d'aide à la conduite ou de gestion d'urgences.

Cette convergence va sans doute impliquer des changements dans la façon de concevoir les systèmes embarqués et donc dans la façon de les tester avec la mise en place de stratégies globales et d'outils avec une couverture fonctionnelle plus large.

Notes

- [1] <http://www.esterel-technologies.com/>
- [2] <http://www.college-de-france.fr/site/gerard-berry/course-2013-2014.htm>
- [3] <https://fr.mathworks.com/>
- [4] “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Std 61012-1990, pp. 1–84, Dec. 1990
- [5] S. M. A. Shah, D. Sundmark, B. Lindström, and S. F. Andler, ‘Robustness Testing of Embedded Software Systems: An Industrial Interview Study’, IEEE Access, vol. 4, pp. 1859–1871, 2016
- [6] <https://www.ietf.org/rfc/rfc2396.txt>
- [7] Apache JMeter, <http://jmeter.apache.org/>
- [8] <https://www.soapui.org/>
- [9] <https://smartbear.com/>
- [10] <http://www.testing-whiz.com/web-services-testing>

Les tests, une nouvelle mesure de complexité

11.1. Introduction

Dans ce chapitre un peu prospectif nous voulons éléver le débat et regarder les tests comme une activité fondamentale de l'ingénierie des systèmes qui va structurer le futur de nos sociétés hyper connectées, alors que les prévisions font état de 50 milliards d'objets interconnectés via Internet vers 2030. Dans les projets informatiques, la tendance vers de plus en plus d'intégration est irréversible. Les projets dits *d'intégration* sont de plus en plus nombreux, et dans ces projets l'activité IVVT est prépondérante. Le travail consiste alors à élaborer des scénarios de tests, avec les données qui conviennent, éventuellement obtenues par simulation. Dans ces projets, on comprend tout de suite que la complexité ne réside évidemment pas dans les lignes de code constitutives des modules que l'on intègre mais dans les interactions entre ces modules compte tenu des nouveaux usages. Ce sont ces interactions qu'il faut valider, et c'est pour cela que les scénarios de tests et l'effort consacré à leur mise en œuvre sont un indicateur de complexité pertinent du point de vue du chef de projet. D'où la définition que nous proposons :

Définition : Un système S_A est plus complexe qu'un système S_B si le coût IVVT de S_A est supérieur au coût IVVT de S_B .

Notons que cette définition prend en compte les coûts d'ingénierie des modules constitutifs du système qui entrent dans le processus d'intégration décrit précédemment, lesquels peuvent être appréhendés via un modèle d'estimation type COCOMO ou Points de fonctions. L'accent est mis ici sur les interactions que les protocoles d'interfaces autorisent.

Intégrer l'information dans l'entreprise numérique, c'est mettre en cohérence trois ensembles de trajectoires évolutives, bien distinctes mais étroitement corrélées :

- Les processus métiers et/ou industriels qui créent la valeur de l'entreprise, compte tenu des missions que l'entreprise s'est assignées et des nouveaux usages rendus possibles par l'innovation technologique. De fait, une co-évolution des métiers, de la technologie et de l'ingénierie.
- Les programmes, c'est-à-dire le support informatisé, numérisé, des règles et des comportements de tout ou partie de ces processus qui représentent aujourd'hui des dizaines de millions de lignes de code et des milliards d'enregistrements dans les bases de données.
- La technologie numérique, c'est-à-dire les plates-formes et infrastructures TIC, aujourd'hui extraordinairement diversifiées, ubiquitaires, mobiles, microscopiques mais puissantes ... tant au niveau des capacités de traitement que du stockage de l'information.

Chacune de ces trajectoires a sa dynamique évolutive et sa complexité propre. Chacune a ses modalités organisationnelles mais toutes ont en commun la contrainte de se transformer et de s'adapter en permanence. Le PROJET est le moteur de cette co-évolution vitale pour l'entreprise.

Ce qui fait la complexité du numérique est la nécessaire cohérence de ces trois trajectoires prises individuellement, mais surtout de rendre leurs combinaisons deux à deux, et trois à trois également cohérentes, ce qui est beaucoup plus difficile compte tenu de la taille des patrimoines applicatifs des entreprises. C'est la recherche de l'équilibre et du bon compromis entre ces trois trajectoires, de l'organisation du patrimoine applicatif, qui donnera un avantage compétitif à l'entreprise qui saura comment organiser toutes ces interactions. La clé de la découverte de ce nouvel équilibre est l'architecture système, de tous les systèmes, industriels et/ou tertiaires.

Les architectes des systèmes numériques sont confrontés à deux grands types de complexité :

1. Une complexité statique appréhendée de longue date par les mécanismes de hiérarchisation qui sont au cœur de la technologie informatique depuis son origine. Hiérarchiser, classer, rechercher, créer/effacer, restructurer, ... sont les opérations au centre des technologies de management de l'information.
2. Une complexité dynamique liée aux évolutions des environnements organisationnels et humains qui demandent toujours plus d'interactivité, aux évolutions du système lui-même qui se précise et s'adapte durant la réalisation, et aux évolutions de la technologie qui aujourd'hui permettent un asynchronisme massif des traitements, complexité qui se matérialise par des flux de transactions qui se chiffrent désormais en dizaines de milliers par seconde. Les *Big Data* et le *Cloud Computing* sont l'image marketing de cette évolution irréversible qui exige fiabilité et performance, sûreté de fonctionnement, au niveau global système.

C'est l'organisation de cette complexité dynamique, pour des systèmes toujours plus sûrs, qui est aujourd'hui l'enjeu fondamental, le clivage décisif entre les entreprises qui développeront la capacité à passer l'obstacle de cette complexité en se formant sérieusement, et celles qui trébucheront. Le mot d'ordre proclamé par tous les acteurs du changement est **agilité**, mais la difficulté de l'agilité est dans le **comment** la mettre en œuvre concrètement.

Rendre les trajectoires cohérentes et sûres, améliorer le contrat de service, c'est réaliser des couplages négociés entre tous les acteurs de ces trajectoires par les **interfaces** contractualisées. Les interfaces sont les éléments stables du système qui permettent l'échange et la transformation de l'information au niveau de finesse requis par l'interactivité. Nous avons donc une suite d'interactions causales :

Plus d'interactivité, plus d'agilité = Découpage plus fin des fonctions = Plus d'éléments en interactions = **intégration plus difficile**, donc plus coûteuse.

Le but de ce chapitre est de montrer en quoi le processus d'intégration IVVT qui est le socle fondamental sur lequel l'entreprise numérique et les usagers du cyberspace pourront développer à la bonne vitesse les potentialités que

l'innovation technologique met à portée de leurs mains, nous donne un indicateur de complexité qui traduit mieux la perception que s'en fait le chef de projet, ce qui rend son action plus efficace.

L'architecture de l'information est la condition nécessaire de l'agilité et de l'intégration réussie.

11.2. Terminologie

Nous utilisons le terme « *numérique* » pour caractériser l'état présent et futur des systèmes informatisés qui gouvernent le fonctionnement de la société et de ses grands équipements (énergie, télécommunications, transport, défense et sécurité...), tant au niveau des entreprises et des administrations, que des particuliers, où l'ordinateur est devenu un artefact quasi microscopique, « qui ne se voit plus », mais omniprésent, dont on ne peut plus se passer. Le meilleur indice de cette « numérisation » massive est la quantité de logiciel que chacun de nous utilise quotidiennement comme par exemple utiliser son smartphone, rechercher un itinéraire avec Google Maps ou son GPS, ou simplement allumer la lumière, ou encore donner un bon coup de frein qui active le dispositif ABS de sa voiture. Rappelons tout de même que le terme « *informatique* » forgé par la fusion de *information* et *automatique* dans les années 60, garde tout son sens, car il s'agit bien d'automatiser le traitement de l'information. Sans ces automatismes, il n'y aurait ni *Cloud Computing*, ni *Autonomic Computing*, ni *Big Data*, ni téléphone mobile, ... car les flux associés dépassent désormais de très loin les capacités humaines.

Au sortir du dernier conflit mondial, cet indice logiciel était strictement égal à zéro ! Aujourd'hui, derrière nos gestes les plus élémentaires se cachent des dizaines de millions de lignes de programmes, fruit du travail et de l'intelligence de millions de programmeurs, logiciels qu'il faut concevoir, développer, valider, entretenir, et retirer proprement du service lorsqu'ils deviennent obsolètes, car on ne les trouve pas à l'état naturel comme les matières premières. Les ordinateurs et les équipements informatisés nomades qui permettent l'interaction en temps réel de millions d'usagers vont mettre en exécution ces logiciels pour rendre à l'usager le service attendu, avec le

niveau de qualité requis par ces usagers qui ignorent tout de la complexité sous-jacente. La puissance de traitement et de stockage de l'information est fournie par des infrastructures regroupant des centaines de milliers de serveurs qui rendent accessibles à tous le *Software as a Service*, [SaaS], un peu comme l'électricité ou le téléphone, par un simple branchement à cette nouvelle source d'énergie. Simplicité des usages, complexité des infrastructures et des plates-formes, complexité de la programmation des équipements et des interfaces, tel est le dilemme de l'intégration pour l'entreprise numérique.

Dans le corps du texte nous serons amenés à parler de taille de programmes, exprimée depuis les années 70-80 en nombre d'instructions source réellement écrites par les programmeurs en COBOL, C, C++, Java, Python, ... [longue liste], à l'exclusion de celles fabriquées par des outils, conformément aux règles de comptage standardisées en vigueur^[1] ; Ainsi 1.000 lignes de scripts peuvent activer un progiciel de grande taille, mais seules les 1.000 lignes ont été écrites. Pour rendre concrète cette notion de ligne source [LS, ou KLS pour 1.000 LS] il est commode de les représenter au format imprimeur, c'est-à-dire en nombre de pages (en moyenne, dans l'édition courante, 50 lignes de texte par page), soit :

- Le coût pour 1.000 lignes de code, c'est-à-dire 1 KLS dans le modèle COCOMO correspondrait à 2 ou 3 hommes_mois de développement. Une erreur dans un tel texte, c'est une erreur sur 1.000 LS, ce qui est déjà considéré comme une bonne performance : c'est une unité de compréhension qui respecte les règles ergonomiques humaines, en particulier la capacité de mémorisation du programmeur, indispensable à la qualité de son travail.
- Pour 10.000 LS, c'est un livre moyen de 200 pages.
- Pour 100.000 LS, c'est 2.000 pages, soit cinq « gros » livres de 400 pages, une taille d'ouvrage considérée comme une limite ergonomique, à la fois pour l'auteur et pour le lecteur ; difficile à mémoriser, y compris pour l'auteur qui doit avoir « en tête » tout le contexte. C'est typiquement le fruit du travail d'une équipe projet agile.

Il est bon d'avoir ces chiffres à l'esprit pour comprendre la nature du travail des programmeurs qui par certains cotés est analogue à un travail d'écrivain

et de traducteur, voire de mathématicien, mais surtout celui d'un créateur et architecte de modèles.

11.3. Le puzzle de l'intégration

11.3.1. Le vrai sens de l'intégration

La plupart des entreprises disposent de cartographies applicatives qui décrivent en grandes masses le contenu de leur patrimoine logiciel. Quelques-unes d'entre elles disposent également de chiffrages de ce patrimoine, soit en termes de Lignes de code Source [LS], soit en termes de Points de Fonctions [PF, une mesure fonctionnelle associée aux données], unités utilisées pour estimer les projets informatiques depuis les années 1980 (voir les réf. bibliographiques [2] et [4]). Dans ce chapitre nous utiliserons de préférence les lignes de code source, unité de comptage plus concrète, et mieux comprise par quiconque a touché à un ordinateur et vu un bout de programme ou un texte HTML/XML dans ses messages Internet.

Ces cartographies sont basées, soit sur l'architecture fonctionnelle des systèmes informatisés, soit sur les processus métiers et/ou les équipements qui les hébergent, soit un mélange des deux. Elles sont représentées à l'aide de diagrammes que l'on trouve dans les outillages qui supportent les méthodologies d'ingénierie, comme les langages UML, SysML, BPML, ... pour les plus courants. Le résultat de ce qu'on appelle communément **urbanisation** ou **architecture d'entreprise** est une cartographie du système qui a été organisé et structuré en module [ou intégrat/ITG].

Il est important pour l'entreprise de connaître la taille du patrimoine logiciel qui lui permet de fonctionner et de se développer, exprimé en lignes de code source [LS]. Et plus encore de bien connaître celles et ceux qui entretiennent ce stock dont la valeur n'est pas vraiment comptabilisée : les programmeurs et les architectes. Aux Etats-Unis, programmeur est un métier noble, correctement valorisé.

Cependant, peu d'entreprises ont une idée précise de la taille et de la valeur de leur patrimoine applicatif.

- Pour une grande banque, dont l'informatisation démarre en même temps que l'informatique, disons dès les années 60, le patrimoine programmatique peut monter jusqu'à 300 millions de lignes sources, majoritairement encore en COBOL.
- Pour des ministères régaliens comme le MinEFI ou le MinDef, informatisés de longue date, on est à minima dans les 150/200 millions de lignes source.
- Pour des entreprises de création récentes comme celles qui gèrent par exemple les diverses réservations de billets d'avions, de trains, d'hôtels, etc. dont on a besoin dans la vie courante, ou bien les clients des opérateurs de téléphonie mobile, ou encore le transport de l'énergie, ... nous sommes dans les 100/150 millions de LS.

Il est commode, pour bien appréhender ce qu'une telle quantité d'information signifie à l'échelle humaine, de se la représenter aux normes de l'édition qui nous sont familières depuis quelques siècles : le livre. Un livre de 400 pages, à raison de 50 lignes par page, c'est l'équivalent d'un programme de 20.000 lignes sources.

Un patrimoine de 100 millions de lignes source c'est l'équivalent d'une bibliothèque de 5.000 volumes.

Mais pour être complet, il faut ajouter à cette bibliothèque, le mode d'emploi, c'est-à-dire. les référentiels système, les aides en lignes [souvent intégrées au texte même du programme], et surtout les scénarios de tests qui permettent de déclarer la bibliothèque « bonne pour le service » aux usagers. Ce qui reviendrait à doubler sa taille !

Un patrimoine de 100 millions de lignes source, c'est un coût de développement incompressible d'environ 25.000 homme_année, mais beaucoup plus en coût réel si on introduit les facteurs de réussite bien connus (moins de 30% des projets), soit trois fois plus.

Pour information les analyses statistiques faites sur des milliers d'entreprises donnent comme productivité moyenne des programmeurs le chiffre de 4.000 lignes, à 10%, de programmes réellement écrites (et non fabriquées par des outils) par programmeur et par année calendaire « normale », c'est-à-dire 220 jours ouvrés de 8 heures . Trois à cinq fois plus pour des programmeurs chargés de la maintenance selon le taux moyen de modifications à effectuer.

Les programmes sont la composante « intelligente » qui permettent de manipuler les données de l'entreprise grâce aux opérations basiques : CREER une donnée, RECHERCHER/LIRE une donnée, TRANSFORMER/MODIFIER une donnée pour la mettre à jour, ou encore EFFACER une donnée qui n'a plus d'intérêt [résumé par l'acronyme CRUD : CREATE, RETREIVE, UPDATE, DELETE]. Les programmes permettent également l'échange d'information via les messages et événements EMIS/REÇUS, par d'autres programmes et/ou l'opérateur du système [opérations SEND / RECEIVE]. Les données de l'entreprise, encore plus que les programmes, sont la vraie richesse de l'entreprise. Une immobilisation au sens comptable du terme, plus précieuse qu'un stock.

