# Lisibilite et comprehension des programmes informatiques:analyse du probleme dans un service informatique de l'EDF

J.C. Sperandio, A. Drouin

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports Techniques

N° 26

# LISIBILITÉ ET COMPRÉHENSION DES PROGRAMMES INFORMATIQUES : ANALYSE DU PROBLÈME DANS UN SERVICE INFORMATIQUE DE L'EDF

Jean-Claude SPERANDIO
Annie DROUIN

Juillet 1983

Rapport technique INRIA

# LISIBILITE ET COMPREHENSION DES PROGRAMMES INFORMATIQUES : ANALYSE DU PROBLEME DANS UN SERVICE INFORMATIQUE DE L'EDF

Jean-Claude SPERANDIO
avec la collaboration d'Annie DROUIN

juin 1983

**Résumé**

Le présent rapport fait la synthèse de 22 interviews menées dans un grand service d'informatique de gestion (à EDF/GDF) et centrées sur les difficultés de lecture et de compréhension de programmes existants. Les principaux facteurs abordés sont : la standardisation des règles de dénomination, la structuration, les commentaires et la documentation associée aux programmes.

**Summary**

This report synthesizes 22 interviews carried out in a big computerized management department (EDF/GDF). The interviews aimed at investigating the difficulties of program understanding. The main issues concerned : the standardization of labelling rules, the program structure, the comments and the documentation associated to the programs.

**Mots-clés**

Lisibilité, compréhension, dénomination, commentaires, structuration, documentation.

**Key words**

Readibility, understanding, labelling, commenting, structuration, documentation.

## I - POSITION DU PROBLEME

Une des activités intellectuelles centrales de la programmation, à côté de et souvent associées à la tâche de résolution de problème qu'est l'élaboration d'une procédure informatique, est la compréhension de programmes existants (écrits par d'autres programmeurs ou par le lecteur depuis un certain temps). On a même écrit avec raison (J. Arsac) qu'"un programme est destiné à être compris par des humains avant d'être exécuté par l'ordinateur", mais ce n'est que depuis quelques années seulement, que l'on se préoccupe vraiment d'améliorer la lisibilité et la compréhension comme des qualités importantes des langages et des programmes. L'intérêt est opérationnel et se mesure en temps gagné. Les astuces et l'hermétisme de programmation cessent peu à peu d'être valorisées dans la mentalité des programmeurs. Cela tient sans doute à l'évolution des matériels et des systèmes d'exploitation, qui rendent plus négligeables qu'il y a quelques années le gain de quelques fractions de secondes d'exécution ou de place en mémoire s'il en coûte en échange un allongement du temps de programmation et surtout du temps de la maintenance technique ultérieure des programmes. Cette incidence est donc opérationnellement d'autant plus importante que :

- l'équipe de programmation est nombreuse et diversifiée dans la formation et la compétence,
- la programmathèque est volumineuse,
- les programmes sont longs ou compliqués,
- la part de maintenance logicielle est importante par rapport à la part d'écriture initiale,
- l'utilisation de progiciels se développe.

Pratiquement, la nécessité de lire et de comprendre des programmes existants apparaît lorsque l'on veut les utiliser, les modifier ou leur adapter d'autres programmes ou les fichiers nécessaires.

Lire et comprendre sont deux activités distinctes, mais étroitement reliées. Plus encore que dans la lecture de texte, le processus de lecture d'un programme (écrit par un autre programmeur) est contrôlé par le processus de compréhension. Inversement, la stratégie de lecture va induire une certaine hiérarchisation des niveaux de compréhension. Il faut donc conjointement étudier la lecture et la compréhension.

Comme en perception de formes et en lecture de textes, l'étude de la lecture des programmes passe par la recherche des facteurs de lisibilité, c'est-à-dire les facteurs qui rendent plus ou moins rapide cette lecture et surtout qui déterminent le cours de cette lecture, son cheminement à l'intérieur du programme et les points d'arrêt. Moins encore que dans la lecture de textes, la lecture des programmes est rarement linéaire surtout si le programme est long et complexe. Il est probable qu'elle s'apparente davantage à l'exploration de formes, par recherche de repères et par assimilations successives de zones de compréhension. La logique du chaînage des étapes d'assimilation et les inférences faites pour passer d'une zone de lecture à une autre est particulièrement importante pour l'efficacité de la compréhension.

La littérature sur cette question commence à peine à se développer. Le processus de lecture lui-même est encore très peu étudié. Plus pragmatiquement, les auteurs ont tenté tout d'abord de déterminer les facteurs qui modulent la performance à des épreuves de compréhension passées à l'issue de lectures de programmes effectuées en temps limité. Parmi tous les facteurs évoqués dans la littérature, les plus fréquemment pris en compte sont les suivants :

a) <u>les facteurs propres aux programmeurs</u>

- le degré de connaissance du langage de programmation utilisé (règles syntaxiques/sémantiques/logiques),
- le degré de connaissance du "dialecte local", s'il y a lieu, c'est-à-dire les restrictions ou les extensions valides du langage standard, ainsi que les règles d'usage propres à une équipe donnée,
- le degré de connaissance du référentiel, c'est-à-dire l'environnement du programme, en particulier :
    - les dénominations plus ou moins signifiantes des variables, des fichiers ou des autres programmes évoqués,
    - le but opératoire des fichiers et des autres programmes associés,
    - la tâche réalisée par le programme et le produit final.

b) <u>les facteurs propres aux programmes</u>

- le langage utilisé : les auteurs de nouveaux langages se plaisent souvent à affirmer que leur langage est plus "naturel", plus "compréhensible" que d'autres. Cela peut s'évaluer expérimentalement.

