Génie logiciel

par **Bertrand MEYER**

Ancien Élève de l'École Polytechnique et de l'École Nationale Supérieure des Télécommunications

Master of Science (Stanford University) ; Docteur ès Sciences.

Professeur associé à l'Université de Californie

Président de la Société Interactive Software Engineering, Inc.

1.	**Introduction**	H 2 050 - 2
1,1	Du slogan à la réalité	– 2
1,2	Logiciel, génie logiciel	– 2
1,3	Quelques questions fondamentales	– 2
2.	**Cycle de vie**	– 3
2,1	Définition et interprétations	– 3
2,2	Étapes	– 4
2,3	Variantes : le prototypage	– 4
3.	**Objectif majeur : la qualité du logiciel**	– 6
4.	**Méthodes**	– 7
4,1	Programmation structurée	– 7
4,2	Méthodes de conception descendantes et ascendantes	– 7
4,3	Méthodes de spécification	– 7
4,4	Méthodes à objets, types abstraits, modularité	– 8
4,5	Méthodes empruntées à l'intelligence artificielle	– 9
4,6	Méthodes de contrôle de la qualité	– 9
4,7	Méthodes de gestion de projets	– 10
5.	**Mesures et modèles**	– 11
5,1	Modèles de coût	– 11
5,2	Modèles de fiabilité	– 12
6.	**Langages**	– 13
7.	**Outils**	– 15
7,1	Outils d'aide à la construction des programmes	– 15
7,2	Outils de gestion de projets et de configurations	– 15
7,3	Outils de contrôle et de validation	– 15
7,4	Au-delà des outils : les environnements	– 16
8.	**Vers une discipline scientifique**	– 17
Index bibliographique		– 19

© TECHNIQUES DE L'INGÉNIEUR 6-1986.

1 Introduction

1,1 Du slogan à la réalité.

Au commencement ce ne fut guère qu'un jeu de mots, à tout le moins une provocation. Lorsqu'en 1968 le comité scientifique de l'OTAN (Organisation du Traité de l'Atlantique Nord) se mit en tête d'inviter quelques-uns des plus grands spécialistes mondiaux de la programmation à réfléchir ensemble sur ce que l'on commençait d'appeler la crise du logiciel, le titre choisi pour la conférence sonna comme un défi : parler de *software engineering*, de génie logiciel, et donc suggérer l'analogie avec de respectables disciplines telles que le génie civil ou le génie chimique, ce ne pouvait être qu'une manière de dérision, un procédé pour alerter les informaticiens sur le caractère rudimentaire de leurs méthodes et de leurs outils quotidiens.

La conférence se déroula à Garmisch, en Bavière, à l'automne 1968, et se termina sur un cri d'alarme dont on peut retrouver l'écho dans les actes publiés un peu plus tard [I.b. 33]. Face à l'extraordinaire développement de la puissance du matériel, le logiciel n'avait pas suivi. Projets en retard, budgets dépassés, informatisations manquées, systèmes monstrueux, graves erreurs de fonctionnement : innombrables étaient les exemples qui montraient l'urgence d'une réflexion en profondeur sur les moyens d'améliorer la qualité des logiciels et leur mode de production.

Dix-sept ans après la conférence de Garmisch, le terme de génie logiciel et son équivalent anglais ne sont plus ressentis comme oxymorons. C'est même plutôt leur succès qui frappe : on ne compte plus les enseignements universitaires, les congrès, les revues et les groupes de travail qui s'en réclament. Mais cette faveur même peut sembler suspecte : au-delà de l'engouement pour une appellation bien tournée, le métier qu'elle désigne est-il vraiment devenu un *génie*, une discipline d'ingénieur ? On peut en débattre. Que les concepts aient progressé, que des idées majeures aient fait leur chemin, il serait difficile de le nier ; mais l'effet réel de ces progrès dans les entreprises, là où se développe le logiciel qui devrait être de qualité industrielle, c'est une autre affaire. Les réflexions qui suivent apporteront peut-être quelques éléments de réponse.

1,2 Logiciel, génie logiciel.

Avant de commencer notre promenade, il est utile de préciser l'objet de l'étude. Qu'appelle-t-on exactement génie logiciel ?

Une définition parfois proposée (pas tout à fait sérieusement) est que cette expression sert à désigner ce que l'on appelait naguère tout bonnement programmation, et doit être employée en présence de ses supérieurs hiérarchiques, des jeunes personnes à qui l'on fait la cour, et plus généralement chaque fois que l'on souhaite donner une apparence plus noble à ce que l'on fait, le terme de programmation étant réservé aux circonstances où l'on doit vraiment se faire comprendre, par exemple lorsqu'on affecte une tâche à l'un de ses subordonnés. Au-delà de la boutade, cette formule a le mérite de rappeler que la distance entre la simple notion de programmation et celle de génie logiciel n'est pas aussi grande que pourraient le faire croire certains clivages entre chercheurs.

A l'autre extrême, on rencontre aussi une définition très limitée qui ne s'intéresse qu'aux circonstances spécifiques de la production industrielle de logiciel : gestion de grands projets, gestion de configurations, organisation des équipes, mesures et modèles de coût et de fiabilité, etc. Cette définition exclut tout ce qui se rapporte aux problèmes classiques de la programmation (langages, outils de programmation, théorie) ; elle procède d'une vue selon laquelle les difficultés de la production de logiciel dans l'industrie sont dues à des problèmes économiques et d'organisation beaucoup plus qu'à des questions strictement techniques. Nous ne partageons pas cette opinion, et la définition qu'elle implique nous paraît exagérément restrictive.

Nous allons tenter de proposer une définition satisfaisante du génie logiciel ; mais il convient pour cela de préciser d'abord le sens du mot *logiciel*.

Définition : on regroupe sous le terme de *logiciel* les différentes formes des programmes qui permettent de faire fonctionner un ordinateur et de l'utiliser pour résoudre des problèmes, les données qu'ils utilisent, et les documents qui servent à concevoir ces programmes et ces données, à les mettre en œuvre, à les utiliser et à les modifier.

Cette définition permet d'inclure dans le logiciel non pas seulement les programmes (sous leurs différentes formes, c'est-à-dire sources aussi bien qu'objets), mais aussi les données et la documentation associée : spécifications, documents de conception, manuels d'utilisation, etc. La définition couvre le logiciel de base (compilateurs, systèmes d'exploitation, etc.) aussi bien que les applications.

Nous suivrons l'usage en utilisant aussi le mot « logiciel » comme adjectif et en écrivant *un logiciel* pour *un produit logiciel*.

Munis de cette première définition, nous pouvons nous attaquer au *génie logiciel* proprement dit.

Définition : on appelle *génie logiciel* l'application de méthodes scientifiques au développement de théories, méthodes, techniques, langages et outils favorisant la production de logiciel de qualité.

Cette définition cherche à réunir les éléments qui nous paraissent essentiels :

– la présence d'une base scientifique, sans laquelle on ne peut parler de génie ;

– cinq directions de développement fondamentales, de la plus abstraite (les théories) à la plus pratique (les outils) ;

– le rôle essentiel de la notion de qualité.

Nous considérons le mot « production » suffisamment général pour qu'il ne soit pas nécessaire d'ajouter « et la maintenance » (ou l'« entretien »).

Cette définition peut apparaître très générale ; on notera cependant qu'elle limite le génie logiciel aux activités de deuxième degré : développer un outil de construction de programmes, c'est faire du génie logiciel ; mais ce n'est pas nécessairement en faire que de développer un simple programme d'application (même de bonne qualité).

1,3 Quelques questions fondamentales.

Il est tout d'abord utile de préciser quelques-uns des grands axes qui définissent les degrés de liberté d'un projet logiciel.

Le premier axe est celui du temps. Un projet logiciel suit un déroulement chronologique ; un modèle de ce déroulement recueille aujourd'hui un accord à peu près général, au moins dans ses grandes lignes : le modèle du *cycle de vie*. Nous le présenterons au paragraphe 2, et nous discuterons ses limitations et quelques variations possibles.

Le deuxième axe correspond à l'un des mots essentiels de la définition précédente : le mot de *qualité*. Le but du génie logiciel étant de permettre la production de logiciel de qualité, il importe de savoir précisément ce que recouvre ce concept. Tel est l'objet du paragraphe 3.

Vient ensuite l'axe des *méthodes* : la mise en place d'une politique de génie logiciel exige des méthodes s'appliquant aux aspects techniques (spécification, conception, validation) et à l'organisation des projets. Quelques-unes de ces méthodes sont brièvement présentées au paragraphe 4.

Certaines des entités caractéristiques des projets logiciels (coûts, délais, taux d'erreurs) se prêtent à une analyse quantitative, et des techniques de modélisation mathématique ont été proposées pour estimer à l'avance la valeur de certains paramètres. C'est l'axe des *mesures et modèles* (§ 5).

Un rôle fondamental est joué par les *langages* utilisés aux différentes étapes de la réalisation des produits logiciels. Cet axe est étudié au paragraphe 6.

Enfin, les *outils* ont une influence considérable en génie logiciel comme dans les autres disciplines. Cette question fait l'objet du paragraphe 7.

Un dernier axe important est orthogonal à tous les précédents ; c'est l'axe des applications, qui permet de parcourir les problèmes spécifiques dus au fait que les produits logiciels ont des objets différents : calcul scientifique, gestion, contrôle-commande, informatique système, etc. Nous nous limiterons, dans cet exposé d'introduction, à l'étude des problèmes généraux du génie logiciel, sans examiner ceux qui ne se posent que dans un domaine d'application particulier.

2 Cycle de vie

2,1 Définition et interprétations.

Un paradigme, dans une discipline scientifique, est un concept fondamental ou un ensemble de concepts (comme la notion d'invariant en physique ou celle de structure en mathématiques), qui implique tout un mode de raisonnement et sert de référence commune aux spécialistes de la discipline à une certaine étape de son évolution historique [I.b. 21]. L'un des paradigmes principaux du génie logiciel est la notion de cycle de vie, pierre de touche de l'étude chronologique des projets.

Le cycle de vie est une modélisation conventionnelle de la succession d'étapes qui préside à la mise en œuvre d'un produit logiciel. Il est souvent représenté sous la forme dite **modèle de la cascade** en raison de l'analogie qu'évoque le schéma correspondant (fig. **1**).

Les huit étapes indiquées sur ce schéma se répartissent en deux époques : jusqu'à la phase d'écriture des programmes, il s'agit de construire, au-delà, de mettre en place.

Il faut noter que le détail de la division en étapes varie dans les différentes présentations publiées. Une norme de l'IEEE (Institute of Electrical and Electronic Engineers) propose cependant une terminologie de référence [I.b. 18].

La notion de cycle de vie se prête à deux interprétations :

– l'interprétation neutre selon laquelle le cycle de vie est une description plus ou moins idéalisée (un **modèle** au sens mathématique du terme) de ce qui se passe dans la plupart des projets logiciels ;

– l'interprétation volontariste qui propose le modèle de la cascade (modèle signifiant ici exemple à suivre) comme une **méthode** à respecter dans l'organisation d'un projet, impliquant en particulier [I.b.6] :

- que toutes les étapes, sans exception, soient exécutées ;
- que l'ordre indiqué soit respecté ;

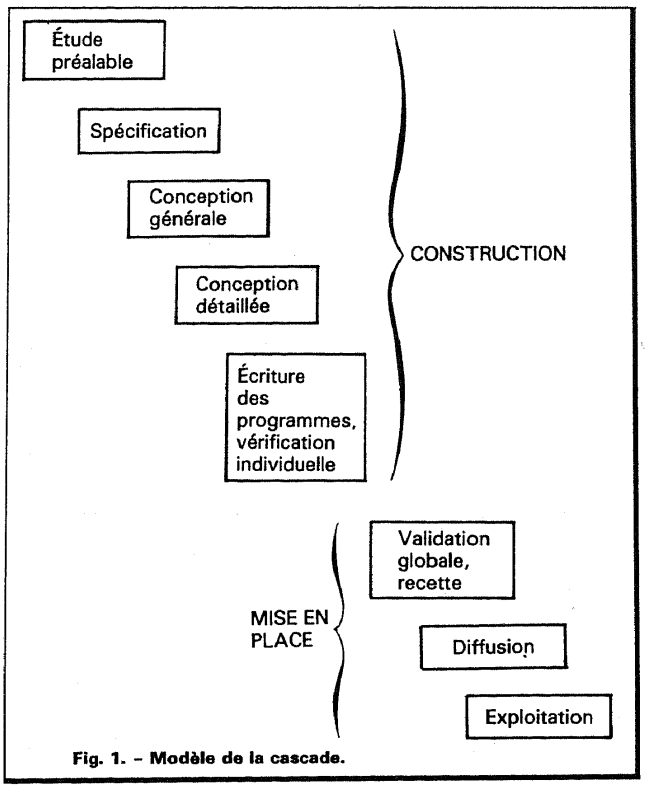

Fig. 1. – Modèle de la cascade.

• que l'on ne passe pas à l'étape *n* sans avoir terminé l'étape *n* – 1 et fait avaliser formellement ses résultats par un comité technique différent de l'équipe de développement, selon une procédure définie à l'avance ;

• que les remises en cause, inévitables en pratique, ne puissent se propager qu'au niveau immédiatement supérieur (on ne peut changer à l'étape *n* que des décisions prises à l'étape *n* – 1) et passent obligatoirement par la même procédure de validation.