On peut également se représenter les fichiers et les bases de données avec les normes de l'édition. On parlera alors de lignes de données [LD]. Un grand fichier comme celui des contribuables, des clients d'EDF, des assurés sociaux, c'est au bas mot 25/30 millions de références clients. Si l'on compte une page d'information par référence, soit environ 2.500/3.000 caractères, on atteint facilement les 100 milliards de caractères. Si on édite ces fichiers sous forme de tableaux de type Excel avec un taux de remplissage de 50% par ligne tenu des blancs et des règles visuelles d'alignement, nous arrivons [en prenant 50 caractères par ligne] à 2 milliards de lignes de données soit 40 millions de pages, ou encore une bibliothèque de 100.000 livres.

Un grand fichier de 25 millions de références, c'est l'équivalent d'une bibliothèque de 100.000 volumes.

Des fichiers de ce type, il y en a quelques centaines dans un pays comme le nôtre. Une grande administration comme la CNAM-TS gère plus de 1

milliard de bordereaux de remboursements par an, pris en charge par la sécurité sociale et/ou les mutuelles pour ceux des français qui en bénéficient. A supposer qu'il n'y ait que 250 caractères par bordereau, soit 10 fois moins que précédemment, on aboutirait cette fois à une bibliothèque de 400.000 volumes, en une seule année ! Pour les fiches FADET de facturation du téléphone, c'est 10 à 100 fois plus. A ce stade, nous entrons dans le domaine des *Big Data* où les manipulations « à la main » ne sont plus possibles, l'usage des outils devenant indispensable.

Si on s'intéresse à la qualité de ces données et aux erreurs éventuelles qu'elles contiennent suite à des manipulations erronées, on est alors pris de vertige, et il faut bien s'accrocher à la rampe ... nous ne ferons pas le calcul ici. Mais c'est une raison supplémentaire pour connaître ce patrimoine en détail et être rigoureux sur les tests comme il est fortement recommandé dans cet ouvrage.

Pour revenir à la cartographie, on peut donc séparer la partie **programmes**, c'est-à-dire les procédures codées dans tel ou tel langage, et la partie **données**, c'est-à-dire l'information transformée et exploitée par les programmes (cf. figure 11.1). Du point de vue du comptage, l'entreprise numérique doit comptabiliser séparément la partie **procédurale** du patrimoine, les programmes proprement dits déduits des règles de transformation et des contraintes à respecter, et la partie **données**, c'est-à-dire les fichiers et bases de données, conformément à la logique de comptage des modèles d'estimation aujourd'hui largement utilisés en management de projet (voir réf. [4]).

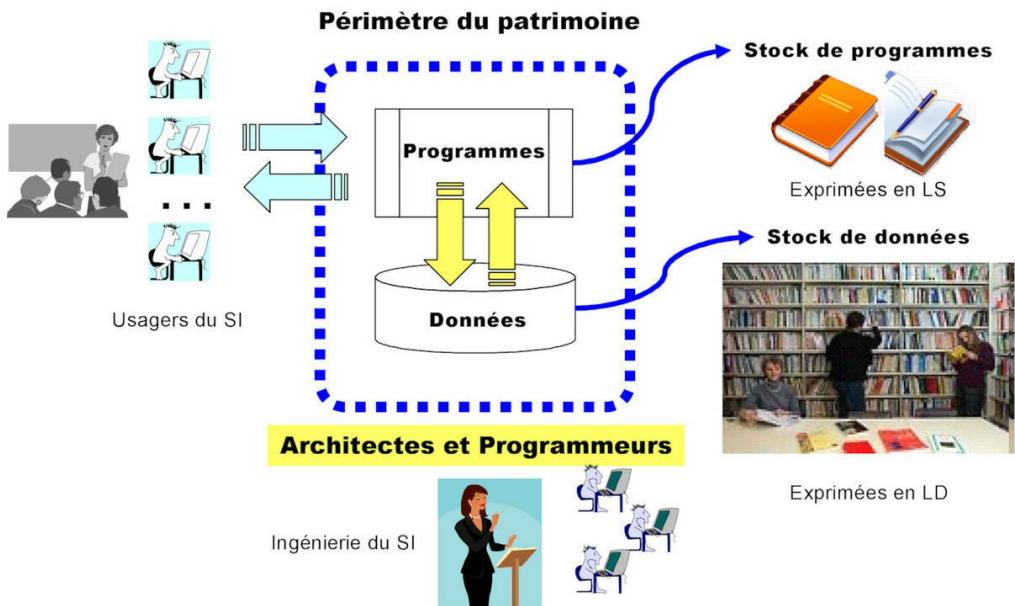


Fig. 11.1 Données et Programmes

C'est une première cartographie, certes pas très utile, mais qui cependant montre bien la nature des liens que cette cartographie va entretenir avec son environnement. Là où les choses deviennent vraiment intéressantes est lorsqu'il va s'agir de structurer cette « masse » par rapport aux organisations qui l'utilisent et qui la gèrent (cf. figure 11.2).

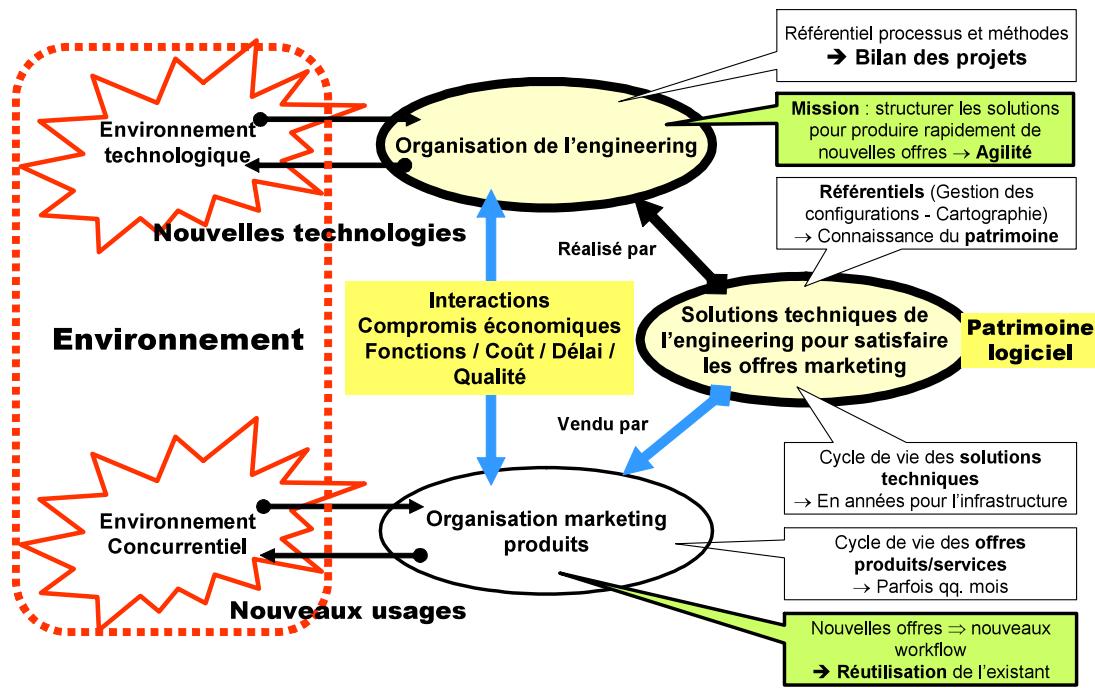


Fig. 11.2 Interaction des organisations – Ecosystème

Selon que l'on prend le point de vue de l'engineering, du marketing (c'est-à-dire des usagers), ou des technologies utilisées, on obtient des cartographies très différentes. Pour ce qui concerne l'intégration, c'est la relation engineering/patrimoine logiciel qu'il faut privilégier, mais en n'oubliant jamais qu'elle est au service des usagers. C'est l'engineering qui intègre les solutions, mais ce sont les clients usagers qui découvrent les erreurs résiduelles qui ont échappé à la vigilance de l'engineering.

Du point de vue de l'engineering, le patrimoine logiciel se présente comme un gigantesque puzzle (cf. figure 11.3), dont les « pièces » élémentaires sont les blocs de code source et des blocs de données associés à ces blocs de code ; d'où la terminologie souvent utilisée en intégration : *building block*.

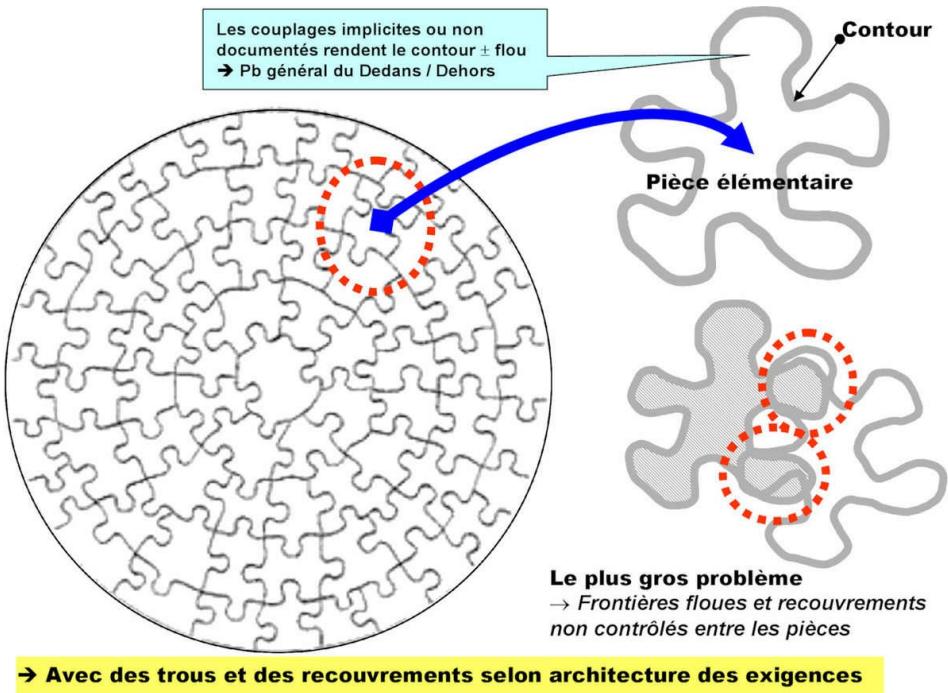


Fig. 11.3 Le puzzle de l'intégration

A ce stade on peut comprendre que tout le problème de l'intégration, et donc de l'architecture, sera de contrôler le nombre et la nature des relations que les pièces entretiennent entre elles. Mais avant, il nous faut regarder du côté de ceux qui sont en charge de l'ingénierie de ces pièces.

11.3.2. Unité de sens/compréhension – Performance des programmeurs

Une ligne de code source, c'est-à-dire une instruction, n'a généralement aucun sens pour le programmeur. Ce qui compte c'est l'agrégation des lignes de code en blocs sémantiques, l'équivalent de phrases, qui traduisent une action effectuée dans la réalité des métiers et/ou des équipements gérés par le programme. Des études d'ergonomie ont montré que ce qu'un programmeur correctement formé peut raisonnablement manipuler en termes de quantité d'information dépend de la capacité de mémorisation et d'organisation qu'il a su développer. On en connaît les limites statistiques :

- Notre champ visuel nous permet d’appréhender sans trop de difficulté deux pages de textes, c’est-à-dire un livre ouvert, soit une centaine de lignes, avec des conventions typographiques dont nous sommes familiers depuis l’enfance.
- Pour des textes plus longs, il faut une structuration qui prenne en compte le nombre de concepts qu’un cerveau correctement éduqué/instruit est capable de mémoriser et de manipuler pour agir de façon pertinente. Ce nombre, parfois qualifié de « magique », vaut 7 ± 2 ; il est connu de tous les ergonomes. Appliqué à nos deux pages, avec une table de matière commentée nous arrivons à une unité de sens programmeur de l’ordre de 20 pages de texte, soit environ 1.000 lignes source/instructions [quel que soit le langage].

C'est la raison pour laquelle l'unité de compte du modèle d'estimation COCOMO est précisément 1.000 LS, ce qui représente un effort de développement approximatif moyen de 2,5 hm, soit 10 semaines calendaires [c'est évidemment une grandeur statistique].

Un bloc de code d'environ 1.000 LS est ce qu'un programmeur correctement formé est en moyenne capable d'appréhender de façon sûre et durable : c'est une unité de compréhension programmeur [UCP], un quantum d'information validé qui définit le « grain » sémantique ultime de son travail de modélisation. Tout dépassement de ce seuil doit être géré comme un risque.

NB : En programmation objet, avec des langages comme Java, C#, Python, ... c'est une classe de 8-10 méthodes, chacune comptant en moyenne 100-125 LS, soit deux pages de texte, c'est-à-dire une entité qui a du sens pour le programmeur.

Quelques données ergonomiques qui structurent l'activité des programmeurs :

- Une lecture attentive de 20 pages de programme, c'est 2-3 heures de concentration intense, sans distraction ni interruption.

- On peut estimer qu'une reprise de ce code par le programmeur qui l'a conçu et développé, après un an, c'est environ 10% de l'effort initial, soit cinq jours de travail. Un bon programmeur gardera en mémoire la structure générale du programme, mais oubliera les détails qu'il reconstruira cependant sans difficulté.
- Le même travail de reprise, par un tiers, sans connaissance préalable, c'est 20-30%, soit 2 à 3 semaines d'appropriation, si la documentation est bien faite, pour effectuer des modifications.
- Une productivité de 20.000 LS par an est une performance atteinte par moins de 10% des programmeurs [c'est 5 fois la moyenne statistique de 4.000 LS à $\pm 10\%$]. Si le programmeur qui annonce de tels taux n'est pas classé comme excellent, il est probable que plus de la moitié du travail a été oublié ou omis : programmation peu rigoureuse sans respect des règles de bonnes pratiques, absence de documentation, tests incomplets, etc. ... d'où la notion d'âge « moyen » en Compétence/Expérience de l'organisation de l'engineering [sous-traitants inclus] qui est une moyenne pondérée, c'est-à-dire un barycentre, de l'expérience de chacun.

On comprend pourquoi un turnover excessif dans une organisation d'ingénierie peut vite devenir un handicap insurmontable. Un turnover de 20-30% rend impossible l'atteinte des niveaux 4-5 sur l'échelle CMMI. Les méthodes agiles insistent sur la régularité de l'effort, et en particulier l'effort de test. Les meilleurs programmeurs sont des marathoniens, pas des sprinters. Dans la Silicon Valley il est courant d'accorder une journée par semaine de travail libre aux programmeurs pour les fidéliser.

11.3.3. Nombre de pièces à intégrer – Combinatoires

Le nombre de pièces du puzzle dépend de la taille des pièces élémentaires que sont capables de fabriquer les programmeurs dans une organisation de l'ingénierie (cf réf. [5]).

À titre d'exemple dans une équipe projet agile, c'est-à-dire 8-10 personnes maximum, qui fonctionne sur des durées de 12-18 mois, on raisonne en binômes programmeurs testeurs.

- Un binôme va pouvoir gérer/produire, en moyenne, des blocs de code d'environ 20 KLS, soit une vingtaine de pièces « usinées » en 12-18 mois.
- Une équipe de 4-5 binômes va pouvoir gérer/produire, en moyenne, des blocs applicatifs d'environ 100 KLS, soit une centaine de pièces.

Pour faire le dénombrement du nombre de pièces d'un patrimoine, nous avons donc trois unités « naturelles » de comptages, faisant sens pour l'organisation de l'ingénierie :

- L'unité de compréhension programmeur UCP qui se réfère au programmeur individuel qui *in fine* écrira le texte du programme de la pièce, et le corrigera en cas de modification.
- L'unité de compréhension du binôme programmeur testeur UCB, environ vingt fois plus grosse que l'UCP. Dans les méthodes agiles il y a deux rôles importants interchangeables : celui qui programme et celui qui valide le programme, c'est-à-dire celui qui produit la preuve de bon fonctionnement : les tests.
- L'unité de compréhension équipe projet agile UCE, environ cent fois plus grosse que l'UCP, à charge de l'équipe de s'organiser, ou de s'auto-organiser comme le recommandent certains auteurs, selon les compétences de chacun de ses membres, sous la responsabilité d'un programmeur senior.

