Titre: Title:	Application des modèles de compréhension des programmes au pseudocode schématique
Auteur: Author:	Martin Truchon
Date:	1996
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Truchon, M. (1996). Application des modèles de compréhension des programmes au pseudocode schématique [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/9027/

UNIVERSITÉ DE MONTRÉAL

APPLICATION DES MODÈLES DE COMPRÉHENSION DES PROGRAMMES AU PSEUDOCODE SCHÉMATIQUE

MARTIN TRUCHON
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
AOÛT 1996

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

APPLICATION DES MODÈLES DE COMPRÉHENSION DES PROGRAMMES AU PSEUDOCODE SCHÉMATIQUE

présenté par: TRUCHON Martin

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BOUDREAULT Yves, M.Sc., président

M. ROBILLARD Pierre N., D.Sc., membre et directeur de recherche

M. ROBERT Jean-Marc, D.Sc., membre

REMERCIEMENTS

J'aimerais exprimer toute ma gratitude à tous ceux et celles qui m'ont encouragé et soutenu lors de la réalisation de ce mémoire.

Mes premiers remerciements vont tout d'abord à mes parents et à ma famille qui m'ont supporté et encouragé tout au long de mes études malgré la distance qui nous séparait. J'aimerais leur exprimer toute ma reconnaissance et mon affection, car c'est grâce à leur soutien que j'ai pu me rendre jusqu'au bout de cette aventure. Un merci tout spécial à mes anges gardiens, maman et grand-maman, qui veillent sur moi jour et nuit.

Je voudrais aussi remercier toute l'équipe du Laboratoire de Recherche en Génie Logiciel pour leur aide et leur collaboration. Particulièrement à MM. Benoit Lefebvre, François Trudel et Marc-André Poulin pour leurs conseils et leurs encouragements qui fûrent grandement appréciés.

Je ne manquerai pas de remercier chaleureusement mon directeur de recherche, M. Pierre N. Robillard, pour ses judicieux conseils ainsi que pour la confiance qu'il m'a témoignée durant ces deux années. Je lui suis également reconnaissant pour son aide financière.

RÉSUMÉ

Depuis environ vingt-cinq ans, les chercheurs tentent d'appliquer au développement de logiciels et au génie logiciel une foule de concepts appartenant au domaine de la psychologie cognitive. Ces travaux ont mené à l'élaboration de plusieurs modèles et théories expliquant en partie le comportement du programmeur et, entre autres, les processus de compréhension des programmes.

La compréhension des programmes a un impact majeur dans le développement des logiciels, et plus particulièrement dans la phase de maintenance, suite à la mise en opération de ceux-ci. Une foule d'outils ont été développés de façon tout à fait empirique, sans tenir compte de la psychologie du programmeur, croyant qu'ils pourraient faciliter la maintenance des logiciels. Quelques-uns apportent une certaine contribution, mais plusieurs ne sont qu'accessoires.

Parallèlement, le pseudocode schématique est une forme de représentation de l'information qui permet de visualiser graphiquement les structures de contrôle d'un programme indépendamment du langage de programmation utilisé. Il intègre le concept de raffinements successifs permettant de subdiviser la fonctionnalité en plusieurs niveaux d'abstraction. Il facilite aussi la documentation du code source. Sous plusieurs aspects, on peut observer des similitudes entre les structures du pseudocode schématique et certains concepts de la psychologie cognitive relatifs à la compréhension des programmes.

Les objectifs de ce mémoire sont donc: 1) de définir certains concepts utilisés dans les différents travaux sur la psychologie cognitive et la compréhension des programmes informatiques, 2) de répertorier les différents modèles de compréhension des

programmes et les expérimentations effectuées dans le domaine et 3) d'étudier comment on peut appliquer les résultats de ces travaux en utilisant le pseudocode schématique dans le développement des logiciels.

Plusieurs chercheurs ont tenté de comprendre les mécanismes mis en cause dans la compréhension des programmes informatiques. Nous avons répertorié cinq modèles couramment cités: 1) le modèle de Brooks, 2) le modèle de compréhension basé sur la connaissance de Letovsky, 3) le modèle de Pennington, 4) le modèle syntaxique/sémantique de Shneiderman/Mayer et 5) le modèle intégré de von Mayrhauser/Vans.

De plus, nous avons bâti un répertoire de plus de vingt-cinq expérimentations portant sur la compréhension des programmes dans le contexte de la psychologie cognitive. Chacune de ces expérimentations est classée selon la méthode expérimentale utilisée et l'expérience des sujets.

Finalement, nous avons discuté des possibilités d'application des résultats de ces expérimentations dans un contexte pratique et professionnel, en tenant compte de l'utilisation du pseudocode schématique comme support à cette intégration. L'utilisation de ce formalisme permettra sûrement une union efficace entre les théories sur la compréhension des programmes et la pratique professionnelle du génie logiciel.

ABSTRACT

For the past twenty-five years, researchers have been trying to adapt many concepts from cognitive psychology to software development and engineering. These works led to the development of different models and theories which explain programmer's behavior and also program comprehension process.

Program comprehension has a major impact in software development and most of all during maintenance stage. Many tools have been built in an empiric way without any concern for the programmer's psychology. Few of them contribute to the maintenance but, most of them are only accessories.

Schematic pseudocode is a way to visualise graphically structural control of a program without depending on a programming language. This representation integrates the concept of stepwise refinement that subdivises functionalities in many levels of abstraction. It also facilitates documentation of source code. In many ways, we can see similar concept between the structures of schematic pseudocode and some topics of cognitive psychology relating to program comprehension.

This research's objectives are: 1) to define certain concepts used by different works on cognitive psychology and program comprehension, 2) to classify different program comprehension models and experiments done in this field and 3) to study how these works' results can be applied by using schematic pseudocode in building software.

Many researchers have tried to understand mechanism involved in program comprehension. We have classified five well-known models: 1) Brooks' model, 2) Letovsky's knowledge-based understanding model, 3) Pennington's model, 4)

Shneiderman/Mayer's syntactic/semantic interaction model and 5) the integrated metamodel of code comprehension of von Mayrhauser/Vans.

Also, we have prepared a list which includes more than twenty-five experiments on program's comprehension in cognitive psychology. Each of these experiments is classified based on the experimental method used and on subjects' experience.

Finally, we have discussed different possibilities as how the experiments' results can be used in a practical and professionnal context, based on the use of schematic pseudocode as an integration support. In using this formalism will certainly allow a useful combination of program comprehension theories and professionnal practice of software engineering.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

INTRODUCTION

Bien malin celui ou celle qui pourrait affirmer ce qu'affiche le programme suivant lorsqu'on l'exécute:

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')()#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n'
')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dci@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

Ce code source, écrit en langage C, peut parfaitement être compilé et exécuté. Il a été intercepté sur une base de messages du célèbre réseau Internet. Ce programme affiche tout simplement, sur la sortie standard, une petite chansonnette intitulée «On the First Day of Christmas», composée de douze couplets dont le contenu de chacun répète celui du précédent en y ajoutant un élément. Le texte produit par le programme se retrouve à l'annexe I.

Après plusieurs minutes d'analyse, on pourrait probablement en arriver à comprendre une partie du code source. On peut remarquer dans celui-ci l'utilisation récursive de la fonction principale *main* ainsi que d'un paramètre ayant comme identificateur le symbole "_". De plus, comme le programme est appelé sans paramètre, la valeur du paramètre t prend la valeur "1" (convention du langage C) lors du premier appel à la fonction principale *main*. Enfin, on observe l'absence d'indentation, de commentaires et

de noms de variables significatifs. Bien qu'elles soient permises par le langage, ces pratiques ne sont pas très recommandables. Pourtant, ce programme ne contient qu'une vingtaine de lignes de code. Pourquoi éprouve-t-on tant de difficulté à en déchiffrer le contenu ?

Cet exemple peut sembler exagéré au premier abord. Par contre, si on se place dans le contexte d'un logiciel composé de plusieurs dizaines de milliers de lignes de code source, même si celui-ci respecte les principes de base de la "bonne" programmation, un programmeur peut avoir l'impression de se retrouver devant un programme ayant l'allure de ce qui est présenté au début de cette introduction.

Pourquoi a-t-on autant de difficulté à comprendre le code source composé par d'autres programmeurs ? Qu'est-ce qui fait qu'un programmeur novice a plus de difficulté à saisir la fonctionnalité d'un bout de code qu'un programmeur ayant plusieurs années d'expérience ? Quels processus mentaux font en sorte qu'un individu, à force de lire et de relire le code source d'un programme ainsi que la documentation l'accompagnant, en arrive à se faire une image du fonctionnement d'un logiciel ? Qu'est-ce qui fait que certains éléments comme les structures des langages de programmation, l'indentation, les commentaires ou encore la nomenclature des identificateurs facilitent la compréhension du code source ?

Depuis environ vingt-cinq ans, plusieurs chercheurs ont tenté de trouver des réponses à ces interrogations en réalisant différentes expérimentations et en proposant divers modèles illustrant les processus mentaux qui opérent lors de certaines tâches de programmation. Toutefois, plusieurs aspects de ces processus nous échappent encore, nous empêchant de comprendre adéquatement ces mécanismes.

Parallèlement, certaines études (Korbi, 1989) nous révèlent que la maintenance corrective, perfective et évolutive constitue environ 80 % du travail dans le développement de systèmes logiciels. De plus, entre 50 et 80 % des efforts de maintenance par les programmeurs servent à essayer de comprendre ce qui a été fait par leurs prédécesseurs et ce, malheureusement trop souvent avec grande difficulté. En se basant sur les aspects importants qui contribuent à la compréhension, peut-on concevoir des programmes qui tiennent compte de ces facteurs ?

La compréhension des programmes joue donc un rôle de premier plan dans presque toutes les activités de maintenance ainsi que dans plusieurs des phases initiales de la création d'un logiciel (Brooks, 1983). Voilà toute l'importance de maîtriser les processus de la compréhension des programmes et ce, dans le but de se doter d'outils tenant compte de ces mécanismes afin d'améliorer la compréhensibilité des programmes lors des tâches de maintenance.

En effet, les outils de développement existants sont bien souvent mal adaptés et supportent difficilement le processus mental du programmeur dans l'élaboration d'un logiciel, les éditeurs de texte servant à l'écriture du code source n'offrant qu'un support à la manipulation de caractères par toute sorte de fonctionnalités dont l'utilité est trop souvent discutable. Nous croyons qu'il est possible de concevoir des outils et des pratiques supportant et facilitant le cheminement mental du programmeur.

L'objectif principal de ce projet de recherche consiste donc à appliquer les concepts provenant des modèles de compréhension des programmes à la pratique du génie logiciel et, plus particulièrement, à l'utilisation du pseudocode schématique. Nous portons une attention marquée au pseudocode schématique, car il constitue la seule forme connue de représentation de design de ce genre qui permet une extension jusqu'au code source. Nous ne désirons toutefois pas discuter des outils existants qui utilisent le pseudocode

schématique, car nous voulons privilégier l'application des concepts reliés à la compréhension des programmes plutôt que la critique des outils disponibles sur le marché.