- la qualité de l'écriture, en particulier :
  - la dénomination des variables (certaines dénominations sont plus "parlantes" que d'autres, plus faciles à apprendre, à retenir et à décoder),
  - l'utilisation d'instructions plus ou moins compactes, plus ou moins astucieuses, plus ou moins dangereuses (ex. : l'imbrication d'instruction de boucles, d'aiguillages, de rupture de séquence), etc.
- le type de structure du programme,
- la logique algorithmique suivie,
- les aides internes, en particulier :
  - le titrage plus ou moins explicite des parties de programmes,
  - les commentaires,
- les aides externes, en particulier :
  - la schématisation éventuelle du programme ou de parties du programme (ex. : organigrammes),
  - la documentation associée à un programme.

Ces différents facteurs ont constitué la trame d'un ensemble d'entretiens que nous avons mené dans un grand service d'informatique de gestion. Nous avons essayé de déterminer dans quelle mesure chaque programmeur, compte tenu de sa propre expérience et de ses tâches habituelles, était confronté à des difficultés de compréhension et quelle pouvait être l'importance de ces différents facteurs. Le premier objectif de cette étude était donc de relativiser l'intérêt même d'une étude qui doit être menée en utilisant une méthodologie plus objective. Le second objectif était de préparer les étapes et les modalités de cette étude ultérieure.

## II - GENERALITE SUR LE SERVICE INFORMATIQUE DE REFERENCE ET METHODOLOGIE

Le service d'accueil est le SIM (Service Informatique et Méthodes), relevant de la Direction de la Distribution de l'EDF/GDF et situé à Issy-les-Moulineaux.

Historiquement, au début des années 60, l'informatique de la Direction de la Distribution se répartissait en :

- 5 centres de traitement de la facturation clientèle (appelés SITI, Services Intérégionaux du Traitement de l'Information),

- 11 ORGE (Organismes Régionaux de Gestion), chargés de traiter les éléments variables des activités du personnel des établissements.

En 1975, le Schéma Directeur Informatique de l'EDF a préconisé :

- la fermeture progressive des SITI et des ORGE,
- la création progressive d'ATIC (Atelier de Traitement de l'Information des Centres) au niveau départemental et d'ERI (Equipe Régionale Informatique) au niveau régional,
- la création du SIM (Service Informatique et Méthodes) au niveau national.

La mission du SIM est de mener la déconcentration sur 95 Centres de Distribution (un ATIC par Centre de Distribution), impliquant tous les moyens humains, matériels et logiciels devant être mis en place à l'horizon 85. Le SIM assure notamment l'implantation et la mise en oeuvre des traitements BATCH et Transactionnels : création, lancement et maintenance des programmes existants et à concevoir.

Deux constructeurs se partagent les ordinateurs de traitement équipant les centres : IBM (série 4341) et CII-HB (série DPS 7). Pour l'informatique transactionnelle, deux constructeurs équipent les centres : CII-HB (DKU 7002 - QUESTAR) et TRANSAC (ALPHA 20 et TE 12).

Le SIM comprend plusieurs départements, ayant chacun une mission définie :

- Groupe de "régionalisation de l'informatique distribution", chargé du lancement des ATIC sur les plans réseaux, matériels, recrutement, formation.

- Département "facturation accueil", chargé de la création, du lancement et de la maintenance des programmes BATCH relatifs à la facturation de la clientèle.

- Département "applications évolutives", chargé des programmes TEMPS REEL des applications transactionnelles inter-directions,

- Département "organisation de l'informatique et étude des méthodes", chargé des études et de la conception des moyens en matériels et logiciels liés à l'évolution de l'informatique de gestion.

La formation, le niveau de recrutement, l'expérience et les responsabilités des programmeurs du SIM sont très variables. Notre étude a porté sur 22 personnes, appartenant aux différents départements du SIM, en fonction de leur ancienneté.

5 avaient moins d'un an dans le service
6 " entre 1 et 5 ans " " "
6 " entre 5 et 10 ans " " "
5 " plus de 10 ans " " "

Nous avons procédé par entretiens individuels centrés sur les questions principales suivantes :

- formation en informatique et expérience dans le service,
- le travail de programmation actuel et passé (type d'application, type de tâches),
- pratiques personnelles et difficultés éventuelles rencontrées pour la lecture et la compréhension de programmes existants rédigés par d'autres programmeurs,
- pratiques personnelles pour écrire les commentaires,
- utilisation personnelle de la documentation existante,
- la conception et la rédaction de la documentation relative à de nouveaux programmes,
- la documentation accessible sur terminal.

Les questions sur lesquelles l'entretien a été centré ont été abordées dans l'ordre indiqué ci-dessus, sauf si la personne interviewée abordait spontanément une question ultérieure.

Chaque entretien a duré en moyenne 45 minutes, parfois moins pour les débutants, parfois beaucoup plus pour les plus anciens ! L'entretien se déroulait soit dans le bureau de l'agent, soit le plus souvent dans une salle de réunion disponible.

Seul l'auteur du présent rapport a mené et recueilli les entretiens, sans magnétophone, mais en prenant des notes au fil du discours.

