INTRODUCTION

1 Le génie logiciel

Le génie logiciel est une activité d'*ingéniérie* (« génie ») pour produire des *logiciels*. Comparé aux branches traditionnelles de l'ingénierie (Génie civil, Génie chimique, génie électrique, *etc.*), le génie logiciel est relativement récent et cette branche a la particularité d'avoir été crée *ex nihilo* lors d'une conférence scientifique de l'OTAN le 11 octobre 1968, à la suite de la « crise du logiciel » des années 60 (on y reviendra).

Les buts du génie logiciel sont de proposer des méthodes et des techniques de développement de logiciels pour assurer la *fiabilité* des logiciels, et surmonter la difficulté que la programmation et les programmes ne suivent que très peu de *lois générales quantitatives* (au sens de la physique) en raison de leur nature essentiellement *logique* : une fonction entière écrite très maladroitement et pleine d'erreurs peut souvent faire beaucoup moins de dégâts sur le comportement final d'un logiciel qu'une simple erreur de frappe si elle est dans une portion critique du logiciel.

Il est important de ne pas faire la confusion entre, d'une part, une collection de petits programmes écrits « à la demande », de manière souvent informelle, et qui, par accumulation, construisent petit à petit un environnement de travail, et d'autre part, le développement d'un logiciel commandé à une équipe d'informaticiens ou de bio-informaticiens. Ce dernier

- est un développement professionnel d'un produit et non une somme de développements individuels,
- doit respecter des normes et suivre un processus de conception et développement (souvent appelé « cycle de vie du logiciel »),
- fait appel à des spécifications qui génèrent une lourde documentation assurant la communication entre les équipes et avec le client sur les besoins et services rendus par chaque partie du logiciel,
- met en œuvre des méthodes systématiques de validation du logiciel par rapport à ses spécifications,
- doit assurer un niveau adapté de qualité et de fiabilité du logiciel,
- gère rationnellement l'évolution des besoin des utilisateurs et les évolutions du logiciel sans interrompre le service rendu.

Comme on le voit, adopter une démarche de génie logiciel pour développer un produit logiciel implique une certaine lourdeur. Cela a un coût : en informatique, le coût est équivalent au temps ingénieur car le prix du matériel informatique et du temps calcul est généralement négligeable par rapport au poids des salaires des ingénieurs, contrairement à la biologie « humide » où les expériences constituent le coût principal. Face à une demande de développement de logiciel, il faut donc savoir décider si l'on va, ou non, appliquer des méthodes de génie logiciel. Pour cela, trois questions jouent un rôle fondamental :

1. Les erreurs sont-elles lourdes de conséquences ? (vies humaines, gros investissements matériels, très lourdes pertes de temps en cas d'erreur logicielle, retombées majeures attendues en cas de réussite[ou d'échec], *etc.*). C'est par exemple le cas des transports en commun, du nucléaire, de la sécurité d'accès à des salles dangereuses, des domaines spatial et aéronautique, de la finance, de l'étude de phénomènes majeurs rarissimes, *etc.*
2. Plusieurs personnes ou équipes doivent-elles être impliquées dans le développement du logiciel ? (partage des tâches, développement modulaire, communications entre les partenaires, suppression des ambiguïtés, intégration des différentes parties logicielles, correction cohérente des erreurs, *etc.*)
3. Les utilisateurs sont-ils externes aux équipes de concepteurs ? (définition précise des besoins, évolution et modifications des besoins, ignorance profonde des utilisateurs, nécessité d'interfaces conviviales, non adaptation des utilisateurs aux défauts logiciels connus par des « contournements », *etc.*)

Si *deux* parmi ces trois questions ont une réponse positive, on doit appliquer des méthodes de génie logiciel.

2 Les principales difficultés

Tout d'abord il faut être conscient qu'*il n'existe pas de logiciels sans erreurs*. Certaines erreurs peuvent ressortir après plusieurs années d'utilisation (un exemple bien connu est le remplissage optimal d'un four industriel pour la poterie dont un bug majeur a été identifié 20 ans après sa mise en service).

Un enjeu majeur du génie logiciel est donc de limiter ces erreurs dites résiduelles à des fonctionnalités non critiques, ou prévoir des « voies alternatives » comme en biologie pour les fonctionnalités critiques.

La correction d'une erreur entraîne souvent d'autres erreurs :

- problèmes d'explosions en chaîne des erreurs à corriger
- mesures d'évolution du logiciel au cours du temps peu fiables

— mais des formes caractéristiques peuvent inciter à stopper prématurément un développement logiciel ; ne pas hésiter à tout jeter et recommencer à zéro dans ces cas là.
— faire des gestions de versions pour revenir en arrière si les erreurs découvertes sont bien identifiées.

Il est en fait très complexe d'assurer une mesure de qualité d'un logiciel. Il est donc fortement rentable d'assurer la qualité des méthodes et techniques de développement et de maintenance (« bonnes pratiques du génie logiciel »).