La stratégie utilisée sera de faire d'abord une classification et une synthèse des travaux reliés à l'étude des composantes contribuant à la compréhension des programmes, pour ensuite discuter des applications possibles dans la pratique du génie logiciel.

Nous constatons que les travaux effectués jusqu'à ce jour suivent deux approches. Certains chercheurs ont adopté une approche conceptuelle en élaborant des modèles de compréhension des programmes et en tentant de définir une foule de concepts, comme la théorie des schémas. D'autres chercheurs ont toutefois utilisé une approche expérimentale par la conduite d'expérimentations portant sur l'indentation ou l'utilisation de certaines structures de boucle, par exemple.

Bien que nous effectuions un survol des travaux utilisant ces deux approches, nous avons choisi de laisser de côté les aspects conceptuels dans l'application à la pratique du génie logiciel car le grand nombre d'opinions variées, et parfois contradictoires, ainsi que l'irrégularité du vocabulaire utilisé dans les travaux utilisant cette approche nous démontrent un manque de maturité et de consensus dans ce domaine relativement nouveau. Il reste encore du chemin à faire avant d'en arriver à des modèles reconnus et validés à partir desquels nous pourrons faire des prédictions. Nous nous attarderons donc plus en détails aux aspects expérimentaux.

Cette démarche nous a amené à constater que le pseudocode schématique procure plusieurs avantages qui pourraient être renforcés par les améliorations proposées. De plus, plusieurs autres aspects étudiés, indépendants du pseudocode schématique, auraient avantage à être plus utilisés dans la pratique professionnelle.

Plus concrètement, le premier chapitre de ce mémoire traite des concepts les plus courants en compréhension des programmes. Le deuxième chapitre traite un peu plus en détails des différents modèles, expérimentations et théories dans le domaine. Par la suite, le troisième chapitre introduit brièvement le pseudocode schématique et fournit l'exemple d'un court programme écrit selon cette notation. Enfin, le dernier chapitre vise à discuter de l'application pratique au génie logiciel des résultats des nombreux travaux sur la compréhension des programmes.

CHAPITRE I -

SCIENCES COGNITIVES ET GÉNIE DU LOGICIEL

Les objectifs de ce chapitre sont de montrer de quelle façon le sujet du présent projet de recherche s'inscrit dans la perspective du génie logiciel et de définir certains concepts utilisés dans les différents travaux sur la psychologie cognitive et la compréhension des programmes informatiques.

1.1 Les sciences cognitives

Il y a déjà plusieurs dizaines d'années que les sciences essaient de comprendre comment la connaissance est développée, représentée en mémoire et utilisée. Que ce soit pour expliquer les différences provoquées par l'expérience des individus ou par la tâche à accomplir, les chercheurs ont tenté de développer des modèles capables de rendre compte des mécanismes d'apprentissage, de création et de compréhension. L'apparition d'un nouveau champ d'études, que Tom Love appelle "La psychologie du logiciel" (*Software Psychology*), provient de la nécessité de comprendre de quelle façon les connaissances du programmeur sont acquises, emmagasinées et utilisés pour développer un logiciel.

Conscients de toutes les difficultés reliées au développement des logiciels de moyenne et grande envergure, les chercheurs tentent d'appliquer les connaissances de la psychologie cognitive au domaine du génie logiciel. En effet, il devient de plus en plus évident que nous ne pouvons pas faire abstraction des mécanismes reliés à la connaissance dans le développement de systèmes logiciels. Son invisibilité qui, selon Brooks (1986),

constitue une des propriétés essentielles du logiciel et l'absence de loi naturelle le régissant font en sorte que les processus de création et de compréhension du logiciel jouent un rôle primordial. Le développement accru d'outils de support à la programmation ne peut solutionner seul le problème si ceux-ci ne tiennent pas compte des aspects psychologiques de la création et de la compréhension des programmes lors du développement de logiciels.

Bien que l'application de la psychologie cognitive en génie logiciel puisse s'effectuer sur différents aspects allant du design jusqu'à la formation des professionnels, nous nous sommes restreint, dans le cadre de ce projet de recherche, aux aspects liés à la compréhension de programmes. Le Tableau 1.1 présente les tâches et activités reliées à la maintenance requérant la compréhension du code selon von Mayrhauser et Vans (1995). Comme nous pouvons le constater, la compréhension des programmes y est pour beaucoup.

Tableau 1.1 - Tâches et activités reliées à la maintenance des logiciels (von Mayrhauser et Vans, 1995)

Tâches	Activités
Maintenance adaptative	• Comprendre le système • Définir des spécifications • Réaliser le design préliminaire et détaillé • Modifier le code source • Déverminage • Tests de régression
Maintenance perfective	• Comprendre le système • Définir les spécifications concernant les améliorations • Réaliser le design préliminaire et détaillé des améliorations • Ajout et modification au code source • Déverminage • Tests de régression
Maintenance corrective	• Comprendre le système • Générer et évaluer des hypothèses concernant le problème • Corriger le code source • Tests de régression
Réutilisation	• Comprendre le problème et trouver une solution basée sur la réutilisation de composantes existantes • Trouver les composantes à réutiliser • Intégrer les composantes à réutiliser
Adaptation du code	• Comprendre le problème et trouver une solution basée sur des composantes prédéfinies • Réévaluer la solution afin d'augmenter la probabilité d'utiliser des composantes prédéfinies • Obtenir et modifier les composantes prédéfinies • Intégrer les composantes modifiées

La problèmatique étant exposé, nous sommes maintenant en mesure d'introduire certains éléments qui nous aideront à comprendre le répertoire présenté au chapitre II ainsi que les travaux repertoriés.

1.2 La théorie des schémas

La théorie des schémas porte sur l'organisation des connaissances en mémoire et sur les mécanismes de mise en oeuvre de ces connaissances (Détienne, 1988). Le concept de schéma a été introduit en 1932 par Bartlett et a été développé dans le domaine de l'intelligence artificielle et dans les études psychologiques sur la mémorisation des phrases.

Détienne (1988) définit un schéma comme un ensemble organisé de connaissances qui représente, sous forme générique, des concepts, des procédures, des événements ou des séquences d'événements. On peut s'imaginer le schéma comme étant un gabarit contenant des espaces vides mais dont le contenu est variable. Lors de l'activité de compréhension d'un programme, le schéma est instancié, c'est-à-dire que des valeurs particulières sont associées aux différents espaces vides qu'il contient. Par exemple, une structure en arbre ou un algorithme de tri sont des schémas. Lors de l'instanciation, on précisera s'il s'agit d'un arbre binaire ainsi que le contenu de ses feuilles. De la même façon, on pourra préciser l'index sur lequel s'effectue le tri.

Détienne (1988) énumère trois types de schémas:

- des schémas élémentaires représentant des connaissances sur la structure de contrôle et sur les variables;
- des schémas algorithmiques, représentant des connaissances sur des catégories d'algorithmes, qui sont eux-mêmes formés de schémas élémentaires;
- des schémas représentant des connaissances sur des types de problèmes particuliers, relatifs au domaine d'application.

À partir des études sur la mémorisation des phrases, on fait très souvent le rapprochement entre la compréhension de textes narratifs et la compréhension des programmes. On pose alors deux postulats:

1. les programmes informatiques peuvent être considérés comme des textes, avec cette particularité qu'ils sont écrits dans des langages formels;
2. les programmes informatiques, comme les textes narratifs, présentent certaines structures narratives.

Plusieurs études ont été effectuées sur la base de ces postulats (Soloway, Ehrlich, Bonar et Greenspan, 1982; Soloway et Ehrlich, 1984). Nous verrons les résultats de ces études dans le répertoire présenté au chapitre II.

L'utilisation d'expérimentations étant très courante dans les travaux reliés à la compréhension des programmes, nous discuterons, dans les prochaines pages, de certains paramètres de ces expérimentations.

1.3 Expérimentations

Le but des expérimentations en sciences est de mesurer ou d'évaluer certains attributs afin de vérifier des hypothèses préalablement énoncées, pour en arriver à formuler une théorie, la valider ou encore créer un modèle représentant le phénomène étudié. Dans le cas de la compréhension des programmes informatiques, on s'intéresse aux comportements des programmeurs et on cherche souvent à étudier les avantages et inconvénients de certaines structures syntaxiques et sémantiques des langages de programmation.

Dans ces expérimentations, on doit arriver à contrôler l'environnement de façon à réduire le nombre de variables indépendantes. On place alors un nombre restreint de sujets, ayant le plus souvent des niveaux d'expérience différents, dans une situation fictive où ils doivent accomplir une tâche bien particulière dans un temps déterminé. Le danger de ce type d'expérimentations est d'influencer le comportement normal des sujets par les contraintes imposées sur ceux-ci, ce qui peut faire en sorte qu'on n'observera pas les mêmes comportements que dans une situation réelle. On peut toutefois limiter ces effets en planifiant l'expérimentation de façon soignée.

1.3.1 Sujets

Étant donné le caractère psychologique du domaine et des projets de recherche, une expérimentation implique forcément l'utilisation de sujets et dans ce cas-ci, d'êtres humains. Selon le caractère et les objectifs de l'expérience, on cherche souvent à faire ressortir les effets d'un certain paramètre en fonction de l'expérience du sujet dans les domaines d'application ou de la programmation. Pour ce faire, les chercheurs font appel à des sujets ayant des niveaux d'expérience variés et les regroupent en un certain nombre de catégories.

Par exemple, on pourrait utiliser des étudiants inscrits à deux cours: un cours de baccalauréat et un cours au cycle supérieur de niveau maîtrise. Dans ce cas bien précis, on pourrait juger que les étudiants du cours de baccalauréat sont de niveau novice, tandis que les étudiants de niveau maîtrise sont plutôt du niveau expert, car ils possèdent une grande connaissance du langage de programmation utilisé dans l'expérience. Toutefois, un autre chercheur réalisant une expérimentation utilisant des professionnels et des étudiants comme candidats pourrait juger ces derniers comme appartenant au niveau novice même si certains de ceux-ci suivent des cours au cycle supérieur et possèdent une

connaissance profonde du langage de programmation. C'est une question de point de vue.

En fait, c'est ce que l'on observe dans la littérature. Les termes utilisés pour caractériser le niveau d'expérience des sujets varient d'un auteur à l'autre. Il est alors bien difficile de comparer deux expérimentations l'une par rapport à l'autre d'un seul coup d'oeil.

Pour cette raison, il serait souhaitable de développer une terminologie permettant de normaliser le niveau d'expérience des sujets. De plus, étant donné que le génie logiciel s'intéresse majoritairement au développement de systèmes logiciels par des professionnels, nous croyons que cette terminologie doit tenir compte du grand niveau d'expertise de ce milieu en évitant de classer, dans la catégorie des professionnels, des étudiants ayant complété trois cours de programmation. Shneidermann (1976) propose une nomenclature afin de catégoriser l'expérience des participants à ses expérimentations. Malheureusement, étant donné qu'aucun professionnel ne participe à celles-ci, cette nomenclature tient uniquement compte du niveau de scolarité des étudiants qui agissent comme sujets.