En prenant des patrimoines : PETIT, 1 MLS, MOYEN, 10 MLS, et GROS, 100 MLS, on a un nombre de pièces basiques qui sont les nœuds du graphe correspondant du système informatisé de l'entreprise ; soit :

Nombre de pièces à intégrer :

	Unité de comptage	1 MLS	10 MLS	100 MLS
UCP		1.000	10.000	100.000
UCB		50	500	5.000
UCE		10	100	1.000

Dans certains systèmes temps réel critiques, l'unité de compréhension est parfois l'instruction du langage de programmation, comme par exemple les

instructions qui permettent la synchronisation des traitements et des événements, ou le partage des données [en programmation objet, ce sont les bibliothèque RMI].

Le paramètre le plus important, du point de vue de l'intégration, sont les relations que les pièces entretiennent les unes avec les autres. Une relation existe entre la pièce P1 et la pièce P2 si un couplage quelconque existe entre P1 et P2. Le couplage peut être fonctionnel, par exemple P1 et P2 ont des données communes, ou organique, P1 et P2 sont hébergées dans un même serveur, ce qui fait que si P1 bloque ou sature le serveur, P2 sera *ipso facto* bloquée également.

L'étude des matrices de couplages, appelées matrice N^2 , est un aspect fondamental de l'ingénierie système (cf réf. [3]), car le taux de remplissage de la matrice va donner une première indication sur la quantité de travail à effectuer sur le système pour valider l'intégration. Dans le cas le plus favorable, avec N pièces, on aura à valider $N-1$ relations de couplage ; les pièces constituent une simple chaîne, c'est-à-dire la diagonale de la matrice. Dans le cas le plus défavorable, quand la matrice est saturée, chaque pièce est couplée à toutes les autres, directement ou indirectement, on aura alors à valider $N(N-1)$ relations, avec l'hypothèse restrictive que lorsqu'une pièce est sollicitée plusieurs fois, elle réagit toujours de la même manière. Une pièce qui, du fait de sa programmation, ne se comporterait pas de la même façon à chacune des sollicitations, comptera comme autant de pièces différentes compte tenu des états possibles de la pièce en question, par exemple : état NOMINAL, tout marche bien, état NON NOMINAL, la pièce ne rend plus le service et le signale, un état intermédiaire avec MODALITE où une partie du service a été fait. Dans ce cas, il faut analyser les chemins possibles qui conduisent à cet état grâce à des traces ad hoc résultant de l'architecture.

Une pièce qui occasionne des « fuites de mémoire » sur le serveur qui l'héberge va progressivement dégrader son contrat de service, premier problème, mais surtout occasionner des dysfonctionnements aléatoires dans les autres pièces qui partagent le même serveur. Un tel comportement est catastrophique dans le cas du *Cloud Computing*, où la pièce doit pouvoir être réallouée dynamiquement en fonction du niveau de saturation de l'infrastructure.

Dans un univers simple, comme celui des instructions machines, on peut raisonner Vrai/Faux, mais dès que la taille des pièces devient significative, dès le niveau individuel UCP avec ses 1.000 lignes/instructions source, la logique devient modale, en fonction du contexte.

La logique d'intégration, du point de vue de l'architecte, consiste donc à se poser systématiquement trois questions :

1. Quel est le bon nombre de pièces à considérer, de quelle taille ?
2. Quels sont les couplages que la pièce entretient avec les autres pièces, c'est-à-dire quelles sont les interfaces externes de la pièce [le contrat d'interface], y compris l'environnement ? Et ce, pour toutes les pièces.
3. Quel est le degré d'indépendance [isolation, confinement] de la pièce compte tenu des sollicitations ? Par défaut : être totalement indépendant, mais c'est une propriété qui doit être vérifiée à la construction [architecture] et validée, par exemple via les tests de performance et de fatigue, les tests de charge, etc.

11.3.4. Ebauche d'une typologie des pièces à intégrer

La nature des pièces à intégrer a une grande importance du point de vue de la combinatoire. Dans la figure 11.1, nous avons distingué la typologie la plus fondamentale, la typologie **données / programmes** qui est le fondement du modèle de Von Neumann, la base de tous nos ordinateurs, ce qui correspond à la séparation programmes / bases de données dans tous les systèmes informatisés.

Du point de vue de la combinatoire c'est insuffisant, et il faut aller un cran plus loin dans la décomposition hiérarchique pour bien en mesurer l'importance. Les critères de classification indiquent l'importance que l'architecte attribue aux différentes entités qui constituent la classification. Dans la figure 11.4, on a fait apparaître les métadonnées et les métaprogrammes qui constituent une partie du référentiel système. Les informations correspondantes sont particulièrement importantes car elles doivent être partagées par tous les acteurs de l'ingénierie et de ceux des métiers qui sont en interactions avec l'engineering [voir figure 11.2]. On a également fait apparaître les messages et événements qui sont des interfaces fondamentales devant être contractualisées. On a aussi fait apparaître la

distinction entre les programmes qui dépendent d'un équipement et/ou d'une plate-forme d'exécution et ceux qui dépendent des processus métiers car leur cycle de vie est totalement différent : on ne change pas les équipements tous les jours, alors qu'un nouveau besoin métier peut exiger une réponse de l'engineering dans un délai de quelques semaines. Cette dernière distinction est cruciale pour l'agilité du système.

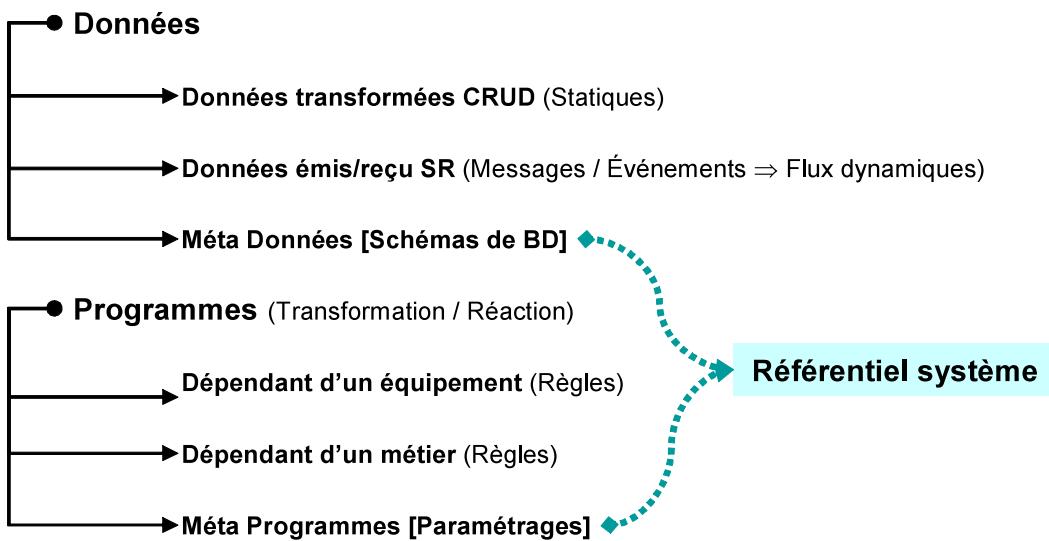


Fig. 11.4 Typologie

Vous noterez que sur l'arborescence, nous n'avons pas fait apparaître une typologie actuellement en vogue, celle des « services », pour des raisons de simplification. Un service, dans la mesure où il est partagé est astreint à des règles strictes d'évolution/adaptation de même nature que celles des données partagées entre plusieurs programmes nécessitant un service d'accès.

Dans tous les cas, c'est à l'architecte de définir ce qui est important pour le cycle de vie du système et de le faire figurer comme tel dans l'arborescence des entités constitutives du système. Ces entités, avec leurs attributs, sont des types et doivent être rigoureusement documentées dans le référentiel du système.

11.3.5. Le problème de l'intégration

Toute adaptation/transformation significative du système va se traduire par la création de nouvelles pièces, la modification de pièces existantes en général en restant compatible [Principe de compatibilité ascendante] pour ne pas occasionner de régression auprès des usagers qui n'en aurait pas le besoin, la suppression de pièces obsolètes. Cet inventaire définit le périmètre de l'intégration. [cf figures 11.3 et 11.5]. Les pièces de cette nomenclature seront appelés *modules* ou *building blocks*, parfois traduit en Intégrat.

Les pièces dépendantes d'un équipement ou d'une plate-forme sont la partie émergée d'un « iceberg » d'interfaces système et/ou progiciel dont il convient de bien mesurer la hauteur [voir figure 11.6].

C'est l'un des deux problèmes majeurs de l'intégration, le second étant la validation des propriétés transverses globales : interopérabilité, performance, disponibilité, sûreté de fonctionnement, facilité d'emploi, etc. pour ne parler que des plus importantes.

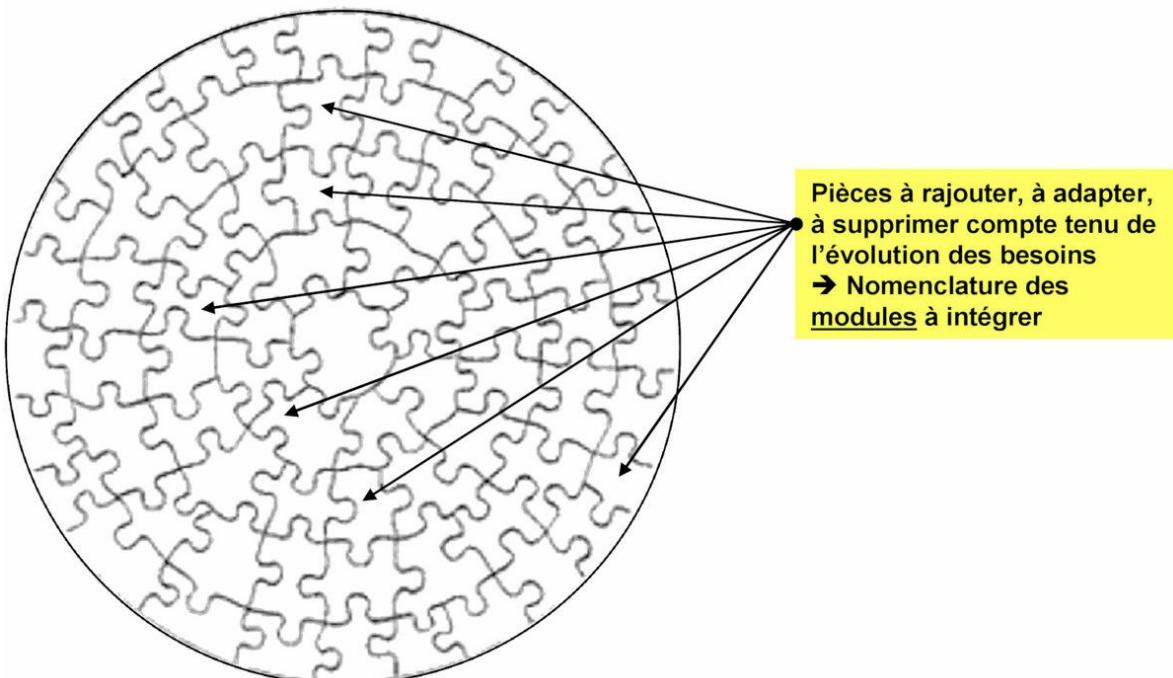


Fig. 11.5 Modifier le puzzle

Cet inventaire ayant été fait, on peut dresser la liste des modules externes qui sont en interactions directes [couplages], avec l'un quelconque des modules appartenant à l'inventaire.

La gestion de configuration système est indispensable à ce stade.

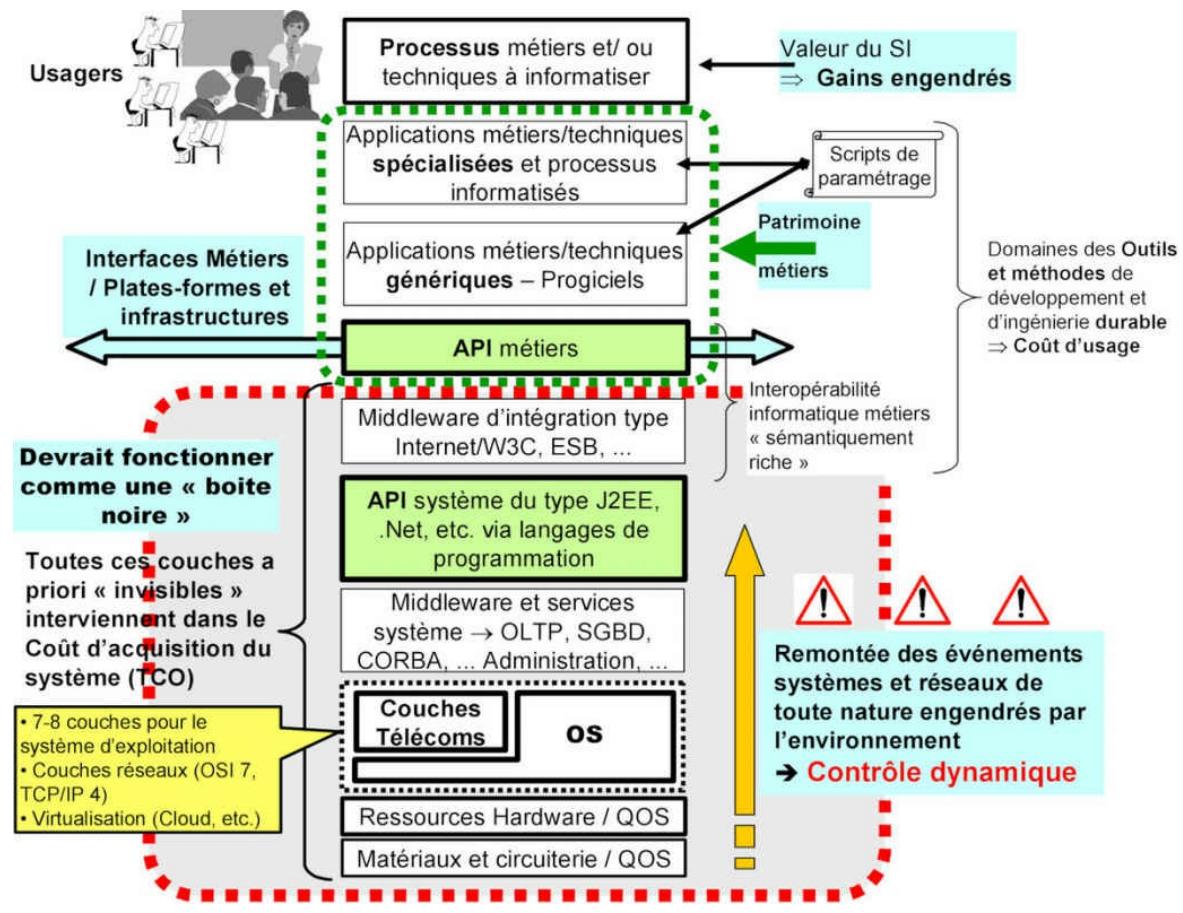


Fig. 11.6 Pile des interfaces systèmes

Le problème fondamental de la pile des interfaces systèmes est la remontée des événements en provenance des plates-formes et/ou des infrastructures, progiciels compris, qui doivent être récupérées par les API métier, car ils doivent être traduits en termes métiers pour pouvoir être exploités de façon cohérente. Ce contrôle ne peut être que dynamique (cf. le concept d'*autonomic computing*, introduit par IBM, et repris par tous : HP, Microsoft, Oracle, ... et les outillages d'accompagnement en *systems management* que l'on trouve dans ces offres).