3 L'éthique

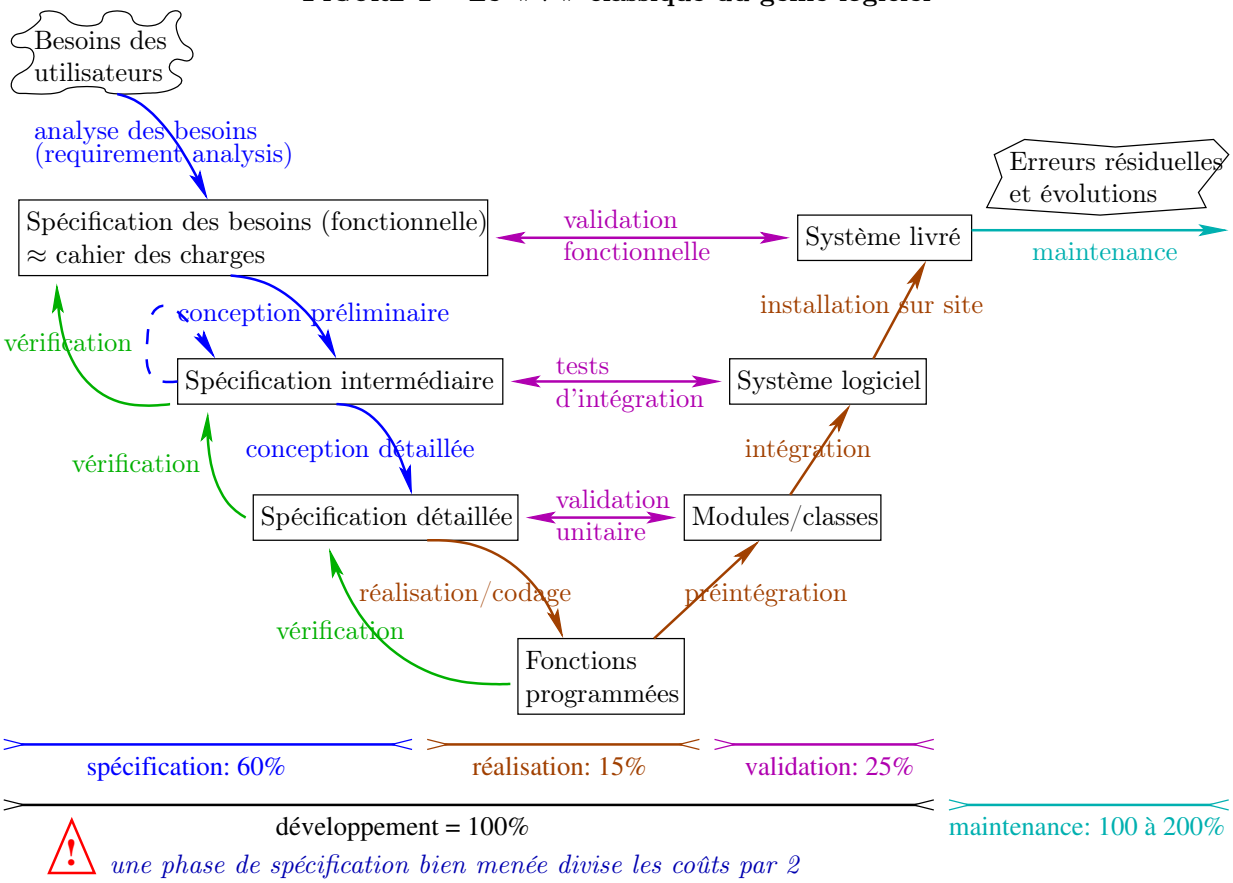
Un autre aspect du métier de bio-informaticien ne doit pas être négligé et entre entre autres dans l'activité de génie logiciel : l'*éthique*. En tant qu'ingénieur connaissant les STIC (Sciences et Techniques de l'Information et de la Communication), l'activité de développement de logiciel ne doit pas être aveugle de la portée des produits logiciels délivrés. Il est répréhensible d'utiliser son savoir, et *a fortiori* l'ignorance des utilisateurs, pour cacher aux utilisateurs certaines fonctionnalités, ou pour minimiser à leur yeux la portée des données produites par le logiciel.
— Ne pas utiliser vos connaissances pour détourner ou rendre implicites des fonctionnalités sur les machines des utilisateurs.
— Respecter la vie privée et les principes de base de confidentialité si des données sensibles doivent être manipulées (anonymisation des données médicales par exemple).
— Ne pas développer de logiciels contraires à l'intérêt général, et (seconde priorité) développer des logiciels en accord avec les besoins du client ou de l'entreprise. Cela implique donc de maintenir l'intégrité de son jugement.
— Le génie logiciel est un travail d'équipe et en tant que tel un comportement collectif élégant, en tant que membre ou à la direction d'une équipe, est la seule attitude professionnelle.

4 Les méthodes de développement logiciel

Ces différentes méthodes définissent ce que l'on appelle souvent le « Cycles de vie » des logiciels. Elles reposent sur des activités de différent types combinées entre elles durant le temps de développement et de maintenance du logiciel. Il existe 4 types d'activités principaux :
— la spécification et la conception (parfois par prototypage),
— le développement logiciel (étapes de codage),
— la validation et le test de logiciel,
— la maintenance et l'évolution du logiciel.
Le « cycle de vie » le plus classique (parce qu'il est calqué sur la méthode la plus classique en technologie) est donné en figure 1.

La partie descendante du « cycle en V » du logiciel est la *conception* du logiciel.
— Partant de la formulation de ses besoins par le client (souvent informelle, incomplète, floue, voire contradictoire) il faut d'abord établir une *spécification des besoins* qui précise clairement sur papier ce que l'utilisateur attend du logiciel à créer. Cette spécification des besoins est à peu de choses près le futur mode d'emploi du logiciel, si ce n'est que le document est sans doute un peu plus technique. En particulier la spécification des besoin peut être également vue comme un cahier des charges. L'activité de création de la spécification des besoins à partir de leur expression informelle par le client est appelée *L'analyse des besoins*.
— Il s'agit ensuite de commencer à structurer les différentes parties du futur logiciel qui seront utiles pour réaliser toutes les fonctionnalités demandées dans la spécification des besoins. Cela conduit à écrire une spécification qui commence à décrire, d'une part, comment combiner ces parties pour obtenir le logiciel global, et d'autre part, à spécifier proprement ce que devra faire chaque partie. De telles spécifications sont appelées des *spécifications intermédiaires* et l'activité de les concevoir est appelée le *raffinement de spécification* ou encore la *réification de spécification*. Si les partie de logiciel mises en jeu doivent offrir des fonctionnalités elles-mêmes assez compliquées, on doit elles aussi les raffiner et cela peut donc donner lieu à plusieurs niveaux de spécifications intermédiaires si le logiciel à développer est particulièrement conséquent. Ces différents niveaux sont donc des étapes de conception intermédiaire.
— À chaque étape de raffinement, il est nécessaire de se convaincre que la nouvelle spécification « de bas niveau » réalise correctement la spécification précédente « de plus haut niveau ». Ces activités se nomment la *vérification*.
— Arrive enfin un niveau de détail où un programmeur n'aura pas de difficulté à programmer les fonctions demandées dans chaque partie définie par la spécification. Cette dernière étape est appelée la conception détaillée et la dernière spécification est appelée la *spécification détaillée*.

Figure 1 – Le « V » classique du génie logiciel



En bas du V se trouve l'étape de *réalisation du logiciel* (ou codage) qui produit donc des (morceaux de) programmes.

Il faut ensuite regrouper ces programmes pour obtenir des *modules* (ou des *classes* si le langage est orienté objet) et cette étape est la *pré-intégration* et réalise chacune des parties de la spécification détaillée. Les modules sont ensuite interconnectés pour réaliser chacune des parties de la spécification intermédiaire et ces différentes étapes (en nombre qui dépend du nombre de spécifications intermédiaires) constituent l'*intégration* du logiciel.

Les étapes de pré-intégration et d'intégration doivent faire l'objet de *validations* par rapport aux spécification de même niveau, souvent par des tests de logiciel, on y reviendra.

À ce stade, on a une version complète du logiciel mais elle tourne sur les machines et l'environnement logiciel de l'équipe de réalisation du logiciel. Il se peut que la présence de tout l'outillage logiciel à disposition sur le site de développement, qui n'est pas présent sur la/les machine/s cible/s du client, « cache » des dysfonctionnements potentiels. La dernière activité est donc de délivrer le logiciel sur son site cible.

Ensuite commence l'exploitation du logiciel, la correction des bugs résiduels, l'adaptation des paramètres, *etc.* et cette étape est la *maintenance* du logiciel et l'évolution des besoins conduira à modifier le logiciel pour le faire évoluer.