1.3.2 Taille des programmes étudiés

La taille des programmes informatiques utilisés dans les expérimentations varie très peu. D'après un tableau publié par von Mayrhauser et Vans (1995), la très grande majorité des programmes utilisés contient moins de neuf cents lignes de code source. Ces deux auteures affirment que la disponibilité des données provenant d'ingénieurs experts en logiciel représente tout un défi, à moins que les entreprises qui maintiennent du code de grande envergure encouragent leurs ingénieurs à participer à de telles expériences. On est aussi en droit de se demander si l'état actuel de la connaissance des processus cognitifs intervenant dans la compréhension des programmes ne nuit pas à l'utilisation

de programmes de taille comparable à ceux de l'industrie. et auxquels le génie logiciel s'intéresse particulièrement. Notre compréhension des processus cognitifs étant relativement limitée, il est sûrement plus simple, pour l'instant, de restreindre la taille des programmes étudiés dans les expérimentations. Toutefois, nous devrons en arriver à augmenter le volume des programmes utilisés dans les études afin d'être en mesure comprendre tous les impacts de l'utilisation des techniques de génie logiciel sur la compréhension des programmes.

1.3.3 Tâches et collectes de données

Différentes techniques sont utilisées afin de recueillir des données pertinentes dans les études menées. Certaines de ces techniques ont été développées par le passé, dans le but d'effectuer le même genre d'expérimentation, mais dans d'autres domaines d'application de la psychologie. On choisira le type d'expérimentation le plus approprié selon l'étude que l'on désire réaliser.

Les sections qui suivent décrivent les tâches expérimentales les plus couramment utilisées dans les expériences sur la compréhension des programmes. Shneiderman (1980) en fait d'ailleurs un très bon exposé.

1.3.3.1 Tâches de compréhension

Une tâche de compréhension consiste à soumettre le code source d'un programme à un groupe de sujets et à mesurer ou évaluer leur degré de compréhension de la fonctionnalité du programme. Cela est fait le plus souvent par un questionnaire composé de questions à choix multiples ou à réponses ouvertes. Shneiderman (1980) note que l'on doit s'assurer de mesurer tous les niveaux de compréhension d'un programme:

compréhension de chaque ligne de code, compréhension de l'algorithmie et des structures de données et compréhension du fonctionnement général du programme.

1.3.3.2 Tâches de réalisation

Sime, Green et Guest (1977), Spohrer et Soloway (1986) ainsi que Davies (1991) ont eu recours à des tâches de réalisation dans leurs travaux. Une tâche de réalisation consiste à soumettre aux sujets les spécifications d'une solution informatique à un problème en leur demandant de réaliser un programme qui implante cette solution. Malheureusement, il est très difficile d'évaluer le résultat, car il n'existe pas une seule et unique réponse. Pour faciliter le traitement des données après l'expérimentation, il est préférable de préparer à l'avance un ensemble de points sur lesquels portera l'évaluation.

1.3.3.3 Tâches de correction d'anomalies (*Debugging*)

Dans une tâche de correction d'anomalies, on demande aux sujets de localiser ou de corriger des anomalies introduites volontairement ou involontairement (corrigées au préalable) dans un programme. Il ne serait pas approprié d'utiliser des questions à choix multiples dans ce type d'expériences.

On peut fournir du matériel complémentaire au code source, comme un document de spécifications, des échantillons de résultats corrects, etc. On peut aussi se rapprocher de la réalité en ne fournissant pas le nombre d'anomalies présentes dans le code source.

1.3.3.4 Tâches de modification

La tâche de modification est probablement celle qui s'apparente le plus au travail de tous les jours puisque la maintenance adaptative et évolutive constitue une grande partie de la

tâche d'un ingénieur logiciel. On fournit habituellement aux sujets certains éléments de documentation ainsi que la liste des modifications que l'on désire effectuer. On évaluera le résultat de façon similaire à la tâche de réalisation. Le succès dans une tâche de modification peut constituer une excellente mesure du degré de compréhension.

1.3.3.5 "Remplir-la-case"

La méthode de collecte de données "Remplir-la-case" (de l'anglais "*Fill-in-the-blank*") s'appuie sur les tâches de réalisation et de compréhension. Elle consiste à compléter un programme dans lequel on a substitué certaines lignes par un espace blanc. Comme les sujets n'ont jamais vu le programme auparavant, ils n'ont pas à se rappeler, comme dans la technique de mémorisation, mais plutôt à utiliser leurs connaissances ainsi que les indices fournis par les lignes présentes. Soloway a abondamment utilisé cette technique dans ses travaux (Soloway et Ehrlich, 1984; Ehrlich et Soloway, 1984; Soloway, Adelson et Ehrlich, 1988; Détienne et Soloway,1990).

1.3.3.6 Verbalisation

La verbalisation est une technique de collecte de données basée sur une méthodologie développée par Newell et Simon (Newell et Simon, 1972), consistant à recueillir, à l'aide de divers moyens techniques (enregistrement audio et vidéo, par exemple), les propos, actions et expressions faciales des sujets (Adelson et Soloway, 1985). Souvent, on encourage même directement les sujets à s'exprimer à haute voix et en abondance, plus que ce qu'ils font en situation normale. On analyse ensuite les enregistrements afin d'en retirer des comportements typiques de programmeurs lors de l'exécution de ces tâches de programmation.

Détienne et Soloway (1990) ont utilisé cette technique en conjonction avec la méthode «Remplir-la-case» dans une de leurs expérimentations. Letovksy (1987), Sutcliffe et Maiden (1992) ainsi que Von Mayrhauser et Vans (1995) s'en sont également servis. Détienne (1988) a utilisé une variante de cette technique en présentant un programme, instruction par instruction, aux sujets et en leur demandant d'énoncer après chaque instruction les hypothèses sur le fonctionnement du programme.

1.3.3.7 Mémorisation

La mémorisation du code source d'un programme est considérée comme une mesure valable de sa compréhension (Shneiderman, 1976). On présente le code source d'un programme pendant un certain temps, puis on demande aux sujets de réécrire ce qu'ils ont vu comme dans une tâche de réalisation. Plusieurs auteurs ont eu recours à cette technique afin d'évaluer le degré de compréhension d'un programme par les sujets (Mynatt, 1984; Davies, 1994; Shneiderman, 1977; Wiedenbeck, 1986 et 1991). Cette technique est basée sur l'hypothèse voulant que l'on se rappelle mieux ce que l'on comprend.

1.3.3.8 Mesures subjectives

Les mesures subjectives s'ajoutent aux autres techniques présentées comme méthode de mesure de la performance des sujets. Il s'agit en fait de demander aux sujets une auto-évaluation de leurs performances à effectuer une tâche demandée, de leur compréhension ou encore de leurs préférences. Il est rare qu'on les utilise seules, car elles ont le fâcheux désavantage d'être difficilement reproductibles puisqu'elles dépendent de l'échantillon des sujets utilisés. Elles peuvent toutefois très bien compléter une épreuve de réalisation ou de compréhension en venant appuyer les autres observations.

1.4 Modélisation de la compréhension des programmes

Von Mayrhauser et Vans (1995) font un excellent inventaire des modèles de compréhension des programmes dans leur article intitulé «Program Comprehension During Software Maintenance and Evolution». Les deux auteures exposent clairement les concepts de base de ce type de modèle et présentent brièvement six modèles parmi les plus populaires et les plus discutés: le modèle de Letovsky, le modèle de Shneiderman et Mayer, le modèle de Brooks, le modèle de Soloway, Adelson et Ehrlich, le modèle de Pennington et celui de von Mayrhauser et Vans. Voici un bref aperçu des concepts de base utilisés dans l'élaboration de ces modèles.

Un modèle de compréhension des programmes consiste en une représentation simplifiée des processus mis en oeuvre par le programmeur, dans le but de comprendre un programme par l'étude de son code source et de sa documentation. Les modèles de compréhension sont considérés comme des éléments essentiels à la conception d'outils d'assistance aux activités de maintenance (Détienne, 1988). Les processus de compréhension utilisent des informations déjà connues par le programmeur (la connaissance) en vue d'acquérir de nouvelles informations qui permettront d'accomplir certaines tâches de programmation reliées majoritairement à la maintenance d'un système.

Tous les modèles de compréhension comprennent au moins les trois éléments suivants (von Mayrhauser et Vans, 1995):

1. une base de connaissance emmagasinée dans la mémoire à long terme;
2. une représentation mentale du code source contenue dans la mémoire de travail;

3. un processus combinant, dans une représentation mentale, les connaissances de la mémoire à long terme avec les nouvelles informations extraites des artefacts logiciels (code source et documentation).

1.4.1 Stockage des connaissances

La plupart des modèles de compréhension distingue la mémoire à court terme et la mémoire à long terme (Curtis, 1984). La mémoire à court terme a une capacité limitée de 7 ± 2 items selon Miller (1975). Elle contient les éléments d'information sur lesquels le programmeur porte son attention. Le processus d'agglomération ("chunking") permet de circonvenir les limites de cette mémoire à court terme.

On considère généralement que la mémoire à long terme a une capacité illimitée. Les effets de l'expérience et de la formation se font ressentir sur la mémoire à long terme. Il ne suffit pas d'accumuler des concepts pour bâtir cette base de connaissance, mais bien d'en arriver à l'organiser efficacement.

Von Mayrhauser et Vans (1995) identifient deux types de connaissance: la connaissance générale et la connaissance spécifique d'un logiciel. La connaissance générale est principalement constituée d'informations concernant les langages de programmation, les principes de programmation, les algorithmes solutionnant des problèmes génériques, etc. La connaissance spécifique est pour sa part reliée à la connaissance d'un programme en particulier.

Soloway, Adelson et Ehrlich (1988) suggèrent une subdivision de cette connaissance générale en deux autres types. D'abord, les schémas de programmation composés de fragments de programme représentant des séquences d'actions typiques (boucle de recherche, boucle d'itération, etc.). Il y a ensuite ce qu'ils appellent les règles de

programmation qui spécifient certaines conventions de programmation telles que la signification des identificateurs. Shneidermann et Mayer (1979) divisent plutôt cette connaissance générale en deux domaines: sémantique et syntaxique. La connaissance sémantique est indépendante des langages de programmation et décrit les concepts généraux de la programmation (fonctionnement d'une boucle, boucle de recherche, etc.), alors que la connaissance syntaxique concerne l'information spécifique aux langages de programmation.