2,2 Étapes.

Définissons maintenant plus précisément l'objet des différentes étapes citées ; le résultat de chacune d'elles (ce qui doit en sortir concrètement) est résumé sur le tableau **I**. Nous suivons ici les auteurs qui se rattachent à ce que nous venons d'appeler l'interprétation volontariste ; même si l'on ne s'impose pas la discipline stricte qu'implique cette interprétation, les définitions suivantes fournissent un cadre de référence utile.

Tableau I. – Résultats des étapes du cycle de vie.

Étape	Résultat
Étude préalable	*Phase exploratoire* : – dossier d'entretiens ; – décision (ne pas faire, acheter, faire faire, faire) ; – budget approximatif.
	Phase conceptuelle : – cahier des charges ; – plan général du projet ; – budget précis ; – définition des contraintes.
Spécification	Document de spécification (fonctions et performances) ; première version du manuel d'utilisation ; plan détaillé du reste du projet ; plan de validation (tests en particulier).
Conception générale	Définition des principales structures de données ; décomposition du système en modules (architecture) et description du rôle de chaque module.
Conception détaillée	Description détaillée des structures de données et des modules.
Écriture des programmes, vérification individuelle	Texte des programmes, chaque module vérifié séparément.
Validation globale, recette	Compte rendu de recette ; rapports d'inspection et de validation.
Diffusion	Versions des programmes et de leur documentation adaptées aux différentes catégories d'utilisateurs.
Exploitation	Programme en fonctionnement. Rapports d'incidents et de correction.

Étude préalable : il s'agit tout d'abord *(phase exploratoire)* de déterminer, en général après avoir mené une série d'entretiens avec les utilisateurs potentiels, s'il y a lieu de réaliser l'application et, si oui, d'en fixer les conditions générales, débouchant *(phase conceptuelle)* sur un cahier des charges et un plan du projet.

Spécification : passage de la description non formelle que représente le cahier des charges à une définition précise des objets manipulés par le système, des tâches qu'il doit effectuer sur ces objets (spécification fonctionnelle), des contraintes de performances et de la planification détaillée des étapes suivantes. On peut en particulier préparer dès cette étape le *plan de validation* du système.

Conception générale : on passe ici de la définition à la réalisation. Il s'agit de concevoir l'architecture du système, c'est-à-dire de décrire les principales structures de données (internes aussi bien qu'externes, la seconde catégorie incluant les fichiers et les bases de données) et la décomposition du système en un certain nombre de modules (qui, dans la méthode « à objets », sont précisément bâtis autour des structures de données : cf. § 4,4).

Conception détaillée : affinage des éléments précédents (structures de données, modules) jusqu'à l'obtention d'une forme permettant d'écrire immédiatement les programmes.

Écriture des programmes et vérification individuelle : écriture des textes des programmes et mise en œuvre des structures de données ; vérification interne de chaque module par ses auteurs.

Validation globale, recette : validation de l'ensemble du système ; il s'agit cette fois de valider par rapport aux fonctions à assurer (on distingue la *vérification,* qui a pour objet de contrôler la cohérence interne d'un élément de logiciel, de la *validation,* qui contrôle le respect de conditions fixées extérieurement à l'objet lui-même). Cette phase s'appuie sur le plan de validation défini à l'étape de spécification ; elle est normalement confiée à une équipe différente de l'équipe de développement. Son résultat est un document de certification souvent appelé compte rendu de recette.

Diffusion : préparation et distribution des différentes versions.

Exploitation : mise en place du système dans son environnement opérationnel ; prise en compte des rapports d'incident ; correction des erreurs.

Dans ce modèle, il convient de noter l'absence d'étapes associées à la **documentation** (activité nécessaire tout au long du projet et non pas phase indépendante du cycle de vie) et à la **maintenance** (qui est couverte par l'« exploitation » pour ce qui est de la correction des erreurs, et doit être considérée comme relevant d'un nouveau projet pour ce qui est des modifications de spécification).

2,3 Variantes : le prototypage.

La notion de cycle de vie, avec ce qu'elle implique de rigidité dans le déroulement séquentiel des étapes, a donné lieu à un certain nombre de critiques ; des schémas plus souples ont été proposés. Il est en particulier beaucoup question depuis quelques années de **prototypage rapide**.

Le prototypage rapide s'oppose non seulement à la méthode du cycle de vie, mais aussi à celle du prototypage lent, qui est certainement l'une des façons les plus courantes de développer du logiciel.

Les tenants de cette méthode considèrent qu'il est vain de vouloir figer à un stade précoce des spécifications qui, de toute façon, vont être remises en cause. Ils recommandent donc de travailler par ap-

proximations successives, en construisant d'abord, le plus rapidement possible, une première version sommaire, un prototype.

Les avis diffèrent sur le rôle exact qui est assigné au prototype ; on peut distinguer deux grandes tendances.

• Dans le premier cas, le prototype est simplement destiné à mettre certaines hypothèses à l'épreuve, hypothèses qui peuvent porter par exemple sur l'interface du système avec ses utilisateurs, l'efficacité de certains algorithmes, la faisabilité de certaines tâches, etc. ; une fois le résultat obtenu, on ne conserve pas le prototype et le développement repart de zéro, fort cependant de l'expérience gagnée. Avec cette méthode, le prototype peut être réalisé avec des outils et un langage différents de ceux qui seront retenus pour le produit final ; certains langages de très haut niveau, de mise en œuvre peu efficace mais permettant d'obtenir très vite des programmes exécutables, peuvent ici être mis à profit (cf. § 6).

• Un prototype de la seconde forme ne comporte que certaines des fonctions de base du système envisagé ; il s'agit de le faire évoluer vers un produit complet par additions successives, sans rupture de continuité.

On peut utiliser les termes de *prototype jetable* et de *prototype incrémental* pour distinguer ces deux variantes (certains auteurs parlent de maquette dans le premier cas et réservent le terme de prototype au second, mais cette terminologie n'est pas universelle).

Le débat entre partisans de la méthode séquentielle et du prototypage n'est pas clos. Il est intéressant de noter que les arguments des deux écoles se nourrissent de la même constatation de base : tous les résultats d'études sur le coût des erreurs en logiciel montrent que les erreurs commises tôt dans le cycle de vie (aux étapes d'étude préalable et de spécification) et décelées tard ont des conséquences énormes sur le coût des projets. Ce phénomène est illustré de façon particulièrement frappante par une courbe résumant les résultats d'un certain nombre d'analyses de grands projets et publiée par B. W. Boehm [I.b. 5] (fig. **2**). Les données sur ces projets proviennent de l'Armée de l'Air américaine (projet Safeguard) et des sociétés américaines IBM, TRW et GTE.

On voit pourquoi ces résultats peuvent servir à justifier des méthodes très différentes :

– une conclusion possible est qu'il faut consacrer une attention considérable aux phases initiales du cycle de vie, afin d'éviter à tout prix que des erreurs commises lors de la définition du système passent inaperçues jusqu'aux étapes de mise en place : c'est l'un des arguments majeurs des recherches sur la spécification formelle ;

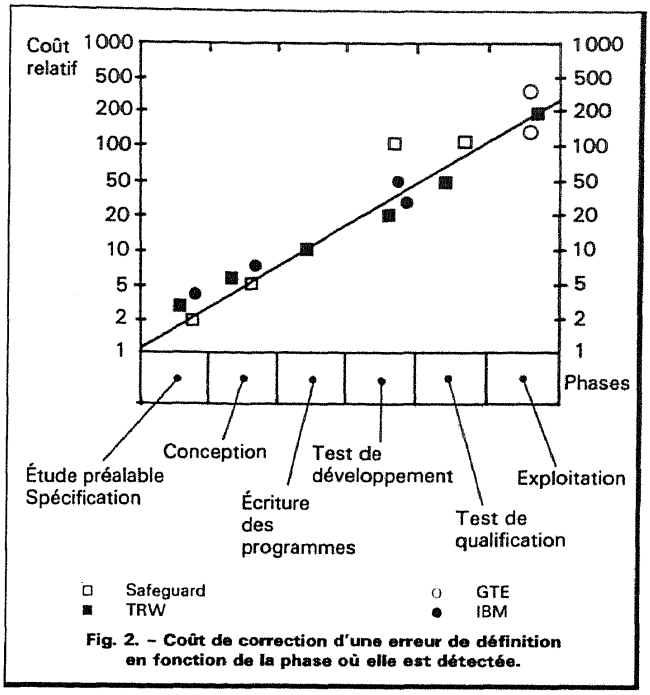

Fig. 2. – Coût de correction d'une erreur de définition en fonction de la phase où elle est détectée.

– on peut au contraire en déduire que, les erreurs de définition étant inévitables, la seule chose raisonnable est d'essayer d'en limiter les conséquences ; il convient donc de passer le plus vite possible à l'expérimentation séparée sur des segments du système, segments suffisamment bien délimités pour que les phénomènes décrits par la figure 2 n'aient pas d'effet catastrophique sur le projet dans son ensemble. C'est la voie qui mène au prototypage sous l'une de ses deux formes.

Les deux méthodes (prototypage et spécification complète avant toute réalisation) ne sont cependant pas aussi totalement incompatibles que cette discussion peut le laisser supposer. En particulier, un prototype jetable peut dans certains cas faire bon ménage avec une spécification : s'il est naïf de croire que la réalisation d'un prototype dispense d'une réflexion initiale approfondie, il peut être tout à fait raisonnable d'intégrer à cette réflexion une composante expérimentale, sous la forme d'un prototype destiné à valider certaines hypothèses qu'il est difficile d'analyser autrement.

L'auteur a porté ses recherches dans la direction de la spécification plus que dans celle du prototypage (encore qu'il ait, comme tout le monde, pratiqué sans l'avoir cherché le prototypage incrémental lent). Mais on ne peut nier que la technique du prototypage rapide puisse rendre des services lorsque les conditions de son application sont réunies (une liste de conditions nécessaires est donnée en [I.b. 7]).

[I.b.*n*] renvoie à la référence *n* de l'Index bibliographique placé en fin d'article.

3 Objectif majeur : la qualité du logiciel

La définition du génie logiciel sur laquelle repose cet article accorde une place primordiale à la notion de qualité. Il est donc nécessaire d'examiner d'un peu plus près ce que recouvre cette notion dans le cas du logiciel.

Un certain nombre d'auteurs en ont proposé des définitions ; l'une des premières à reposer sur une étude systématique était celle de Boehm et de ses collaborateurs à TRW [I.b. 4]. La définition dont nous nous inspirerons ici est celle de James McCall, résultat d'une étude effectuée en 1977 par General Electric pour le compte de l'armée de l'air américaine [I.b. 22].

McCall prend bien soin de distinguer les deux sortes de conditions qui déterminent la qualité d'un produit logiciel :

- les caractéristiques *externes* de la qualité d'un logiciel, comme l'extensibilité (facilité d'adaptation à des changements de spécifications), qui peuvent être perçues par les utilisateurs du produit ;
- les caractéristiques *internes* de la qualité, qui peuvent être analysées par les informaticiens à l'examen du programme et des autres documents techniques associés : un exemple est la modularité du système, c'est-à-dire sa division en unités logiques simples et cohérentes.

Cette distinction, qui n'apparaissait pas toujours clairement dans les travaux antérieurs sur la qualité du logiciel, est importante car elle évite de confondre causes et symptômes. McCall appelle **facteurs** les caractéristiques externes et **critères** les caractéristiques internes.

Seuls les facteurs comptent en dernier ressort, puisqu'ils déterminent l'utilisation du produit ; mais c'est le respect des critères au cours du développement qui conditionne la production d'un logiciel conforme à ces facteurs. Les exemples précédents illustrent cette remarque : la modularité (critère) est un élément important pour assurer l'extensibilité (facteur).

Plus généralement, McCall propose une matrice montrant l'influence de chacun des critères sur chacun des facteurs. Cette influence peut être positive, comme dans l'exemple de la modularité et de l'extensibilité, négative ou nulle. Cette analyse est complétée par la définition d'un certain nombre de mesures, permettant d'associer à chaque critère des vérifications quantitatives.

Le tableau **II** donne la définition de dix facteurs voisins de ceux de McCall.

L'une des caractéristiques essentielles d'une telle définition de la qualité est qu'elle est plurielle : ce sont *des* qualités possibles que l'on met ici en évidence. Or un simple examen montre que ces qualités ne sont pas nécessairement deux à deux compatibles. On ne saurait par exemple atteindre l'optimum tout à la fois en matière de fiabilité et de facilité d'emploi : le premier but entraîne inévitablement l'inclusion de protections, de barrières qui sont préjudiciables au second.

Une telle constatation n'a rien de vraiment surprenant : dans toute branche d'ingénierie, la qualité s'obtient au prix d'un compromis entre des objectifs souvent contradictoires : coûts, délais de réalisation, étendue des fonctions offertes, sécurité d'emploi, etc. Mais en logiciel ce compromis est trop souvent encore réalisé de façon inconsciente, l'un des facteurs (l'efficacité par exemple) étant exagérément privilégié.