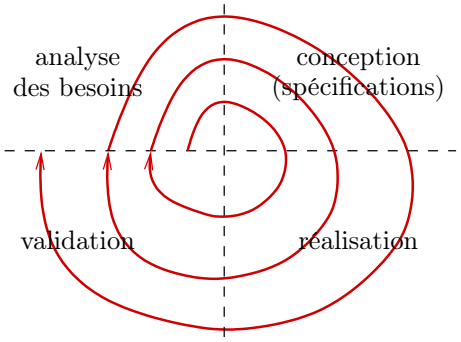
La maintenance est une activité de première importance puisqu'elle dépasse (en temps et donc en coût) la conception, et pourtant, elle est beaucoup trop souvent sous-valuée au yeux des ingénieurs. La maintenance couvre principalement trois types d'activité :

- La réparation des bugs dits « résiduels », éventuellement induits par les adaptations du logiciel en maintenance, ou encore les vulnérabilités du logiciel découvertes en exploitation. Ce type d'activité représente un peu moins d'un quart du temps de maintenance
- L'adaptation aux changements environnementaux (évolution des librairies logicielles, des systèmes d'exploitation, les changements de plateforme matérielle, *etc.*). Ce type d'activité représente un peu moins de 20% du temps de maintenance.
- Enfin l'ajout ou l'extension de fonctionnalités répond à l'évolution des besoins et de l'organisation des clients. C'est l'activité la plus coûteuse ; elle peut remettre en cause une partie de l'architecture même du logiciel. ce type d'activité représente un peu moins de 60% du temps de maintenance.

Le défaut principal du développement en V est qu'il n'est pas incrémental : toutes les fonctionnalités du logiciel doivent être prévues dès le départ et réalisées ensemble, sans réelle considération de priorité entre les fonctionnalités. Bien souvent, lorsque le logiciel n'est pas soumis à des réglementations drastiques (qui accompagnent généralement le premier critère du génie logiciel, les erreurs lourdes de conséquences) on peut se permettre de repousser à plus tard la conception et la réalisation des fonctionnalités « secondaires » et de commencer par faire un logiciel « noyau » ne contenant que les fonctionnalités principales, puis de l'enrichir par étapes.

Chaque étape d'enrichissement du logiciel donne lieu aux activités du V : analyse des besoins, spécification, réalisation et validation et l'on parle alors souvent de *développement en spirale* représenté en figure 2. Le premier « tour de

Figure 2 – Le développement en spirale



spirale » effectue une analyse des besoins restreinte aux fonctionnalités indispensables du logiciel, les spécifie, les réalise et les valide. On obtient ainsi le « noyau » sus-mentionné. Chaque tour supplémentaire détermine quelles fonctionnalités doivent être ajoutées (analyse des besoins), les spécifient de manière cohérente avec les spécifications préexistantes du « tour d'avant », les réalise et valide l'*ensemble* du logiciel car l'introduction de nouvelles fonctionnalités peut introduire des erreurs globales.

Enfin, avec la possibilité de mettre ensemble de larges modules issus de bibliothèques de logiciels, il arrive souvent que des logiciels conséquents puissent être conçus sans pour autant appliquer des techniques de génie logiciel au sens strict du terme. L'idée est la suivante : si le premier critère du génie logiciel n'est pas requis (i.e. les erreurs ne sont pas lourdes de conséquences) on peut tenter d'échapper aux lourdeurs du génie logiciel en s'organisant pour que l'un des deux autres critères devienne également non requis. Comme on ne peut pas trop réduire les équipes de développement si le logiciel est conséquent (deuxième critère), on doit tenter que les utilisateurs *participent* au développement pour supprimer le troisième critère. C'est là la base des *méthodes agiles* de développement du logiciel : réduire autant que possible la taille des équipes de conception du logiciel et faire participer le client à l'équipe de développement pour éviter l'écritures de spécifications et les remplacer par des prototypes successifs. En somme, c'est une méthode inspirée du développement en spirale dans laquelle les étapes de spécification sont supprimées, remplacées par la présence d'un membre de l'équipe de développement délégué par le client.

Comme on l'a souligné, ces méthodes incrémentales ne sont pas toujours applicables, en particulier quand les conséquences des erreurs peuvent être lourdes et/ou la réglementation impose des normes précises (transports, nucléaire, aérospatiale, *etc*).

En fait, le génie logiciel recouvre un large palette d'activités d'ingénierie car il existe une *énorme variété de logiciels* avec des finalités très diverses. Il faut donc *au préalable* choisir une méthode de développement qui cadre bien avec les exigences liées aux services que doit rendre le logiciel, aux contexte contractuel entre les développeurs et les utilisateurs, *etc.*

5 Quelques chiffres

5.1 sur la nécessité d'appliquer les techniques du génie logiciel

Les nombreuses contraintes de qualité, la taille des logiciels à développer, le coût de leur développement, *etc.* imposent une ingénierie rigoureuse pour produire ces logiciels. À l'inverse, ces techniques d'ingénierie du logiciel coûtent cher (environ 3 fois le coût d'un développement artisanal) et il ne faut les appliquer que si un développement artisanal risque d'échouer. On a vu les trois critères qui permettent de décider si l'on doit ou non entrer dans une démarche de génie logiciel, avec la règle du 2 sur 3. Ils sont précieux car les échecs de la méthode artisanale sont cuisants. Voici

les chiffres résultant des développements artisanaux qui étaient de mise au début des années 1980, avant que le génie logiciel s'impose comme incontournable. Ils sont donnés en pourcentage du coût total, sur une étude ayant porté sur 7M$ durant le début des années 1980.

Cinq « niveaux de réussite » ont été considérés :
— les logiciels livrés au client et utilisés sans modifications majeures représentent seulement **2%**,
— les logiciels livrés et utilisés après modifications majeures représentent seulement **3%**,
— les logiciels livrés, ayant fait l'objet de modifications majeures, mais rapidement abandonnés représentent **19%**,
— les logiciels livrés mais jamais utilisés par le client représentent **48%**
— enfin ceux jamais livrés, donc non payés par le client, représentent **28%**.
Comme on le voit, au moins 95% des coûts des logiciels de cette étude ont été dépensés en pure perte avec des approches artisanales. Les clients sont maintenant moins naïfs vis à vis des produits informatiques et les deux derniers cas, soit 76% des coût du développement artisanal, seraient refusés par le client, c'est-à-dire non livrés, non payés. Il est donc crucial de ne pas « rater » la décision d'appliquer des méthodes de génie logiciel pour éviter de grosses pertes pour l'entreprise.