1.4.2 Modèle mental du programme

Le modèle mental est une des composantes que l'on retrouve dans la majorité des modèles de compréhension des programmes. On le définit comme étant la représentation interne du logiciel qui fait l'objet de la tâche de compréhension. Il peut représenter plus ou moins fidèlement les caractéristiques réelles du système logiciel selon que la compréhension est bonne ou mauvaise. Bien entendu, plus le modèle mental est complet et véridique, plus il pourra supporter efficacement les tâches de programmation accomplies par le programmeur.

1.4.2.1 Éléments statiques du modèle mental

Le modèle mental est composé de divers éléments statiques tels des structures textuelles, des agglomérations d'énoncés ("*chunks*"), des schémas, des balises ("*beacons*"), des hypothèses et des règles de programmation.

Pennington (1987) utilise le concept des structures textuelles pour expliquer la connaissance du flux de contrôle. Ces structures sont en fait des segments de code source organisés selon une certaine structure (exemple: déclarations de variables, structures conditionnelles).

Les agglomérations ("*chunks*") sont des macrostructures contenant des structures textuelles à plusieurs niveaux d'abstraction. Von Mayrhauser et Vans (1995) donnent comme exemple la microstructure que constitueraient les énoncés d'un algorithme de tri en bulle, alors que la macrostructure serait une abstraction de ce bloc de code, identifiée simplement par l'étiquette «tri en bulle».

Brooks (1983) et Letovsky (1987) traitent du concept d'hypothèses dans leurs travaux. Letovsky les définit comme étant le résultat de l'activité de compréhension. Brooks considère, lui, que la compréhension n'est totale qu'au moment où le modèle mental contient une hiérarchie complète d'hypothèses vérifiées par rapport au code source.

1.4.2.2 Éléments dynamiques du modèle mental

Selon Von Mayrhauser et Vans (1995), une stratégie de travail pour comprendre un programme sert à établir les correspondances entre les schémas de programmation qui font partie des connaissances du programmeur et les artefacts du logiciel (code source et documentation). Elle dirige les actions du programmeur afin qu'il atteigne le but de la tâche de programmation. Par exemple, la stratégie employée par le programmeur pourrait être de lire le code source, ligne par ligne, afin d'en arriver à une bonne compréhension d'un certain bout de code.

On rapporte deux processus de compréhension qui sont guidés par ces stratégies. Tout d'abord, le processus d'agglomération construit des structures plus abstraites à partir de structures moins abstraites, en substituant le détail de ces dernières par une étiquette qui résume le contenu de celles-ci. Finalement, le processus des références croisées permet de relier l'information appartenant à des niveaux d'abstractions différents (exemple: une partie du programme à sa description fonctionnelle).

1.5 Résumé du chapitre

Dans ce chapitre, nous avons vu que la compréhension des programmes informatiques occupe une grande partie de l'ensemble des tâches accomplies dans le développement de logiciels et, plus particulièrement, durant la phase de maintenance. Depuis plusieurs années, des chercheurs tentent de faire la lumière sur les processus reliés à la compréhension des programmes.

Dans cette optique, plusieurs théories et concepts ont été développés, comme la théorie des schémas et les modèles de compréhension. De plus, plusieurs expérimentations ont été faites afin de valider ces concepts. Nous allons voir dans le chapitre suivant le détail de ces modèles et de ces expérimentations.

CHAPITRE II -

ASPECTS COGNITIFS DE LA COMPRÉHENSION DES PROGRAMMES

L'objectif de ce chapitre est de présenter une revue de différents travaux portant sur la compréhension des programmes. Le tout est divisé en trois parties: la première présente des modèles de compréhension des programmes, la seconde rapporte des résultats d'expérimentations et la dernière traite de divers aspects non couverts par les deux précédentes.

2.1 Présentation du répertoire

Après avoir présenté brièvement le domaine de recherche et abordé quelques concepts de base reliés à celui-ci, nous présentons les différents travaux portant sur la compréhension des programmes informatiques que nous avons répertoriés dans la littérature. Nous avons retenu principalement ceux qui traitent des modèles de compréhension des programmes, qui rapportent des expérimentations ou qui élaborent des théories en s'appuyant sur ces modèles. Ce type d'articles se retrouve majoritairement dans les publications traitant de psychologie appliquée au domaine de l'informatique (ex. International Journal of Human-Computer Studies).

2.2 Des modèles de compréhension des programmes

Nous présentons dans cette section le contenu de cinq modèles de compréhension des programmes qui ont été abondamment discutés et cités par différents auteurs dans la littérature.

Tableau 2.1 - Modèles de compréhension des programmes

Auteurs	Modèle connu sous le nom de	Modèle basé sur expérimentation
Brooks (1983)	-	Non
Letovsky (1987)	Modèle de compréhension basé sur la connaissance	Oui
von Mayrhauser et Vans (1995)	Modèle intégré de compréhension du code	Oui
Pennington (1987)	-	Oui
Shneiderman et Mayer (1979)	Modèle syntaxique/sémantique	Non

Chacun de ces modèles est présenté dans les sections suivantes selon l'ordre chronologique de leur publication.

2.2.1 Modèle syntaxique/sémantique de Shneiderman et Mayer

C'est sûrement le modèle le plus populaire et le plus cité dans la littérature. Shneiderman et Mayer (1979) ont reproché principalement aux études, à cette époque, de ne s'attarder qu'à des aspects spécifiques de la tâche de programmation en évitant de produire des modèles pouvant tenir compte du comportement du programmeur dans son ensemble. Selon eux, un tel modèle devrait pouvoir tenir compte des structures et des processus cognitifs mis en jeu dans chacune des cinq tâches de programmation suivantes: réalisation, compréhension, modification, correction d'erreurs et apprentissage. Le modèle proposé par Shneiderman et Mayer tient compte, lui, de l'ensemble de ces tâches.

Shneiderman et Mayer s'inspirent d'une étude, récente à cette époque (Greeno, 1973), en proposant la structure de mémoire illustrée à la Figure 2.1: premièrement, une mémoire à court terme qui capte l'information provenant de l'extérieur et y effectue une analyse très sommaire. Cette mémoire ne peut conserver que très peu d'informations (7 items ± 2), comme le propose Miller (1956) dans son célèbre article. Par conséquent, elle a durée de persistance très courte. Deuxièmement, une mémoire à long terme dans laquelle le programmeur conserve ses connaissances permanentes avec une capacité illimitée. Troisièmement, une mémoire de travail dans laquelle les nouvelles informations contenues dans la mémoire à court terme et des éléments de la mémoire à long terme sont intégrés en de nouvelles structures. La durée de persistance de cette mémoire se situe entre la mémoire à long terme et celle à court terme.

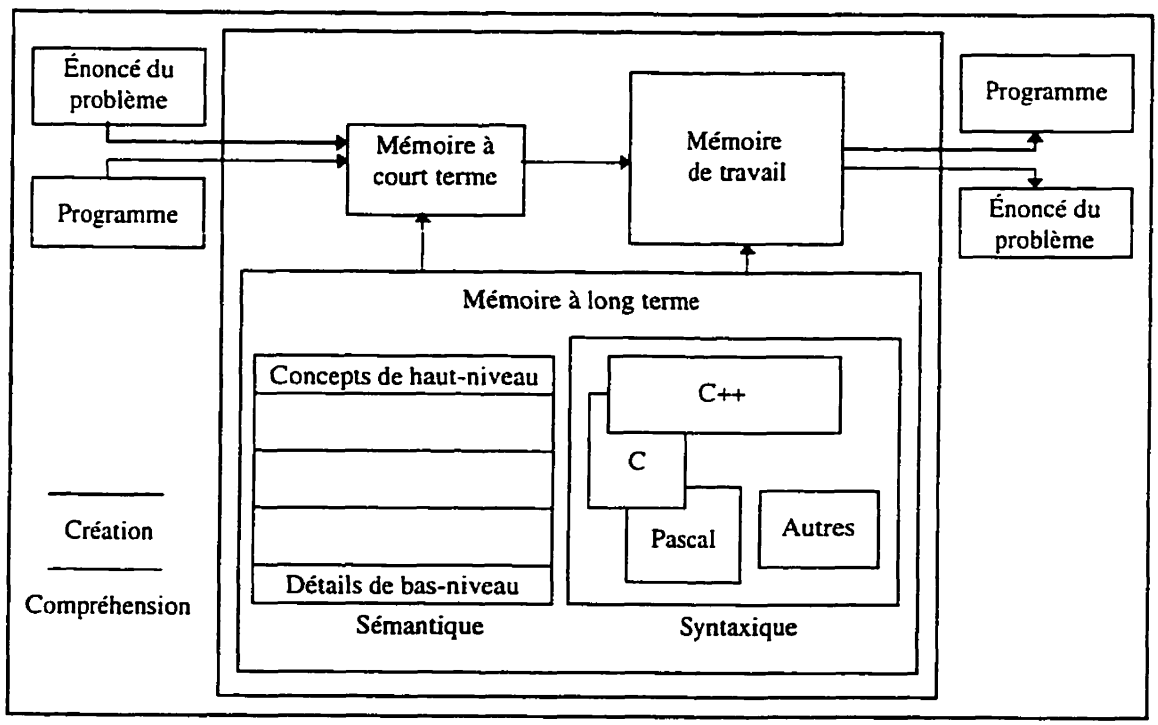


Figure 2.1 - Modèle de Shneiderman et Mayer (von Mayrhauser et Vans, 1995)

Deux types d'informations concernant les techniques et concepts de programmation se retrouvent dans la mémoire à long terme. Premièrement, la connaissance sémantique qui consiste en divers concepts généraux de programmation, indépendants du langage de programmation utilisé. Cette connaissancs peut être de bas niveau, comme les types élémentaires de données, allant jusqu'à des connaissances de haut niveau, comme les méthodes de tri. Deuxièmement, la connaissance syntaxique est plus détaillée et précise mais plus volatile que la connaissance sémantique. Elle regroupe de l'information concernant les structures de contrôle et les énoncés des différents langages de programmation.

Shneiderman présente aussi comment son modèle s'applique aux différentes tâches de programmation, comme la création de programmes, la modification et la correction d'erreurs dans ceux-ci et l'apprentissage de nouveaux concepts de programmation. Toutefois, nous nous restreindrons ici à la partie compréhension de programmes de ce modèle. Il est important de noter que la compréhension de programmes constitue une partie majeure de toutes les autres tâches de programmation, car le programmeur doit d'abord comprendre et se faire un modèle mental du programme avant d'être en mesure de le modifier. Ce modèle mental est en fait une structure sémantique à plusieurs niveaux. Au niveau le plus élevé, on retrouve les objectifs du programme. À l'opposé, au niveau sémantique le plus bas, le programmeur regroupe certaines séquences d'énoncés ou des algorithmes qu'il reconnaît.