11.4. L'organisation de l'intégration

Dans la pratique, le processus d'intégration consiste à rassembler les nouveaux modules et les modules ayant fait l'objet d'adaptations dans un environnement spécifique, c'est-à-dire la plate-forme d'intégration, qui simule raisonnablement l'environnement d'exploitation réel. L'intégration ne se fait jamais directement dans l'environnement d'exploitation pour des raisons de sûreté de fonctionnement et de contrat de service évidentes.

La séquence qui conduit de la plate-forme d'intégration à la plate-forme de production, au minimum trois étapes, est représentée par le schéma de la figure 11.7.

En amont de la plate-forme d'intégration il y a la ou les plates-formes de développement qui ne sont pas représentées sur la figure, mais dont le responsable de l'intégration doit s'assurer d'une compatibilité raisonnable, en particulier pour ce qui concerne la pile des interfaces systèmes rappelée par la figure 11.6. C'est ce qui est parfois appelé à juste raison *usine logicielle* [software factory] dont le rôle est de développer et valider toutes les pièces logicielles constitutives des modules à intégrer de façon standardisée.

En entrée du processus d'intégration, le responsable dispose de la liste des pièces/modules à intégrer et de la place de ces modules dans l'architecture du système, tant au niveau fonctionnel qu'au niveau organique. Il dispose également de la traçabilité de ces modules avec les exigences qui ont nécessité ces adaptations, conformément au triangle d'or de l'ingénierie système [voir les 12 principes de modélisation].

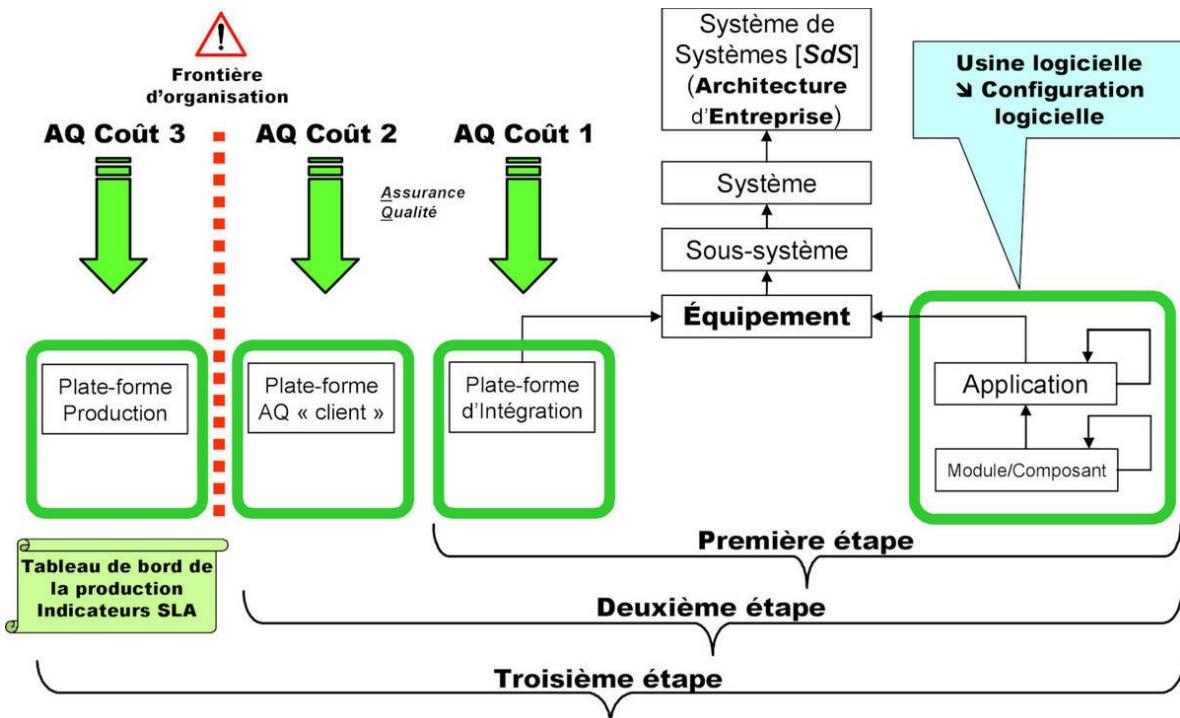


Fig. 11.7 Etapes du processus d'intégration

L'ordre dans lequel les modules vont pouvoir être intégrés dépend de la qualité du travail architectural. En effet, comme il a été dit ci-dessus, si c'est le laisser-faire qui prédomine, il faut s'attendre à ce que tous les modules dépendent les uns des autres, ce qui va se traduire par la saturation de la matrice de couplage. Au contraire, si l'architecte a mis en œuvre les bonnes pratiques architecturales : architecture modulaire hiérarchique, architecture en couches, architecture des interfaces, les relations entre modules vont pouvoir être traduites dans la structure centrale qui pilote l'intégration : l'arbre d'intégration (cf. réf. [4]). Cet arbre d'intégration est l'un des résultats fondamentaux du processus de conception de l'architecture. Pour l'intégration, la situation se présente comme suit [figure 11.8].

$$h \cong \log_{scale}(N)$$

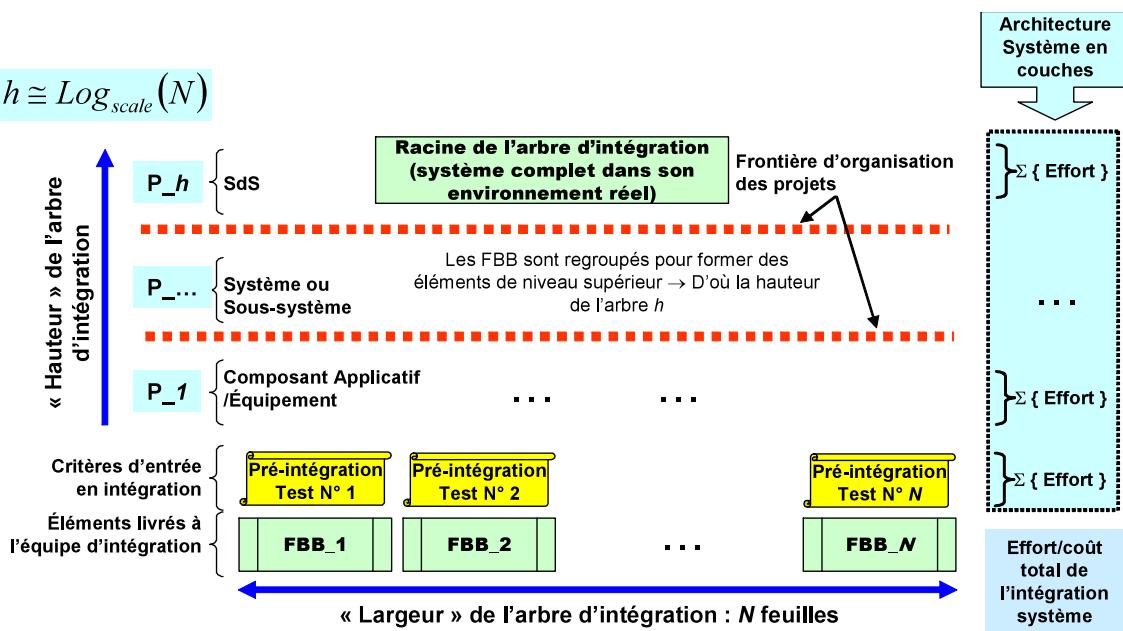


Fig. 11.8 L'arbre d'intégration

Chacun des modules entrant en intégration se présente en entrée du processus avec un ensemble de tests dit de *pré-intégration*, requis par le responsable d'intégration, et qui valident le niveau de qualité du module et son aptitude à supporter les tests d'intégration proprement dits. Le but des tests d'intégration est de découvrir les erreurs liées aux couplages des modules ; ces tests ne sont pas un complément des tests fonctionnels du module, lesquels font partie des critères d'entrée en intégration, ils matérialisent la validation des aspects transverses.

Les N blocs qui constituent les modules feuilles de l'arbre d'intégration sont regroupés par petits paquets, généralement aux alentours de 7 ± 2 , pour que leurs interactions restent compréhensibles en cas de dysfonctionnement ; la combinatoire de ces modules pouvant être décrite par des workflows ad hoc (c'est-à-dire une grammaire des enchaînements autorisés). La taille moyenne des paquets définit l'échelle de l'arbre d'intégration, et finalement sa hauteur qui varie comme le Log du nombre de modules, comme indiqué sur la figure 11.8.

L’arbre d’intégration, déduit de l’architecture, traduit la mécanique de regroupement et d’enchaînements des modules jusqu’à la racine de l’arbre qui est le système complet.

La somme des tests à effectuer sur chacun des nœuds de l’arbre, en partant des tests de pré-intégration est une bonne mesure de la complexité de l’intégration, et la somme des coûts, à droite sur la figure 11.8, traduit le coût de l’organisation de cette complexité.

L’un des principaux résultats qui émerge des études sur les tests, au sens IVVT, a été en effet de valider que toute la remontée du cycle de développement dans sa formulation en V doit, mais surtout peut être anticipée dès la phase de conception du système. Après, il est trop tard, ou en tout cas beaucoup plus coûteux, pour faire les réajustements indispensables. La qualité du travail architectural doit donner suffisamment d’information pour mettre en œuvre des modèles d’estimation comme ceux que nous avons présentés dans nos ouvrages récents (cf [2] et [4]). Du point de vue de la complexité et des coûts, on peut représenter la situation comme suit (figure 11.9).

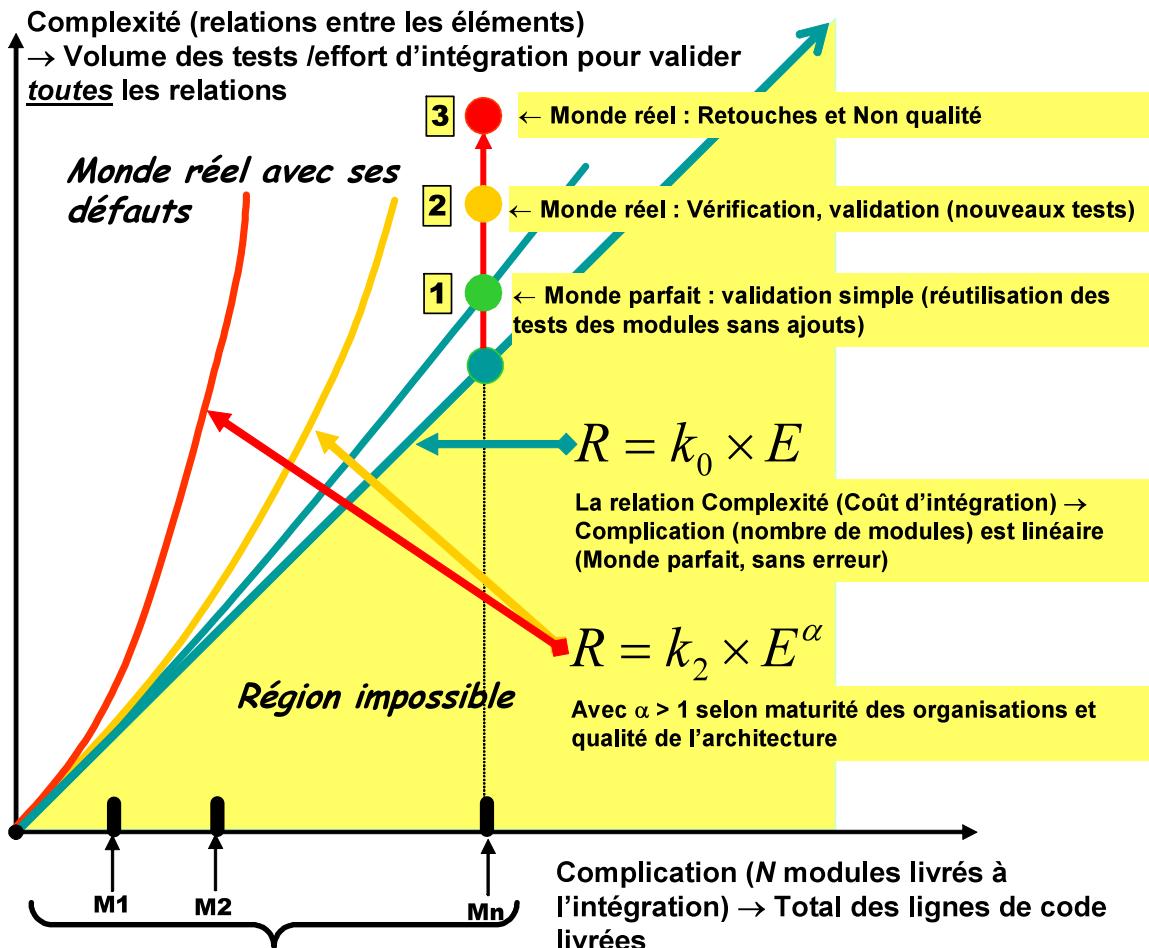


Fig. 11.9 Complexité complication de l'arbre d'intégration

Dans un monde parfait, sans erreur ni défaut, la complexité et le coût suivent la trajectoire 1 quasi linéaire. C'est ce que traduit la première relation $R=k_0\times E$ dans laquelle E correspond à l'effort (volume de tests) à fournir pour valider tous les contrats d'interfaces des modules à intégrer.

Dans le monde réel raisonnablement organisé pour purger les erreurs, la complexité va croître comme la taille de l'arbre d'intégration, d'où l'apparition d'une loi de puissance (réf. [4]). L'arbre d'intégration matérialise le coût d'organisation de la complexité, en fait la construction d'une arborescence directement issue de l'architecture, ce qui permet d'anticiper la nature des coûts. On pourrait ici introduire des distinctions faites par le

modèle d'estimation COCOMO II, dans sa version 2000, qui fait intervenir la maturité de l'organisation.

Une maturité excellente, niveaux 4 ou 5, dans l'échelle CMMI se traduira par un facteur d'échelle α légèrement supérieur à un, voire inférieur si réutilisation. Les tests sont réutilisés au maximum ce qui fait que le coût additionnel croît, mais cette croissance est faible.

Une maturité moyenne, niveaux 2 ou 3, se traduira par un coefficient α élevé, typiquement 1,2 à 1,5. Le travail a été moins bien anticipé, on découvre plus de problèmes au fil de l'eau, donc des reprises de travail que l'on croyait acquis sont nécessaires, mais la bonne maîtrise des domaines clés du niveau 2 limite la casse.

L'absence de maturité se traduit toujours par des coûts d'intégration élevés et un taux de retouches important après livraison à la production [SLA médiocre, MCO coûteux]. Dans le cas pire, encore fréquent comme l'indiquent les statistiques du *Standish Group*, on peut découvrir dans la période d'intégration des problèmes graves liés à la nature transverse de certaines exigences comme les performances ou la sûreté de fonctionnement ; la valeur du coefficient α va tendre vers deux. Les projets d'interopérabilité sont souvent dans cette catégorie par optimisme (et manque de lucidité !), manque d'anticipation et absence de modèles sémantiques qui caractérisent la nature de l'information qui s'échange entre les systèmes.