5.2 sur l'importance majeure de l'étape de spécification

Bon nombre de chefs de service peu sensibilisés aux contraintes du génie logiciel ont tendance à vouloir démarrer la phase de codage au plus tôt. En effet, pour évaluer l'état d'avancement du projet, il paraît plus facile de compter de nombre de lignes de code produites, ou le nombre de fonctions déjà réalisées (voire l'avancée des démonstrations des parties déjà codées s'il n'a aucune compétence en informatique) que d'évaluer l'état d'avancement de l'ensemble de documentations que représente les différents niveaux de spécification. Ils ont donc hâte de clore la phase de spécification bien avant les quelques premiers 60% du temps de conception qu'elle devrait occuper. C'est une erreur majeure car des spécifications mal faites allongent fortement le temps nécessaire dans les phases suivantes, en augmentent considérablement le coût, et retardent par conséquent fortement la fin du projet. Pour s'en convaincre voici les résultat d'une étude sur le coût en temps (= coût tout court, puisque l'essentiel de coût du projet est le salaire des concepteurs) d'une erreur de conception, selon le moment où on la découvre :
— si l'erreur de conception est découverte dès l'étape de conception préliminaire, on va considérer qu'elle prend une unité de temps pour être réparée (qu'il s'agisse d'un jour, d'une semaine ou d'un mois, selon la taille du projet). **Coût = 1**
— si elle est découverte en conception détaillée : **coût = 1,5**
— si elle est découverte durant l'étape de réalisation : **Coût = 2 à 3**
— si elle est découverte en cours d'intégration : **Coût = 3 à 5**
— si elle est découverte en phase de validation fonctionnelle : **Coût = 5 à 20**
— enfin si elle est découverte en maintenance : **Coût = 10 à 100**.
En pratique, les deux derniers cas sont considérés comme des catastrophes, ils induisent généralement des pertes conséquentes de l'entreprise sur ce produit logiciel, et impactent son image de marque.

5.3 sur le choix du langage de programmation

À taille égale de « ce que fait le logiciel », les coûts de développement et de maintenance sont proportionnels au nombre d'instructions. Par ailleurs le nombre d'erreurs de programmation est également proportionnel à ce nombre d'instructions.

Il en résulte qu'il vaut mieux choisir un langage de programmation bien adapté au problème abordé par le logiciel à concevoir, afin de réduire le nombre de lignes de code. Cela revient à dire qu'il faut privilégier les langages de haut niveau, qui permettent au programmeur de coder *dans les termes du problème abordé* plutôt que *dans les termes de la machine*. En ce sens, les langages impératifs sont souvent moins performants que les langages orientés objet, eux-mêmes moins performants que les langages fonctionnels ou les langages logiques. Par ailleurs, il existe des langages dits *orientés métier* qui offrent des primitives spécifiques au domaine abordé, réduisant d'autant le nombre de lignes de code (à supposer que le langage orienté métier soit lui-même validé, car s'il contient des erreurs, on ne gagne rien à l'utiliser !). Souvent, un langage orienté métier est un simple enrichissement d'un langage existant par des bibliothèques spécialisées. C'est par exemple le cas de bioPython.

Malheureusement, en pratique, on ne peut pas choisir librement le langage de programmation le plus adapté au projet en cours de développement... En effet, il faut prendre en compte le temps et la difficulté de *formation* des équipes de développement. Ces derniers connaissent *a priori* un nombre restreint de langages et cela limite le choix, à cause du coût de formation des personnels. Il est donc souvent rentable de spécialiser des équipes sur certains types de logiciels ou de métiers cibles pour éviter la dispersion. En revanche, si plusieurs projets à venir relèvent d'un certain domaine,

il faut gérer la formation permanente des développeurs en ce sens pour améliorer l'efficacité des équipes sur le moyen ou long terme.

5.4 sur la négociation des délais

À produit fixé, le développement par plusieurs personnes ou plusieurs équipes multiplie automatiquement par au moins 3 le coût du logiciel par rapport à une seule personne ou une seule équipe. Par conséquent mettre 3 personnes sur un logiciel ne permet pas de diminuer le temps de développement du produit ; il en faut plus... et cela s'accompagne par un alourdissement des spécifications et des procédures de communication.

Il est donc très rentable de négocier la date de rendu du produit logiciel : plus le temps de conception/développement est grand, moins le nombre de personnes à affecter au projet est grand, et moins les procédures de spécification détaillées et de synchronisation des équipes sont coûteuses.

5.5 sur les jugements de valeur

Il faut totalement bannir les jugements de valeur au sein d'une équipe de conception/développement. C'est infondé et contre-productif.

Par exemple, en moyenne, 90% des développeurs d'un logiciel produisent 15% du logiciel final (donc 10% des développeurs produisent 85% du logiciel final). Cela signifie-t-il que seuls 10% des développeurs travaillent vraiment ? Non. En fait, la productivité d'une personne dépend, de manière presque intrinsèque, de son rôle dans l'équipe. L'expérience suivante a été menée à plusieurs reprise, avec systématiquement le même résultat :
- on constitue 10 équipes de bonne taille pour développer chacune un logiciel ;
- on observe invariablement dans chacune de ces équipes la répartition 90%-15% sus-mentionnée ;
- on « mélange » ces équipes de telle sorte que l'une des équipes est constituée des 10% les plus productifs des équipes précédentes ;
- non seulement on constate que cette équipe n'est pas notablement plus performante que les autres sur de nouveaux projets,
- mais de plus on observe de nouveau la répartition 90%-15% sus-mentionnée au sein de cette équipe.

On va maintenant reprendre les activités successives mises en évidence dans la section 4, en suivant l'ordre du V du génie logiciel, et expliquer comment les aborder de manière rationnelle pour un développement de qualité.

LES SPÉCIFICATIONS DE LOGICIEL

Comme on l'a vu, l'activité de spécification est cruciale pour limiter le nombre d'erreurs de conception et de programmation. La fonction d'une spécification est de décrire précisément ce que doit faire le logiciel ou la portion de logiciel qu'elle décrit. Les spécifications servent à la communication avec le client, et entre les développeurs. Elles servent de plus de document de référence pour les activités de vérification et de validation.

6 Le quoi et le comment

En premier lieu il faut bien avoir en tête que décrire « ce que doit faire le logiciel » ne signifie aucunement décrire « comment fera le logiciel ». Le slogan est : *une spécification doit décrire le QUOI et pas le COMMENT*. C'est bien sûr évident pour la spécification des besoins, qui s'adresse aux utilisateurs et non aux concepteurs, mais c'est plus subtilement également important pour les spécifications intermédiaires et pour la spécification détaillée. Il est vrai que les spécifications intermédiaires doivent décrire comment seront réalisées les fonctionnalités demandées par la spécification de niveau au-dessus mais la réalisation d'un ensemble de fonctionnalités de plus haut niveau repose sur la combinaison de fonctionnalités de plus bas niveau. Une fois expliquée la manière dont on combine les fonctionnalités de plus bas niveau, qui est effectivement une partie décrivant le « comment », l'essentiel de la spécification est de décrire ce que doivent faire les fonctionnalités de plus bas niveau qu'on utilise. Et là, il s'agit de décrire ce que doivent faire ces fonctionnalités et non pas comment elles doivent le faire (comment elle doivent faire sera décrit dans les spécifications de plus bas niveau).