Si la nécessité d'opérer des compromis est une des constantes de l'ingénierie, le problème se pose de façon particulièrement aiguë en génie logiciel du fait de la complexité des logiciels que l'on réalise aujourd'hui. On peut sans risque affirmer que certains de ces logiciels, comme ceux qui équipent les autocommutateurs téléphoniques ou certains logiciels militaires, dépassent en complexité tous les systèmes de quelque nature précédemment construits par l'humanité.

Certains se demandent si nous n'approchons pas des limites au-delà desquelles cette complexité ne pourra plus être maîtrisée. C'est ainsi qu'un célèbre informaticien américain, David Parnas, a démissionné de son poste de conseiller scientifique auprès de l'*Initiative de Défense Stratégique* du Président Reagan, non par désaccord politique, mais parce qu'il estime techniquement irréalisable le logiciel nécessaire au succès de cette entreprise ; sa prise de position [I.b. 37] a suscité un débat passionné parmi les spécialistes du domaine.

Tableau II. – Dix facteurs de la qualité du logiciel.

Facteur	Définition
Validité	Aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et la spécification
Fiabilité	Aptitude d'un produit logiciel à fonctionner dans des conditions anormales
Extensibilité	Facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qui lui sont demandées
Réutilisabilité	Aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications
Compatibilité	Facilité avec laquelle un logiciel peut être combiné avec d'autres
Efficacité	Utilisation optimale des ressources matérielles (processeurs, mémoires internes et externes, dispositifs de communication, etc.)
Portabilité	Facilité avec laquelle un produit peut être transféré dans différents environnements matériels et logiciels
Vérifiabilité	Facilité de préparation des procédures de recette et de validation (particulièrement des jeux d'essai), et des procédures de détection d'erreurs et de suivi d'incidents en exploitation
Intégrité	Aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés
Facilité d'emploi	Facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des résultats, de rattrapage en cas d'erreur d'utilisation

4 Méthodes

De nombreux travaux ont été menés depuis une quinzaine d'années sur la méthodologie de construction des programmes. Nous avons retenu quelques étapes particulièrement marquantes.

4,1 Programmation structurée. – Le terme de programmation structurée est un de ceux qui ont eu le plus de succès, à la suite des travaux de Dijkstra et Hoare (à partir de 1967 environ). Pourtant, les idées de ces précurseurs n'ont souvent atteint les praticiens de la programmation que sous une forme tronquée.

Il est trop courant de voir la programmation structurée présentée comme un simple ensemble d'interdictions relatives aux structures de contrôle dans les langages de programmation : ne pas employer de branchements, se limiter à trois figures de base (enchaînement, choix, boucle) et à quelques variantes.

En réalité, la programmation structurée se voulait une remise en cause profonde des méthodes habituelles de construction de logiciel, et ses créateurs insistaient sur la nécessité d'une démarche rigoureuse, mathématique. Un rôle important était dévolu à la notion de démonstration de validité des programmes : un programme doit être construit, selon Dijkstra et Hoare, comme on prouve un théorème ; sa validité doit être évidente, et mathématiquement vérifiable.

Ces méthodes rigoureuses exigent que l'on parte d'une description du problème qui soit elle aussi mathématiquement irréprochable ; elles mènent donc au problème de la spécification formelle. Nous en donnerons quelques exemples après avoir abordé ce thème (§ 4,3).

4,2 Méthodes de conception descendantes et ascendantes. Les méthodes de conception proposent des principes généraux pour guider le concepteur d'un système complexe à travers le labyrinthe des problèmes à résoudre.

Les méthodes le plus souvent associées à la programmation structurée sont de type **descendant**, c'est-à-dire qu'elles appliquent une démarche systématique, dite « par affinages successifs », qui part de l'expression la plus générale du problème à résoudre et décompose répétitivement les tâches à effectuer en sous-tâches plus simples, jusqu'à ce que tout ait été exprimé en termes d'opérations assez élémentaires pour être mises en œuvre directement dans un langage de programmation.

Les méthodes **ascendantes** ne doivent pas être négligées pour autant : elles cherchent à favoriser la réutilisation de logiciel existant et la construction de nouveaux systèmes par combinaison d'éléments prédéfinis. Elles répondent ainsi à l'un des problèmes clés du génie logiciel, celui de la réutilisabilité : trop d'investissements sont perdus en raison du faible taux de réutilisation des programmes et de l'habitude (qui, dans l'état actuel de la technique, est souvent une nécessité) de repartir de zéro pour chaque nouveau produit.

La vérité en matière de méthodes de conception consiste sans doute en une judicieuse combinaison de descendant et d'ascendant, permettant d'appliquer à chaque nouveau problème une étude systématique partant de la spécification, tout en favorisant au maximum la réutilisation d'éléments de logiciel précédemment réalisés.

4,3 Méthodes de spécification. – Les méthodes de spécification sont destinées à faciliter l'analyse des problèmes à résoudre et la description externe des systèmes.

Dans le cycle de vie, la spécification précède les phases de conception et de mise en œuvre, au cours desquelles les objets du système (programmes et structures de données) seront effectivement construits. La spécification a pour but de définir auparavant les propriétés précises que devront respecter ces objets.

Au-delà d'une certaine taille de projet, cette tâche se révèle souvent la plus importante et la plus ardue. Plusieurs méthodes ont été proposées pour la faciliter. Parmi celles qui sont réellement employées dans certaines branches de l'industrie, les unes (SADT, Merise) cherchent essentiellement à faciliter le dialogue entre les informaticiens et leurs clients, en proposant un support commun de description et de discussion (graphique dans le premier cas cité). D'autres (SREM, PSL/PSA) mettent l'accent sur les outils informatiques permettant d'exercer un contrôle serré de l'évolution du projet, de ses spécifications et de sa documentation.

A côté de ces méthodes issues en général de l'industrie, de nombreux chercheurs ont proposé des **langages formels** permettant, grâce à des notations de type mathématique, d'exprimer les spécifications sous une forme rigoureuse et non ambiguë. Les méthodes associées sont en général plus difficiles à mettre en œuvre, car elles exigent de la part des spécificateurs une culture mathématique et une habitude du raisonnement abstrait que tous les informaticiens ne possèdent pas ; c'est, selon les partisans des méthodes formelles, le prix à payer pour obtenir des systèmes informatiques dont la validité et la fiabilité pourront être garanties. On trouvera dans l'article [I.b. 29] une discussion de l'utilité comparée des spécifications formelles et informelles.

On peut distinguer, dans les travaux sur la spécification formelle, trois directions principales : *validation, construction, exécution.* Ces trois façons de considérer la spécification ne sont pas nécessairement incompatibles ; elles se distinguent par l'utilisation prioritaire à laquelle on destine les spécifications.

Les méthodes de la première catégorie (par exemple FDM, HDM, Affirm) mettent l'accent sur la production de spécifications qui puissent servir de base à la **validation** automatisée des programmes ; des progrès importants ont été réalisés dans ce domaine depuis quelques années, et l'on sait désormais valider mathématiquement (au prix d'un effort qui reste considérable) des programmes de plusieurs dizaines de milliers d'instructions, incluant des processus parallèles.

D'autres méthodes ont pour but essentiel non pas de permettre la validation de programmes développés indépendamment de la spécification, mais plutôt d'utiliser la spécification elle-même comme point de départ de la **conception** du programme, le processus de validation étant intégré au processus de construction. Partant de la spécification, on peut procéder par déduction, par transformation ou par utilisation de schémas prédéfinis (ou encore par une combinaison de ces techniques) pour obtenir des versions exécutables. Les travaux de Dijkstra [I.b. 12] ont donné une forte impulsion à ce domaine ; de nombreuses équipes s'y intéressent : projet CIP à Munich, méthodes développées à Nancy (méthode déductive, SPES), méthode VDM (Jones, Bjørner), travaux menés par Abrial, Manna, Sintzoff, van Laamswerde, et par de nombreux chercheurs gravitant autour du groupe de travail 2.1 de l'IFIP (International Federation for Information Processing) et en France du groupe Anna Gram ; voir aussi [I.b. 24].

La figure 4 (p. 9) cherche à donner une idée, sur un petit exemple, de ce que peut être une telle méthode de construction des programmes par affinage de leur spécification.

Le problème traité ici est celui de la recherche d'un élément dans une table supposée triée. Il s'agit bien entendu d'un exemple de petite taille, mais qui présente quelques pièges. On s'en persuadera en consultant d'abord la figure **3***a* (p. 8), qui présente quatre tentatives de solution du même problème ; le lecteur est invité à vérifier

Pour la table analytique, se reporter à la première page de cet article.

```
i := 1 ; j := n;
tant que i ≠ j faire
      m := ⌊(i + j)/2⌋;
      si t[m] ≤ x alors
            i := m
      sinon
            j := m
      fin
fin;
présent := (x = t[i])
```

Programme 1

```
i := 1 ; j := n; trouvé := faux;
tant que i ≠ j et non trouvé
faire
      m := ⌊(i + j)/2⌋;
      si t[m] < x alors
            i := m + 1
      sinon si t[m] = x alors
            trouvé := vrai
      sinon j := m − 1 fin
fin;
présent : = trouvé
```

Programme 2

```
i := 0; j := n;
tant que i ≠ j faire
      m := ⌈(i + j)/2⌉;
      si t[m] ≤ x alors
            i := m + 1
      sinon
            j := m
      fin
fin;
si i ≥ 1 et i ≤ n alors
      présent := (x = t[i])
sinon
      présent := faux
fin
```

Programme 3

```
i := 0; j := n + 1;
tant que i ≠ j faire
      m := ⌊(i + j)/2⌋;
      si t[m] ≤ x alors
            i := m + 1
      sinon
            j := m
      fin
fin;
si i ≥ 1 et i ≤ n alors
      présent := (x = t[i])
sinon
      présent := faux
fin
```

Programme 4

ⓐ programmes

Pour chacun des quatre programmes, on trouvera ci-après une ou plusieurs valeurs du tableau t et de l'élément x pour lequel le programme est incorrect. Dans tous ces contre-exemples, t contient un ou deux éléments, mais il existe une infinité d'autres valeurs qui soulèvent les mêmes problèmes.

Programme 1
$t = [1\ 2]$, $x = 2$ (bouclage infini)

Programme 2
$t = [1\ 2]$, $x = 0$ (débordement du tableau)
$t = [1]$ (résultat erroné : *présent* mis à faux alors que l'élément est là)

Programme 3
$t = [1\ 2]$, $x = 0$ (débordement du tableau)

Programme 4
$t = [1\ 2]$, $x = 0$ (débordement du tableau)

ⓑ contre-exemples

La notation $\lfloor r \rfloor$, où r est un réel, désigne le plus grand entier inférieur ou égal à r (partie entière) ; de la même façon, la notation $\lceil r \rceil$ désigne le plus petit entier supérieur ou égal à r (partie supérieure). Si p est un entier, $\lfloor p/2 \rfloor$ est la division de p par 2 et $\lceil p/2 \rceil$ est égal à $\lfloor (p + 1)/2 \rfloor$.

Fig. 3. – Quatre programmes (faux) de recherche dichotomique.

que toutes ces solutions, à première vue acceptables, sont en fait erronées ; il suffit pour cela de montrer pour chacune d'entre elles que certaines valeurs du tableau t et de l'élément x produiront un résultat incorrect (*présent* mis à vrai alors que l'élément est absent, ou inversement) ou entraîneront une erreur à l'exécution (débordement de mémoire, bouclage infini). Un contre-exemple est fourni dans la figure 3*b* pour chacun de ces quatre programmes.

Le programme (correct) de la figure **4** calcule le plus grand indice i tel que $t[i] \leqslant x$ (ou 0 s'il n'existe pas de tel indice : la spécification plus précise est donnée sur la figure). Pour obtenir la réponse à la question « x apparaît-il dans t ? », on doit le faire suivre d'une instruction de la forme :

```
si i ⩾ 1 et i ⩽ n alors
      présent : = (x = t[i])
sinon
      présent : = faux
fin
```

On notera à quel point tous les détails de la construction sont importants pour obtenir un programme correct : valeurs initiales, signes $<$ ou $\leqslant$, affectation de m à i et de $m - 1$ à j, choix de la partie supérieure du résultat de la division, etc.

La troisième voie de développement citée à propos de la spécification formelle est celle qu'empruntent certains auteurs qui cherchent à abattre la cloison entre la spécification, forme descriptive du système, et le programme, forme prescriptive (exécutable). On obtient ainsi la notion de **spécification exécutable**. Cette approche a été particulièrement illustrée par les utilisateurs de certaines méthodes issues de l'intelligence artificielle et en particulier du langage Prolog (cf. § 4) : en utilisant des notations formelles soumises à certaines restrictions, on peut obtenir des textes interprétables comme spécifications mais pouvant également être exécutés sur un ordinateur. On doit accepter pour cela une perte de puissance expressive (du point de vue de la spécification) et d'efficacité (du point de vue de l'exécution), mais on gagne l'unicité de la description. C'est une des méthodes qui tendent à réconcilier la spécification et le prototypage.

4,4 Méthodes à objets, types abstraits, modularité. – Il convient de mentionner un ensemble de méthodes qui peuvent s'appliquer aux différentes étapes de la première partie du cycle de vie (la phase de construction) et s'opposent assez nettement à la démarche traditionnelle. Dans une programmation (conception, spécification) par objets, on décrit un système de façon globale, moins par la fonction qu'il remplit que comme un ensemble de classes d'objets, caractérisés par leurs propriétés abstraites.