Ce qui caractérise le mieux l'absence ou le manque de maturité est la qualité du travail effectué pour expliciter rigoureusement l'architecture des interfaces. La raison d'être des interfaces est de découpler les modules ou les systèmes qui sont de part et d'autre de l'interface, comme les API métiers de la figure 11.6. Si l'interface ne joue pas ce rôle, alors il y aura automatiquement des couplages non maîtrisés. Les projets correspondants ne seront pas indépendants, ou beaucoup moins qu'ils pourraient l'être ; c'est une situation typique des projets d'interopérabilité mal conçus. Dans ce cas la trajectoire projet nominale de type 1, s'écartera de plus en plus pour aller vers des trajectoires de type 2 ou 3, catastrophiques du point de vue des coûts et du SLA.

11.5. Mesurer la complexité par les tests d'intégration

La pertinence d'une mesure de la complexité qui repose sur les tests d'intégration est donc bien fondée. Ces tests sont directement issus de la conception (rôle majeur des interfaces et des protocoles de mise en œuvre) dont ils sont la traduction en termes de coûts.

Le mot complexité est doté d'une riche polysémie qui peut rendre son emploi douteux si l'on ne prend pas quelques précautions^[2]. Dans le monde des systèmes informatisés on peut la définir en faisant une distinction de bon sens entre le nombre d'éléments constitutifs du système, et le nombre de relations, statiques et dynamiques, que ces éléments entretiennent entre eux et avec l'extérieur.

On dira qu'un système est **complexe** s'il y a de nombreuses relations entre les éléments constitutifs du système et avec l'environnement dans lequel il s'intègre. Décompter les relations/couplages est un exercice plus difficile que de compter les éléments car il y a des relations bien visibles, explicites, et des relations cachées, implicites, qui ne se révéleront qu'à l'usage : c'est le phénomène dit d'*émergence* , bien connu des théories de la complexité [systèmes dynamiques]. La panne d'un système est un phénomène émergent redouté qui résulte d'une incohérence dans l'état global du système, lequel état n'a pas été visité lors de la validation, vérification et tests, c'est-à-dire lors de l'intégration. On en donnera un exemple simple ci-après avec la mutualisation et l'interopérabilité.

On dira qu'un système est **compliqué** s'il comporte beaucoup d'éléments. D'où l'importance des analyses simplement volumétriques : on compte les instructions des programmes, on compte les défauts constatés [RA/Rapports d'anomalies] et les actions correctrices [mises à jour ponctuelles, patches, ...] qui en résultent, on compte les équipements informatiques, on compte les usagers, on compte les programmeurs, on compte les messages d'erreurs et les aides en ligne, etc. ... etc. Mais on oublie souvent de dénombrer les états du système qui est cependant le facteur prépondérant de la complication, les états mémoire dont la combinatoire peut vite devenir immense.

Pour bien appréhender la complexité, il faut donc soigneusement distinguer son aspect complication, c'est-à-dire un simple dénombrement (par exemple le nombre de règles et/ou de contraintes, le nombre de lignes source, le nombre de modules, etc.), et son aspect relations/couplages. Nous réservons

l'usage du mot *complexité* à l'aspect dénombrement des relations/couplages qui prend en compte la dimension dynamique des interactions, lesquelles seront matérialisées par des tests d'intégration, à un moment ou à un autre.

Par exemple, rendre deux systèmes interopérables c'est objectivement augmenter la complication, par le simple fait que les lignes de code et/ou les données (en y incluant les événements) des deux systèmes s'additionnent. Mais c'est aussi et surtout augmenter la complexité, car les deux systèmes vont échanger des informations et interagir de diverses façons. Il va donc y avoir de nouveaux couplages qui par définition n'existaient pas avant l'opération de fusion. Des éléments de chacun des systèmes vont ainsi se trouver « couplés ». Ainsi se crée une nouvelle relation qui stipule que toute modification de l'un des éléments couplés ne peut se faire sans l'accord de l'autre. Si l'un des systèmes a besoin d'une fonction qui existe déjà chez l'autre, il est inutile de la re-développer, il suffit de pouvoir l'utiliser en l'état à l'aide d'un protocole *ad hoc*, là où elle est, par exemple avec un simple workflow. C'est un nouveau type de couplage, c'est-à-dire une relation de partage. Cette nouvelle situation est schématisée par la figure 11.10.

Le schéma met en évidence un aspect totalement contre-intuitif relatif au nombre d'états résultant de ce couplage. Il faut en effet considérer l'ensemble des parties des états de chacun des systèmes, d'où la combinatoire « explosive » (si rien n'est fait, car fonction du nombre de pièces) en $2^{n_1+n_2}$, c'est-à-dire 2 à 2, 3 à 3, ... etc., soit l'ensemble des parties.

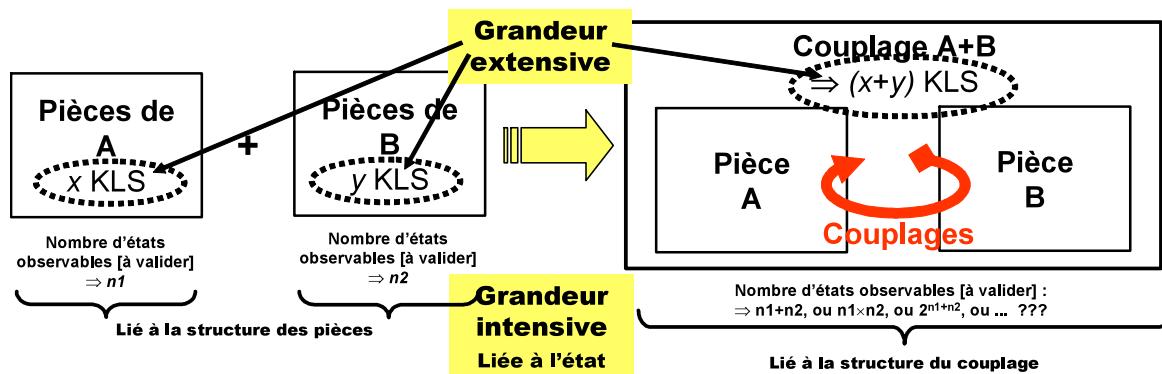


Fig. 11.10 Influence du couplage sur le nombre d'états à valider

Le nombre de lignes de code est une grandeur additive : celles-ci s'additionnent. Le coût moyen de ces lignes de code est une grandeur intensive car le coût dépend de l'environnement, comme nous le précise le modèle COCOMO. 1.000 LS seules ne sont pas équivalentes à 1.000 LS parmi 500.000 LS car le facteur d'échelle α prend en compte ce contexte. En ce sens, le coût/effort « abstrait » selon COCOMO fonctionne comme une « température » ou une « pression » (voir réf. [4]).

Des lignes de code isolées sont plus « froides » que des lignes de code intégrées, c'est-à-dire moins coûteuses. Dans le cyberspace, par défaut, tout est intégré, tout est lié, tout est plus « chaud », ce qui maximise la potentialité de découvrir de nouveaux usages.

Dans une opération de mutualisation d'éléments d'un [ou plusieurs] systèmes, on baisse objectivement la complication car le volume des textes (code et/ou données mutualisés) se trouve factorisé dans une bibliothèque et/ou une base de données. Mais on augmente corrélativement la complexité, car là où il n'y avait pas de relation, il en existe nécessairement de nouvelles (couplages par réutilisation, par partage de ressources, etc.), ne serait-ce que pour utiliser/référencer/modifier l'élément mutualisé.

Pour que la mutualisation soit rentable il est impératif que le gain engendré par la baisse de la complication (moins de code et/ou de données), soit largement supérieur au coût de la mutualisation. Un élément mutualisé sera par définition plus sollicité après l'opération de mutualisation qu'avant. Or l'occurrence des pannes est une fonction monotone croissante de l'usage. Par conséquent son niveau de maturité doit être très supérieur à ce qu'il était avant mutualisation. Moralité : de nouveaux scénarios de tests seront nécessaires pour « durcir » l'élément, car s'il tombe en panne, ce sont tous les éléments appelants qui à leur tour ne rendront pas le service attendu. Si rien n'est fait il y aura dégradation inévitable du contrat de service, conformément à la loi générale d'augmentation du désordre (entropie du système). Le code mutualisé doit être programmé en priorité à tout autre et intégré le plus vite possible de façon à l'exposer le plus fréquemment possible via la mécanique d'intégration continue.

Beaucoup de DSIs ont été abusés par des décisions hâtives, sans une vraie réflexion, souvent poussés « au crime » par les éditeurs et/ou leurs consultants, vendant de la simplification sans s'intéresser à la complexité, quand celle-ci n'est pas purement et simplement ignorée : le cas des EAI, technologies au demeurant intéressantes, mais extrêmement dangereuses une fois mises entre des mains inexpérimentées. Le non-dit étant avant tout d'éviter de parler de ce qui fâche, et si cela ne marche pas, on rajoutera une couche d' *add on* censés régler le problème.

On peut illustrer ces différents phénomènes à l'aide du diagramme figure 11.11.

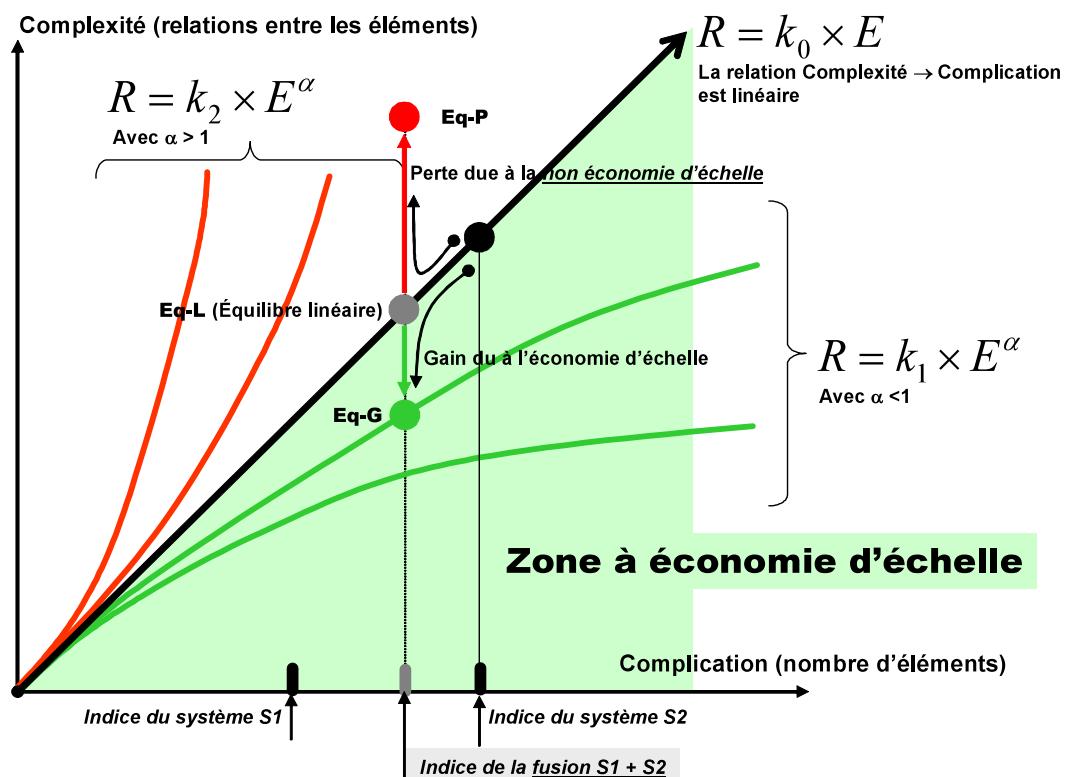


Fig. 11.11 Economie d'échelle et complexité

Ce qu'il faut absolument garantir dans une opération d'interopérabilité c'est que le coût après fusion des systèmes S1/S2, reste linéaire, c'est-à-dire que la baisse de l'indice de complication s'est accompagnée corrélativement d'une baisse proportionnelle du nombre des relations (ingénierie des interfaces),

une fois les systèmes interopérés/fusionnés ; ceci est matérialisé dans la figure par le point d'équilibre linéaire Eq-L.

Une opération réussie se caractérisera par une économie d'échelle mesurable, d'où l'obtention d'un gain. Ce gain s'obtient grâce à un investissement en architecture qui permet de faire chuter le nombre de relations et d'atteindre le point Eq-G, grâce à l'architecture des interfaces entre S1 et S2. L'interopérabilité de plusieurs systèmes est d'abord un problème d'architecture qui doit être posé comme tel, indépendamment des technologies utilisées, conformément aux objectifs de cette mise en interopérabilité.

Une opération ratée, résultant de l'impasse sur l'architecture des interfaces, va se traduire certes par une baisse de la complication dans un premier temps, par exemple moins de lignes de code à gérer, partie visible de l'opération, mais accompagnée d'une augmentation corrélative du nombre de relations qui va se traduire par une augmentation du coût d'intégration et/ou du coût de non qualité. Du fait de l'augmentation du nombre des relations, il y a une augmentation du travail de vérification/validation indispensable au maintien du contrat de service (SLA). En conséquence, on se retrouve au point Eq-P. Si rien n'est fait ce sont les usagers qui paieront la note via le SLA médiocre et le MCO coûteux.

La définition intuitive de la complexité : **plus complexe = plus de tests = plus coûteux**, peut donc être considérée comme validée. Mais à une condition : expliciter rigoureusement l'architecture du système, tant au niveau fonctionnel qu'au niveau organique, et surtout les interfaces.

La structure du coût de cette complexité peut être anticipée et estimée comme un résultat de l'architecture, en particulier par la connaissance des matrices de couplages qui est un critère de terminaison du processus de conception du système.

11.6. Recommandations et bonnes pratiques

Il est contre-productif, voire futile, de continuer aujourd'hui à séparer artificiellement l'ingénierie des systèmes d'information et l'ingénierie des

équipements massivement informatisés, anciennement baptisés systèmes industriels. La frontière entre les deux est devenue si poreuse que la distinction n'est désormais plus pertinente. Les langages de programmations généraux se sont banalisés, il y a du Java partout, dans nos téléphones mobiles, dans les systèmes de défense et de sécurité, dans tous les IHM des systèmes d'information récents et dans leurs portails Internet, etc. ... etc., en attendant d'autres langages, comme peut-être Python, dans la communauté Internet. L'ingénierie des systèmes informatisés, quelle que soit leur nature, est **une** : c'est celle de l'entreprise numérique qui doit savoir gérer toute l'information qui en constitue l'intelligence même, et en exploiter la valeur.

Il ne faut plus penser ni plateforme, ni infrastructure, il faut penser flux d'**information**.

La machinerie numérique, plates-formes et infrastructures, qui supporte l'ensemble des processus est aujourd'hui complètement banalisée et virtualisée, il n'y a plus de machines propres aux systèmes industriels, et depuis longtemps, sauf dans quelques niches très spécifiques. Une grande variété d'organes d'interactions, capteurs et effecteurs, entre les hommes et les machines, entre les machines elles-mêmes dont certaines sont des robots capables d'initiatives, ont vu le jour, s'adaptant de mieux en mieux aux besoins de performance des acteurs, et mobiles de surcroît.

Les méthodes d'ingénierie sont fondamentalement les mêmes, ce qui se traduit par des familles d'outils de plus en plus génériques, avec des langages de modélisation à large spectre comme BPML, WSDL, UML, SysML, etc. ..., voire des DSL, permettant une description rigoureuse des systèmes et des services, au bon niveau d'abstraction.

Les langages de modélisation sont désormais des instruments indispensables à une bonne compréhension des comportements et de la dynamique des systèmes, mais la notion fondamentale que le chef de projet doit mettre en œuvre au sein de ses équipes est bien celle du savoir penser **modélisation**, c'est-à-dire la pensée systémique qui est celle de l'ingénierie système.