Il en résulte qu'à chaque niveau de spécification, une partie « chapeau », très minoritaire, explique comment combiner les fonctionnalités de plus bas niveau pour obtenir les fonctionnalités demandées par la spécification de niveau au-dessus, et le reste de la spécification, majoritaire, explique ce que doivent faire ces fonctionnalités de plus bas niveau, donc le *QUOI*.

En quelque sorte, le « comment » relève plus de la structure des spécifications intermédiaires ou détaillées que de leur contenu textuel. Le passage de la spécification des besoins aux spécifications détaillées indique par cette structure même comment sera réalisé le logiciel :
- séparation entre les fonctions d'*Interface Homme-Machine* (IHM) et *noyau fonctionnel* du système,
- choix de stockage des données (base de données et son schéma),
- découpage en modules ou classes,
- types/structures de données utilisées.

7 En maintenance

On a vu que la maintenance représente en moyenne un investissement en temps largement supérieur à celui de la conception et réalisation du logiciel. Par ailleurs il est évident que pour effectuer la maintenance d'une fonctionnalité on a crucialement besoin des spécifications des fonctionnalités de plus bas niveau qu'elle utilise. Il en résulte que les spécifications de tous les niveaux (pas seulement la spécification des besoins) font partie intégrante du produit logiciel. *Ces spécifications doivent donc être mises à jour elles aussi au cours de la maintenance.* La majorité des bugs introduits par une maintenance logicielle résulte d'une connaissance approximative des fonctionnalités de plus bas niveau au moment de modifier une fonctionnalité qui les utilise. Il est donc important de tenir à jour les spécifications des fonctionnalités à tous les niveaux pour disposer d'un socle solide lors des évolutions, de version en version, du logiciel.

8 Ambiguïté et incomplétude

Les principaux ennemis lors de l'écriture d'une spécification sont l'*ambiguïté* et l'*incomplétude*.

Ambiguïté : chaque phrase d'une spécification doit être pensée et pesée en essayant d'imaginer quelles interprétations peuvent en être faites, autres que celle qu'on a en tête. La majorité des erreurs de conception/réalisation d'un logiciel résulte d'une interprétation différente de ce que signifie une phrase dans une spécification, entre, d'une part, la personne ou l'équipe qui utilise une fonctionnalité et, d'autre part, la personne ou l'équipe qui réalise cette fonctionnalité.

Incomplétude : tous les cas de figure d'appel d'une fonctionnalité doivent être prévus, même ceux pour lesquels on a

de bonne raisons de penser qu'elles n'arriveront jamais. Bien souvent un bug peut conduire à appeler la fonctionnalité avec des arguments théoriquement inatteignables, et la maintenance peut tout à fait conduire à des extensions de fonctionnalités qui atteignent ces valeurs non prévues au départ. Les incomplétudes les plus fréquentes sont l'absence de spécification des cas exceptionnels ou erronés (`pop` sur une pile vide par exemple, ou division par 0, *etc.*). Lorsqu'une fonctionnalité est appelée avec des valeurs erronées, il faut préciser très clairement quel sera son comportement (e.g. lever une exception). De même lorsqu'elle fait appel à une fonctionnalité de plus bas niveau qui lui retourne une exception, il faut spécifier comment elle récupère l'erreur ou bien comment elle la transmet aux fonctionnalités de plus haut niveau.

9 La faisabilité et la réutilisation

Naturellement, il serait absurde de prévoir dans une spécification une ou des fonctionnalités infaisables par programme (cf. la notion de calculabilité ou de décidabilité en algorithmique), et en pratique, tout aussi absurde de prévoir une fonctionnalité trop coûteuse à réaliser. Cette préoccupation se nomme *la faisabilité*. Elle repose essentiellement sur l'expertise des concepteurs, donc peut paraître difficile à assurer pour un débutant, mais cette expertise s'acquiert très vite, peu de projets suffisent.

La faisabilité est un élément suffisamment important pour imposer que les équipes de développement participent lourdement à l'analyse des besoins. On peut, par exemple, être amené à tempérer les promesses d'un commercial...

La réutilisation de modules ou de classes déjà écrites pour d'autres projets antérieurs est un élément à prendre en considération pour évaluer la faisabilité. Cependant attention, il faut être sûr qu'on va pouvoir réutiliser *sans aucune modification* ces morceaux de code car sinon cela engendre des révisions de code qui restent assez coûteuses, et surtout induisent de forts risques d'erreurs.
La réutilisation de *librairies* open-sources est également un élément important pour évaluer la faisabilité. Par exemple bio-python est très riche en fonctionnalités sur les séquences. Un point important, dans ce cas, est de n'utiliser que des fonctionnalités qui ont été suffisamment validées, donc largement utilisées par d'autres programmeurs dans le monde.
Les décisions de réutilisation se font principalement lors de la spécification détaillée mais leur existence peut influer sur la structure des spécifications de plus haut niveau.

10 La testabilité

Enfin, il faut toujours garder en tête que les spécifications seront utilisées de manière cruciale lors des étapes de validation. Par conséquent, elles ne doivent pas contenir d'affirmations difficiles à tester. Par exemple une spécification qui stipulerait directement que dans une situation engendrant de lourdes pertes, le logiciel devra réagir de telle ou telle façon, serait une mauvaise spécification car on ne va pas pouvoir se placer dans cette situation pour tester le logiciel ! On dit que de telles situations sont *non opérables*. L'opérabilité est la capacité de placer le logiciel dans l'état initial décrit par la spécification. Pour les situations non opérables facilement, il vaut mieux faire un inventaire des conditions (opérables elles) qui peuvent caractériser de telles situations (ou mener à de telles situations) et indiquer les réactions du logiciel lorsqu'il détecte ces conditions.
De la même façon, il faut que les réactions requises par la spécification soient *observables* ou *mesurables* lors de la validation.
Lorsqu'on aborde les spécifications intermédiaires ou détaillées, il se peut que cela conduise à enrichir les spécifications avec des fonctionnalités non requises par la spécification de plus haut niveau. Ces fonctionnalités supplémentaires ne servant que lors de la validation, pour rendre atteignables et observables les comportements requis par la spécification de plus haut niveau.

11 Les spécifications formelles

Lorsque les conséquences d'une erreur sont énormes (vies humaines, données cruciales, *etc.*) il peut être utile de *prouver* mathématiquement qu'un programme réalise correctement ce que la spécification requiert. Étant donné la taille des logiciels, il n'est pas pensable de faire cette preuve à la main : elle serait encore plus susceptible de contenir des erreurs que le logiciel lui même. Il faut donc faire appel à des démonstrateurs de théorème assistés par ordinateur, et cela implique que les spécifications soient sous une forme que ces démonstrateurs manipulent. Il s'agit de formules logiques.