Les processus cognitifs font en sorte que le programmeur ne peut comprendre un programme seulement à partir de la syntaxe et en lisant le code source, ligne par ligne. Il construit plutôt une structure sémantique interne d'une manière semblable au processus d'agglomération (*chunking*) décrit par Miller (1956). Le programmeur reconnaît la fonctionnalité accomplie par un groupe d'énoncés et les regroupe en un bloc. À leur tour, ces blocs seront assemblés pour former d'autres blocs de plus en plus gros, jusqu'à

ce que le programme complet soit compris. La structure sémantique résultante est relativement bien gravée en mémoire et permet, par exemple, d'expliquer à d'autres gens et avec aise, les tenants et aboutissants du programme. Shneiderman soutient que le processus d'agglomération est mieux supporté par la programmation structurée (sans "*goto*"), mais cet argument demanderait à être raffiné et développé.

2.2.2 Modèle de Brooks

Brooks (1983) fait très bien ressortir toute l'importance de l'activité de compréhension de programmes en notant que celle-ci joue un rôle majeur dans plusieurs tâches liées au développement d'un logiciel et, plus particulièrement, pendant la phase de maintenance. De plus, bon nombre d'activités de tests et de contrôle de qualité sont basées sur la compréhension de programmes, comme les techniques de walkthrough et d'inspection[1]. Voilà la nécessité de fournir des mécanismes expliquant les aspects les plus importants de la compréhension de programmes.

Brooks décrit le processus de création de logiciels comme étant l'élaboration d'une correspondance entre les éléments des divers domaines du problème et les éléments du domaine de la programmation. À l'opposé, la tâche de compréhension en sera une de reconstruction de l'information que le programmeur a utilisée pour faire le lien entre le problème et le programme résultant.

---

[1] Le walkthrough permet, par la simulation de la tâche à réaliser, de trouver les défauts par la vérification de l'exactitude des spécifications, de la logique dans les différents diagrammes, de la pertinence des structures de données ou de la bonne implantation des algorithmes utilisés. Pour sa part, l'inspection permet, par une vérification serrée des documents, de trouver les défauts portant autant sur le design que sur le code. Elle permet de s'assurer que toutes les **normes** et les **lignes de conduites** sont respectées.

L'approche adoptée par Brooks utilise le raffinement successif descendant ("*top-down*") d'hypothèses sur les domaines du problème et établit des relations entre ces hypothèses et les différents artefacts logiciels. La Figure 2.2 illustre cette approche. Aussitôt que le programmeur connaît le but du programme étudié, il est en mesure d'exprimer une première hypothèse. Afin de vérifier la validité de celle-ci, il doit trouver un segment de code ou un élément de documentation qui confirme cette hypothèse.

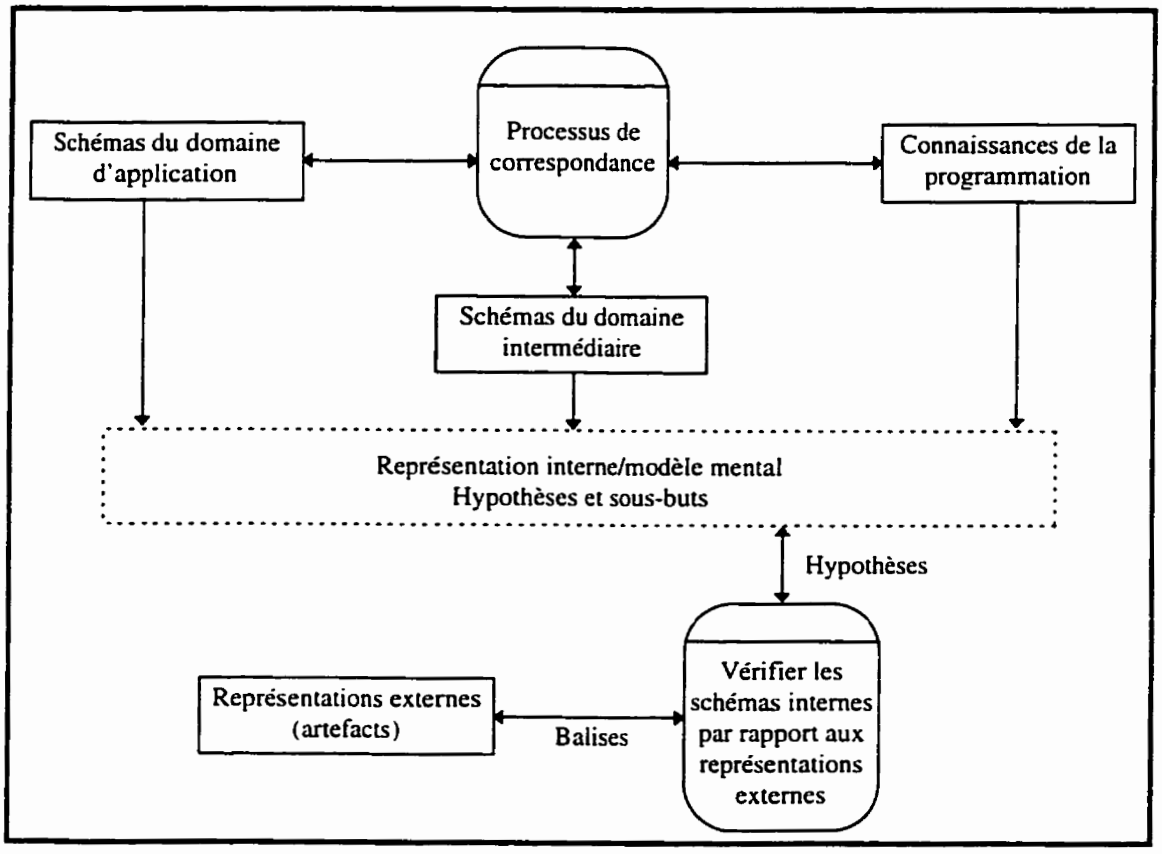


Figure 2.2 - Modèle de Brooks (von Mayrhauser et Vans, 1995)

Comme la première hypothèse est habituellement très globale et vague, on doit énoncer des hypothèses subordonnées additionnelles, selon une structure hiérarchique: des hypothèses les plus abstraites vers les hypothèses de plus en plus spécifiques et concrètes. Les éléments au bas de cette hiérarchie sont habituellement suffisamment détaillés pour pouvoir être associés à des éléments du code ou à d'autres formes de

documentation. Ces éléments portent le nom de balise (en anglais "*beacon*"). Le processus de vérification des hypothèses consiste précisément à trouver des balises qui confirment l'hypothèse en question. Différentes stratégies peuvent être utilisées à cette fin. La découverte d'une balise importante confirme l'hypothèse et permet d'élaborer de nouvelles hypothèses à partir de celle-ci.

Il se peut que ce processus de génération d'hypothèses échoue. Par exemple, le programmeur peut être incapable de trouver une balise permettant de confirmer son hypothèse ou, encore, il peut avoir associé le même fragment de code à deux hypothèses différentes. Le programmeur devra alors revenir sur son hypothèse et possiblement remonter la hiérarchie afin de modifier une hypothèse qui semble erronée.

2.2.3 Modèle de compréhension basé sur la connaissance de Letovsky

Ce modèle de compréhension des programmes, illustré à la Figure 2.3, a été élaboré par Letovsky (1987) à partir des résultats d'une expérience de verbalisation dans laquelle les programmeurs étaient invités à réfléchir à haute voix pendant une tâche de maintenance évolutive d'un programme, une telle activité requérant, bien entendu, que le programmeur comprenne celui-ci. Plusieurs types de comportements ont été identifiés, mais Letovsky a retenu les deux principaux dont il a tenu compte dans son modèle: le questionnement et l'expression d'hypothèses.

On peut diviser le modèle selon les trois composantes suivantes:

1. Une base de connaissance qui est principalement constituée de l'expérience du programmeur impliqué dans la tâche de compréhension.
2. Un modèle mental du programme étudié. Ce modèle sera appelé à évoluer pendant le processus de compréhension.

3. Un processus d'assimilation qui fait le lien entre les artefacts (code source, documentation, etc.) et la base de connaissance afin de bâtir le modèle mental.

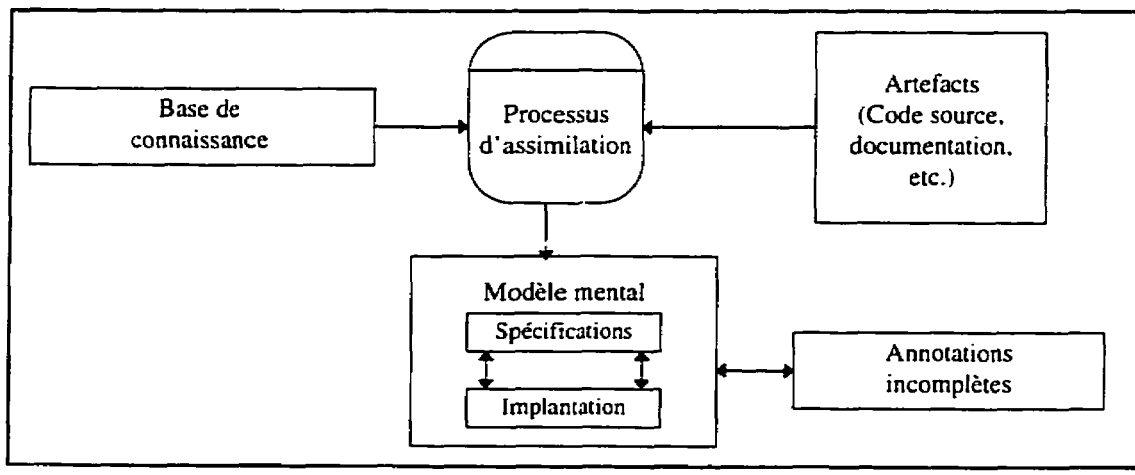


Figure 2.3 - Modèle de Letovsky (von Mayrhauser et Vans, 1995)

La base de connaissance comprend six types de connaissance qui ont été identifiés par différentes études en intelligence artificielle ou par des méthodes empiriques:

- sémantique des langages de programmation;
- objectifs (ex. recherche ou tri dans un vecteur);
- schémas (solutions à des problèmes déjà rencontrés);
- connaissance du bon fonctionnement;
- connaissance des domaines d'application;
- conventions de programmation.

Le modèle mental du programme étudié se bâtit tout au long du processus de compréhension. Il ne sera complet que lorsque toutes les spécifications pourront être reliées à l'implantation, c'est-à-dire à une action ou à une structure de données dans le code source.

Le processus d'assimilation est plus complexe. Il doit construire les liens entre les spécifications du programme et son implantation à partir des artefacts et de la connaissance du programmeur. Il peut procéder par un processus ascendant ("*bottom-up*") en construisant les annotations à partir de l'implantation et en reconnaissant les schémas utilisés ou, à l'inverse, représenter d'abord les spécifications et, par une approche descendante ("*top-down*"), développer leurs implantations possibles pour ultimement les faire correspondre avec le code source. Letovsky considère que la compréhension de programmes utilise avantageusement ces deux approches en alternant de l'une à l'autre au fur et à mesure que l'information devient disponible.