## III - SYNTHESES DES RENSEIGNEMENTS RECUEILLIS

### 1 - Formation en informatique et première expérience dans le service

L'ensemble des personnels consultés présente une grande variété quant aux filières de formation suivies et quant aux itinéraires personnels à l'intérieur de l'entreprise EDF/GDF. Les plus anciens ont été les acteurs du développement de l'informatique dans le service (ou dans les structures qui lui ont précédé). Ils en connaissent donc l'histoire et les rouages. Pour eux, l'état actuel de l'organisation générale ou l'état actuel de certains détails techniques de la programmation se trouve sinon justifié, du moins expliqué par le développement historique. Il n'en est pas de même pour les plus jeunes. Connaissant moins le fil historique de l'informatique à l'EDF, ils ont par contre eu souvent l'occasion au cours de leur formation de voir des aspects de l'informatique autres que ceux propres au service, et peuvent donc se référer plus ou moins à d'autres contextes. Ils sont donc plus critiques. Cela dépend surtout de leur formation antérieure en informatique qui est très hétérogène. Les plus anciens n'ont pas eu de formation particulière : ils ont appris l'informatique dans le service, sur le tas, avec l'aide de stages internes éventuellement. Parmi les plus jeunes que nous avons rencontrés, tous n'ont pas reçu une formation informatique antérieure, mais la plupart d'entre eux ont bénéficié soit d'une initiation pendant leur scolarité (IUT, école d'ingénieur ou pendant le service militaire), soit même d'une expérience professionnelle limitée (stage, emploi provisoire, etc.).

Les difficultés rencontrées au cours des premiers mois de travail dans le service sont donc très variables selon l'origine et les antécédents. En règle générale, ceux qui ont débuté la programmation dans le service estiment qu'une formation plus systématique aurait été profitable, que la documentation disponible est insuffisante ou inadaptée, que les stages sont insuffisants ou inadaptés, trop tardifs surtout et que le "bouche-à-oreille" ne suffit pas. A cet égard, il semble y avoir des différences importantes selon les équipes, certains programmeurs plus anciens jouant le rôle de pédagogues appréciés.

Les difficultés sont moindres pour ceux qui arrivent dans le service avec un début de formation; ces difficultés apparaissent d'autant moindres que le bagage antérieur est plus important. Pour ceux-là, les difficultés sont d'un autre ordre. Ayant déjà une expérience ou du moins des connaissances en pro-

grammation, éventuellement même en COBOL, il s'agit pour eux de minimiser la durée de la période transitoire pendant laquelle ils découvrent l'organisation du service et (pour certains) de l'EDF/GDF, l'organisation des différentes applications, les sigles, enfin les programmes existants et les tâches que l'on attend d'eux. La durée de cette période transitoire semble variable selon les individus et les équipes (de quelques semaines à plus d'une année). Il est d'ailleurs malaisé de définir avec précision à partir de quand un programmeur est ou s'estime être "opérationnel". Cela dépend des tâches exigées et des applications. Quelqu'un peut être ou se sentir "opérationnel" pour écrire des petits programmes, mais non pour assurer la maintenance de programmes existants; cela dépend surtout de la nature des problèmes de maintenance à résoudre. En règle générale, une année entière est considérée comme nécessaire pour se familiariser avec l'ensemble des tâches de programmation du service, mais la variabilité est grande sur cette appréciation et semble surtout dépendre du bagage informatique antérieur, du type d'application, ainsi que du mode d'organisation de l'équipe.

## 2 - Le travail de programmation actuel et passé

Vu par un observateur extérieur, le travail de programmation dans le service présente des aspects diversifiés :

- analyse du problème,
- écriture des programmes,
- maintenance des programmes,
- écriture de la documentation (à usage interne du service ou destinés à des utilisateurs extérieurs),
- formation.

Certains des programmeurs interviewés parmi les plus anciens, ont été chargés de toutes ces tâches, à un moment ou un autre, mais les plus jeunes n'ont qu'un expérience limitée à l'une ou à quelques-unes de ces tâches.

Il ressort très nettement pour l'ensemble du service une organisation tendant à dissocier l'analyse et l'écriture des programmes. Cela est plus ou moins net selon les applications. En général, les personnes intérrogées sont peu favorables à cette dualité (dite "interne/externe") et nombreux sont ceux qui estiment que pour bien analyser un problème, il faut aller jusqu'à rédiger

les programmes et les maintenir (mais certains sont d'un avis contraire). Le peu de liens avec les utilisateurs des programmes est jugé, par les plus jeunes surtout, comme une faiblesse de l'organisation du système. Les plus anciens, qui en général ont une meilleure connaissance de l'ensemble de l'entreprise (ne serait-ce que parce qu'ils ont eux-mêmes précédemment travaillé dans les services utilisateurs), se sentent moins "coupés" des utilisateurs que les plus jeunes.

La spécialisation des tâches de programmation proprement dite varie selon les applications, mais n'est pas très stricte, semble-t-il. En général, les responsables d'application commencent par confier aux jeunes programmeurs la rédaction de petits programmes nouveaux, soit totalement nouveaux, soit constituant une nouvelle version d'un programme existant, ou une modification d'une partie limitée d'un programme. La maintenance des programmes est, en règle générale, confiée à celui qui a écrit le programme, lorsque c'est possible. La maintenance est en effet considérée comme une tâche difficile qui ne peut être réalisée que lorsque le programmeur est "à l'aise" dans le programme et l'environnement. Dans certaines applications, semble-t-il, chaque programmeur est affecté à un nombre limité de programmes qu'il a la charge de maintenir. Dans d'autres applications, la polyvalence est de rigueur et semble être bien appréciée, mais elle n'est possible que dans les applications jeunes et de taille restreinte (en nombre de programmes et en nombre de programmeurs).