Il existe pour ce faire des *langages de spécification formelle* qui ne sont autres que des adaptations de telle ou telle

logique ou théorie mathématique pour décrire le plus aisément possible les propriétés requises sur le comportement du logiciel. *Computation Tree Logic* (CTL) est un exemple de logique utilisable pour les spécifications formelles. D'autres logiques ou cadres mathématiques précis peuvent être utilisés ; par exemple les « spécifications algébriques » se fondent sur la logique dite du premier ordre (qui est sans doute la plus classique des logiques), les « réseaux de Pétri » permettent de spécifier des programmes qui accèdent à des ressources de manière concurrente (e.g. réservations de billets) et utilisent CTL, ou encore le langage de spécification « B » repose sur la théorie des ensembles, *etc.*

Les spécifications dans ces langages, ainsi que les méthodes de preuves de correction de programme par rapport à ces spécifications, supposent une relative expertise en logique. Ces approches ne sont pas très différentes de celles utilisées pour étudier modèles discrets de réseaux biologiques. Dans le cadre de ce cours, nous n'aborderons que la *Logique de Hoare* qui permet de prouver partiellement la correction de programmes impératifs.

Notons enfin que le choix des spécifications formelles permet non seulement de prouver la correction des programmes, mais elle permet aussi de prouver qu'une spécification de plus bas niveau réalise effectivement sa spécification de plus haut niveau. Autrement dit, les étapes de vérification du V du génie logiciel sont elles aussi assistées par ordinateur. Dans ce contexte, on parle souvent de « réification » ou de « raffinement » de spécifications formelles.

Les langages de représentation graphique de spécifications, comme UML, sont parfois appelés « langages de spécification semi-formels » mais ne sont en fait pas « formels » au sens de l'assistance aux preuves.

RÉALISATIONS LOGICIELLES ET CODAGE

12 Le choix des structures de données abstraites

Losrque l'étape de spécification/conception est achevée, les spécifications détaillées sont assez concrètes pour qu'un programmeur n'aie pas de difficultés majeures pour réaliser les fonctionnalités demandées, ni de cas particuliers spécifiés de manière imprécise qui l'obligeraient à faire des choix implicites.

La première activité de réalisation est alors de réfléchir aux structures de données qui découlent de la spécification détaillée, à un niveau abstrait qui ne dépend pas encore du langage de programmation.

On rappelle qu'un langage de programmation est essentiellement défini par deux volets : (i) les structures de données « natives » (concrètes) et (ii) les primitives de contrôle. Le choix des « structures de données abstraites » complémentaires aux structures de données natives (e.g. ensembles, matrices, arbres, graphes en tous genres, *etc.*) est de loin celui qui pilotera la réalisation et c'est donc sur cette base que les choix de programmation se font.

Il faut donc commencer par faire l'inventaire exhaustif des structures de données qui permettront de réaliser les spécifications détaillées. Ensuite, les fonctionnalités à réaliser deviennent principalement des techniques de remplissage, de modification ou d'exploration d'une structure de donnée à partir d'une autre (e.g. fabriquer un arbre binaire de recherche à partir d'un ensemble), ou de communication entre plusieurs structures de données (e.g. explorer les chemins de longueur donnée dans un graphe à partir de calcul matriciel). Le fonctions ne sont alors plus que des moyens de navigation entre structures de données.

13 Choix d'un langage

Vient alors l'étape du choix d'un langage de programmation. Les résultats en « calculabilité » nous enseignent que tout ce qui est programmable dans un langage de programmation est *a priori* également programmable dans tout autre langage de programmation. Cela relativise l'importance de ce choix, et les batailles de chapelles voulant établir la supériorité intrinsèque d'un langage sur un autre n'ont rien de scientifique. En premier lieu, comparer des langages de programmation n'a de sens que pour un projet donné : un langage peut être mieux adapté à un projet qu'un autre et que ce soit l'inverse si l'on change de projet. On rejoint là la question de l'énorme champ d'application du génie logiciel.

D'un point de vue « idéal », pour un projet donné, l'*abstraction*, la *modularité* et l'*encapsulation* sont les éléments de jugement clefs pour choisir un langage de programmation et les librairies utilisées.

13.1 L'abstraction

C'est la capacité de programmer en des termes proches du problème traité plutôt qu'en des termes proches de la machine. Cela touche tous les aspects : nom « parlant » des fonctions et des variables utilisées, nom parlant et adéquation des types de données manipulés et de leurs opérateurs de manipulation, structures de contrôle lisibles (typiquement, incrémenter ou décrémenter un indice dans une boucle `while` est pour le moins une programmation dans les termes de la machine !), *etc.*

Par exemple, un langage « orienté métier », comme bio-python face à python, offrira des fonctions primitives spécialisées, déjà validées, qui permettent d'écrire des lignes de code plus lisibles en termes de biologie. Ces langages, *via* leurs librairies, offrent aussi des structures de données d'assez haut niveau qui facilitent la lisibilité des programmes.

Naturellement, l'abstraction est aussi une recommandation de style de programmation : bien choisir ses noms de fonctions et de variables, qu'ils soient explicites (même si sont plus longs à taper...), ne pas hésiter à définir de nouveaux types de données parlants, même s'ils diffèrent peu de structures de données classiques ou déjà programmées par ailleurs.

Enfin le contrôle lui-même, tel que défini par le langage, peut être plus ou moins bien adapté à un problème donné, en termes de lisibilité. Par exemple les langages fonctionnels typés (voir plus loin) sont particulièrement bien adaptés pour écrire très rapidement des prototypes, ou pour réaliser le « noyau dur » des logiciels ; en revanche, ils sont généralement moins bien adaptés pour écrire des IHM sophistiquées. Les langages orientés objets (LOO) sont généralement préférables pour développer des IHM complexes.

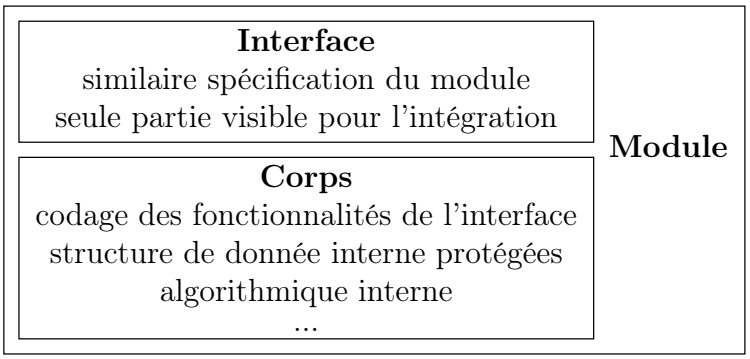
13.2 La modularité

Lorsque plusieurs personnes ou *a fortiori* plusieurs équipes réalisent un logiciel, il est nécessaire de mettre en place des principes de fragmentation du code pour le découper en fichiers indépendants, tout en évitant que plusieurs équipes développent des fonctions similaires (duplication de code). De plus, des problèmes de nommage apparaissent souvent : par exemple deux fonctions, trop mineures pour être dans les spécifications détaillées, auxquelles deux équipes vont donner le même nom, mais qui ne font pas la même chose ; *idem* pour les noms de variables globales. Évidemment, lors de l'intégration, cela crée des erreurs. De plus, si l'on résout la question en mettant préalablement d'accord toutes les équipes sur des nommages conventionnels, il se peut que, sachant ce que fait telle ou telle variable manipulée par l'équipe d'à côté, une autre équipe l'utilise, par exemple pour consulter efficacement sa valeur et s'en servir dans une quelconque fonction. Cette pratique est malheureusement elle aussi source d'erreurs graves.