Le bien-fondé de cette approche est particulièrement évident dans des domaines comme la conception des systèmes d'exploitation ou des systèmes de contrôle-commande, où les objets coopérants sont les gestionnaires des différentes ressources disponibles ; mais la même idée se transpose avec bonheur aux autres domaines d'application.

La programmation par objets a été introduite en 1967 par le langage de programmation Simula 67, dont les principales idées ont depuis été reprises par Smalltalk. Certains éléments de cette méthode ont également influencé la conception du langage Ada. La base théorique a été fournie par les travaux menés à partir de 1974 autour de la notion de **type abstrait**, modélisation mathématique des concepts informatiques de type et d'objet.

Donnée : un tableau $t[1..n]$;
un élément x.
Bien entendu, le programme ne peut modifier les valeurs de ces objets.
On suppose $n \geq 0$ (si $n = 0$, le tableau est vide).
Hypothèse : t trié, c'est-à-dire

$$t[i] \leq t[j] \text{ pour } 1 \leq i \leq j \leq n$$

Résultat cherché : un entier i tel que

$$0 \leq i \leq n \text{ et}$$

$$\begin{cases} t[k] \leq x \text{ pour } 1 \leq k \leq i \\ t[k] > x \text{ pour } i+1 \leq k \leq n \end{cases}$$

Note : il est important de vérifier qu'il existe toujours un index i unique conforme à ces propriétés. En particulier, i vaudra 0 si $x < a[1]$; i vaudra n si $x \geq t[n]$ (on se souviendra qu'une propriété de la forme « pour tout x appartenant à E, $P(x)$ » est toujours vraie, quelle que soit la propriété P, si l'ensemble E est vide).

MÉTHODE
Le but recherché peut s'écrire sous la forme

$$c_1 \text{ et } c_2$$

où c_1 est défini comme

$$\begin{cases} 0 \leq i \leq j \leq n \text{ et} \\ t[k] \leq x \text{ pour } 1 \leq k \leq i \text{ et} \\ t[k] > x \text{ pour } j+1 \leq k \leq n \end{cases}$$

et c_2 est la condition $i = j$.
Considérons c_1 comme un invariant et c_2 comme un but. Le programme peut s'écrire :

établir c_1 de façon simple ;
tant que *c_2 n'est pas vérifié* **faire**
rapprocher i de j
en conservant c_1
fin

MISE EN ŒUVRE
Pour établir c_1 de façon simple, il suffit de prendre $i = 0$, $j = n$. Pour rapprocher i de j, considérons la valeur médiane :

$$m = \lceil (i+j)/2 \rceil$$

Pouvons-nous affecter à i ou à j la valeur de m sans pour autant remettre en cause la validité de l'invariant c_1 ? Cela dépend des valeurs relatives de x et $t[m]$.

- Si $t[m] \leq x$, nous savons que $t[k] \leq x$ pour $1 \leq k \leq m$ puisque t est trié. Nous pouvons donc choisir m comme nouvelle valeur de i.
- Si $t[m] > x$, nous savons que $t[k] > x$ pour $k \geq m$, c'est-à-dire pour $k \geq (m-1)+1$. Nous pouvons donc choisir $m-1$ comme nouvelle valeur de j.

Mais attention, ce raisonnement n'est correct qu'en raison de deux propriétés que le lecteur est invité à vérifier soigneusement :

- dans les deux cas, le nouveau choix de i ou de j fait décroître la valeur de la quantité entière $j - i$ (le variant de la boucle), tout en la maintenant positive ou nulle : le processus est donc assuré de se terminer ;
- la valeur choisie pour m est telle que, dans les conditions d'exécution du corps de boucle (c_1 vérifié, c_2 non vérifié), $1 \leq m \leq n$: l'élément $t[m]$ utilisé pour les comparaisons est donc bien défini.

SOLUTION
Nous obtenons finalement le programme suivant :

```
i := 0 ; j := n ;
tant que i ≠ j faire
    m := ⌈(i + j)/2⌉ ;
    si t[m] ≤ x alors i := m
    sinon j := m − 1 fin
fin
```

Fig. 4. – Développement d'un programme selon une méthode rigoureuse : exemple de la recherche dichotomique.

La notion de programmation par objets se combine particulièrement bien avec les techniques de programmation modulaire introduites à partir de 1972 par David Parnas (cf. tout particulièrement [l.b. 35] et [l.b. 36]). La principale contribution de Parnas est la notion de **masquage de l'information**, selon laquelle chaque module ne doit laisser filtrer vers l'extérieur que ses propriétés officielles, en conservant par-devers lui les détails de sa mise en œuvre, ou *secrets*. En autorisant les autres modules à se servir de ces propriétés internes, on rendrait en effet très difficiles, voire impossibles, les remises en cause ultérieures, bloquant ainsi toute évolution du système. Parmi les facteurs de la qualité cités au paragraphe 3, c'est donc à l'*extensibilité* et à la *réutilisabilité* que le principe du masquage de l'information doit toute son importance.

4,5 Méthodes empruntées à l'intelligence artificielle. – Les spécialistes d'intelligence artificielle ont depuis longtemps (plus précisément depuis 1959, date de diffusion de Lisp) utilisé des méthodes et des langages les plaçant quelque peu en marge du génie logiciel. L'une de leurs contributions les plus importantes est la mise au point d'environnements de programmation avenants, tel Interlisp, disponibles dans les laboratoires d'intelligence artificielle bien avant que l'industrie se préoccupât de la question.

Remarque : les langages de l'intelligence artificielle et les systèmes associés se distinguent en particulier par le caractère dynamique des objets manipulés : il est possible dans ces environnements de créer librement de nouveaux objets de programme à l'exécution. Les langages utilisés en génie logiciel classique sont bien plus statiques ; il faut en général prévoir dès avant l'exécution le nombre et la taille des objets dont on aura besoin (cela est vrai de Fortran et de Cobol, mais aussi dans une large mesure de langages comme Pascal, C et Ada dont les possibilités de création dynamique d'objets voient leur utilité pratique limitée par l'absence de mécanismes automatiques de récupération de la mémoire). Cela nous paraît une des limitations techniques majeures des langages classiques. On notera que les langages à objets tels que Simula et Smalltalk n'en souffrent pas.

Plus récemment, le bruit fait autour des **systèmes experts** et de la programmation heuristique a soulevé quelques espoirs : peut-on espérer que des systèmes intelligents d'aide à la programmation fourniront une solution radicale aux grands problèmes du génie logiciel ? Quelques applications expérimentales de systèmes experts d'aide à la construction ou à la correction de programme ont été réalisés, mais aucune expérience en vraie grandeur n'a encore fourni de réponse définitive. Un numéro spécial des *Transactions* de l'IEEE sur le génie logiciel [l.b. 31] fait le point sur les perspectives de fécondation croisée entre génie logiciel et intelligence artificielle.

4,6 Méthodes de contrôle de la qualité. – La qualité : telle est, on l'a dit, la préoccupation principale en génie logiciel. Un vieil adage des ingénieurs d'assurance de la qualité, qui s'applique au logiciel comme aux autres domaines, dit que la qualité ne se contrôle pas a posteriori : elle se construit a priori. On ne peut produire du logiciel conforme aux facteurs du paragraphe 3 qu'en intégrant la recherche de la qualité au processus de construction, tout particulièrement aux premières étapes du cycle de vie : lorsque le logiciel existe, il est trop tard pour se préoccuper de sa qualité. Aussi les méthodes examinées précédemment s'appliquent-elles à la *production* de logiciel de qualité.

Il serait déraisonnable, cependant, de s'en remettre aveuglément au processus de construction, si rigoureux soit-il, pour garantir la qualité : trop d'erreurs et d'imprévus guettent le logiciel sur la route de la perfection pour qu'un processus de contrôle permanent ne soit pas indispensable.

Les méthodes de contrôle de la qualité développées jusqu'à présent s'appliquent surtout au produit ultime, le code. Il s'agit en particulier des méthodes de test, qui aident à la préparation des données d'essai et à la conduite des essais eux-mêmes.

On distingue les tests dits « boîte noire », où l'on déduit les données d'essai de la structure des spécifications, et les tests en boîte blanche, qui prennent en compte la structure des programmes. Cette distinction recoupe celle que l'on opère entre validation et vérification (cf. § 2,2).

Les méthodes en boîte blanche sont bien adaptées à l'étape de vérification individuelle, où l'on veut en général contrôler, pour cha-

que élément de programme, que les données d'essai choisies ont permis de suivre tous les chemins possibles, de couvrir pour chaque variable un large spectre de valeurs, etc. ; les méthodes en boîte noire, elles, s'appliquent tout naturellement à l'étape suivante (validation globale et recette), où l'on veut s'assurer, sans se laisser influencer par la structure interne des programmes, qu'ils remplissent bien les fonctions définies par la spécification.

Les méthodes de test se prêtent aussi à la distinction entre « ascendant » et « descendant » introduite à propos de la conception. Avec une méthode ascendante, on teste chaque module avant les modules qui l'utilisent ; on est donc certain de tester chaque élément dans des circonstances qui se rapprochent de son utilisation opérationnelle, mais on ne peut s'attaquer aux modules de plus haut niveau (ceux dans lesquels les erreurs sont les plus graves et les plus difficiles à corriger) qu'à la fin du processus. Une méthode descendante permet en revanche de commencer par les éléments de plus haut niveau ; mais elle exige que l'on écrive, à la place des modules non encore développés, des pseudo-modules ou échafaudages (*stubs* en anglais), versions très simplifiées des modules manquants, destinées à permettre au test de se dérouler normalement. La méthode descendante permet de mieux répartir l'effort de test au cours du projet, mais elle implique, outre l'effort supplémentaire consacré aux échafaudages, le risque que les conditions de test diffèrent par trop des conditions réelles d'exploitation.

Bien entendu, le choix d'une méthode de test ascendante ou descendante est étroitement lié au choix correspondant quant à la méthode de conception.

Tester les programmes implique qu'on les exécute. Les limites de cette technique sont bien connues : les tests ne sauraient être exhaustifs ; ils renseignent sur les erreurs qu'ils détectent, mais non sur celles qui leur échappent ; dans le cas de systèmes interactifs, parallèles ou non déterministes, ils sont difficiles à mettre en œuvre et ne peuvent être reproduits à l'identique. D'autres méthodes de validation et de vérification ont été développées :

- les techniques de **démonstration** mathématique de la validité des programmes, étroitement liées aux méthodes de spécification formelle (cf. § 4,3), sont théoriquement idéales mais encore difficiles à mettre en œuvre ;
- plus modestes sont les techniques d'**analyse statique**, qui permettent d'examiner le texte des programmes pour y détecter les anomalies potentielles : des tests sans exécution, en quelque sorte. Nous citerons au paragraphe 7,3 certains outils d'analyse statique.

Avec quelques procédures manuelles qui ressortissent plutôt à la gestion de projets, comme la procédure des inspections de code (réunions d'analyse et de critique des programmes en cours de développement), les techniques que nous venons de voir (le test et ses variantes) constituent l'essentiel de ce qui est pratiqué aujourd'hui en matière de contrôle de la qualité.

Ces techniques s'appliquent surtout, on l'aura noté, aux phases finales du cycle de vie et à leurs produits, les programmes. Elles ne répondent donc pas vraiment à l'objectif défini au début de cette discussion du contrôle de la qualité : l'idée d'un contrôle permanent du projet et de ses produits.

Nota : le modèle de la cascade, qui semble présenter la vérification et la validation comme des étapes chronologiques et non comme des activités continues, n'est pas entièrement satisfaisant à cet égard.

Il est de toute évidence nécessaire de développer des méthodes de contrôle de la qualité qui s'appliquent à tous les produits d'un projet (programmes, mais aussi données, cahiers des charges, spécifications, documents de conception, manuels et autres documents techniques), et qui puissent être mis en œuvre tout au long de la vie du projet. Le contrôle de la qualité rejoint ici tout naturellement la gestion de projets.

4,7 Méthodes de gestion de projets. – A côté des problèmes purement techniques, la réalisation de logiciel soulève de nombreuses difficultés quant à la gestion des projets. Dans une certaine mesure, ces difficultés sont les mêmes que dans les autres applications de l'ingénierie ; mais les particularités du logiciel et la complexité des systèmes que l'on réalise dans ce domaine ont rendu nécessaire la mise au point de méthodes spécifiques.

Les travaux sur la gestion des projets logiciels ont porté par exemple sur l'organisation des équipes : comment éviter le phénomène souvent observé selon lequel, au-delà d'une certaine taille de l'équipe, les problèmes de communication l'emportent sur les problèmes techniques ? Certaines méthodes d'organisation très strictes ont été préconisées en réponse à ces difficultés, comme la méthode des « équipes à programmeur en chef » d'IBM [I.b. 2].

Brooks, l'architecte principal du système OS 360, résume ainsi, dans son livre intitulé *Le Mythe de l'Homme-Mois* [I.b. 8], l'insuffisance des recettes simples pour résoudre les problèmes de la gestion de projets : « Si l'on ajoute du personnel à un projet en retard, on ne fait que le retarder encore. »

Une autre question liée à la gestion des projets est celle de la maîtrise des coûts et des délais. Il est essentiel dans ce domaine de pouvoir effectuer des estimations à l'avance ; c'est ici qu'interviennent les modèles.