La documentation est écrite par l'auteur du programme, en règle générale. Plus rarement, elle est établie après coup par un autre programmeur lorsqu'un ancien programme était non documenté ou mal documenté. Si tous estiment que la documentation est une tâche nécessaire, ils reconnaissent également qu'elle n'est pas prioritaire : il faut que les programmes "tournent" avant tout ! Nous reprendrons ce point plus en détail.

Quant à la formation, elle fait l'objet de beaucoup de critiques, surtout pas les plus jeunes : trop peu de stages, trop peu de documents pédagogiques appropriés. Les nouvelles réalisations en matière d'aides pédagogiques (notamment sur COBOL, CODEX, ITF, etc.) étaient trop récentes au moment des entretiens et de ce fait n'avaient pas été utilisées. Tous plus ou moins, mais en particulier les plus jeunes, estiment que la formation n'est pas suffisamment systématisée et que l'apprentissage "sur le tas" n'est pas une bonne chose.

En résumé, la diversité des tâches, la diversité des applications et l'hétérogénéité des filières de formation constituent trois facteurs d'hétéro-

généité de la population interviewée quant à l'estimation individuelle des premières difficultés rencontrées par chacun, mais c'est avant tout la formation qui fait l'objet des critiques les plus vives.

### 3 - Lecture et compréhension de programmes existants, écrits par d'autres

Il est clair que cette question préoccuppe moins les programmeurs très confirmés que les débutants, mais elle se pose ou s'est posée plus ou moins à chacun. Reprenons les points noirs de la compréhension les plus souvent évoqués :

- le degré de connaissance du langage de programmation utilisé, le COBOL, n'est pas le facteur jugé essentiel ici, dans la mesure où le COBOL n'est pas un langage considéré comme difficile et que chacun estime que ce cap de difficulté est ou sera vite surmonté. Cependant, certains débutants ont fait part des difficultés qu'ils ont éprouvées ou éprouvent pour apprendre le COBOL.

- le degré de connaissance du "dialecte local", appelé ici le COBOL "compatible" (1). Ce point n'est pas non plus jugé crucial, dans la mesure où :
  - les différences par rapport au COBOL standard ne sont pas très importantes,
  - la plupart des programmeurs n'ont pas eu auparavant une expérience du COBOL standard,
  - les différences sont soit des restrictions (par exemple, certaines instructions à ne pas utiliser), soit des recommandations (par exemple, certaines règles de dénomination des variables, fichiers, programmes, etc.), soit des outils nouveaux plus performants et reconnus comme tels (par exemple, clauses COPY, Data automatique, etc.).

Dans l'ensemble, le dialecte local ne fait pas l'objet de critiques majeures. Les aspects positifs sont : performance et facilité de communication entre personnes initiées. Les aspects négatifs : formalisme un peu lourd,

---

(1) Compatible dans le sens de "commun à IBM et CII", ou plus exactement non dépendant de telle ou telle machine. On dit aussi "COBOL maison".

quoique pratique, et surtout tendance à standardiser les programmes, ce que certains estiment ne pas être une bonne chose.

- le degré de connaissance de l'environnement informatique : c'est ici le facteur de difficulté de compréhension jugé essentiel par tous et surtout par les débutants. Il nous a souvent été dit : "nos programmes ne sont pas très compliqués, c'est l'environnement qui est compliqué". La connaissance de cet environnement pourrait être facilitée par une formation systématique. Les plus jeunes (moins d'un an expérience) sont confrontés tout d'abord aux dénominations des variables ou des fichiers qui utilisent des sigles inconnus ou codés selon des règles qu'ils ne connaissent pas ou maîtrisent encore mal. Surtout, le but des programmes et la tâche qu'ils réalisent apparaissent explicitement comme des obstacles à la compréhension tant qu'ils ne sont pas bien assimilés.

- la qualité de l'écriture : il y a, semble-t-il, des "styles" de programmation que certains préfèrent, sans pour autant pouvoir expliciter pourquoi ces styles leur sont plus compatibles. Probablement, il faut en rechercher l'origine dans la formation de chacun, et ce d'autant plus que la formation a été faite "sur le tas". Plusieurs programmeurs ont affirmé qu'ils comprenaient bien ceux qui programment comme eux. Cette parenté peut porter sur les habitudes de dénomination des variables, sur la structuration du programme, sur l'utilisation de certaines astuces connues, etc. Certains programmeurs ont été habitués à programmer ensemble. Dans certaines équipes, c'est presque la règle. D'autres préfèrent rester solidaires. Il s'en suit une manière plus ou moins partagée de programmer, qui rejaillit ainsi sur la compréhension.

- la structuration des programmes est un des facteurs de compréhension le plus souvent évoqués au cours des entretiens. Les plus jeunes, en particulier ceux qui ont appris l'informatique en école, sont généralement des adeptes convaincus de la "programmation structurée" et éprouvent des difficultés pour comprendre des programmes écrits selon d'autres règles. Cela dit, il faudrait s'entendre sur ce que l'on appelle "structure", puisque les programmeurs plus anciens font ressortir à juste titre un système de structuration fondé sur un autre principe : isomorphie de la séquence du programme et de la séquence de

l'ordinogramme moyennant des règles de lecture de celui-ci de haut en bas, et de gauche à droite, et un codage des parties de programme se référant aux pavés de l'ordinogramme.
Cependant, ce n'est pas le type de structuration adoptée qui est ici en cause, mais le fait de structurer ou de ne pas structurer. Ce qui est critiqué (par tous, jeunes ou anciens) c'est la programmation "sauvage". Mais qui programme "sauvagement" ?
En tout cas, s'il existe des adeptes de telle ou telle forme de structuration, il ne semble pas que les programmeurs souhaitent l'adoption de règles normatives générales pour structurer. La normalisation, en particulier pour certaines dénominations, est jugée positive, mais chacun ressent également le danger pour leur propre métier d'une trop grande standardisation normative des pratiques de programmation. Le droit au choix de la structuration semble faire partie de cette part réservée à la flexibilité et à l'individualisme.