En effet, un point important dans le développement de logiciel est la liberté nécessaire à chaque équipe de choisir l'algorithmique qui réalisera les fonctions demandées dans la spécification détaillée. Il est fréquent qu'une première version ne donne pas les performances voulues et que l'équipe modifie ses choix algorithmiques et les structures de données sous-jacentes en accord avec cette nouvelle algorithmique. Cela a pour conséquence que, sans changer une virgule des spécifications, le nommage interne va changer… et cela engendre immédiatement des bugs chez toutes les autres équipes qui s'appuyaient sur l'ancien nommage.

La modularité supprime cet écueil en considérant des *modules* ou des *classes*[1] qui regroupent toutes les fonctions travaillant sur une structure de données, et en séparant le *corps* de chaque module, invisible/inconnu des autres parties du logiciel, et son *interface* qui est constituée de l'ensemble spécifications des fonctions aptes à lire ou manipuler les données du corps (Figure 3). Ainsi on peut changer à volonté l'algorithmique du corps, sans changer l'interface : les programmes du corps sont de véritables « pièces détachées ». En imposant à toutes les autres parties du logiciel de n'utiliser que les fonctions de l'interface, on résout le problème sus-mentionné.

Figure 3 – Structure d'un module et encapsulation



Le découpage en modules, fortement contraint par des spécifications intermédiaires et détaillées, est généralement guidé par l'inventaire des structures de données utiles (section 12 précédente). Un module est alors constitué d'une interface qui inventorie les fonctionnalités disponibles pour les autres modules, et d'un corps (privé) qui réalise ces fonctionnalités sur la structure de données qu'il gère mais le corps n'est pas accessible par les autres modules.

Un autre avantage de la modularité est de permettre la *compilation séparée* : dans les grandes lignes, l'idée est que changer le corps d'un module n'impacte pas les autres modules, donc seulement la re-compilation de ce module est nécessaire. Toutefois, si l'on change l'interface d'un module, tous ceux qui l'utilisent doivent également être modifiés : il faut donc maintenir un arbre de *dépendances* entre modules. Il existe des outils (comme `make`) pour ne recompiler que le strict nécessaire.

Enfin, la programmation modulaire finit par donner lieu à des bibliothèques (librairies) de modules, ou de classes, qui facilitent la réutilisation de code : au lieu d'écrire *ex nihilo* toutes les parties du projet, on réutilise des modules ou des classes déjà validées, présents dans des bibliothèques dédiées. Lorsqu'un projet est réalisable dans un langage en se contentant « coller » ensemble des modules extraits de bibliothèques, on parle souvent de « langage de glue ».

1. La différence principale entre un langage de programmation *modulaire* et un langage de programmation *orienté objets* est l'héritage.

13.3 L'encapsulation

La modularité n'est pleinement efficace qu'avec l'encapsulation : la structure des modules ou des classes est alors « protégée intrinsèquement » par le langage de programmation qui intègre pleinement cette séparation corps/interface et ne permet *que* l'usage des fonctions déclarées dans l'interface. C'est généralement le cas des langages orientés objets.

C'est néanmoins aux programmeurs du module ou de la classe d'assurer une constante cohérence des données manipulées au sein du corps. Il s'agit d'une cohérence d'usage partagé : si plusieurs programmes font appel à ce module, chacun engendrera des modification interne des données du corps selon ses propres critères. Il ne faut pas que la cohérence du résultat final repose sur les programmes qui font appel au module. Si la cohérence devait dépendre de l'ordre des appels à l'interface alors ce serait aux programmes appelants d'assurer cette cohérence. Mais si chacun des programmes appelants garantissait individuellement la cohérence du résultat final, alors l'interlacement des actions de plusieurs programmes au cours du temps pourrait aboutir à des incohérences graves.
En conséquence, il revient au corps de chaque module d'assurer la cohérence de ses données quel que soit l'ordre d'appel des fonctions de son interface.

13.4 En pratique

Tout ce qui précède aide à choisir un langage de programmation bien adapté au projet du moment. Cependant, en pratique, le premier critère pour choisir un langage est souvent qu'il doit faire partie des compétences de l'équipe de développement. En effet, former toute une équipe à un nouveau langage qui répondrait mieux aux critères « idéaux » pour un projet donné ne serait pas viable économiquement, du moins à court terme.

Il n'en reste pas moins qu'il faut exploiter au mieux la formation permanente, pour élargir et actualiser le champs de compétence des équipes. De plus, une entreprise qui doit faire face à un très gros projet, ou qui devra fournir d'assez nombreux logiciels relevant de domaines similaires, a tout intérêt à investir dans la formation de ses personnels sur un langage bien adapté.

De plus, une bonne formation en informatique rend facile les changements de langages pour des informaticiens diplômés. C'est souvent à cause de la pénurie d'informaticiens, qui conduit à embaucher des personnels n'ayant eu que des formations réduites à quelques semaines, que le choix d'un langage est si limité en entreprise. Pour un esprit bien fait, bien connaître les principes d'un langage dans chacune des classes de langages de la section suivante suffit : tous les langages d'une même classe se ressemblent beaucoup, aux mots-clefs près.

14 Classification des langages

Le style de programmation est lourdement impacté par la classe de langages de programmation utilisée. Il y a principalement trois classes de langages et une quatrième qui est transversale aux deux premières :

1. les langages impératifs,
2. les langages fonctionnels,
3. les langages logiques,
4. les langages orientés objets.

Les langages *impératifs* sont ceux que l'on connaît par défaut : Python, C, Basic, langage machine, Fortran, Cobol, Pascal, ADA (qui est de plus modulaire), *etc.* Ces langages sont « impératifs » en ce sens qu'un programme indique pas à pas comment gérer la mémoire de l'ordinateur, de manière impérative. Par exemple « `i=i+1` » impose, dans la mémoire, une nouvelle valeur à la variable `i`. Le style impératif ne se limite pas à la mise à jour des données, par exemple l'instruction de contrôle `while` est une manière impérative d'imposer à la machine une stratégie de calculs.