Une hypothèse importante dans le modèle de Letovsky repose sur le fait que la personne qui étudie un programme dans le but de le comprendre évaluera continuellement, pendant le processus, la cohérence et la complétude du modèle mental partiel du programme. C'est cette évaluation qui fait surgir les différentes questions concernant les parties nébuleuses du modèle mental. Letovsky identifie quatre types de questions:

1. "Comment et pourquoi": ce type de questions survient lorsqu'une portion du modèle mental est incomplète. Les questions de type "Comment" sont générées de façon descendante ("*top-down*") à partir des spécifications, tandis que les questions "Pourquoi" sont générées de façon ascendante ("*bottom-up*") à partir des artefacts logiciels.
2. "Qu'est-ce ?": on se pose ce type de question lorsqu'on rencontre une référence à une variable, routine ou type de données inconnus.
3. "Doute": on devra répondre à ce type de question s'il y a des ambiguïtés dans les annotations entre les spécifications et les artefacts logiciels.
4. "Divergence": ce type de question témoigne de contradictions dans le modèle mental.

Ces questions donneront souvent accès à des réponses à l'aide de processus de conjectures, en suggérant des solutions possibles à celles-ci. Ces solutions deviendront éventuellement des conclusions.

2.2.4 Modèle de Pennington

Selon le modèle de Pennington (1987), lors de la compréhension d'un programme, le programmeur développe deux représentations mentales différentes de celui-ci. D'abord, le programmeur se bâtit une abstraction du flot de contrôle, que Pennington appelle modèle du programme. Celui-ci est construit à partir de balises représentant des blocs de contrôle selon une approche ascendante. La construction de ce modèle du programme repose sur des processus d'agglomération de microstructures, constituées d'énoncés pour former des macrostructures de plus haut niveau, et sur la connaissance des schémas de programmation.

Ensuite, à partir d'une représentation abstraite du flot de données et de la fonctionnalité, le programmeur se construit un modèle contextuel via une approche ascendante en se servant d'informations provenant du modèle du programme. La connaissance des schémas du domaine lui sert à se représenter mentalement des éléments du code source en terme de concepts du monde réel.

2.2.5 Modèle intégré de von Mayrhauser et Vans

À la base de ce modèle, von Mayhrauser et Vans (1995) soutiennent que la maintenance fait en sorte que le programmeur alterne très souvent entre une approche de compréhension descendante et une approche ascendante. De plus, elles croient que l'utilisation de la connaissance durant la phase de maintenance correspond à une activité

de réingénierie qui reconstruit le design de haut niveau à partir du code source du programme. Le modèle intégré de von Mayrhauser et Vans s'inspire largement des travaux de Soloway et Ehrlich, pour ce qui est de l'approche, et du modèle de Pennington, pour les modèles contextuel et du programme.

Le modèle proposé par von Mayrhauser et Vans pour expliquer les processus cognitifs mis en jeu dans la compréhension d'un programme informatique se divise en quatre parties. Les trois premières parties sont des processus de compréhension accompagnés de leur représentation interne dans la mémoire à court terme: le modèle du programme ("*program model*"), le modèle contextuel ("*situation model*"), le modèle du domaine ("*domain model*"). La dernière partie, la base de connaissance ("*knowledge base*"), est nécessaire afin de bâtir les trois modèles précédents.

Dans le cadre de ce modèle de compréhension, le modèle du programme consistant en une abstraction du flot de contrôle du programme est la première représentation mentale que le programmeur se construit lorsque le code est complètement nouveau. Contrairement au modèle du domaine, le modèle du programme est bâti de façon systématique, en étudiant le flux de contrôle et de données.

Pour ce qui est du modèle contextuel, on peut le construire à partir d'approches opportuniste ou systématique selon la situation. Ce modèle est développé une fois que le modèle mental du programme est construit. Il représente une abstraction fonctionnelle du flot de données et de la fonctionnalité du programme. Par exemple, un programmeur expérimenté qui connaît bien le domaine d'application cherchera dans le code des indices afin de trouver la partie du programme responsable d'une certaine fonctionnalité, mais lorsqu'il l'aura trouvé, il ne lui sera pas nécessaire de lire ligne par ligne le code source pour comprendre. Il connaît déjà le contexte dans lequel cette fonctionnalité est employée et comprend déjà son fonctionnement.

Un modèle du domaine est formé lorsque le programmeur est familier avec le code ou le style de programmation. La connaissance utilisée dans ce modèle consiste en une connaissance des domaines d'application et a été acquise lors d'expériences précédentes. Elle est nécessaire pour une décomposition descendante d'un système en ses composantes et utilise une approche opportuniste de compréhension, c'est-à-dire le programmeur ne parcourt pas le code source ligne par ligne, mais recherche plutôt certains éléments d'algorithmie ou de structures de données en particulier.

La base de connaissance (mémoire à long terme) essentielle aux processus de compréhension se compose de schémas tels que présentés à la section 0. Selon les auteures de ce modèle, les schémas seraient regroupés en partition selon le processus de compréhension qui les utilise.

2.3 Expérimentations

Un grand nombre d'expérimentations sur la compréhension des programmes ont été réalisées par les chercheurs, surtout durant les années 1980. Par ces expériences, ils essaient de vérifier certaines hypothèses sur le comportement des programmeurs ou encore de confirmer les théories et les modèles qu'ils élaborent. Cette partie constitue l'approche expérimentale de la compréhension des programmes.

2.3.1 Sujets

Comme nous l'avons mentionné auparavant, dans la section 1.3.1, nous considérons qu'il est important d'utiliser une terminologie normalisée pour désigner le niveau d'expertise des sujets utilisés dans les expérimentations. Dans le Tableau 2.2, nous proposons donc une nomenclature composée de quatre niveaux d'expertise. Celle-ci est

utilisée dans le répertoire afin de pouvoir comparer convenablement le niveau d'expérience des sujets. Nous avons donc déduit le niveau d'expérience des sujets à partir des informations fournies par l'auteur.

Tableau 2.2 - Nomenclature des niveaux d'expérience des sujets

Niveau	**Description**
Étudiants	Personnes qui en sont à leur premier cours de programmation. Elles sont en processus d'apprentissage
Étudiants avancés	Personnes ayant complété au moins un cours de programmation et qui ont une expérience limitée de la programmation.
Professionnels	Personnes dont les tâches de programmation constituent leur travail de tous les jours.
Experts	Professionnels ayant au moins deux ans d'expérience dans le domaine.

Dans les divers travaux répertoriés, nous tenons à faire ressortir la volonté des chercheurs d'effectuer leurs expérimentations sur des sujets ayant des niveaux d'expérience différents. C'est ce que l'on appelle l'aspect novice/expert de l'expérimentation.

2.3.2 Méthode de classification

Von Mayrhauser et Vans (1995) ont effectué un exercice de classement des expérimentations, similaire au nôtre, dans leur article intitulé «Program Understanding: Models and Experiments». Dans celui-ci, elles répertorient les cinq modèles de compréhension des programmes présentés précédemment ainsi que quelques dizaines d'expérimentations sur le sujet. Pour construire le Tableau 2.4, nous nous sommes inspiré de la formule de présentation d'un tableau apparaissant dans leur article et

montrant le sujet des expérimentations par rapport aux attributs de celles-ci. À la manière de von Mayrhauser et Vans, on retrouve chacune des expérimentations placées dans un tableau où chaque colonne représente un thème relié à la compréhension des programmes, et chaque ligne représente un attribut expérimental.

Contrairement à von Mayrhauser et Vans, nous n'avons pas indiqué la taille du code source utilisé ni le langage de programmation. Nous indiquons plutôt la tâche ou la technique de collecte de données utilisée dans chacune des expérimentations. Celles-ci ont toutes été présentées dans la section 1.3.3. De plus, nous avons ajouté le niveau d'expérience des sujets tel que défini dans le Tableau 2.2. Enfin, nous indiquons, dans la section Novice/Expert, si l'auteur de l'expérimentation compare ou non les résultats en fonction de l'expérience des sujets.

Le Tableau 2.3 contient la liste des travaux qui ont été classés dans le Tableau 2.4. Notons qu'une expérimentation peut se retrouver dans plusieurs colonnes si plusieurs thèmes sont abordés par celle-ci.

Le Tableau 2.4 démontre bien l'intérêt pour l'étude de la compréhension des programmes, la grande diversité des approches utilisées (tâches et techniques de collecte de données) ainsi que la plage étendue d'expérience des candidats. Ces facteurs nous indiquent que la compréhension des programmes informatiques est effectivement un processus complexe qui ne peut être facilement cerné. Malgré le travail réalisé à ce jour, le problème n'est pas encore résolu.

Tableau 2.3 - Expérimentations répertoriées

a) Adelson et Soloway (1985)	i) Gilmore et Green (1988)	q) Sime et al. (1977)
b) Davies (1990)	j) Iselin (1988)	r) Soloway, Bonar et Ehrlich (1983)
c) Davies (1991)	k) Kesler et al. (1984)	s) Soloway et al. (1983)
d) Davies (1994)	l) Khalil et Clark (1989)	t) Soloway et Ehrlich (1984)
e) Détienne (1988)	m) Miara et al. (1983)	u) Teasley (1994)
f) Détienne et Soloway (1990)	n) Mynatt (1984)	v) Wiedenbeck (1986)
g) Ebrahimi (1994)	o) Robertson et Yu (1990)	w) Wiedenbeck (1991)
h) Ehrlich et Soloway (1984)	p) Shneiderman (1977)	x) Wiedenbeck et al. (1993)

Tableau 2.4 - Attributs et contenu des expérimentations

Attributs		Thèmes									
		Schémas de programmation	Règles de programmation	Balises	Structures d'un langage de programmation	Stratégies cognitives	Connaissances	Modèles mentaux	Processus mentaux	Complexité cognitive	Styles de programmation
Tâche ou technique de collecte de données	Compréhension	g			g, j			x		l	k, m, u
	Réalisation	g, s			g, q, r	c, r					
	Fill-in-the-blank	f, t	t			f	h				
	Correction	b, i									
	Modification									l	
	Mémorisation	b, t	t	v, w			d, h	x	p	l, n	
	Verbalisation	f, t				a, f			e		
	Mesures subjectives										m
	Autres	o									
Expérience des sujets	Étudiants	s, t	t		q, r	c, r			p	l	k, m
	Étudiants avancés	b, f, g, i, o, s, t	t	v, w	g, j, r	c, f, r	d, h	x		l, n	m, u
	Professionnels			w	j, r	a, r	d				m
	Experts				j, r	a, c, r		x	e		m
Novice/ Expert	Traité	s, t	t	v, w	j, r	a, c, r	d, h	x			m, u
	Non-traité	b, f, g, i, o			g, q	f			e, p	l, n	k

2.3.3 Condensé des résultats des expérimentations répertoriées

Cette section présente un sommaire des thèmes abordés et des résultats présentés dans divers articles, rapportant des expérimentations reliées à la compréhension des programmes informatiques.