- la logique algorithmique suivie est aussi un facteur de plus ou moins grande compréhension, mais il a été souvent remarqué que ce n'est pas la complexité de la logique de programmation qui fait le plus problème ici, mais plutôt celle de l'environnement. Le risque peut être de confondre la finalité d'une séquence d'instruction avec une autre et/ou de ne pas saisir telle finesse ou tel détail dans une procédure par ailleurs globalement comprise, lorsque l'on "reconnait" une procédure ou partie de cette procédure. C'est là que doivent intervenir les commentaires, en signalant les pièges ou les variantes par rapport à une procédure classique. L'existence de "programmes types" semble, à la quasi unanimité, être considérée comme un élément positif à la fois parce qu'ils facilitent l'écriture et parce qu'ils facilitent la compréhension. Ceci est à mettre au crédit de la standardisation, avec la réserve évoquée plus haut.

- les commentaires insérés dans les programmes sont considérés comme importants, nécessaires même, mais assez mal employés. Il n'y a pas de règles bien précises. On reconnaît volontiers, même les plus anciens, ne pas très bien savoir comment faire. Les programmeurs ont tendance à commenter abondamment les parties de programme où ils ont éprouvé des

difficultés de conception, mais ce ne sont pas nécessairement ces parties-là qui, par la suite, occasionneront des difficultés de compréhension aux lecteurs ultérieurs. A cet égard, le concepteur d'un programme n'est pas le mieux placé pour "se mettre à la place du lecteur" et prévoir ses difficultés. En tout cas, il semble qu'il y ait d'assez grandes différences entre les programmeurs. Certains reconnaissent être plutôt prolixes en commentaires, tandis que d'autres estiment (à tort ou à raison) ne pas en introduire suffisamment. Certains esthètes sont d'avis qu'un programme bien conçu, bien structuré, utilisant des libellés "parlants", etc. doit se suffire à lui-même et que les commentaires sont en quelque sorte la marque d'une faiblesse de conception! Les commentaires tentent de remplir plusieurs fonctions : signal de pièges à éviter, ou renseignement pédagogique, ou titre de paragraphe ou simple borne séparative. On attend surtout d'eux qu'ils soient des jalons pour faciliter le diagnostic d'anomalies éventuelles ou les modifications ultérieures.

## 4 - Le processus de lecture

Après les "points noirs" de la compréhension, nous avons demandé si, en abordant des programmes écrits par d'autres, le programmeur adopte une démarche systématique. De plus, à trois programmeurs interviéwés dans leur propre bureau, qui venaient de recevoir un programme à étudier qu'ils ne connaissaient pas, nous avons demandé de "lire tout haut", ou plus exactement de découvrir ce programme et d'expliciter leur démarche de lecture, ce qu'ils comprenaient et ce qu'ils ne comprenaient pas. Ce point a trait à une expérience future en projet.

Les informations recueillies sur ce point de l'interview sont très labiles. Bien entendu, ce ne sont pas des résultats expérimentaux, mais simplement des indices susceptibles de nourrir une expérience ultérieure.

Deux méthodes ont été évoquées :

a) lecture exhaustive, instruction par instruction, une première fois, et relectures successives de certaines parties.

b) exploration analytique, en recherchant les repères permettant d'identifier la structure générale, en revenant ultérieurement sur des niveaux de détails inférieurs.

Sur les 22 personnes interviewées, 14 estiment procéder selon la seconde méthode (dont 2 sur les 3 programmeurs "observés").

Mais de l'avis général, le mode d'approche d'un nouveau programme semble dépendre en premier lieu du programme lui-même et du degré de connaissance que le programmeur en a. Sauf exception, celui-ci connaît au moins la finalité générale du programme, il s'attend à y trouver certaines procédures, certaines instructions; il s'attend à une certaine structure éventuellement. Si ce n'est pas le cas, une première phase exploratoire a pour but d'apporter ces connaissances minimales : les commentaires jouent à cet égard un rôle apprécié.

L'identification de la structure générale, au moins à un niveau macroscopique, est considérée comme l'un des tout-premiers objectifs à atteindre. D'où l'intérêt évident d'une programmation qui d'une façon ou d'une autre explicite bien cette structure (mais les avis sont partagés sur les avantages de telle ou telle méthode d'analyse qui peut induire tel ou tel mode de structuration).

Cependant, il ne nous est pas apparu clairement que la procédure de compréhension la plus générale suivait une hiérarchie descendante systématique, allant des grandes parties du programme jusqu'au niveau de l'instruction élémentaire. Cette approche hiérarchique est peut-être (mais ce n'est pas sûr) adoptée par les programmeurs ayant eu une bonne formation à la programmation dite structurée, mais ce n'est pas la majorité des personnes interviewées. Il se pourrait aussi que l'intérêt d'une telle approche hiérarchique apparaisse variable selon la longueur et la complexité du programme. Ceux qui nous ont dit procéder de cette manière admettent que, le cas échéant, ils s'attardent sur certains segments du programme qui présentent soit une bizarerie apparente, soit une similitude avec des connaissances antérieures (appel de sous-programmes connus, références à un fichier connu, etc.).