5 Mesures et modèles

L'une des idées essentielles de la science moderne est que l'on ne connaît bien que ce que l'on sait mesurer. Le génie logiciel ayant pour ambition d'asseoir la production de logiciel sur des bases scientifiques, il est naturel que de nombreux spécialistes se soient demandé si l'on pouvait appliquer à ce domaine des techniques quantitatives. Des méthodes de modélisation et de mesure ont ainsi été proposées ; elles sont encore loin de constituer ce que certains auteurs ont prématurément appelé une « physique du logiciel » mais contiennent des éléments qui peuvent être utiles au praticien.

Les modèles applicables au logiciel se répartissent en deux catégories principales : modèles de coût, modèles de fiabilité. Les premiers permettent d'évaluer a priori les dépenses liées à un projet, les seconds d'estimer le taux d'erreurs dans un système.

5,1 Modèles de coût.

5,11 Forme générale. – Les modèles de coût ont été proposés en grand nombre. Beaucoup d'entre eux se présentent, au moins dans leur version la plus simple, sous la forme :

$$E = a\, I^b$$

où a et b sont des constantes ; une telle formule donne l'effort nominal E nécessaire à la construction du logiciel, exprimé en hommes-mois, en fonction du nombre de milliers d'instructions du programme final I.

Ainsi, selon le modèle Cocomo de Boehm [I.b. 5 et 7], qui donne $a = 2{,}8$ et $b = 1{,}2$ pour un logiciel intégré à un système complexe, l'effort total nécessaire à la construction d'un produit dont le code comporte 20 000 lignes est :

$$2{,}8 \times 20^{1{,}20} = 102 \text{ hommes-mois} \approx 9 \text{ hommes-années}$$

étant entendu qu'il s'agit de l'effort total, incluant toutes les phases de construction (spécification, etc.) et la rédaction de la documentation ; on se place dans l'hypothèse d'un logiciel réalisé dans des conditions industrielles.

Quelques précisions sont nécessaires pour bien comprendre l'application d'une formule de ce genre (obtenue par interpolation à partir de diverses bases de données contenant des informations collectées sur des projets industriels).

Il est bien connu que l'homme-mois est une unité douteuse [I.b. 8] ; pour prendre un cas extrême, 365 personnes ne font pas en un jour le travail d'une en un an. Il convient donc d'interpréter le résultat E avec précaution. Cocomo (comme certains autres modèles) permet d'ailleurs d'estimer non seulement l'effort total E mais aussi le **délai nominal** donné, selon ce modèle (et toujours pour la catégorie des logiciels intégrés à des systèmes complexes), par la formule :

$$D = 2{,}5\, E^{0{,}32}$$

Pour l'exemple précédent, cela donne 11,5 mois (en l'absence des multiplicateurs de l'effort mentionnés plus loin). Par division, on obtient la taille moyenne nominale T de l'équipe de développement, 8,5 personnes environ dans l'exemple choisi.

5,12 Variations. – Le terme *nominal* appliqué aux résultats précédents (effort E, délai D, taille de l'équipe T) signifie que, pour Boehm, il existe pour tout produit logiciel une série de valeurs optimales, celles que fournissent les formules du modèle de base. Que se passe-t-il dans le cas bien réel où le responsable du projet souhaite soit aller plus vite (en mettant plus de monde au travail), soit diminuer la taille de l'équipe (en acceptant de retarder la fin du projet) ?

Comme tout gestionnaire de projets le sait d'expérience, ce genre d'aménagement n'est pas gratuit en termes d'effort total et ne peut de toute façon s'opérer que dans certaines limites. Cocomo fournit des formules pour réviser en conséquence les estimations précédentes. On notera que la formule correspondant au premier cas (diminution du délai) ne s'applique qu'à une plage de délais allant de 75 % à 100 % du délai nominal D : pour Boehm, il est vain d'espérer gagner plus de 25 % sur D, quelles que soient les ressources supplémentaires en personnel dont on dispose. C'est le théorème de Brooks cité plus haut qui réapparaît sous une forme plus précise.

5,13 Discussion. – L'examen d'un modèle de ce type appelle plusieurs commentaires.

Il faut noter tout d'abord le caractère assez fruste de la mesure que constitue L, nombre d'instructions du programme (en milliers). La définition précise est **nombre de kilolignes d'instructions-sources livrées** ; en d'autres termes, on ne considère que les instructions du programme-source, écrit dans un certain langage dont le niveau d'abstraction influe inévitablement sur cette mesure. Le mot *instruction* couvre aussi bien les déclarations que les instructions exécutables, mais exclut les commentaires. Enfin seules sont prises en compte les instructions *livrées* : les programmes de test ou les outils de développement n'entrent dans le calcul que s'ils font partie du produit livré au client ; un appel de sous-programme compte pour une instruction, mais les instructions du sous-programme lui-même ne sont pas comptabilisées s'il appartient à une bibliothèque d'intérêt général antérieure au projet.

Cette mesure prête le flanc à la critique et, de fait, nombreux sont les auteurs qui ont proposé des critères de mesure plus fins. Certains de ces critères cherchent ainsi à prendre en compte la complexité du graphe de contrôle des programmes. Mais il faut bien admettre que les mesures plus ambitieuses ne donnent pas, dans les études qui ont été menées sur la corrélation entre les divers critères proposés et l'effort réel observé expérimentalement sur les logiciels correspondants, des résultats très supérieurs à celui de la simple mesure I, qui présente l'avantage de pouvoir être calculée facilement par un simple programme de comptage.

L'application de modèles de ce type suppose, par ailleurs, que l'on connaît la taille du programme final, qui n'est pas toujours facile à déterminer à l'avance, même pour un responsable de projet expérimenté.

Une autre objection évidente est que les formules universelles comme celle de Cocomo ne permettent pas de rendre compte des conditions particulières à chaque projet : il est certainement irréaliste de traiter de la même façon un système de pilotage automatique d'avion, réalisé sur un microprocesseur à mémoire strictement limitée, par une équipe qui connaît mal le matériel et le langage, et un programme de comptabilité écrit en Cobol sur *IBM 370* par une société de service rompue à ce genre d'exercice (même si le nombre d'instructions final des deux programmes est le même). Mais Cocomo permet de prendre en compte cette remarque : d'une part, le modèle comporte non pas une mais trois formules de base (celle que nous avons citée pour les logiciels intégrés à un système complexe, une autre pour les logiciels autonomes et moins critiques et la troisième pour les cas intermédiaires) ; mais surtout des multiplicateurs associés aux particularités de chaque projet permettent d'ajuster les estimations de base fournies par ces formules.

La figure **5** (p. 12) donne les plages de variation pour chacun des facteurs de coût.

Ainsi, selon ce tableau, un multiplicateur de 1,34 est associé au facteur « expérience de la machine virtuelle » ; cette valeur est le

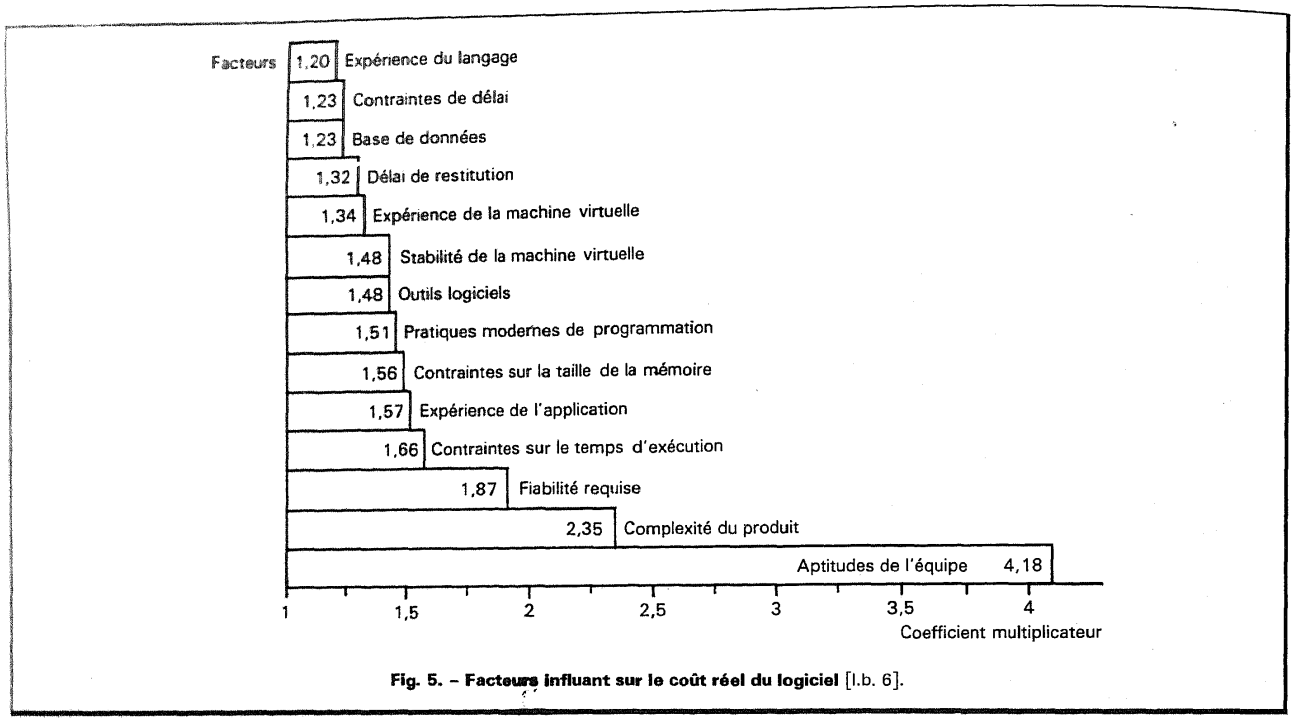

Fig. 5. – Facteurs influant sur le coût réel du logiciel [I.b. 6].

rapport entre les valeurs maximale et minimale de la plage de variation de l'effort réel selon que l'équipe de développement connaît particulièrement bien ou particulièrement mal la machine virtuelle (terme sous lequel Boehm regroupe l'ordinateur, le système d'exploitation, le compilateur, et les autres outils qui constituent ensemble la « machine » vue par les membres du projet).

L'examen de ces multiplicateurs (par exemple du dernier, relatif à la compétence de l'équipe, qui peut aller jusqu'à 4,18.) peut évidemment laisser sceptique sur l'applicabilité réelle de modèles tels que Cocomo ; à quoi sert un modèle qui donne, d'un côté, des formules mathématiques précises et de l'autre des coefficients de variation aussi importants, se rapportant à des facteurs dont l'analyse est inévitablement subjective ?

Ce type de modèle est en effet difficile à défendre comme source de prévisions *absolues*. Il serait illusoire (même si Cocomo donne quelques indications générales sur ce point) de chercher à savoir de but en blanc si l'*aptitude de l'équipe* doit être évaluée à 2,3 ou 2,7. Mais les choses changent si l'on considère plutôt des valeurs relatives. Une société qui a sérieusement entrepris de collecter des mesures sur ses projets pendant un certain temps peut comparer les valeurs observées avec les prédictions du modèle, et donc *calibrer* celui-ci en fonction de ses caractéristiques propres. Il devient alors beaucoup plus facile d'estimer les paramètres non pas dans l'absolu mais par comparaison avec les valeurs retenues pour des projets précédents, sur lesquels le modèle a été essayé et affiné.

C'est de cette façon qu'il faut comprendre un modèle tel que celui de Cocomo : non pas un oracle qui donne une réponse infaillible à une question précise, mais un élément d'estimation qui suppose une politique de collecte de mesures et la mise en place d'une base de données sur les coûts de développement de logiciel dans l'entreprise.

Notons pour terminer sur ce sujet que nous avons seulement donné un aperçu du modèle Cocomo, qui inclut aussi des techniques de prévision plus fines, dont on trouvera la description dans [I.b. 5 et 7].

5,2 Modèles de fiabilité.

Les modèles mathématiques de fiabilité [I.b. 38], transposition au logiciel de la théorie classique de la fiabilité des systèmes matériels, permettent d'obtenir un certain nombre d'informations sur les erreurs qui subsistent dans un programme à partir d'informations sur le projet (particulièrement sur les erreurs précédemment détectées), de données de référence et d'hypothèses statistiques sur la répartition des erreurs.

Implicitement, ces modèles traitent donc les défaillances des systèmes logiciels de la même façon que celles des dispositifs matériels. On peut trouver cette assimilation hasardeuse, puisque les défaillances des systèmes matériels proviennent non seulement de fautes de conception ou de réalisation mais aussi de la simple usure physique, phénomène qui bien sûr n'existe pas en logiciel.

Dans la suite, nous appelons *erreur* une anomalie présente dans le logiciel, et *défaillance* un cas observé de mauvais fonctionnement du logiciel. A la différence des défaillances matérielles, les défaillances d'un logiciel sont toujours dues à des erreurs.