2.3.3.1 Les schémas et règles de programmation

Les règles de programmation (Soloway et Ehrlich, 1984) correspondent aux conventions que les programmeurs respectent lors de l'écriture d'un programme telles l'indentation et la nomenclature des variables. Ces règles jouent le même rôle que les balises, de sorte qu'il peut être difficile de les différencier.

Soloway et Ehrlich (1984) (t) ont fait une expérimentation tout à fait intéressante. Celle-ci est basée sur deux types de connaissances qu'ils définissent de la façon suivante:

- schémas de programmation ("*programming plans*"): fragments de programmes représentant des séquences d'actions typiques en programmation. Exemples: Parcours d'un tableau, tri d'un tableau de données.
- règles de programmation ("*rules of programming discourse*"): règles qui spécifient certaines conventions de programmation. Exemples: nomenclature des identificateurs, organisation logique des fichiers.

Le but poursuivi par Soloway et Ehrlich est de déterminer si les programmeurs experts possèdent ces deux types de connaissance. Dans l'affirmative, un programme ne respectant pas ces schémas et ces règles devrait être plus difficile à comprendre. On s'attend aussi à ce que les programmeurs débutants ne possèdent pas ces connaissances. Ils devraient donc être moins influencés par un programme ne respectant pas celles-ci.

Finalement, on s'attendra à ce que les programmeurs experts performent de façon semblable aux débutants face à des programmes ne respectant pas ces règles.

Il est intéressant de remarquer que les programmes utilisés dans l'expérimentation comportent deux versions: l'une respecte les types de connaissances énumérés ci-haut et l'autre ne les respecte pas, les deux versions ayant la même taille. Si on se réfère aux méthodes de calcul de la complexité des programmes proposées dans la littérature (métrique de Halstead (1977), nombre de lignes de code), les deux versions de ce même programme sont de complexité semblable. Et pourtant, Soloway et Ehrlich démontrent une différence marquée dans la compréhension entre les deux versions. On peut donc s'attendre à ce que la complexité des programmes soit influencée par certains aspects cognitifs dont les méthodes classiques de calcul de la complexité ne tiennent aucunement compte.

Les auteurs concluent que les programmeurs utilisent leur connaissance des schémas et des règles de programmation dans une tâche de compréhension de programmes. Mais quand ils rencontrent un programme ne respectant pas ces schémas et ces règles, ils utilisent d'autres mécanismes, comme la simulation par exemple. Les programmeurs experts s'attendent énormément à ce qu'un programme respecte ces schémas et règles. Quand ce n'est pas le cas, ils performent de piètre façon.

En fait, ce que les auteurs démontrent, c'est que les programmeurs experts, et bien plus que les programmeurs novices, ont certaines attentes face à la manière dont est écrit un programme. Cela revient à dire que si le guide de programmation d'une organisation n'est pas respecté consciencieusement, on devra s'attendre à ce que le code produit soit beaucoup plus difficile à comprendre et à modifier avec tout ce que cela comporte (coût, délai de livraison, etc.).

Davies (1990) (b) apporte certaines nuances vis-à-vis la théorie des schémas. D'abord, il soutient que même en prouvant que les schémas de programmation existent, cela ne signifie en rien que ceux-ci constituent la source primaire d'information servant à la compréhension des programmes. De plus, il n'a pas été démontré clairement que cette théorie soit applicable à tous les langages de programmation. Finalement, il apparaît de plus en plus clair que les schémas de programmation sont fortement reliés au niveau d'expérience du programmeur.

Gilmore et Green (1988) (i) considèrent, pour leur part, que le concept des schémas de programmation revêt une grande importance dans le développement d'environnements de programmation, mais qu'il est prioritaire de déterminer leur nature dans des termes indépendants des langages. Enfin, Robertson et Yu (1990) (o) supportent et approuvent les différentes études en faveur de la théorie des schémas et des plans de programmation.

La lecture de ces travaux nous fait prendre conscience de l'instabilité du vocabulaire et des concepts entourant les schémas de programmation. Les auteurs ne s'entendent pas sur une définition unique de ce qu'est un schéma. Un effort de clarification et de stabilisation de ces concepts s'avère nécessaire.

2.3.3.2 Les balises

Certains éléments, comme les balises et les règles de programmation, facilitent les processus de compréhension. Certains éléments du code source agissent comme des balises (Wiedenbeck, 1986; Wiedenbeck, 1991) en permettant au programmeur de repérer plus facilement certains schémas de programmation. Un exemple classique est l'échange des valeurs de deux variables à l'intérieur d'une boucle qui indique très souvent la présence d'un algorithme de tri.

Deux articles rapportent les tentatives plutôt concluantes de Wiedenbeck (1986, 1991) (v,w) qui démontrent l'existence de certains éléments typiques révélant la présence de structures de programmation particulières, et leur utilisation dans la compréhension des programmes informatiques. Ces éléments portent le nom de balises, comme nous l'avons vu au chapitre I. Il semblerait que celles-ci rendent un programme plus facile à comprendre pour un programmeur expérimenté, même dans le cas où celui-ci n'est pas familier avec le programme et le domaine d'application. On a aussi découvert qu'une balise située de façon erronée pouvait conduire à une mauvaise interprétation de la fonctionnalité et que la présence de certaines structures, bien que ne constituant pas elles-mêmes des balises, peuvent appuyer leurs effets. Enfin, les balises jouent un rôle, à l'intérieur du modèle de compréhension des programmes de Brooks, dans le processus de formulation d'hypothèses.

Malheureusement, la seule balise étudiée dans les expérimentations menées par Wiedenbeck est la permutation de deux valeurs dans un algorithme de tri. La présence d'une telle permutation à l'intérieur d'une boucle suggère fortement la présence d'une partie de code source effectuant le tri de certaines données. Toutefois, on est en droit de se demander s'il existe bien d'autres balises du même genre et les études laissent entrevoir un très bon potentiel dans ce sens.

2.3.3.3 Les structures d'un langage

L'étude des structures des langages de programmation, du point de vue psychologique, ne remonte pas à hier. Sime, Green et Guest (q), dans un article publié en 1973, s'employaient déjà alors à comparer deux types de structures conditionnelles. L'objectif de leurs travaux n'était toutefois pas d'en arriver à déterminer laquelle était la meilleure, mais plutôt d'évaluer l'approche consistant à comparer, non pas la totalité de deux langages, mais seulement un type de structures à la fois.

Soloway, Bonar et Ehrlich (1983) (r) font remarquer que les caractéristiques d'un langage utilisé par un programmeur débutant sont d'une importance capitale dans la réalisation de programmes. On peut espérer que des professionnels s'adaptent aux contraintes d'un certain langage, mais si celui-ci ne convient pas cognitivement à un programmeur débutant, cela pourra être une source énorme de problèmes. Les auteurs ont donc vérifié l'hypothèse suivante: une structure d'un langage de programmation se rapprochant de la stratégie cognitive d'un individu sera plus facile à utiliser efficacement.

Deux stratégies sont identifiées dans l'emploi des boucles en programmation: traiter/lire et lire/traiter. Les auteurs soutiennent que la deuxième stratégie est plus naturelle puisqu'elle fait en sorte que le *i*ème élément est lu et ensuite traité dans la *i*ème itération. Bien que la stratégie traiter/lire soit bien supportée en Pascal par la boucle *while*, on peut aussi implanter la stratégie lire/traiter. Par contre, pour ce faire, on devra ajouter certains éléments (utilisation de variables de contrôle, double contrôle) afin d'obtenir le bon comportement. Cependant, on peut imaginer un nouveau type de boucle: *loop...leave...again* qui permet d'implanter facilement une stratégie lire/traiter à l'aide de la sortie en milieu de boucle. On pourra alors éliminer l'utilisation de variables de contrôle mentionnée précédemment. Dans l'étude rapportée dans cet article, ce type de boucle est supporté par le langage hypothétique Pascal L.

Afin de vérifier leur hypothèse, Soloway, Bonar et Ehrlich se sont posé les questions suivantes: quelle stratégie de boucle les programmeurs utilisent-ils naturellement ? Est-ce que les programmeurs écrivent des programmes corrects plus souvent s'ils utilisent un langage qui supporte leur stratégie préférée ? Indépendamment de leur stratégie préférée, est-ce que les programmeurs écrivent des programmes corrects plus souvent s'ils utilisent la stratégie supportée par le langage ? Est-ce que l'exactitude de la solution, la préférence pour une stratégie particulière et la sensibilité à la stratégie

facilitée par le langage dépendent de l'expérience du programmeur ? Les auteurs ont mis sur pied une expérimentation afin de trouver des réponses à ces questions. Celle-ci consistait à faire résoudre un problème simple, utilisant une boucle, par trois groupes de sujets de niveaux différents, en leur imposant un certain langage (Pascal ou Pascal L). Il ne restait plus qu'à observer quelle stratégie de boucle était utilisée.

L'analyse des résultats révèle que la stratégie lire/traiter est fortement préférée par les étudiants. De plus, le langage Pascal L a permis d'écrire plus de programmes corrects que le Pascal. On peut donc en conclure que les gens écriront plus souvent des programmes corrects s'ils utilisent un langage facilitant leur stratégie préférée. Il apparaît aussi que la sensibilité à la stratégie facilitée par le langage peut avoir un effet significatif sur la performance. On s'aperçoit aussi que l'exactitude et la préférence pour une stratégie peuvent varier selon l'expérience.

Les auteurs ont clairement démontré que la stratégie sous-entendue par l'utilisation correcte de la boucle *while* en Pascal n'est pas la stratégie la plus naturelle. De plus, ils ont montré que la performance augmente de façon importante lorsqu'une structure de type *loop...leave...again*, supportant une stratégie lire/traiter plus naturelle, est utilisée. On peut aussi tirer une autre conclusion de cette étude: les gens écrivent plus souvent des programmes corrects en utilisant une boucle permettant de sortir au milieu de celle-ci. Cependant, certains s'opposent à ce genre de boucle (non-structure, lisibilité).

La formulation des conditions dans les structures conditionnelles a, potentiellement, un effet sur la compréhension. C'est ce que Iselin (1988) (j) a tenté de vérifier en s'intéressant aux effets des conditions positive/négative et vrai/faux. Il en conclut que la compréhension et l'interprétation des énoncés conditionnels sont facilitées par une formulation positive de l'énoncé (ex. vérifier qu'une variable contient une certaine valeur au lieu de vérifier qu'elle ne la contient pas). Iselin obtient aussi des résultats qui

supportent les conclusions de Soloway, Bonar et Ehrlich (1983) (r) sur les stratégies de boucle.

Enfin, Ebrahimi (1994) (g) a étudié, pour sa part, les erreurs les plus fréquentes faites par les programmeurs novices par rapport aux structures des langages et à la composition des plans de programmation dans le cadre d'une expérimentation utilisant quatre langages. Il a observé une forte corrélation entre le nombre d'erreurs dans les structures de programmation et le nombre d'erreurs dans la composition des plans et en conclut que la compréhension des structures d'un langage contribue à une bonne composition des plans.