Avec les trois programmeurs "observés", nous avons essayé de suivre leur démarche en identifiant les points obscurs). Il s'agissait de 3 débutants (moins de 6 mois dans le service). L'objectif de leur tâche était de faire une modification partielle du programme, sans changer la structure. Les points obscurs rencontrés :

**Cas 1** :

- pas de structure évidente sur un programme d'une centaine d'instructions,

- trop peu de commentaires donnant la finalité des parties du programme,
- une série de "PERFORM" non explicitement justifiée,
- la partie à modifier avait été d'emblée indiquée par le chef d'équipe, mais le programmeur ne comprenait pas exactement quand, à l'exécution, cette partie du programme intervenait,
- l'utilité de la partie suivant immédiatement la partie à modifier n'apparaissait pas; on y accédait après un emboîtement d'aiguillages (IF...THEN...ELSE) dont la logique restait obscure.

**Cas 2**

- petit programme d'une cinquantaine d'instructions,
- ce programme fonctionnait, mais il fallait le réécrire parce que deux modifications antérieures l'avaient déformé,
- la structure générale semble être assez claire au programmeur après une lecture directe plutôt superficielle (environ 3 minutes), mais il bute sur les deux fichiers qui sont lus, car il ne connait pas la logique de lecture de ces fichiers,
- il y a un GO...TO qui lui paraît suspect,
- le programmeur ne dispose pas de la documentation correspondant au programme, mais il sait que la documentation existe, et semble décidé à la consulter.

**Cas 3**

- petit programme d'une soixantaine d'instruction,
- objet de la modification à faire : adapter les instructions de lecture à certaines modifications faites sur un fichier,
- le programmeur parcourt le listing en repérant les grandes divisions (explicites en COBOL). Il revient très rapidement sur la "PROCEDURE DIVISION". Repère les instructions d'édition et de lecture. Il dit : "apparemment, ici on lit la bande, mais je ne sais pas ce qu'elle contient". Puis il recherche sur le listing la FILE SECTION et à l'intérieur de celle-ci recherche le nom du fichier en question, s'y attarde, puis revient sur l'instruction READ correspondante. Il dit : "mon collègue connaît bien ce fichier. Je vais attendre qu'il revienne pour qu'il m'explique comment il est fait. Pour l'instant, je nage complètement".

De ces quelques informations éparses, nous ne retirons aucune conclusion, sinon quelques idées sur la nécessité d'une approche expérimentale plus précise dont les facteurs principaux seraient :

- le niveau d'expérience du programmeur,
- le degré de connaissance sur l'objet du programme et son environnement éventuel,
- la longueur et la complexité du programme,
- l'explicitation de la structure générale,
- les aides ponctuelles apportées par les commentaires.

## 5 - L'utilisation de la documentation associée aux programmes

En principe, chaque programme dès sa mise au point fait l'objet d'une documentation comportant, outre l'analyse fonctionnelle (cahier des charges, spécifications, etc.), une schématisation plus ou moins détaillée (organigramme) et des explications sur la rédaction et le fonctionnement du programme. L'analyse fonctionnelle est généralement rédigée par celui ou ceux qui ont été chargés de la conception fonctionnelle du produit : cette partie constitue une trace historique de l'analyse du programme (ou de l'ensemble des programmes d'un produit donné) et sera utile pour mieux comprendre la finalité du produit et certains aspects de la procédure. Lorsque les programmes sont écrits et mis au point, une double documentation est faite : une destinée aux utilisateurs et une autre destinée aux programmeurs, orientée vers la maintenance.

En pratique, la documentation est lacunaire (d'après les avis exprimés). D'une part, certains programmes n'ont pas été documentés ou l'ont été partiellement ou superficiellement. D'autre part, la documentation n'est pas toujours mise à jour lorsqu'interviennent par la suite des modifications dans les programmes.

L'état de la documentation, toujours d'après les opinions exprimées, semble très inégal selon les applications. Certaines sont considérées comme bien documentées, un effort ayant été fait pour réaliser systématiquement une documentation et pour la maintenir à jour, mais cela ne semble pas être le cas général. Plusieurs qualités sont requises d'une documentation pour qu'elle soit utilisée :

- l'exhaustivité : on ne fait l'effort d'aller consulter la documentation que si celle-ci n'est pas trop lacunaire, c'est-à-dire si on pense

avoir de bonnes chances de trouver le renseignement cherché. Plus une documentation est partielle, moins elle sera utilisée, même pour les parties bien documentées.

- la facilité d'exploitation : certaines documentations sont considérées comme difficilement abordables, parce que trop denses, trop touffues ou mal structurées. "Pour trouver ce que l'on cherche, dit-on, c'est un vrai labyrinthe. Seuls les initiés peuvent s'en servir". Ceci explique que les débutants se trouvent désemparés.
- l'accessibilité : la documentation peut exister, mais on ne l'utilise que si on l'a facilement "sous la main". On ne sait pas toujours où se trouve la documentation.

De ces qualités requises, il s'en suit une très inégale utilisation de la documentation existante selon les tâches, selon les applications et selon l'ancienneté du programmeur.