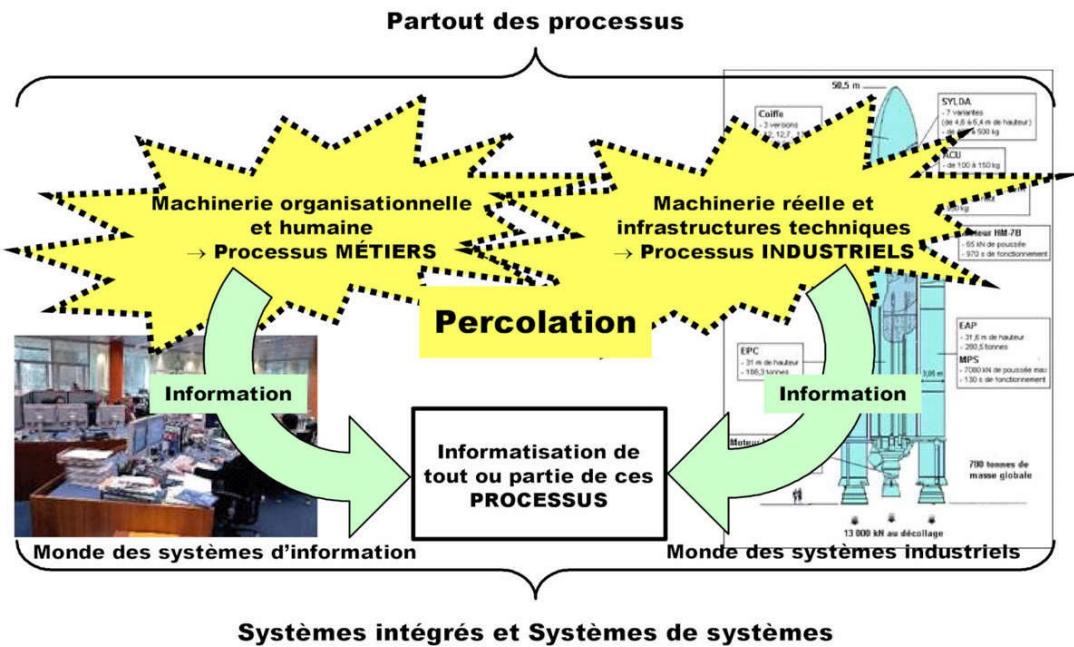


Fig. 11.12 Unification des ingénieries

Tant pour l'organisation des processus métiers que pour l'agencement des processus industriels l'architecture est le point de passage obligé d'une ingénierie proactive, prédictive et durable, qui sait prévoir ce qui va se passer et en mesurer l'avancement.

Il faut cesser de penser que l'architecture est un centre de coûts. Les bonnes pratiques qualité comme la mise en œuvre intelligente du CMMI montrent à l'évidence que tout investissement effectué en amont sera largement compensé par des gains en aval d'un facteur 10 à 50 pour l'intégration, beaucoup plus (> 100) après la mise en production.

Une incohérence d'interface non détectée peut conduire à des retours en développement / maintenance de centaines de pièces qui dépendent directement ou indirectement de cette interface.

Le résultat visible du processus de conception du système est une famille de **modèles cohérents** qui vont permettre aux architectes de répondre aux questions de toute nature que pose la réalisation, de s'assurer que les pièces

du puzzle forment un tout qui satisfait aux exigences globales et que ces pièces peuvent être assemblées de façon systématique :

- Découpe fonctionnelle et organique permettant aux équipes projets de travailler en parallèle : c'est le fondement de l'**agilité**.
- Simultanément à la découpe, ingénierie des tests permettant de valider fonctionnellement et organiquement le découpage, pour chacune des pièces prises individuellement : c'est le concept d'**architecture testable**, c'est-à-dire le *Test Driven Development*, fortement recommandé par les méthodes agiles.
- **Contractualisation des interfaces** à tous les niveaux et ingénierie de ces interfaces : tout doit être explicité pour démarrer en parallèle les scénarios de validation système, les pilotes et les « bouchons » pour le monitoring des tests.
- Arbre d'intégration qui est le protocole d'ordonnancement des pièces à assembler et des tests d'intégration à appliquer à chacune des étapes d'assemblage : c'est le concept d'**usine système logiciel et d'intégration** continue [*software factory*, 6-Sigma], analogue à une chaîne de montage, indispensable à la mise en œuvre de l'agilité.
- **Ingénierie des exigences**, en particulier toutes celles qui intéressent la globalité du système comme les performances, la sûreté de fonctionnement, la sécurité, l'interopérabilité, l'évolutivité de façon à disposer le plus tôt possible de scénarios de validation globale, de bout en bout : c'est le préalable de la démarche « centrée architecture » qui garantit que le système réalisé satisfera les exigences client.

Chacun doit se convaincre qu'il est désormais plus simple, moins coûteux, plus sûr de raisonner sur des modèles partagés entre tous les acteurs et parties prenantes que sur le système lui-même. L'arbre d'intégration qui en est le résultat est l'instrument concret, pragmatique, qui permet de synchroniser et de contrôler les trajectoires des différents projets qui contribuent à la réalisation du système conforme aux besoins de ses usagers réels.

Les textes associés à ces modèles sont la matière première de la mesure de complexité fondée sur l'activité IVVT.

Les bonnes pratiques de modélisation et leur mise en œuvre par chacun des acteurs concernés est le sésame qui permet à l'entreprise numérique d'exploiter son bien le plus précieux : l'INFORMATION, au service de ses usagers, et d'en apprécier la valeur au sens économique du terme.

11.7. Bibliographie du chapitre

- [1] Y.Caseau, *Urbanisation, SOA et BPM*, Dunod, 2011, 4^e édition.
- [2] B.Mesdon, *Les points de fonctions en ingénierie logicielle*, Hermès 2012.
- [3] J.Printz, *Architecture logicielle*, Dunod, 2012, 3^e édition.
- [4] J.Printz, N.Trèves, B.Mesdon, *Estimation des projets de l'entreprise numérique*, Hermès, 2012.
- [5] J.Printz, B.Mesdon, *Écosystème des projets informatiques*, Hermès 2006.

Notes

[1] Fixées par le SEI ; voir <http://www.sei.cmu.edu/reports/92tr020.pdf>

[2] Voir notre étude sur le site du LIX, [laboratoire Informatique de l'Ecole Polytechnique] :
http://www.lix.polytechnique.fr/~liberti/cal07/presentations/printz-mesures_de_complexite.pdf

Conclusion

La conception de logiciels reste avant tout une activité humaine ; aussi, n'est-elle pas exempte d'erreurs. Des erreurs seront commises lors de l'expression des besoins et vont se traduire par des spécifications incomplètes ou encore incohérentes. Des erreurs seront faites lors de la définition de l'architecture et vont se traduire par une application inefficace, difficilement testable ou encore non maintenable de sorte que toute correction d'anomalie en entraînera de nouvelles. Enfin, des erreurs seront commises lors des phases de codages. Toutes ces erreurs vont engendrer des défauts dans la réalisation finale qui pourront conduire à des défaillances ou dans les cas extrêmes à des pannes totales du système logiciel.

Quelles que soient les méthodes de conception utilisées, ces erreurs ne peuvent être totalement évitées ; il convient donc à la fois de réduire leur impact par la mise en place d'une architecture défensive et à la fois de viser à leur élimination de façon aussi complète que possible, comme on peut le faire dans le domaine chirurgical où toutes les précautions sont prises avant l'intervention mais aussi pendant et après celle-ci. Pour cela il est nécessaire de tester à tous les niveaux du cycle de vie des logiciels en employant une stratégie adaptée. Lors des phases de spécification, on privilégiera des tests dits *statiques*, c'est-à-dire basés sur l'analyse des différents textes produits

(au sens large). Une fois le logiciel réalisé et assemblé, on emploiera plus fréquemment des tests dits *dynamiques* c'est-à-dire basés sur l'exécution du logiciel. Parmi ces différentes étapes, la phase d'intégration est certainement la phase la plus sensible et celle à laquelle il faudra apporter le plus grand soin, en particulier du point de vue des tests (voir à cet effet les chapitres 6 et 11).

S'il est simple d'exprimer le besoin de tester aux différents stades de développement, il est plus complexe de mettre en œuvre cette exigence. En effet, tester est un problème difficile. Difficile au sens mathématique car des résultats d'indécidabilité rendent caduc tout espoir de résoudre ce problème de façon simple et purement automatique. Difficile également du point de vue pratique car, même dans les cas « faisables » au sens théorique du terme, la combinatoire associée aux différentes exécutions de n'importe quel logiciel rend impossible une démarche brutale basée sur la seule énumération de cas de test. Par ailleurs, les différents facteurs, humains et techniques, à prendre en compte lors de la conception et l'exécution des tests demandent une grande habileté aux chefs de projet et aux équipes en charge des tests.

Il faut donc mettre en place une démarche empirique, c'est-à-dire basée sur **l'expérimentation**, qui permettra de définir des objectifs de test clairs (qui seront principalement la découverte de défauts), des moyens d'observations et qui, en s'appuyant sur des méthodes éprouvées, permettra de sélectionner les cas de tests « pertinents ». Pour cela, le testeur pourra faire reposer son raisonnement sur les seules spécifications, et l'on parle alors de tests *boîtes noires*, ou se permettra d'utiliser en plus des spécifications des détails des réalisations (comme le code) pour construire ses tests, et l'on parlera alors de tests *boîtes blanches*. Dans ce dernier cas, il faudra prendre garde de n'utiliser ces détails que pour exercer une instruction particulière ou parcourir un chemin donné tout en fondant le choix de l'objet du test et du résultat attendu sur la seule analyse des spécifications (l'expérience du testeur pourra aussi bien entendu être utilisée avec profit). Si les détails de réalisation ne sont pas utilisés pour déterminer les jeux de valeurs employés pour les tests, ils peuvent être le moyen de mesurer la qualité des tests produits en caractérisant les chemins parcourus lors de l'exécution des tests en fonction de leur couverture des instructions, globalement ou sous certains critères liés à l'accès aux variables. Dans certains cas particuliers, l'utilisation de code

permet néanmoins de générer des tests pertinents de façon automatique (comme nous l'avons abordé au chapitre 9).

Tous ces aspects ont été étudiés à travers cet ouvrage ainsi que le nécessaire besoin de mettre en place des procédures de gestion des tests incluant l'estimation des coûts, la planification des activités, et point essentiel dans une démarche agile de développement, une gestion en configuration des tests. Il est en effet illusoire de vouloir employer avec profit une démarche basée sur l'itération et l'incrémantation sans avoir une gestion de configuration sérieuse et outillée.

Après de nombreuses années dans l'accompagnement d'industriels confrontés à la problématique de la qualité des logiciels produits et dans le constat que les tests demeurent un point essentiel et indispensable dans l'amélioration ou le maintien de cette qualité nous pouvons retracer dans cette conclusion quelques facettes saillantes de ces enjeux.

Historiquement, les tests ont eu un étrange destin dans le développement des technologies de l'information. Dès le départ il était clair que l'on était sur un point dur, comme on peut s'en convaincre en lisant l'histoire du projet SAGE^[1], et les conférences du *NATO Science Committee* qui fondèrent le génie logiciel, en 1968-1969. Les modèles d'estimation des coûts/délais de développement des projets logiciel, fondés sur l'analyse statistique des données de centaines de projets, donnent dès les années 1970-1980 des fourchettes de coût de 30-40 % pour la partie test. La NASA estime que pour les projets spatiaux comme la navette, 70 % du coût est consacré aux tests.

On sait donc tout cela dès le départ, mais la communauté industrielle, et surtout la communauté académique du génie logiciel, ne font rien, ou presque. La préférence va à l'étude des langages de programmation, puis aux outils de conception dont UML est une sorte d'aboutissement. Attitude d'autant plus surprenante de la part de scientifiques, puisque le fondement de la démarche scientifique est l'expérimentation. Les tests ne sont rien d'autre que des protocoles expérimentaux dont le but est de rechercher les failles. La validité d'une théorie – un programme est une théorie – est fondée sur l'expérimentation. Un scientifique (cf. A. Einstein) propose une théorie, et des dizaines, centaines, ... d'autres essayent de la « falsifier », pour reprendre la terminologie de K. Popper^[2]. Le summum est atteint avec le LHC,

instrument colossal développé par le CERN, pour valider l'hypothèse de l'existence du boson de Higgs, pierre angulaire du modèle standard des particules et qui a donné entre l'écriture de la seconde et la troisième version de cet ouvrage des résultats spectaculaires dans cette démonstration.

La communauté académique n'a pas initialement fait l'effort de conceptualisation et d'abstraction concernant la validation, au sens large, des grandes constructions que sont devenus les programmes, dès les années 1970-1980. Les tests ont longtemps été vus comme un vaste bricolage informe dont il vaut mieux se détourner. C'est tout le contraire de ce qui a été fait pour le hardware, où dès le départ on s'est intéressé au « *design to tests* » comme une activité indispensable et inséparable de la conception des circuits. Inutile de concevoir ce que l'on ne saura pas valider ; on peut néanmoins noter que depuis quelques années cette démarche a fait son chemin dans la conception des logiciels et que la démarche théorique visant à ne voir les tests que comme un sous domaine des *preuves mathématiques* est en train de laisser place à une vision plus pragmatique tout en capitalisant sur ces efforts de recherche anciens et variés ; voir à cet effet le chapitre 10.

Pour le logiciel, créé de toutes pièces par l'activité des programmeurs, la présence de défauts est imputable à celui qui l'a conçu. Ce qui fait que le statut de l'erreur est dans ce domaine complètement différent de ce que l'on peut rencontrer dans les domaines de la mécanique ou de l'électronique. Les matériaux que l'on trouve dans la nature contiennent des défauts, il faut faire avec et il n'y a pas de discussion possible. De même penser que l'on peut programmer sans erreur est une illusion que seuls ceux qui n'ont jamais programmé peuvent imaginer. La différence entre un bon programmeur et un mauvais programmeur n'est pas dans le nombre d'erreurs commises, mais dans la vitesse avec laquelle les erreurs vont être diagnostiquées et corrigées^[3]. Poincaré, l'un des plus grands mathématiciens français, était connu pour faire des erreurs, mais elles étaient faciles à corriger et ne remettaient jamais en cause la logique générale de ses démonstrations.

La leçon à tirer est que les erreurs sont consubstantielles à l'acte de programmation. Culpabiliser est contre-productif. Certes, il vaut mieux en faire le moins possible, mais plus important, il faut surtout s'organiser en conséquence. D'où les approches comme le CMMI dont nous reparlerons plus loin.

Les seuls industriels, jusque dans les années 1980, qui se sont intéressés sérieusement aux tests sont les constructeurs d'ordinateurs, les réalisateurs de systèmes informatisés où les contraintes de sûreté sont fortes (défense, spatial, aéronautique, nucléaire, etc.), les éditeurs de logiciels. Nécessité a fait loi. Retirer un logiciel du marché parce que trop défectueux est une expérience qui peut conduire à la faillite. La pression du risque incite à la prudence, et finalement à la sagesse.

La démarche qualité que l'on voit se développer tout au long de la décennie 1980 met l'accent sur le préventif, en expliquant que c'est moins coûteux que le curatif. On se souvient du titre provoquant du livre de Crosby, « *Quality is free* ». Si l'on sait faire les tests qui vont éviter les défaillances lors de l'exploitation réelle du système, c'est gagné ! Mais les industriels du milieu défense, spatial, ... sont minoritaires, et il ne parle pas le même langage que ceux des services, banques, assurances, etc. L'informatique de gestion, les systèmes d'information, sont souvent opposés à l'informatique industrielle, jugée plus « scientifique ». Le message admis maintenant « qualité = test » ne passe pas dans ces années.