Les langages *fonctionnels* sont redoutablement puissant (notablement moins de lignes de code à problème fixé). L'approche de ces langages est de considérer que finalement, un programme n'est jamais qu'une fonction (au sens mathématique du terme) qui prend ses entrées, quelle qu'elles soient, en arguments, et retourne pour résultat d'autres données. Donc tout est fonction dans ces langages. Dès lors, programmer revient à définir de nouvelles fonctions à partir de fonctions natives offertes par le langage. La récursivité est alors un élément basique du style de programmation. Par exemple pour parcourir une liste, on n'utilise pas une boucle `while` qui gère un indice entre 0 et le prédécesseur de la longueur de la liste, on écrit une fonction récursive qui indique le résultat si la liste est vide (cas de base) et qui, sinon, explique comment calculer le résultat à partir du premier élément de la liste et du résultat obtenu récursivement sur le reste de la liste. Les langages fonctionnels les plus connus sont LISP et ses nombreuses variantes, Scheme, ML et ses nombreux dialectes (comme CAML), APL, *etc.*

Le style fonctionnel peut paraître déroutant pour qui s'est plié au style impératif depuis longtemps, cependant si l'on songe au temps passé à réfléchir quelles sont les bornes d'une boucle `while`, et les nombreuses erreurs que cela engendre, le style récursif est notablement plus rapide, plus court et plus fiable. Plus généralement, la diminution du nombre de lignes de code des langages fonctionnels par rapport aux langages impératif est notable... et on sait que le nombre de bugs est proportionnel au nombre de lignes [2].

Les langages *logiques* considèrent qu'un programme a pour objectif d'extraire, à partir d'un ensemble de données ou bien d'un ensemble de connaissances, une ou plusieurs données qui répondent à des propriétés souhaitées. Ces propriétés, ainsi que les connaissances, sont exprimées en logique du premier ordre, avec un langage conventionnel. La langage le plus connu est celui d'origine, appelé PROLOG. Une version plus récente qui intègre de la résolution automatique de contraintes est par exemple ASP. Les programmes dans ces langages commencent par écrire des formules logiques qui traduisent les connaissances, puis une formule logique qui caractérise les résultats souhaités, et le programme retourne, selon les cas, un exemple de donnée résultat, ou parfois l'ensemble des données résultats.
Les langages logiques qui partent de données plutôt que de connaissances logiques sont généralement liés aux bases de données. SQL est l'un d'eux, sans doute le plus connu.

Enfin les langages *orientés objets* sont caractérisés par la possibilité de définir des classes, ce qui aide lourdement à remplir les critères d'abstraction, de modularité et d'encapsulation sus-mentionnés. De plus une notion d'héritage permet d'éviter des redites dans la définition des classes. La plupart des langages orientés objets sont impératifs (C++, Java, Python, Smalltalk, *etc.*) mais il existe également une version objet de ML appelée OCAML, fonctionnelle donc.

2. Il est tout à fait regrettable que les langages fonctionnels soient peu utilisés en industrie, au point qu'ils ne seraient pas reconnus dans un CV d'ingénieur...

VALIDATION ET TEST DE LOGICIEL

Il s'agit ici de se convaincre que le produit logiciel qu'on réalise est conforme aux spécifications. Durant cette phase, on dispose donc de deux documents de référence : d'une part les sources d'un logiciel, d'un module ou d'une portion de code, et d'autre part les spécifications de ce que sont supposées faire ces sources.
Une erreur fréquente est de vouloir tester un logiciel « dans l'absolu » en ne disposant pas des spécifications, on en ne s'y référant pas. Ça n'a naturellement aucun sens : on ne peut pas valider un produit dont on n'a pas défini précisément le comportement attendu.

De même que les spécifications peuvent être formelles (section 11) ou non, on peut considérer les sources d'un logiciel comme un texte formel à valider, ou comme un objet qui s'exécute sur une machine est dont les résultats doivent être validés.
- Si l'on dispose de spécifications formelles et qu'on considère les sources comme un texte formel, alors on peut faire de la preuve de programme assistés par ordinateur (voir la *logique de Hoare* plus loin).
- Si l'on dispose de spécifications formelles et qu'on veut valider les résultats que le programme fournit à l'exécution, alors on peut utiliser les spécifications formelles pour calculer automatiquement si un résultat est correct ou non (voir la notion d'*oracle* plus loin).
- Si l'on a des spécifications informelles mais qu'on veut étudier les sources comme du texte formel, alors on peut mettre en place des procédures d'*analyse statique* de code.
- Enfin si l'on a des spécifications informelles et qu'on veut valider les résultats que le programme fournit à l'exécution, alors on met en place une stratégie de *test dynamique*, souvent appelé *test* tout court.

	sources (validation statique)	**exécutions (validation dynamique)**
spéc. informelle	analyse de code	test
spéc. formelle	preuve de programme + analyse de code	test avec oracle assisté

Que les spécifications soient formelles ou non, il faut toujours commencer par une campagne de validation statique (faible coût et détection de 90% des erreurs), suivie d'une phase de validation dynamique. En effet, même si toutes les sources d'un programme étaient prouvées correctes formellement, le compilateur ou la machine support peut induire des défections, donc exécuter le logiciel est incontournable.

Par ailleurs, on distingue classiquement différentes phases de validation successives, voir la figure 1 p.3 :
- la *validation unitaire* à l'issue de la préintégration, durant laquelle on valide chaque module individuellement par rapport à sa spécification détaillée,
- les *tests d'intégration* durant lesquels on valide le système logiciel au sein de l'entreprise informatique et sur les machines de développement, par rapport aux spécifications intermédiaires,
- enfin la *validation fonctionnelle*, sur le système livré au client, durant laquelle on vérifie que les fonctionnalités promises dans la spécification des besoins sont correctement opérationnelles.

En pratique, l'analyse statique n'est réalisable que durant les deux premières phases et trouver une erreur en validation fonctionnelle doit être exceptionnel en raison du coût déraisonnable de cela peut induire (cf. section 5.2).

On peut parfois « étager » la validation fonctionnelle si l'on dispose de groupes d'utilisateurs privilégiés :
- on parle d'une phase d'*alpha-test* (α-test) lorsqu'on (pré)délivre le logiciel à des utilisateurs compétents capables de remplir une fiche pour chaque bug rencontré, indiquant précisément sa description et les conditions de son occurrence ;
- on parle ensuite de *beta-test* (β-test) lorsqu'à l'issue de l'α-test, on (pré)délivre le logiciel à des utilisateurs privilégiés qui devront commenter leurs expériences d'utilisation.

Dans tous les cas, les principes de bonne conduite de validation statique ou de validation dynamique sont similaires. On va donc les détailler successivement.

15 L'analyse statique de code

MAINTENANCE ET ÉVOLUTION

LES MÉTHODES INCRÉMENTALES / AGILES