Comme tout modèle mathématique, un modèle de fiabilité calcule un certain nombre de résultats en fonction d'un certain nombre de paramètres d'entrée. Ces paramètres d'entrée peuvent être :

- le temps (selon les modèles, on prend en compte soit le temps calendaire, soit le temps pendant lequel le programme s'exécute) ;
- les taux de défaillances observés en phase de mise au point ou (si l'on est déjà en phase d'exploitation) jusqu'à l'instant courant ;
- des résultats antérieurs relatifs à d'autres projets comparables ;
- des hypothèses statistiques (par exemple, l'un des modèles les plus connus, celui de Musa [I.b. 32], suppose que les erreurs sont uniformément distribuées parmi les instructions du programme) ;
- des paramètres spécifiques du projet.

En sortie, on obtient par exemple :

- une estimation du nombre d'erreurs résiduelles ;
- le taux prévisible d'erreurs $z(t)$ (où t est le temps) ;
- le temps moyen jusqu'à la prochaine défaillance (MTTF : *Mean Time To Failure*) :

$$\text{MTTF}(t) = \frac{1}{z(t)}$$

Ce dernier paramètre est choisi de préférence au temps moyen entre défaillances (MTBF), qui n'a pas de sens si l'on suppose que l'exploitation s'arrête après toute défaillance afin de permettre la

correction de l'erreur, ou des erreurs, qui l'ont causée. Le modèle de Musa, qui utilise le temps d'exécution τ, a pour formule de base :

$$\text{MTTF}(\tau) = \frac{1}{f K N_0} e^{f K \tau}$$

avec f fréquence moyenne d'exécution des instructions du programme (fréquence moyenne d'exécution des instructions divisée par le nombre d'instructions du programme),
K constante (facteur de détection des erreurs),
N_0 nombre d'erreurs restant au début de la phase considérée.

Il est nécessaire de préciser ici (comme pour le modèle de coût Cocomo) que les modèles effectivement utilisés en pratique sont plus fins que ne laisserait supposer cette brève présentation, et que seule une politique de mesure systématique, aboutissant à une base de données propre à l'entreprise, permet de calibrer convenablement les paramètres.

Une technique amusante se rapproche des modèles de fiabilité : celle du *bebugging*, que l'on peut appeler en français « pêche aux bogues » (la bogue, avant d'être le terme recommandé par le Premier Ministre pour traduire l'anglais *bug*, désignait une variété de poisson). Supposez que vous ayez à compter le nombre de poissons dans un lac : vous aurez peut-être l'idée d'y introduire m poissons bagués et, après un temps suffisant, de pêcher n poissons, dont p seront bagués. Le rapport $\frac{m \times (n-p)}{p}$ vous donnera une idée du nombre total de poissons dans le lac.

Si maintenant, au lieu de poissons, vous voulez compter le nombre d'erreurs dans un programme, la méthode correspondante consiste à introduire m erreurs volontaires ; si le processus de mise au point est poursuivi normalement, la détection de n erreurs, dont p sont des erreurs artificiellement introduites, permet d'estimer le nombre d'erreurs naturelles. Une telle méthode n'a de sens que si l'on dispose de données expérimentales fiables sur la nature et le taux de répartition des erreurs ordinairement commises par les programmeurs, afin de garantir que les erreurs introduites leur sont bien statistiquement comparables (les poissons bagués doivent avoir la même probabilité que les poissons du lac de passer à travers les mailles du filet).

Pour intéressantes qu'apparaissent ces différentes techniques, on peut cependant ressentir un certain malaise vis-à-vis de la conception qu'elles reflètent des erreurs en programmation, et plus généralement de la programmation elle-même. S'il apparaît naturel de traiter par des méthodes statistiques les défaillances de systèmes matériels, l'idée selon laquelle la présence d'erreurs dans un logiciel est un phénomène qui peut et doit être mesuré, donc finalement un phénomène normal, peut choquer.

Les erreurs dans les programmes sont des erreurs de raisonnement, et l'analyse quantitative a ses limites dans ce domaine. En phase de mise au point, le nombre d'erreurs restantes n'est pas nécessairement un bon indicateur du travail qui reste à faire (une seule erreur bien corsée peut en valoir beaucoup de vénielles) ; en phase d'exploitation, le passager d'un avion ne sera pas nécessairement réconforté, surtout s'il est lui-même programmeur, par l'annonce que d'après les prédictions du modèle il ne doit rester « que » 0,005 % d'erreurs résiduelles dans le logiciel d'atterrissage automatique.

6 Langages

Les langages de programmation ont pendant longtemps occupé une place sans doute exagérée dans les discussions sur le logiciel, mais l'excès inverse serait tout aussi regrettable : on ne peut négliger l'importance de l'outil d'expression fondamental pour l'écriture des programmes et la description des données. Il convient d'ailleurs d'inclure dans cette discussion les notations dont se servent les informaticiens aux autres étapes du cycle de vie, particulièrement en amont : les langages de spécification et de conception.

La situation en matière de langages de programmation a pendant longtemps été caractérisée par une coupure presque complète entre la réalité industrielle, dominée par les langages des années cinquante et soixante (langages d'assemblage, FORTRAN, BASIC, COBOL, PL/I), et les recherches universitaires qui tournaient autour des dérivés d'Algol 60, d'une part, et de ceux de Lisp, d'autre part. Depuis 1975 environ, cette division a été remise en cause par la percée industrielle, encore limitée mais indéniable, de plusieurs nouveaux langages venus d'horizons divers : **Pascal**, issu de la recherche universitaire et d'abord destiné à l'enseignement, qui a atteint une diffusion importante en mini- et en micro-informatique ; **C**, développé aux Laboratoires Bell (AT & T) qui gagne du terrain en même temps que le système Unix ; **APL**, ancien dans sa conception (1960), mais qui n'est utilisable en vraie grandeur que depuis quelques années.

Un langage plus récent a fait couler beaucoup d'encre : Ada, développé par une équipe de Bull en réponse à un appel d'offres du Ministère Américain de la Défense (DoD). Ada est sans doute le premier langage d'ambition industrielle à avoir été conçu selon des critères qui ressortissent incontestablement du génie logiciel :

- modularité, avec la notion de paquetage ;
- compilation séparée, importante pour la programmation en équipe et pour la gestion des configurations ;

– généricité, qui permet d'utiliser une même structure (procédure, type) dans des contextes différents et favorise donc les facteurs de réutilisabilité et de compatibilité ;

– sévérité des contrôles de types, dans la tradition de Pascal, qui permet d'améliorer la validité et la fiabilité (les erreurs de conception se traduisent souvent, au moment de la mise en œuvre, par des incompatibilités de types, que des langages laxistes toléreront abusivement). Destiné tout particulièrement à la programmation des applications temps réel, Ada permet par ailleurs de construire des systèmes à tâches parallèles, ce qui est une nécessité dans ce domaine.

Dans l'attente de compilateurs qui soient à la fois facilement accessibles, de qualité industrielle et adaptés aux principaux matériels du commerce, Ada reste un pari. Un échec de ce pari aurait sans doute une influence considérable sur l'évolution du génie logiciel.

Au moment où sont rédigées ces lignes, il existe des compilateurs répondant à l'un de ces critères, parfois à deux, mais non aux trois (voir la table fournie régulièrement par C. R. Morgan sous le titre *Matrix of Ada Language Implementation* dans le bulletin *Ada Letters* du groupe Ada de l'ACM ; dernier état en date dans le volume V, n° 1, juillet-août 1985).

L'apparition d'Ada aura de toute façon marqué une étape dans l'évolution des langages. On peut dire qu'Ada constitue le couronnement de la lignée des langages de programmation classiques, dans la filiation FORTRAN - Algol 60 - Algol W - PL/I - Algol 68 - Pascal (filiation certes illégitime à certaines étapes) ; l'effort consacré à la conception du dernier-né semble avoir presque complètement fermé, au moins pour un temps, toute recherche dans cette famille de langages, avec quelques exceptions comme la création du langage Modula 2 par Wirth, l'auteur de Pascal.

Cela ne signifie pas, bien sûr, que les recherches aient cessé sur le thème des langages en général ; on peut noter que ce domaine connaît au contraire depuis quelques années un regain d'intérêt. Les travaux actuels s'orientent pour la plus grande part vers des langages s'écartant notablement de la famille citée précédemment ; il s'agit en général de langages de plus haut niveau, moins impératifs que les langages courants. Comme on l'a souvent fait remarquer, les langages classiques possèdent maintes caractéristiques directement inspirées de l'architecture typique des ordinateurs actuels :

– faible niveau d'*abstraction* des objets manipulés (malgré les primitives de construction de nouveaux types offertes par la plupart des langages depuis Algol W) ;

– importance de la notion de *variable* et donc d'effet de bord (modification dynamique et répétée de la valeur du même objet) ;

– nécessité de raisonner en termes de *commandes* plutôt que de conditions logiques, ce qui accroît la distance de la spécification au programme ;

– obligation pour le programmeur d'indiquer précisément l'*ordre* séquentiel du déroulement des instructions ;

– description « procédurale » du traitement, fondée sur les *actions* à effectuer et non sur la vie individuelle des objets du système.

De nombreux langages proposés depuis quelques années cherchent à s'affranchir de ces différents défauts :

– les langages **ensemblistes** tels que SETL [I.b. 11] offrent les ensembles, les listes, etc. comme structures de données primitives ;

– les langages **fonctionnels**, nouvelles versions de Lisp [I.b. 40] ou créations plus récentes comme FP [I.b. 1], SASL [I.b. 42] et KRC [I.b. 43], s'appuient sur la notion de fonction de préférence à celle de variable ;

– les langages **logiques**, en particulier Prolog [I.b. 9], permettent d'assimiler les programmes à des clauses de logique mathématique (calcul des prédicats du premier ordre) exécutables, en rapprochant leur forme de celle de spécifications formelles ;

– les langages dits **à flot de données** ou à affectation unique tels que VAL, LAU, SISAL ou Lucid, permettent de ne pas préciser l'ordre d'exécution des instructions lorsqu'il n'a pas d'influence sur la sémantique finale du programme, et d'autres langages (fondés sur les concepts des processus séquentiels coopérants de Hoare [I.b. 16] ou ceux du calcul des systèmes communicants de Milner [I.b. 30]) s'orientent vers le parallélisme véritable ;

– enfin, les langages dits **à objets**, dérivés de Simula 67 [I.b. 3] ou de Smalltalk [I.b. 14], comme C++ [I.b. 41], Objective C [I.b. 10] ou Eiffel [I.b. 26], permettent de décrire un système non pas comme une suite d'instructions à exécuter mais comme un ensemble d'objets dynamiques, autonomes et communicants.

Il reste à savoir ce qui, dans ce bouillonnement, va émerger. Les langages fonctionnels et logiques jouent un rôle essentiel dans les recherches sur l'intelligence artificielle ; leur applicabilité au génie logiciel reste à démontrer. Ils sont parfois proposés comme langages pour le prototypage rapide ; SETL a été utilisé avec succès dans ce domaine, ayant servi de langage pour le premier compilateur Ada officiellement validé [I.b. 20]. Les perspectives offertes par les langages à objets nous semblent plus prometteuses ; leur rôle peut à notre sens être fondamental pour résoudre quelques-uns des problèmes cruciaux du génie logiciel, en particulier la réutilisabilité et l'extensibilité. Ce point de vue est développé dans les articles [I.b. 23] et [I.b. 26].

7 Outils

Dans toute branche de l'ingénierie, les outils communément disponibles jouent un rôle considérable. Il ne s'agit pas seulement de leur utilité concrète et quotidienne : à plus long terme, il se forme autour des outils, chez les praticiens du domaine, une culture aussi importante que la culture proprement scientifique. Devient homme du métier celui qui a non seulement maîtrisé les concepts et les connaissances jugés nécessaires à un certain moment de l'évolution des techniques, mais aussi appris à manier et à dominer les outils qui, à ce même moment, constituent le support ordinaire des professionnels.

En logiciel, l'outil est d'abord matériel, devenu depuis quelques années de moins en moins lointain grâce à la diffusion de la micro-informatique et à la décentralisation des fonctions sur les grands systèmes ; nous allons y revenir. Mais les outils qui nous intéressent ici au premier chef sont les **outils logiciels** : programmes destinés à favoriser le développement, la modification et la diffusion d'autres programmes.

Un catalogue des différents types d'outils de génie logiciel dépasserait les limites de cette introduction [l.b. 34]. Nous nous bornerons à citer quelques-uns des domaines les plus prometteurs.

7,1 Outils d'aide à la construction des programmes.

Parmi les outils qui interviennent lors des phases de construction, certains sont classiques, comme les éditeurs de textes sur les systèmes interactifs. Parmi les développements plus originaux, on peut citer :

– les outils d'aide à la conception, en général associés à un langage de conception ou pseudo-code (également appelé PDL : *Program Design Language*), et plus généralement tous les outils (« préprocesseurs » tels que le processeur de macros intégré au langage C [l.b. 19]) grâce auxquels les programmeurs peuvent faire semblant de croire qu'ils disposent de langages de programmation moins primitifs qu'ils ne le sont en réalité ;

– les générateurs de programmes ou « langages de quatrième génération » [l.b.17], progiciels paramétrables permettant de produire, dans un domaine d'application bien délimité, des programmes adaptés aux besoins de chaque utilisateur, à qui il suffira de préciser les paramètres spécifiques de son application (de tels outils existent en informatique de gestion où ils commencent à concurrencer sérieusement COBOL pour les applications de nature répétitive et bien connue, mais aussi dans d'autres domaines où l'on peut traiter les applications les plus courantes dans un cadre normalisé, comme l'analyse syntaxique) ;

– les éditeurs structurels [l.b. 13, 15, 25 et 27], qui permettent de créer, de manipuler et de modifier des textes structurés (programmes, spécifications, etc.) en fonction de leur structure, et non pas comme de simples suites de caractères ; ces outils commencent aujourd'hui à sortir des milieux de la recherche pour gagner l'industrie.