**selon l'ancienneté,**

On pourrait penser que les débutants ont plus de besoins exprimés en documentation que les anciens. Ce n'est pas le cas. Le besoin est surtout manifeste chez ceux qui font de la mantenance : or, ce ne sont généralement pas des débutants. Ceux-ci disent avoir parfois recherché la documentation concernant les programmes qu'ils devaient "apprendre" ou modifier, par exemple, mais en général ils ont été déçus. La documentation, disent-ils, n'est pas pédagogique. Il faut connaître le fil d'ariane du labyrinthe. Elle utilise les mêmes sigles non expliqués que les programmes eux-mêmes. Après quelques tentatives infructueuses, les jeunes programmeurs préfèrent demander conseil directement à un ancien. Nous verrons que cette "transmission orale" du savoir et de l'expérience explique en grande partie la sous-utilisation de la documentation existante.

**selon les tâches**

Quatre types de besoins spécifiques sont apparus, correspondant à quatre situations distinctes :

a) lors d'une maintenance "à chaud", devant une panne ou une anomalie, la documentation sert peu. Les commentaires internes sont plus utiles.

L'organigramme général n'est pas non plus jugé très utile, sauf pour donner une vue globale des grandes parties du programme (un organigramme simplifié pourrait être inclus en commentaires sur le listing). Mais certains rédacteurs de "doc", paraît-il, rédigent en "pensant maintenance". La stratégie de diagnostic (à chaud) repose moins sur la compréhension des détails du programme que sur l'examen des dernières modifications réalisées (qui sont explicitées sur le listing). En général, la documentation n'a pas été mise à jour, sauf si la modification était très importante. En cas de difficulté pour trouver l'origine d'une panne ou d'une anomalie, dans le cas où celui qui fait le diagnostic n'est pas l'auteur du programme, la tendance n'est pas de rechercher la documentation, mais de contacter l'auteur du programme ou un collègue connaissant bien le programme.

b) lors d'une modification à réaliser : la documentation est jugée utile pour comprendre les connexions entre la partie à modifier et le reste du programme. Ceci pour éviter de supprimer indûment des instructions qui sont fonctionnellement impliquées dans plusieurs parties. Pour ce faire, l'analyse fonctionnelle dans le dossier du programme est souvent nécessaire, bien qu'insuffisante. L'organigramme général, s'il a été bien fait et s'il a été maintenu à ce jour, est jugé susceptible de faire apparaître facilement les parties connectées et les parties indépendantes. Toutefois, beaucoup des interviewés admettent qu'ils n'ont pas la patience de décripter les organigrammes.

c) lors d'une réécriture totale d'un programme : les avis sont partagés sur l'utilité de consulter la documentation. Puisqu'il faut faire du neuf, pourquoi s'inspirer du vieux? L'analyse fonctionnelle ancienne, et notamment le cahier des charges et des spécifications, suffit. Dans certains cas, même, cette analyse fonctionnelle ancienne ne convient plus, et dans ce cas on repart complètement à zéro. En résumé, l'intérêt de consulter la documentation dépend de l'amplitude de la réécriture. En pratique, la réécriture des petits programmes est souvent une tâche confiée aux débutants, presqu'à titre pédagogique. La consultation de la documentation, lorsqu'elle existe, leur permet de mieux comprendre la finalité du programme et son environnement. Mais, ce faisant, les débutants pourraient être tentés de trop s'inspirer du

programme existant. Encore faudrait-il que la documentation soit accessible.

d) en situation d'apprentissage : la documentation pourrait avoir un objectif pédagogique. Toutefois, comme nous l'avons dit, les débutants estiment que la documentation leur est inacessible, à cause du jargon employé ou parce que la démarche du rédacteur n'est pas celle qui convient à une initiation.

**selon les applications**

Il y a, semble-t-il, des différences importantes dans l'état de la documentation (exhaustivité, mode de rédaction et suivi de la mise à jour) et dans les besoins exprimés. Une étude plus précise est nécessaire, mais il semble que la taille et surtout l'ancienneté d'une application soit un facteur essentiel à cet égard. D'une part, dans les applications récentes, le nombre de programmes et le nombre de programmeurs sont généralement plus petits que dans les applications anciennes (A 49, par exemple); dans les petites équipes, l'ensemble des programmes est connu de tous; de ce fait, il est plus facile (en tout cas, plus immédiat) d'interroger directement les collègues plutôt que de consulter la documentation elle-même. D'autre part, dans les applications récentes, la tâche des programmeurs est plus souvent l'écriture de nouveaux programmes ou leur mise au point que leur maintenance; de ce fait, la documentation pose plus de problèmes de rédaction que de consultation.

Dans l'ensemble, on ne peut pas dire que la documentation soit vécue par la majorité comme un "problème" important et urgent, bien que ce soit la préoccupation de certains. Non pas que la documentation existante soit jugée satisfaisante, au contraire, mais la plupart préfèrent recourir au savoir des collègues compétents. Certains estiment même que cette communication orale est une nécessité pour le bon fonctionnement du service, à laquelle pourrait nuire un système documentaire trop exhaustif.

**Remarque** : Digne d'intérêt sur le plan social, la communication orale du savoir et du savoir-faire - qui semble bien fonctionner ici - pourrait devenir vulnérable si le turn-over des programmeurs était grand. Ce point mérite

réflexion. Mais l'augmentation du turn-over, en nécessitant une plus grande systématisation de la documentation, augmenterait aussi la nécessité d'une formation plus systématique.