Le basculement s'opère dans les années 1980 avec le développement de l'informatique distribuée. En moins d'une décennie, les ordinateurs centraux éclatent en centaines, puis milliers de serveurs, eux-mêmes environnés de dizaines de milliers de postes de travail « intelligents » (des millions dans le cas d'Internet) qui intègrent des millions de lignes de code. Le besoin d'informaticiens explose, et il faut recruter bien au-delà des filières de formation informatique. Le nombre d'informaticiens, dans une grande banque, peut largement dépasser les 10 000, dont beaucoup ont été formés « sur le tas » !

De façon subreptice, l'informatique des systèmes d'information, est devenue aussi complexe que celle des systèmes industriels, pour d'autres raisons. Le système d'information « vit » au rythme de l'entreprise, et il doit évoluer à la même vitesse. Les systèmes d'information interopèrent entre eux, dans l'entreprise, et hors de l'entreprise, formant ce que l'on appelle une *entreprise étendue*. Mal maîtrisée, l'interopérabilité devient un risque inacceptable : les pannes de l'un se propagent chez l'autre, fragilisant les chaînes de valeur de l'entreprise. La vitesse de diagnostic des

défaillances, paramètre fondamental du contrat de service, est ralenti par l'hétérogénéité des plates-formes de production.

Les DSI ont depuis accepté qu'il était essentiel de s'occuper sérieusement du problème. On peut prendre comme exemple de cette prise de conscience l'intérêt manifesté depuis quelques années de la démarche CMMI (ou de démarches équivalentes comme COBIT, ITIL, etc.). Le cas du CMMI est intéressant car la démarche a été initialement promue par le département américain de la Défense, *via* le *Software Engineering Institute*, installé par le DOD dans l'université Carnegie Mellon, à Pittsburgh. C'est une méthode, en fait d'origine IBM, issue des milieux de la défense, qui met fortement l'accent sur les processus d'ingénierie, et sur les actions qualité intégrées à ces processus. Les processus de validation, vérification, tests y occupent une place centrale. Le développement de la maturité est objectivé à l'aide d'indicateurs permettant de mesurer les progrès accomplis. Les industriels avec lesquels il nous a été donné de travailler (Bouygues Telecom, Orange, PSA, pour n'en citer que deux qui ont participé ou participent à nos enseignements au CNAM, à l'UPMC ou à Paris Nanterre) ont des processus d'intégration, validation, vérification, test (IVVT) extrêmement fouillés et rigoureux. C'est une condition nécessaire à la qualité des livraisons.

La relation entre l'architecture logicielle^[4], et la capacité à découvrir et corriger rapidement les erreurs commence à être mieux comprise. Les promoteurs des méthodes agiles^[5] dont un des courant est « *l'eXtreme Programming* » préconisent le « *Test Driven Development (TDD)* » ou le « *Model Driven Development (MDD)* » ou encore le « *Test Driven Development (TDD)* » qui promeuvent le développement des tests avant celui du logiciel. Dans le monde UP (*Unified Process*) qui accompagne UML, on parle de processus de développement « *Architecture Centric* ».

L'architecture est une condition nécessaire au bon usage des outils d'aides à la fabrication des tests. Là encore, il faut bien comprendre, et en être convaincu, que les tests, comme les programmes, ne se fabriqueront jamais de façon complètement automatique. Faire de bons tests est aussi difficile que de faire de bons programmes, tant la combinatoire des cas possibles est immense, comme on l'a vu dans cet ouvrage. C'est une autre logique qui doit intégrer une connaissance approfondie de l'écosystème dans lequel le

programme sous test va opérer en particulier dans les systèmes embarqués de plus en plus présents.

La profession elle-même s'organise. Les tests sont considérés comme un actif des organisations de développement, au même titre que les programmes. Dans une communication personnelle, la société Atos-Origin qui a réalisé le système d'information des Jeux olympiques de Pékin, nous a communiqué les chiffres de 200 000 heures de travail pour réaliser l'intégration du système et 9 000 scénarios de tests pour la validation, pour garantir le bon fonctionnement du système au jour J, le 8/08/2008, à 8h08.

Dans les projets d'intégration, très nombreux avec l'utilisation des progiciels et/ou de la réutilisation de composants déjà développés par l'entreprise, la part de l'IVVT devient prépondérante.

Le métier de testeur prend corps. Les organismes professionnels comme le CIGREF, qui regroupe les DSI des grands groupes français, et le SYNTEC, qui regroupe les sociétés de services, s'en préoccupent de plus en plus.

Ce présent ouvrage est le reflet de plus de vingt-cinq ans de pratiques et de réflexions sur ce que l'on peut considérer comme les bases du métier de testeur.

Notes

- [1] K. Redmond, T. Smith, *From Whirlwind to Mitre, The R&D story of the SAGE Air Defense Computer*, MIT Press, 2000.
- [2] K. Popper, *Logique de la découverte scientifique*, Payot.
- [3] J. Printz, *Productivité des programmeurs*, Hermès-Lavoisier.
- [4] J. Printz, *Architecture logicielle*, 2^e édition, Dunod, 2009.
- [5] J. Printz, *Écosystème des projets informatiques – Agilité et discipline*, chez Hermès-Lavoisier.

Exemple de QCM

Nous proposons dans ce chapitre quelques questions à choix multiples en relation avec les différents chapitres du livre et couvrant le syllabus de l'ISTQB ; ces questions peuvent être utilisées dans la préparation à la certification ISTQB niveau fondation telle qu'elle est définie lors de la parution de cet ouvrage. La certification, niveau fondation, contient une quarantaine de questions auxquelles il faut répondre en 1 heure. Chaque question ne contient qu'une bonne réponse et il n'y a pas de points négatifs. Pour obtenir la certification il faut avoir plus de 65 % de réponses valides. Pour le niveau avancé, il faut répondre à un questionnaire de 65 questions à réaliser en 1 h 30 avec un niveau de difficulté associé à chaque question (de 1 à 3). Il faut obtenir plus de 65 % des points totaux.

Dans les questions qui suivent, le chiffre entre parenthèses et en italique indiqué à la fin de l'énoncé de la question correspond au niveau de difficulté de celle-ci. Les réponses commentées de ces questions sont disponibles à l'adresse <http://www.dunod.com/contenus-complementaires/9782100706082>.

Q1) Pendant les tests, il est nécessaire de : (1)

- a) Tester toutes les parties du système avec la même intensité, parce que des anomalies peuvent être détectées n'importe où.
- b) Tester l'interface utilisateur en priorité parce que ses anomalies sont les plus dérangeantes pour l'utilisateur.
- c) Tester en priorité les composants du système pour lesquels les défauts génèrent les risques les plus importants.
- d) Tester en priorité les accès aux bases de données de façon à éviter des données erronées et des inconsistances dans la base de données.

e) Tester en priorité les performances de façon à améliorer les temps de réponse.

Q2) Les tests statiques et les tests dynamiques : (1)

a) Sont deux familles de tests complémentaires.

b) Sont incompatibles : il faut choisir avant de commencer la phase de tests.

Q3) Les tests sont : (1)

a) Inutiles la plupart du temps du fait de l'amélioration continue des techniques de développement et d'intégration.

b) Indispensables pour réduire les coûts et les délais tout en augmentant la qualité.

c) Importants mais impraticables donc à éliminer.

Q4) Quelle affirmation est exacte : (1)

a) L'activité de test nécessite d'être curieux c'est pour cela que c'est préférable de la confier à des personnes sans expérience.

b) L'activité de test nécessite d'être curieux mais demande de l'expérience.

c) L'activité de test ne nécessite qu'un bon sens de la communication.

d) L'activité de test nécessite de prendre position quitte à « grossir » ou extrapolier certaines données.

Q5) Une fonction calcule un résultat à l'aide quatre paramètres entiers (codés sur 32 bits) ; il y a donc $2^{32} \times 4$ (2)

= 2 128 ≈ 1 043 possibilités distinctes d'appel de la fonction :

- a) Ce nombre est tellement grand qu'il est inutile d'essayer de tester la fonction ; la tâche est impossible !
- b) En y consacrant de l'énergie on pourra tester toutes les possibilités.
- c) En procédant avec méthode il suffit de tester la fonction avec quelques valeurs pertinentes pour avoir une très grande assurance sur sa correction.
- d) Ce cas n'arrive jamais, il est inutile de se poser ce genre de problème !

Q6) Les tests d'intégration : (1)

- a) Ne peuvent être fait correctement que si les tests unitaires ont été faits correctement.
- b) Remplacent les tests unitaires.
- c) Sont incompatibles avec les tests unitaires.
- d) Se font avant les tests unitaires.

Q7) Les méthodes agiles : (1)

- a) Mettent en avant la souplesse des programmeurs car ils doivent programmer dans différentes postures pour maintenir une forte concentration.
- b) Mettent en avant les tests qui doivent être réalisés très régulièrement.

c) Rendent inutiles les tests car le code ne peut contenir des erreurs avec ces techniques.

d) Déconseillent l'utilisation des tests car ceux-ci risquent de « casser » l'aspect agile des méthodes.

Q8) Quel terme ne désigne PAS une stratégie de tests d'intégration : (1)

a) Méthode incrémentale ascendante.

b) Méthode incrémentale descendante.

c) Méthode de la régression minimale.

d) Méthode du big bang.

Q9) La technique des tests aux limites consiste à : (1)

a) Pousser aux limites les équipes de tests en les mettant fortement sous pression.

b) Essayer d'atteindre la limite des fonctions de maturité du logiciel.

c) Faire fonctionner le logiciel aux limites de ses spécifications.

d) Faire fonctionner le logiciel le plus longtemps possible.

Q10) La technique de partitionnement en classes d'équivalence : (1)

a) Consiste à trouver des domaines sur lesquels le logiciel se comporte de façon homogène.

- b) Consiste à trouver des logiciels équivalant au logiciel à tester et à réutiliser les jeux de tests qui ont été faits pour ce logiciel.
- c) Consiste à diviser l'équipe de test en groupe de tailles et d'expériences équivalentes.

Q11) Comment désigne-t-on également les tests « boîtes noires » ? (1)

- a) Tests unitaires
- b) Tests d'intégration
- c) Tests fonctionnels
- d) Revues de code

Q12) Les tests « boîtes noires » utilisent : (1)

- a) Le code du logiciel à tester.
- b) Les commentaires du logiciel à tester.
- c) Les spécifications du logiciel à tester.
- d) Les commentaires de l'équipe de réalisation.

Q13) Les tests « boîtes blanches» utilisent (plusieurs réponses possibles) : (1)

a) Le code du logiciel à tester.

b) Les commentaires du logiciel à tester.

c) Les spécifications du logiciel à tester.

d) Les commentaires de l'équipe de réalisation.

Q14) La notion de couverture dans les tests : (1)

a) N'est pas utilisée.

b) Permet de « couvrir » une erreur faite lors des tests qui endommage le logiciel à tester.

c) Sert à vérifier qu'un ancien jeu de valeurs peut être réutilisé dans un nouveau contexte.

d) Sert à définir un objectif pour les tests.

Q15) La couverture de « toutes les décisions » (dite encore « toutes les branches ») : (1)

a) Recouvre la couverture « tous les chemins ».

b) Recouvre la couverture « toutes les instructions ».

c) Recouvre la couverture « toutes les conditions ».

d) Recouvre la couverture « tous les chemins ».

Q16) Combien de cas de tests sont nécessaires pour atteindre le critère « toutes les instructions » (C0) sachant que les deux conditions sont indépendantes :

if (condition 1)

then statement 1

else statement 2

(2)

fi

if (condition 2)

then statement 3

fi

...

a) Un cas de tests

b) Deux cas de tests

c) Trois cas de tests

d) Quatre cas de tests

Q17) Combien de cas de tests sont nécessaires pour atteindre le critère « tous les chemins» sachant que les deux conditions sont indépendantes :

if (condition 1)

then statement 1

else statement 2

(2)

fi

if (condition 2)

then statement 3

fi

...

- a) Un cas de tests
- b) Deux cas de tests
- c) Trois cas de tests
- d) Quatre cas de tests

Q18) L'effort de tests dans un projet : (1)

- a) Est de l'ordre de 20 % de l'effort global, voire moins pour les applications peu critiques.
- b) Est proportionnel au nombre de lignes de code livrées.
- c) Inclut également le temps passé par le client dans le cadre de cette activité.
- d) Doit être évalué en début de projet.

Q19) Rechercher les défauts : (1)

- a) Est très coûteux en début de projet, cela mobilise beaucoup trop de monde pour faire des inspections.
-

- b) Il est plus efficace de rechercher les défauts le plus tôt possible.
- c) Pour un défaut donné, son coût de recherche est beaucoup plus élevé en fin de projet qu'en début de projet.
- d) N'a finalement qu'une bonne efficacité en fin de projet.

Q20) Les défauts sont introduits : **(1)**

- a) Tout au long du cycle du projet.
- b) Uniquement dès que l'on programme, les défauts ne sont essentiellement que les conséquences d'erreurs de programmation.
- c) Éventuellement en conséquence d'une spécification ambiguë.
- d) Essentiellement volontairement et de manière malveillante par les équipes projet.

Q21) Que recommanderiez-vous à votre client ? **(1)**

- a) De s'impliquer dans le projet dans les phases de validation.
- b) De faire totalement confiance à la MOE qui garantira la qualité de l'application.
- c) D'être vigilant et de contrôler régulièrement les livraisons intermédiaires en fixant des jalons avec la MOE en début de projet.

d) D'être souple sur les délais de livraison, un retard peut améliorer la qualité du produit livré.

Q22) Je suis chef de projet, mon projet est en retard, que dois-je faire ? (2)

a) Livrer tel quel, je négocierai avec mon client ensuite.

b) Essayer de réduire au maximum le nombre de défauts, quitte à renégocier à la baisse les fonctionnalités à livrer.

c) Demander au client une prolongation du projet.

d) Adopter une démarche préventive, pour éviter de me trouver dans ce type de situation.

Q23) À quel moment dans un projet dois-je penser à l'activité de tests ? (1)

a) Le plus tôt possible, les tests doivent être planifiés.

b) Il est possible d'effectuer les scénarios de tests de recette dès la phase d'expression de besoins.

c) J'ai tout mon temps, le cycle de vie que j'utilise est en V, je commencerai l'activité de tests dès lors que mon équipe aura terminé le codage.

d) Je suis tellement sous pression que nous ferons les tests que si nous avons le temps en fin de projet.

Q24) Quelle affirmation sur les outils de test statiques est exacte : (1)

- a) Les outils de test statiques sont issus de technologies proches de celles utilisées par les compilateurs.
- b) Les outils de test statiques ne peuvent pas donner d'information sur le comportement dynamique du programme.
- c) Les outils de test statiques ne peuvent utiliser que lorsque tout le code est finalisé.
- d) Les outils de test statiques sont récents et donc encore immatures.

Q25) Lorsque l'on introduit un nouvel outil dans une organisation, il est vrai que : (1)

- a) Un outil « Open source » est généralement de moins bonne qualité qu'un outil propriétaire.
- b) Le développement de l'aide en ligne rend inutile la formation des usagers.
- c) L'impact sur l'organisation peut être important.
- d) L'outil étant amené à évoluer il n'est pas nécessaire de gérer cette introduction comme un projet normal.

Bibliographie

Beck Kent, *Test-Driven development*, Addison-Wesley, 2003.

Beizer Boris, *Software Testing Techniques*, Van Nostrand Reinhold, 1990.

Binder Robert, *Testing Object-Oriented Systems : Models, Patterns, and Tools*, Addison-Wesley Professional, 1998.

Burnstein Ilene, *Practical software testing*, Springer, 2003.

Caseau Yves, Urbanisation, *SOA et BPM – Le point de vue d'un DSI*, 4^e édition, Dunod, 2011.

Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach 1st Edition* John Wiley & Sons, 2011

Friedman Michael, Voas Jeffrey, *Software assessment : Reliability, Safety, Testability*, John Wiley & Sons, 1995.

Gilb Tom, Graham Dorothy, *Software inspection*, Addison Wesley, 1996.

Hetzell William, *The complete guide to software testing*, John Wiley & Sons, 1988 (existe en version électronique).

Jorgensen Paul C., *Software Testing : A Craftsman's Approach*, 4th Edition, CRC Press, 2013.

Legeard Bruno, Utting Mark *Practical Model-Based Testing: A Tools Approach*, Elsevier S&T, 2006

Legeard Bruno, Bouquet Fabrice, Pickaert Natacha, *Industrialiser le test fonctionnel – Des exigences métier au référentiel de tests automatisés*, 2^e édition, Dunod, 2011.

B.Mesdon, Les points de fonctions en ingénierie logicielle, Hermès 2012.

Myers Glenford, *The art of software testing*, 2nd edition, John Wiley & Sons, 2004.

Perry William, *A structured approach to systems testing*, 2nd edition, QED Information Sciences, 1988.

Perry William, *Effective methods for software testing*, 3rd edition, John Wiley & Sons, 2006.

Printz Jacques, *Architecture logicielle – Concevoir des applications simples, sûres et adaptables*, 3^e édition, Dunod, 2012.

Printz Jacques, Mesdon Bernard, Trèves Nicolas, *Estimation des projets de l'entreprise numérique*, Hermès-Lavoisier, 2013.

Willcock Colin & alii, *An Introduction to TCCN-3^[1]*, John Wiley & Sons, 2005.

Notes

[1] TTCN signifie *Tree and Tabular Combined Notation* ; c'est un langage de test utilisé dans le monde des télécommunications et normalisé par l'IUT, en complément du langage de spécification SDL.

Index

all pairs, [83](#), [85](#)
all singles, [82](#)
analyse statique de code, [69](#)
application client-serveur
 déploiement, [130](#)
architecture
 critère de simplicité, [15](#)
 testable, [14](#)
besoin
 phase d'analyse, [11](#)
 phase d'expression, [10](#)
Boîte
 blanche, [47](#)
 grise, [49](#)
 noire, [48](#)
bouchons, [29](#), [31](#)

buddy check, [75](#)
classes d'équivalence
construction, [89](#)

invalides, [87](#)
valides, [86](#)

COACH de XP, [21](#)

composants

à état, [95](#)
notion de, [28](#)

conception

détaillée, phase de, [16](#)
globale, phase de, [13](#)

coût, [140](#)

CQFD, [139](#)

critère de couverture

ACU, [114](#), [115](#)
ACU + P, [120](#), [121](#)
AD, [118](#), [119](#)
ADUP, [126](#), [127](#)
APU, [116](#), [117](#)
APU + C, [122](#), [123](#)
AU, [124](#), [125](#)
LCSAJ, [111](#)
MC/DC, [109](#)
PLCS, [112](#)
sévérité, [128](#)
tous les chemins, [102](#)
tous les i-chemins, [110](#)
toutes les branches, [104](#)

toutes les conditions, [105](#)

toutes les conditions multiples, [107](#)

toutes les conditions-décisions, [106](#)

toutes les conditions-décisions modifiées (MC/DC), [108](#)

toutes les décisions, [103](#)

toutes les instructions, [101](#)

cycle

de vie, [8](#)

de vie (projet XP), [22](#)

en V, [7](#)

débogage, [54](#)

délai, [143](#)

développement

agile, [17](#)

itératif « UP », [18](#)

itératif « XP », [20](#)

diagramme états transitions, [97](#)

construction, [99](#)

format général, [98](#)

fonctionnalités, [142](#)

graphe

de contrôle, [66, 100](#)

de flots de données, [67, 113](#)

indépendance métier et plate-forme, [129](#)

inspection, [74, 77](#)

intégrat, [131](#)

aspect modulaire, [133](#)

aspects fonctionnels, [134](#)

attributs indispensables, [132](#)

structure fine, [137](#)

intégration

risques et difficultés, [135](#)

verticale, [136](#)

jeux de tests, [81](#), [84](#), [88](#)

machine à état, [96](#)

mesure du logiciel, [72](#)

métriques, [70](#)

mort par entropie (terminaison), [25](#)

mutant, [61](#)

phase

de conception, [9](#)

de planification, [24](#)

d'exploration, [23](#)

pilotes, [30](#)

Planifier les tests, [138](#)

projet UP, [19](#)

qualimétrie, [71](#)

qualité, [141](#)

rapport d'inspection, [78](#)

résultats (contrôle des ~), [5](#)

révisions, [144](#)

revue, [73](#)

souches, [32](#)

spécifications fonctionnelles (phase de ~), [12](#)

stratégie

du Big Bang, [35](#)

incrémentale, [37](#)

par lots fonctionnels, [36](#)
stubs, [33](#)
table de décision, [92](#), [93](#)
 format, [94](#)
techniques
 aléatoires, [58](#)
 Boîte blanche, [64](#)
 Boîte noire, [62](#)
 de tests, [56](#)
 statiques, choix, [80](#)
test
 alpha, [45](#)
 aux limites, [91](#)
 bêta, [46](#)
 Boîte noire, [63](#)
 concevoir, [3](#)
 conditions de, [4](#)
 de composants, [26](#)
 de confirmation, [53](#)
 de non-régression, [55](#)
 de validation, [38](#)
 dynamique, [57](#)
 dynamique, choix, [79](#)
 d'acceptation, [40](#)
 d'acceptation (contractuelle), [43](#)
 d'acceptation (opérationnelle), [42](#)
 d'acceptation (par les utilisateurs), [41](#)
 d'acceptation (réglementaires), [44](#)

d'intégration, [34](#)
fonctionnel, [50](#)
non fonctionnel, [51](#)
par mutation, [60](#)
planifier, [1](#)
spécifier, [2](#)
statique, [68](#)
statistique, [59](#)
structurel, [52](#), [65](#)
système, [39](#)
unitaire, [27](#)
tester aux limites, [90](#)
traçabilité (matrice de ~), [6](#)
walkthrough, [76](#)

Table des matières

Avant-propos

Chapitre 1. Quelques idées essentielles sur les tests

1.1. Chaîne de l'erreur

1.2. Rôle des tests

1.3. Les sept principes généraux des tests

1.3.1. Principe 1 – Les tests montrent la présence de défauts

1.3.2. Principe 2 – Les tests exhaustifs sont impossibles

1.3.3. Principe 3 – Tester tôt

1.3.4. Principe 4 – Regroupement des défauts

1.3.5. Principe 5 – Le paradoxe du pesticide

1.3.6. Principe 6 – Les tests dépendent du contexte

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

1.4. Processus et psychologie liés aux tests

Chapitre 2. Tester à chaque niveau du cycle de vie

2.1. Les différents modèles de développement

2.2. Préparer les tests lors des phases de conception du cycle en V

2.2.1. Préparer les tests lors des phases d'expression et d'analyse du besoin

2.2.2. Préparer les tests lors de l'écriture des spécifications fonctionnelles

2.2.3. Préparer les tests lors de la conception globaleconception:globale, phase de

2.2.4. Préparer les tests lors de la conception détaillée

2.3. Les tests et les modèles itératifs

2.3.1. Le développement itératif

« UP »développement:itératif « UP »

2.3.2. Le développement itératif

« XP »développement:itératif « XP »

2.4. Les différents niveaux de test

2.4.1. Le test de composantstest:de composants ou tests unitairestest:unitaire

2.4.2. Le test d'intégrationtest:d'intégration

2.4.3. Les tests système et les tests de validation

2.4.4. Les tests d'acceptation, tests alpha et tests bêta

2.5. Les différents types de tests

2.5.1. Tests fonctionnels versus tests non fonctionnels

2.5.2. Tests liés au changement et tests de non-régression

2.6. Conclusion

Chapitre 3. Tester efficacement : les différentes stratégies

3.1. Aperçu des stratégies de tests dynamiques

3.1.1. Techniques aléatoires:aléatoires

3.1.2. Techniques Boîte noire:Boîte noire

3.1.3. Techniques Boîte blanche:Boîte blanche

3.2. Aperçu des stratégies de tests statiques

3.2.1. L'analyse statique de code:analyse statique de code

3.2.2. L'utilisation de métriques

3.3. Revue de code, revue technique, inspection

3.3.1. Revue de type « buddy check »

3.3.2. Revue de type « walkthrough »

3.3.3. Revue de type « inspection:inspection »

3.4. Le processus d'inspection en six étapes

3.4.1. Plan type du rapport d'inspection:rapport d'inspection

3.4.2. Choisir un type de revue

3.4.3. Processus de revue : conclusion

3.5. Tests dynamiques versus tests statiques : synthèse

3.5.1. Choisir une technique de tests dynamiques

3.5.2. Choisir une technique de tests statiques

3.6. Conclusion

Chapitre 4. Concevoir efficacement des jeux de tests grâce aux spécifications

4.1. Réduire le combinatoire avec les techniques All Singlesall singles et All Pairsall pairs

4.1.1. Principes

4.1.2. Illustration

4.1.3. Commentaires

4.2. Tester grâce aux classes d'équivalence

4.2.1. Principes

4.2.2. Utilisation des classes d'équivalence

4.2.3. Construction des classes d'équivalence:classes d'équivalence:construction

4.2.4. Illustration

4.2.5. Commentaires

4.3. Tester aux limites

4.3.1. Principes

4.3.2. Prise en compte de contraintes liant les paramètres

4.3.3. Illustration

4.3.4. Commentaires

4.4. Tester grâce à une table de décision

4.4.1. Principes

4.4.2. Format d'une table de décision

4.4.3. Construction et simplification d'une table de décision

4.4.4. Illustration : le problème des triangles

4.4.5. Illustration : le problème du calcul du lendemain

4.4.6. Commentaires

4.5. Utiliser un diagramme « états transitions »

4.5.1. Principes

4.5.2. Format général d'un diagramme états

transitionsdiagramme états transitions:format général

4.5.3. Construction d'un diagramme états

transitionsdiagramme états transitions:construction

4.5.4. Construction des séquences de test

4.5.5. Commentaires

4.6. Conclusion

Chapitre 5. Utiliser les détails d'implémentation dans les tests

5.1. Définir des objectifs de couvertures par rapport au flot de contrôle

5.1.1. Principes

5.1.2. Construire et utiliser un graphe de contrôle

5.1.3. Critère de couverturecritère de couverture:toutes les instructions « toutes les instructions »

5.1.4. Critère de couverturecritère de couverture:tous les chemins « tous les chemins »

5.1.5. Critère de couverturecritère de couverture:toutes les décisionscritère de couverture:toutes les branches « toutes les branches ou toutes les décisions »

5.1.6. Critère de couverturecritère de couverture:toutes les conditions « toutes les conditions »

- [5.1.7. Critère de couverturecritère de couverture:toutes les conditions-décisions « toutes les conditions – décisions »](#)
- [5.1.8. Critère de couverturecritère de couverture:toutes les conditions multiples « toutes les conditions multiples »](#)
- [5.1.9. Critère de couverturecritère de couverture:toutes les conditions-décisions modifiées \(MC/DC\) « toutes les conditions – décisions modifiées \(MC/DC\) »](#)
- [5.1.10. Critère de couverturecritère de couverture:tous les i-chemins « tous les i-chemins »](#)
- [5.1.11. Autres critères de couverture basés sur le flot de contrôle](#)

[5.2. Définir des objectifs de couvertures par rapport au flot de données](#)

- [5.2.1. Principes](#)
- [5.2.2. Construire et utiliser un graphe de flots de donnéesgraphe:de flots de données](#)
- [5.2.3. Séquences caractéristiques et utilisation statique de l'étiquetage des nœuds](#)
- [5.2.4. Critères de couverture associés aux flots de données](#)

[5.3. Trouver les jeux de valeurs satisfaisant un critère de couverture](#)

[5.4. Conclusion](#)

[Chapitre 6. Processus et tests d'intégration](#)

- [6.1. L'intégration dans le cycle de vie](#)
- [6.2. Intégration dans une architecture client/serveur](#)
- [6.3. Notion d'intégrat](#)
- [6.4. Difficultés et risques de l'intégrationintégration:risques et difficultés](#)
 - [6.4.1. Intégration verticaleintégration:verticale](#)
 - [6.4.2. Intégration continue](#)
- [6.5. Mécanique du processus d'intégration](#)
- [6.6. Dynamique du processus d'intégration](#)
 - [6.6.1. Rappel sur la mécanique du procédé de construction](#)
 - [6.6.2. La machine à intégrer](#)
 - [6.6.3. Retour sur l'intégration continue](#)
 - [6.6.4. Intégration et gestion de configuration](#)
 - [6.6.5. Intégration dynamique](#)

6.7. Stratégie d'intégration pour la validation, la vérification et l'intégration

6.8. Résumé des règles de la dynamique du processus d'intégration

6.8.1. Règle N° 1 – Vitesse d'écriture des tests

6.8.2. Règle N° 2 – Nombre de tests

6.8.3. Règle N° 3 – Effort de vérification des résultats

6.8.4. Règle N° 4 – Effort pour la gestion des configurations

6.8.5. Règle N° 5 – Effort de non-régression

6.8.6. Règle N° 6 – Automatisation des tests

6.8.7. Règle N° 7 – Simplification et réduction de la complexité du processus d'intégration

Chapitre 7. Gérer les tests

7.1. Organisation des tests et répartition des rôles

7.1.1. Choix des types de tests à effectuer

7.1.2. Les différents acteurs

7.2. Planifier les tests

7.3. Définir et évaluer les critères de sorties

7.4. Estimer l'effort de test

7.5. Gérer les tests en configuration

7.6. Conclusion

Chapitre 8. Outils pour les tests

8.1. Typologie attendue des outils de tests

8.2. Les grandes familles d'outils

8.2.1. Outils pour les tests statiques

8.2.2. Outils d'aide à l'exécution des tests dynamiques

8.2.3. Outils d'appui à la gestion des tests

8.3. Conclusion

Chapitre 9. Génération automatique de jeux de test

9.1. Générer des tests à partir de modèles

9.1.1. Principes

9.1.2. Outils disponibles

9.1.3. Avantages / Inconvénients

9.2. Générer des tests à partir du code

9.2.1. Utilisation d'une version de référence

9.2.2. Utilisation d'une version incomplète ou partiellement testée

[9.2.3. Outils disponibles](#)

[9.2.4. Avantages / Inconvénients](#)

[9.3. Conclusion](#)

[Chapitre 10. Tester des systèmes interactifs](#)

[10.1. Test des systèmes embarqués](#)

[10.1.1. Isolation et test des composants ou des sous-systèmes](#)

[10.1.2. Test des interactions](#)

[10.1.3. Test système et de validation](#)

[10.1.4. Normes et standards](#)

[10.2. Test des services Web](#)

[10.3. Conclusion](#)

[Chapitre 11. Les tests, une nouvelle mesure de complexité](#)

[11.1. Introduction](#)

[11.2. Terminologie](#)

[11.3. Le puzzle de l'intégration](#)

[11.3.1. Le vrai sens de l'intégration](#)

[11.3.2. Unité de sens/compréhension – Performance des programmeurs](#)

[11.3.3. Nombre de pièces à intégrer – Combinatoires](#)

[11.3.4. Ebauche d'une typologie des pièces à intégrer](#)

[11.3.5. Le problème de l'intégration](#)

[11.4. L'organisation de l'intégration](#)

[11.5. Mesurer la complexité par les tests d'intégration](#)

[11.6. Recommandations et bonnes pratiques](#)

[11.7. Bibliographie du chapitre](#)

[Conclusion](#)

[Exemple de QCM](#)

[Bibliographie](#)

[Index](#)