7,2 Outils de gestion de projets et de configurations.

L'ordinateur permet aujourd'hui de gérer bien des tâches humaines ; le développement de logiciel ne devrait pas faire exception. De nombreux outils ont été proposés dans ce domaine, allant de simples outils ponctuels (permettant par exemple la prise en compte et le contrôle des emplois du temps individuels dans une équipe) à de véritables systèmes intégrés de gestion de projets, regroupant autour d'une base de données centrale l'ensemble des données : le chef de projet dispose ainsi à chaque instant du tableau de pilotage complet, comprenant des informations techniques et d'autres relatives aux coûts, aux tâches, aux délais, à l'ordonnancement. Il est clair que l'utilisation de tels systèmes n'a de sens que si elle s'accompagne de méthodes rigoureuses pour la gestion et le suivi de projet.

Une activité complémentaire de la gestion de projets est la gestion de configurations, dont le but est de contrôler l'évolution des différents composants d'un logiciel (programmes, spécifications, documents de conception, jeux d'essai, données, documents divers) tout au cours du projet.

On rencontre encore rarement dans la pratique des systèmes intégrés de gestion de projets et de configurations, bâtis autour d'une base de données complète, mais il existe des outils séparés qui n'en rendent pas moins d'appréciables services. Dans le domaine de la gestion de configurations, en particulier, de nombreux environnements offrent des outils inspirés de deux produits disponibles sous Unix : SCCS (*Source Code Control System*), qui permet d'archiver les versions successives d'un document (non pas nécessairement un programme malgré le nom du produit), et Make qui permet de reconstruire un logiciel en exécutant automatiquement les actions de recréation rendues nécessaires par l'évolution de certains composants (par exemple un module-source qui a été modifié postérieurement à la création du module-objet correspondant doit être recompilé).

Plusieurs équipes développent aujourd'hui des outils de gestion de projets et de configurations qui s'appuient sur des systèmes de gestion de bases de données. On trouvera dans l'article [l.b. 28] la description d'un système qui s'appuie sur un modèle binaire des relations entre composants logiciels et sur la notion de contrainte sémantique.

Toujours dans le domaine des outils de gestion, nous avons cité précédemment (§ 5,1) les modèles de coût ; des outils logiciels sont associés à certains de ces modèles (comme *Wicomo* de l'Institut Wang pour Cocomo, *Price-S* de RCA pour le modèle du même nom, etc.).

7,3 Outils de contrôle et de validation.

Comme nous l'avons vu au paragraphe 4,6, le test dynamique, c'est-à-dire l'exécution du système dans des situations prédéterminées et la comparaison des réponses obtenues avec un scénario-type établi à l'avance, reste la méthode de validation la plus répandue malgré ses limitations évidentes.

Il est d'autant plus surprenant de constater que peu d'outils permettant d'automatiser le processus de test existent en pratique. Des générateurs automatiques de jeux d'essai ont été développés par des chercheurs , mais sont rarement applicables en pratique à l'exception de quelques domaines bien définis comme le test des compilateurs, où des batteries de tests normalisées existent pour certains langages comme Pascal [l.b. 39] et Ada (cf. article *Conformité aux normes des langages de programmation* de ce traité).

Des outils de validation de portée théorique limitée mais qui peuvent rendre de grands services sont disponibles dès aujourd'hui. Il s'agit des **analyseurs statiques**, qui permettent d'examiner un texte de programme (indépendamment de toute donnée et donc de toute exécution) pour y détecter des anomalies possibles et fournir

[l.b.*n*] renvoie à la référence *n* de l'Index bibliographique placé en fin d'article.

des éléments de documentation automatisée, et des **analyseurs dynamiques** (ou moniteurs de test) qui permettent de contrôler l'exécution d'un programme sur des jeux d'essai et d'en déduire un certain nombre de résultats et de mesures (taux de couverture). Un exemple de produit commercial combinant l'analyse statique et l'analyse dynamique sur des programmes FORTRAN est RXVP de General Research, qui montre bien les bénéfices pratiques que peuvent apporter ces techniques. Parmi les autres outils de ce type, on notera l'analyseur *lint*, applicable au langage C et disponible sous Unix, l'outil *Auditor* de la société Softool et deux outils français : le *Qualimètre C* de la société IGL et, pour un but légèrement différent (la mesure de complexité des programmes), les *logiscopes* de la société Vérilog.

7,4 Au-delà des outils : les environnements.

Quand les outils deviennent nombreux et complexes, leurs qualités individuelles ne suffisent plus : encore faut-il pouvoir les employer les uns avec les autres. Ce problème de compatibilité a souvent été mal résolu par les systèmes d'exploitation classiques : le compilateur, l'éditeur de textes, l'interprète de commandes s'y présentent en général comme des entités distinctes, construites et utilisées selon des conventions différentes, voire contradictoires, qui provoquent chez l'utilisateur gêne et parfois danger (comme sur tel système où deux outils permettent de copier des fichiers : l'un exige que l'on nomme d'abord la source, puis la cible ; l'autre attend l'inverse).

La notion d'**environnement logiciel intégré** est née d'une réaction contre les multiples incompatibilités qui polluent les systèmes classiques.

La plupart des environnements actuels s'inspirent au moins en partie d'Unix. Unix était au départ un système d'exploitation plus qu'un environnement au sens défini ci-dessus ; il serait en outre exagéré de dire que ce système est complètement intégré. Mais les outils disponibles sous Unix bénéficient d'un niveau de compatibilité encore à peu près unique dans la confrérie des systèmes. Ce remarquable succès a été obtenu grâce à la simplicité de la conception de base du système et à la combinaison d'un petit nombre d'idées fructueuses :

- la structure des fichiers est uniforme : les fichiers normaux sont de simples suites de caractères, accessibles soit en séquence, soit par leur index ;
- l'interactivité est ramenée au même schéma : on considère le terminal comme un fichier, qui peut être utilisé comme entrée ou comme sortie par tout programme apte à lire ou à écrire sur des fichiers ;
- un programme simple sous Unix est en général un *filtre* qui prend en entrée un fichier du type précédent et produit en sortie un fichier du même type ;
- deux ou plusieurs programmes de ce type peuvent donc être composés en soudant la sortie de l'un à l'entrée de l'autre : c'est la notion de **tuyau** ; la notation

refer | *tbl* | *eqn* | *troff*

désigne un tuyau de cette sorte, obtenu en composant les outils Unix de traitement de texte : *refer* (traitement des références bibliographiques), *tbl* (traitement des tables), *eqn* (traitement des textes de type mathématique), *troff* (photocomposition). La sortie de chacun d'entre eux est utilisée comme entrée du suivant.

La simplicité et l'élégance de ces structures de base ont favorisé l'éclosion sous Unix d'une quantité considérable d'outils qui (à la différence de ce qui se passe sur d'autres systèmes) sont souvent deux à deux compatibles. Le tableau III indique quelques-uns de ces outils (non nécessairement tous présents dans toutes les versions commerciales du produit).

Pour les nombreux spécialistes qui cherchent actuellement à développer des environnements logiciels, Unix et ses contemporains (tel Interlisp, dans lequel la structure de référence commune est, plutôt que le fichier, la liste au sens du langage Lisp) ne méritent pas véritablement le qualificatif « intégré ». De Drouas et Nerson [I.b. 44] les appellent environnements de deuxième génération ; ces environnements évitent les incompatibilités inhérentes à leurs prédécesseurs, mais sont essentiellement des boîtes à outils. Pour certains auteurs, seuls peuvent être dits intégrés les environnements dans lesquels les outils sont liés par un fil conducteur plus solide qu'un simple ensemble de conventions homogènes : par exemple un langage (comme dans les environnements Apse définis autour d'Ada par le DoD) ou une méthode intégrée de développement.

Cette vision selon laquelle les environnements doivent être pour ainsi dire totalitaires, bâtis autour d'une idée centrale très forte, ne fait cependant pas l'unanimité ; les ateliers boîtes à outils de type Unix, moins intégrés mais plus souples et ouverts, ont fait leurs preuves et conservent de chauds partisans.

Une chose est certaine pour l'avenir : les nouveaux ateliers intégrés prendront de plus en plus en compte les progrès du matériel. L'élément important ici est le développement des **postes de travail** qui, grâce à la puissance croissante des microprocesseurs et aux progrès des réseaux (locaux et à distance), permettent de concilier les avantages des micro-ordinateurs et ceux des grands systèmes.

Un aspect remarquable de ces nouveaux postes de travail est la qualité de l'interface, utilisant des écrans de haute résolution et des

Tableau III. – Quelques outils sous Unix.

Application	Outils
Compilateurs et interprètes	C, Pascal, FORTRAN 77, Lisp, Prolog, FP ; *yacc* et *lex* (construction de compilateurs).
Courrier électronique	mail, *UUCP* (réseau).
Utilitaires de base	coquille (interprète de commandes) ; gestion de fichiers (copie, déplacement, etc.) ; éditeurs de texte (*ed, vi, emacs*) ; création de processus parallèles.
Traitement de textes	Formatage de textes et photocomposition (*nroff/troff*) ; description de figures (*pic*) et de tables (*tbl*) ; textes mathématiques (*eqn*) ; bibliographies (*refer*) ; contrôle de l'orthographe et du style (*spell, diction, style*).
Traitement de données	Recherche de modèles dans un texte (*grep*) ; comparaison de fichiers (*diff*) ; tri (*sort*) ; transformation de fichiers (*sed, awk*).
Développement de logiciel	Gestion de versions et de configurations (*sccs, make*) ; analyse statique (*lint*) ; mise au point de programmes (*adb/dbx*).

dispositifs d'entrée rapide (souris, tablette). En particulier, nombre d'entre eux (reprenant les idées introduites à l'origine par l'environnement Smalltalk de Xerox, bâti autour du langage de même nom) permettent de diviser un écran en plusieurs zones rectangulaires ou *fenêtres* ; si le logiciel est à la hauteur, c'est-à-dire s'il permet de gérer plusieurs processus actifs en parallèle, affectés chacun à une fenêtre (les possibilités d'Unix en matière de création de processus concurrents font que ce système est bien adapté à des situations de ce genre), le confort des utilisateurs est singulièrement amélioré : ils peuvent en effet poursuivre simultanément plusieurs activités, chacune d'entre elles restant visible grâce à la présence de sa fenêtre sur l'écran.

Ces possibilités, que le lancement du *MacIntosh* par Apple ont fait connaître à un large public, sont particulièrement intéressantes pour les informaticiens, que leur tâche force souvent à être tout à la fois au four, au moulin et à la rivière. Dans une situation typique, par exemple celle de la mise au point d'un module, le programmeur peut être conduit à alterner constamment entre le compilateur, l'éditeur et le langage de commande ; même si l'on peut espérer qu'à l'avenir ces outils seront mieux intégrés qu'il n'est aujourd'hui de règle, la nécessité subsistera de pouvoir passer rapidement d'une vue à une autre. Les systèmes à multi-fenêtrage fournissent ici un élément de solution particulièrement intéressant. Cela n'a de sens, bien entendu, que si la définition de l'écran est suffisante, et (répétons-le) si le système d'exploitation fournit le substrat logiciel adéquat en matière de gestion des processus concurrents.

Ce n'est pas par hasard que nous avons choisi de mentionner les évolutions du matériel au terme de cette brève visite guidée : malgré toute leur superbe, les spécialistes de logiciel sont bien obligés de reconnaître que, pour une large part, l'informatique reste traînée par le matériel (*hardware-driven*), selon le mot de Wirth. Sans matériel, il n'y aurait pas de logiciel, c'est une platitude ; mais si le matériel n'avait pas évolué de façon aussi phénoménale, le génie logiciel ne serait peut-être pas aussi nécessaire, et serait de toute façon beaucoup moins intéressant.

8 Vers une discipline scientifique

On nous permettra, au terme de cette promenade rapide, un point de vue plus personnel. Pour l'auteur, le défi majeur auquel les informaticiens sont aujourd'hui confrontés vient de trois des facteurs de la qualité cités précédemment (§ 2,3) : la *validité*, l'*extensibilité*, la *réutilisabilité*. Le logiciel fabriqué actuellement n'est pas assez sûr ; il est trop difficile à modifier ; il est trop spécifique.

La plupart des autres problèmes du génie logiciel peuvent être rattachés, au moins en partie, à ces trois-là. La question de la maintenance, par exemple, ne sera pas résolue par des recherches visant à améliorer les méthodes de maintenance elle-même, mais par des progrès dans les méthodes de construction, permettant de produire des systèmes plus corrects et plus souples. Les coûts du logiciel, pour prendre un autre exemple, ne peuvent diminuer qu'au prix d'une industrialisation véritable, c'est-à-dire de la mise au point de composants normalisés, réutilisables et combinables. Or en logiciel, nous l'avons déjà signalé, on repart trop souvent de zéro : à l'instant où vous lisez ces lignes, combien de personnes de par le monde sont-elles en train d'écrire, pour la mille-et-unième fois, un programme de tri ou de recherche en table ?