## 6 - L'écriture de la documentation

Il apparaît que les programmeurs expérimentés sont convaincus de la nécessité d'écrire une documentation technique des programmes qu'ils réalisent, même si eux-mêmes déclarent par ailleurs utiliser peu la documentation existante. La partie concernant les détails techniques du programme est écrite la plupart du temps par le programmeur lui-même, en fin de mise au point du programme. Mais ce n'est pas une tâche jugée prioritaire, ce qui explique que certains programmes ne sont pas documentés, ou ne le sont que partiellement, si la charge de travail au moment où ils ont été écrits était importante. L'écriture de la documentation est renvoyée à plus tard.

Quel est le processus d'écriture de la documentation? Il n'y a pas de règles précises, semble-t-il, chacun faisant un peu comme il veut. Certains se disent volontiers "littéraires", n'hésitant pas à produire un texte rédigé, alors que d'autres penchent plutôt pour la notice technique, avec des schémas plutôt que du discours. On retrouve ici la difficulté signalée à propos des commentaires insérés dans les programmes : les difficultés de conception ne sont pas nécessairement les difficultés de compréhension en lecture; or le programmeur a plutôt tendance à souligner les points qu'il juge lui difficiles, sur lesquels il a le plus travaillé.

Un canevas-type pourrait servir de guide lors de l'écriture de la documentation, qui préciserait les parties et les paragraphes, à la manière d'un cahier des charges. Cela faciliterait le passage éventuel ultérieur de cette documentation sur un support informatique. Mais certains signalent le danger de lourdeur et de rigidité d'une telle formule.

## 7 - L'utilisation de la documentation sur support informatique

Nous avons vu qu'une des qualités requises d'une documentation est sa disponibilité. Une autre est la mise à jour permanente. Ces deux qualités peuvent justifier une accessibilité par terminal, actuellement en cours d'étude et de réalisation. Les avis semblent partagés sur l'intérêt pratique

d'une telle solution. On fait remarquer qu'il faudrait disposer d'un nombre supérieur de terminaux dans le service et que la disponibilité de l'information sur un écran ne remplace pas le support papier, ne serait-ce que pour pouvoir s'en servir sur son propre bureau près des listings. En revanche, on en voit bien l'importance pour une mise à jour facilitée et pour l'accessibilité à partir de sites éloignés.

La consultation sur terminal ne devrait pas être un simple changement de support par rapport au papier comme actuellement. On devrait en profiter pour y mettre de nombreux petits renseignements très utiles : noms de personnes compétentes dans chaque type de problèmes, leurs coordonnées, etc. ainsi que des données de type agenda (dates d'effet de telle mesure, de tel transfert, etc.). Et surtout, il est souhaité de pouvoir composer son renseignement à partir de modules, comme à partir d'une base de données. Cela implique une réorganisation générale de la documentation, une réécriture de la plupart des documentations existantes et la définition d'une grammaire efficace d'accès. Mais la conception d'une telle base documentaire, pour qu'elle soit bien adaptée à ceux qui auraient à s'en servir, suppose que l'on ait au préalable mieux étudié les besoins des utilisateurs en documentation, les types de requêtes, les chaînages dans les questionnements, etc. La plupart des interviewés sont d'avis qu'il faudrait pour cela que la documentation devienne une des préoccupations prioritaires dès la conception des programmes, ce qui n'est pas le cas.

## IV - CONCLUSION

En dépit du caractère imprécis et subjectif des renseignements recueillis, lié à la méthode des entretiens individuels, nous pouvons cependant retenir que des difficultés de compréhension des programmes existants constituent réellement une des difficultés du travail de programmeur de gestion et que les facteurs autour desquels les entretiens ont été centrés ont une réalité opérationnelle.

a) Il est clair que la formation joue un rôle capital. L'expérience dans le service et l'auto-apprentissage se surajoutent, mais l'insuffisance de la formation initiale en programmation explique d'une part le délai très important nécessaire pour que chaque programmeur se sente "à l'aise" pour faire face aux tâches qu'on lui confie et d'autre part

l'hétérogénéité des pratiques et des difficultés ressenties.
La formation devrait porter :

- sur le langage utilisé (COBOL et ses particularismes locaux),
- sur la conception et l'écriture des programmes,
- sur l'homogénéisation de certaines règles sociales de programmation (dénominations, commentaires,...),
- sur la lecture et l'approche analytique de programmes nouveaux,
- sur la recherche des pannes et, d'une façon générale, sur la maintenance technique.

Ceci apparaît avec une grande insistance dans la quasi-totalité des entretiens.

b) Sur la réalisation des programmes eux-mêmes, il y a peu à dire. Outre les améliorations d'écriture que l'on peut attendre provenant d'une formation plus homogène et plus systématique, la plupart des personnes interviewées s'accordent pour reconnaître les mérites de certains outils nouveaux qui ont été développés dans le service et qui facilitent à la fois l'écriture des programmes et leur homogénéisation; et donc leur compréhension grâce aux effets de la standardisation. Un point de discussion qui reste ouvert est la structuration des programmes. Plusieurs méthodes coexistent, qui ont chacune leurs partisans et leurs détracteurs.

c) Enfin, la documentation apparaît ne pas jouer le rôle qu'en attendent les différents utilisateurs potentiels. Le passage sur support informatique en accès conversationnel pourrait constituer une étape propice à une recherche plus approfondie sur son utilisation et les besoins différentiels, principalement pour la maintenance. On sait que c'est une question mal résolue d'une façon générale dont l'importance pratique apparaît de plus en plus à mesure que les logiciels se développent et se commercialisent.