Il nous semble donc que les problèmes clés du génie logiciel (ceux dont la solution peut apporter des sauts quantiques, et non pas seulement des améliorations marginales) sont pour l'heure des problèmes techniques : problèmes de méthodes, de langages, d'outils.

Cette opinion, il est juste de le préciser, est loin d'être universelle ; pour toute une école, c'est du côté de la gestion des projets que le bât blesse. B. W. Boehm, par exemple, estime que plus de 50 % des problèmes du développement de logiciel sont des problèmes de gestion. Si l'on part de telles prémisses, il est clair que les solutions seront recherchées en priorité du côté de l'étude des pratiques industrielles, de la collecte de mesures ; si l'on met l'accent sur des méthodes, il s'agira de méthodes de gestion de projets ou d'organisation des équipes plutôt que de spécification formelle ou de conception par objets.

L'importance des problèmes de gestion apparaît clairement à quiconque a observé le déroulement d'un projet mais, dans l'état actuel du métier, ils nous semblent plus un symptôme qu'une cause première. Le génie logiciel n'a pas atteint un niveau suffisant pour que les aspects techniques puissent être considérés comme secondaires par rapport aux questions d'organisation. Il est à peu près

aussi fructueux d'espérer faire progresser la qualité du logiciel par l'analyse des méthodes de travail employées dans les entreprises qu'il l'eût été de fonder les progrès de l'épidémiologie, dix ans avant Pasteur, sur l'étude des pratiques hygiéniques en vigueur dans les hôpitaux.

On peut aussi reprocher à beaucoup de travaux sur l'organisation des équipes ou la gestion des projets de ne pas s'écarter du paradigme classique du développement de logiciel dans les grands projets industriels : le modèle du « toujours plus », qui suppose de grosses équipes, de grosses machines, de gros budgets. Or il existe d'autres paradigmes, qui misent plus sur le talent et la créativité que sur la masse et sur la force brute ; et si l'on considère les systèmes et les concepts qui ont « percé » depuis quelques années, qu'ils s'appellent Unix, Interlisp, Pascal, C, MacIntosh, Simula, Smalltalk ou Prolog, on constate immanquablement qu'ils procèdent de ces nouveaux paradigmes, et résultent du travail de petites équipes ou d'individus brillants plutôt que des méthodes qui permirent en leur temps l'édification des Pyramides ou celle de l'*OS 360*.

Où sont donc les progrès potentiels ? Ce tour d'horizon a fait ressortir quelques domaines particulièrement importants, dans lesquels il ne nous semble pas déraisonnable d'espérer des améliorations considérables si l'on s'en donne les moyens :

– la nécessité d'une démarche plus rigoureuse vis-à-vis de la construction de programmes, fondée sur les mathématiques et plus particulièrement sur la logique : c'est à ce prix que des améliorations significatives pourront être apportées à la validité des logiciels, à travers l'utilisation de spécifications formelles et de techniques de construction systématique ;

– la formation des informaticiens, qui apparaît essentielle en regard du point précédent (les méthodes formelles, on l'a vu, font appel à la culture mathématique) et dont le multiplicateur 4,2 attribué au facteur « aptitude de l'équipe » dans le modèle Cocomo, résultat d'une analyse de la réalité actuelle, fait cruellement ressortir l'importance ;

– l'utilisation de langages et d'outils modernes, en particulier dans le domaine de la conception et de la programmation par objets (meilleure solution connue, à notre sens, au problème de la réutilisabilité et à celui de l'extensibilité), des outils d'aide à la construction des programmes (générateurs de programmes et éditeurs structurels tout particulièrement), de la gestion de configurations ;

– l'utilisation des possibilités des postes de travail modernes comme composants essentiels d'environnements véritablement intégrés.

Il ne reste plus qu'à s'y mettre...

INDEX BIBLIOGRAPHIQUE

La liste ci-après comprend toutes les références citées dans le corps de l'article.

D'une façon générale, on consultera avec profit les principales revues consacrées au domaine par les sociétés savantes : l'IEEE (Institute of Electrical and Electronic Engineers) publie le mensuel *IEEE Transactions on Software Engineering* (à l'origine trimestriel) depuis 1975 et le bimestriel d'orientation plus magazine *IEEE Software* depuis 1984 ; l'ACM dispose depuis 1979, dans un domaine plus spécialisé, de la revue trimestrielle TOPLAS (*Transactions on Programming Languages and Systems*) et publie le bulletin *Software Engineering Notes*. En France, la revue TSI (*Technique et Science Informatiques*), publiée par l'AFCET (*Association Française pour la Cybernétique Économique et Technique*), contient fréquemment des articles sur le génie logiciel ; le groupe de travail « génie logiciel » de l'AFCET publie le bulletin *Bigre + Globule* en collaboration avec l'Institut IRISA de Rennes.

Le principal congrès international est *l'International Conference on Software Engineering*, organisé tous les dix-huit mois depuis 1975 (annuellement à partir de 1987), dont les actes sont publiés par l'IEEE. L'AFCET tient tous les deux ans depuis 1982 son *Colloque de Génie Logiciel* ; il sera complété les années impaires, à partir de 1987, par une conférence européenne (*European Software Engineering Conference*).

Références

1. BACKUS (J.). – *Can Programming be Liberated from the Von Neumann Style ? A Functional Style and its Algebra of Programs.* Communications of the ACM, vol. 21, n° 8, p. 613-41, August 1978.
2. BAKER (F.T.). – *Chief Programmer Team Management of Production Programming.* IBM Systems Journal, vol. 11, n° 1, 1972.
3. BIRTWISTLE (G.), DAHL (O.J.), MYRHAUG (B.) et NYGÅRD (K.). – *Simula Begin.* Student Literatur and Auerbach Publishers, 1973.
4. BOEHM (B.W.), BROWN (J.R.), McLEOD (G.), LIPOW (M.) et MERRIT (M.). – *Characteristics of Software Quality,* TRW Series of Software Technology, North-Holland Publishing Co., Amsterdam, 1978.
5. BOEHM (B.W.). – *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs (New Jersey), 1982.
6. BOEHM (B.W.). – *Les Facteurs de Coût du Logiciel.* TSI (Technique et Science Informatiques), vol. 1, n° 1, p. 1-20, janvier-février 1982. Traduction d'Emmanuel Girard.
7. BOEHM (B.W.). – *Software Engineering Economics.* IEEE Transactions on Software Engineering, vol. SE-10, n° 1, p. 4-21, January 1984.
8. BROOKS (F.P.). – *The Mythical Man-Month.* Addison-Wesley, Reading (Massachusetts), 1974.
9. COLMERAUER (A.), KANOUI (H.) et VAN CANEGHEM (M.). – *Prolog. Bases théoriques et Développements actuels.* TSI vol. 2, n° 4, p. 271-311, juillet-août 1983.
10. COX (B.J.). – *Message/Objet Programming : An Evolutionary Change in Programming Technology.* IEEE Software, vol. 1, n° 1, p. 50-69, January 1984.
11. DEWAR (R.B.K.), GRAND (A.), LIU (S.C.), SCHWARTZ (J.T.) et SCHONBERG (E.). – *Programming by Refinements, as Exemplified by the SETL Language.* ACM Transactions on Programming Languages and Systems, vol. 1, n° 1, p. 27-49, July 1979.
12. DIJKSTRA (E.W.). – *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs (New Jersey), 1976.
13. DONZEAU-GOUGE (V.), HUET (G.), KAHN (G.) et LANG (B.). – *Programming Environments Based on Structured Editors : The MENTOR Experience.* Interactive Programming Environments, ed. BARSTOW (D.R.), SHROBE (H.E.), SANDEWALL (E.), p. 128-40, McGraw-Hill, New York, 1984.
14. GOLDBERG (A.) et ROBSON (D.). – *Smalltalk-80 : The Language and its Implementation.* Addison-Wesley, Reading (Massachusetts), 1983.
15. HABERMANN (N.) et al. – *The Second Compendium of Gandalf Documentation.* Carnegie-Mellon University, Pittsburgh (Pennsylvania), 1982. Une référence plus récente sur Gandalf est le numéro spécial du Journal of Systems and Software (vol. 5, n° 2, mai 1985) consacré à ce système.
16. HOARE (C.A.R.). – *Communicating Sequential Processes.* Communications of the ACM, vol. 21, n° 8, p. 666-77, August 1978.
17. HOROWITZ (E.), KEMPER (A.) et NARASIMHAN (B.). – *A Survey of Application Generators.* IEEE Software, vol. 2, n° 1, p. 40-54, 1985.
18. IEEE. – *Standard for Software Quality Assurance Plans.* ANSI/IEEE Standard 730-1981, 1981.
19. KERNIGHAN (B.W.) et RITCHIE (D.M.). – *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
20. KRUCHTEN (F.) et SCHONBERG (E.). – *Le Système Ada/Ed : une expérience de prototype utilisant le langage SETL.* TSI vol. 3, n° 3, p. 193-200, 1984.
21. KUHN (T.). – *The Structure of Scientific Revolutions.* Second edition, The University of Chicago Press, 1970.
22. McCALL (J.) (Ed.). – *Factors in Software Quality.* General Electric, 1977.
23. MEYER (B.). – *Quelques concepts importants des langages de programmation modernes et leur expression en Simula 67.* Bulletin de la Direction des Études et Recherches d'Électricité de France, Série C (Informatique), n° 1, p. 89-150, Clamart (France), 1979.
24. MEYER (B.). – *A Basis for the Constructive Approach to Programming.* Information Processing 80 (Proceedings of the IFIP World Computer Congress, Tokyo, Japan, October 6-9, 1980), ed. S.H. Lavington, p. 293-8, North-Holland Publishing Company, Amsterdam, 1980.
25. MEYER (B.) et NERSON (J.M.). – *Cépage : Un Éditeur structurel Pleine Page.* Second Colloque de Génie Logiciel (Second Conference on Software Engineering), p. 153-8, AFCET, Nice (France), 1984.
26. MEYER (B.). – *Eiffel : A Design and Implementation Language for Software Engineering.* Inter. University of California, Santa Barbara, November 1985.
27. MEYER (B.). – *Cépage : vers la Conception de Logiciel assistée par Ordinateur.* Convention Informatique, Paris, septembre 1985.
28. MEYER (B.). – *The Software Knowledge Base.* 8th International Conference on Software Engineering, London, August 1985.
29. MEYER (B.). – *On Formalism in Specifications.* IEEE Software, vol. 3, n° 1, p. 6-25, January 1985.
30. MILNER (R.). – *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92, Springer-Verlag, Berlin, 1980.
31. MOSTOW (J.) (Ed.). – *Special Issue on Artificial Intelligence and Software Engineering.* IEEE Transactions on Software Engineering, vol. SE-11, n° 11, November 1985.
32. MUSA (J.D.). – *A Theory of Software Reliability and its Application.* IEEE Transactions on Software Engineering, vol. SE-1, n° 1, p. 312-27, July 1975.
33. NAUR (P.) et RANDELL (B.W.). (Eds.). – *Software Engineering.* NATO Scientific Affairs Division, Bruxelles, 1969.
34. NERSON (J.M.). – *Panorama des outils d'aide à l'amélioration de la qualité des logiciels.* Note Atelier Logiciel n° 41, Électricité de France, Direction des Études et Recherches, Clamart, avril 1983.
35. PARNAS (D.L.). – *On the Criteria to Be Used in Decomposing Systems into Modules.* Communications of the ACM, vol. 5, n° 12, p. 1 053-8, December 1972.
36. PARNAS (D.L.). – *Designing Software for Ease of Extension and Contraction.* IEEE Transactions on Software Engineering, vol. SE-5, n° 2, p. 128-38, March 1979.
37. PARNAS (D.L.). – *Software Aspects of Strategic Defense Systems.* Software Engineering Notes (ACM), vol. 10, n° 5, p. 15-23, octobre 1985. Egalement dans American Scientist, vol. 73, p. 432-40, septembre-octobre 1985.
38. SHOOMAN (M.L.). – *Software Engineering : Design, Reliability and Management.* McGraw-Hill, New York, 1984.
39. SIDI (J.). – *Validation de compilateurs : application à Pascal.* TSI vol. 2, n° 5, p. 345-54, sept.-oct. 1983.
40. STEELE (G.L.). – *Common Lisp,* Prentice-Hall, Englewood Cliffs (New Jersey), 1984.
41. STROUSTRUP (B.). – *The C++ Programming Language.* Addison-Wesley 1986.
42. TURNER (D.A.). – *SASL Language Manual.* Saint Andrews University, 1981.
43. TURNER (D.A.). – *Recursion Equations as a Programming Language.* Functional Programming and its Applications (ed. John Darlington, Peter Henderson and David Turner), Cambridge University Press, 1983.
44. de DROUAS (E.). et NERSON (J.M.). – *Les Ateliers Logiciels Intégrés : Développements Français Actuels.* TSI vol. 1, n° 3, p. 211-32, 1982.