Il serait intéressant de vérifier si la sensibilisation aux aspects cognitifs des structures d'un langage peut améliorer la performance des programmeurs. On doit regarder plus loin que la syntaxe et la sémantique d'un langage pour voir la demande cognitive de certaines structures des langages sur les programmeurs.

2.3.3.4 Les stratégies cognitives

L'expérimentation menée par Détienne et Soloway (1990) (f) avait pour but d'identifier différentes stratégies impliquées dans la compréhension des programmes. Suite à une expérience utilisant la technique de verbalisation, ils ont été en mesure de décrire quatre stratégies cognitives qui jouent un rôle dans la compréhension des programmes: la simulation symbolique, le raisonnement basé sur les règles de programmation, la simulation concrète et le raisonnement sur les contraintes imposées par les plans. L'alternance entre ces stratégies dépend des progrès du processus de compréhension.

Dans une autre étude, Adelson et Soloway (1990) (a) tentent de démontrer quelles sont les habiletés qui disparaissent ou apparaissent lorsque le niveau d'expérience du

domaine d'application change, en se basant sur le fait que l'expertise d'un designer repose sur les connaissances et les habiletés qu'il a développées dans un certain domaine d'application. On s'attend à ce qu'un designer qui travaille dans un domaine non familier n'utilise pas les mêmes connaissances et les mêmes habiletés que s'il évolue dans un domaine familier.

Les auteurs rapportent six comportements relativement semblables à ceux décrits par Détienne et Soloway (1985) et qui se démarquent particulièrement:

1. Élaboration de modèles mentaux: ceux-ci sont construits afin de permettre une simulation du design;
2. Simulation: permet d'intégrer des concepts familiers d'une nouvelle manière dans le but d'observer des effets imprévus de l'utilisation de ces fonctions bien connues dans un nouveau système et d'évaluer le système en développement par rapport aux spécifications;
3. Expansion systématique: permet un bon fonctionnement de la simulation en s'assurant que tous les éléments sont au même niveau d'abstraction;
4. Représentation des contraintes: rend explicites certaines des contraintes implicites du problème pour permettre de construire un modèle mental nécessaire à la simulation;
5. Étiquettes de schémas: utilisées lorsque le designer reconnaît une partie de problème qu'il a déjà résolue dans le passé et dont il détient le schéma en mémoire;
6. Prise de notes: permet de supporter l'expansion systématique en évitant d'oublier certains détails qui ne sont pas au niveau d'abstraction courant.

Ces comportements sont affectés par l'expérience du problème et du domaine d'application par le designer. La simulation et la prise de notes servent seulement lorsque le designer a une connaissance suffisante du domaine d'application. Lorsque le designer n'a pas une connaissance suffisante du problème, il développe des contraintes additionnelles afin de permettre une simulation. Lorsque le designer possède un schéma

approprié, il l'utilisera préférablement à la formulation de contraintes, la simulation ou la prise de notes.

Enfin, Davies (1991) (c) a effectué une analyse empirique des différentes stratégies employées par des programmeurs de différents niveaux d'expérience et utilisant des langages variés. Son travail démontre que plusieurs facteurs peuvent contribuer à l'établissement de stratégies de programmation. Les résultats démontrent que les changements de stratégies sont associés à l'acquisition d'expertise de programmation. Le raffinement des connaissances semble avoir une influence sur les stratégies utilisées.

Les travaux de Soloway, Bonar et Ehrlich (1983) (r) sur les stratégies de boucle correspondent également au thème de cette section.

2.3.3.5 Les connaissances

Ehrlich et Soloway (1984) (h) suggèrent l'existence d'un type de connaissance qui diffère entre les programmeurs débutants et les experts. Ils l'appellent connaissance *tacite* et, selon eux, ce type de connaissance va bien au-delà de la connaissance syntaxique et sémantique d'un langage. Elle ne s'acquiert seulement qu'avec l'expérience du développement et de la programmation. Certaines études (Shneiderman, 1976; Adelson, 1981; McKeithen et al., 1981) ont montré que, contrairement aux débutants, les programmeurs experts se souviennent plus facilement d'un programme lorsque celui-ci est composé de structures ayant une sémantique particulière. Par contre, ils ne performent pas mieux que les débutants si le programme est composé de lignes choisies au hasard, n'ayant aucune signification. Ce n'est donc pas seulement la syntaxe ou la sémantique de chacune des lignes qui est source d'information mais bien la signification des structures qu'elles composent.

Davies (1994) (d) voit les choses un peu différemment en affirmant que c'est l'organisation des schémas en mémoire qui différencie la connaissance d'un expert par rapport à celle d'un novice. Les deux possèdent des plans emmagasinés dans leur mémoire à long terme, mais la façon de les organiser est différente et celle-ci s'améliore avec l'expérience.

2.3.3.6 Les modèles mentaux

Les modèles mentaux constituent la principale composante des modèles de compréhension des programmes. À ce titre, Wiedenbeck, Fix et Scholtz (1993) (x) proposent cinq caractéristiques de ces représentations mentales des programmes informatiques:

- représentation sous forme de buts et sous-buts, organisés hiérarchiquement, formant plusieurs couches ou niveaux d'abstraction;
- correspondance explicite entre les niveaux d'abstraction;
- patron de base récurrent, c'est-à-dire des plans de programmation;
- représentation des interactions entre les parties du programme;
- lien entre les informations contenues dans le modèle et le code source ou la documentation.

L'expérimentation menée par les auteurs a démontré la présence de ces cinq caractéristiques dans le modèle mental construit lors d'une tâche de compréhension. Toutefois, le degré de développement des caractéristiques était moindre dans le cas des programmeurs moins expérimentés.

2.3.3.7 La complexité cognitive

Contrairement à certains travaux qui étudiaient les caractéristiques sémantiques de bas niveau des programmes (nomenclature des variables, indentation), Mynatt (1984) (n) s'est intéressé aux caractéristiques sémantiques de haut niveau. À l'aide de programmes ayant la même fonctionnalité et des mesures de surface identiques (métrique de Halstead), mais utilisant des structures différentes (itération vs récursion, tableaux vs listes chaînées), il a tenté de mesurer la facilité de mémorisation et de simulation manuelle de ceux-ci par des étudiants en programmation ayant différents niveaux d'expérience. Cette étude se base sur la partie connaissance sémantique du modèle de compréhension des programmes de Shneidermann et Mayer (1979).

Les résultats démontrent que la complexité sémantique de programmes ayant des mesures de surface et une fonctionnalité identiques peut varier de façon significative par l'utilisation de structures différentes. Mynatt n'arrive toutefois pas à expliquer pourquoi certaines structures augmentent la complexité.

Les travaux menés par Mynatt démontrent clairement des failles dans les mesures de complexité de Halstead. Mynatt propose des modifications à ces mesures afin d'inclure un facteur pouvant tenir compte de la complexité sémantique des logiciels.

Khalil et Clark (1989) (l) touchent à une autre facette de la complexité cognitive en étudiant l'influence de caractéristiques cognitives comme la différenciation et l'intégration sur des tâches de compréhension et de modification de programmes. Il semblerait que la théorie de la complexité cognitive soit appropriée pour prédire la performance du programmeur dans des tâches de modification de programmes, mais pas dans le cas de tâches de compréhension.

2.3.3.8 Les styles de programmation

Quelques chercheurs se sont intéressés aux effets de l'utilisation de certaines pratiques de programmation, comme les commentaires, l'indentation et la nomenclature des variables, sur la compréhension des programmes informatiques. Si des résultats intéressants sont obtenus, on pourrait les utiliser pour aider les ingénieurs logiciel à construire des guides de programmation basés sur des fondations plus solides que l'intuition et certaines conventions discutables. De plus, cela représenterait un avancement dans l'acquisition de connaissances sur les processus cognitifs, actifs dans la compréhension des programmes.

Teasley (1994) (u) a tenté d'étudier l'effet de la nomenclature des identificateurs dans un programme en se basant sur le modèle de compréhension des programmes de Pennington et sur l'hypothèse qu'une nomenclature peu appropriée produit un effet néfaste sur la compréhension de la fonctionnalité. Il conclut de son expérimentation qu'une nomenclature appropriée constitue un facteur important, facilitant la compréhension des programmes pour les programmeurs novices, mais qu'il n'en va pas entièrement de même pour les programmeurs expérimentés.

L'indentation du code source représente un autre facteur que les chercheurs ont crû utile à la compréhension des programmes. Quoique notant que plusieurs d'entre eux n'ont observé aucune différence significative à l'utilisation de l'indentation, Miara, Musselman, Navarro et Shneiderman (1983) (m) ont quand même tenté d'en savoir un peu plus long sur les effets de cette technique qui semble rendre les structures de contrôle plus faciles à lire. Ils ont remarqué, dans les résultats de l'expérimentation qu'ils ont réalisée, que les programmeurs expérimentés sont en mesure de bien comprendre un programme même si l'indentation est déficiente. De plus, ils ont observé qu'une grande proportion des sujets ont tracé des lignes pour relier les structures de

contrôle lorsqu'il n'y avait pas d'indentation. Les auteurs tirent comme conclusion que l'indentation du code source est nécessaire afin de rendre celui-ci plus facile à lire et à analyser.

2.4 Autres travaux

Nous rapportons dans cette section quelques travaux ne faisant pas appel aux diverses tâches et techniques de collecte de données, mais dont le contenu s'inscrit dans le domaine d'activité discuté dans ce mémoire et qui fait voir d'autres aspects touchés par la psychologie cognitive en programmation.

2.4.1 La réutilisation

Alors que la mode est à la réutilisation de composants promise, entre autres, par le paradigme objet, Curtis (1989) fait remarquer que la réutilisation n'est pas un nouveau concept mais que l'on fait de la réutilisation depuis des années. En effet, le programmeur ne part jamais à zéro chaque fois qu'il commence un nouveau projet. Il réutilise la connaissance qu'il possède déjà sous différentes formes. C'est d'ailleurs la qualité d'un professionnel d'être en mesure de réutiliser la connaissance et l'expérience qu'il possède afin de réaliser son travail plus efficacement. On décrit souvent la réutilisation par l'usage répété d'un même fragment de code source tiré d'une pile de documents, alors que la réutilisation de connaissances constitue la plus grande source de réutilisation en programmation.

Cependant, Curtis note que les processus cognitifs introduisent certaines limitations à cette réutilisation de connaissance. Premièrement, le programmeur tend à modifier les spécifications du problème afin de les adapter à une solution qu'il connaît déjà même si

cela contrevient aux spécifications originales. Deuxièmement, les solutions qu'un programmeur connaît dans un domaine d'application ne se transfèrent pas toujours bien dans un autre domaine. Et finalement, la forme dans laquelle sont données les spécifications peut cacher des indices sur la structure de la solution qui permettrait une réutilisation d'artefacts.

2.4.2 L'enseignement de la programmation

Soloway a discuté dans ses publications d'à peu près tous les aspects de la psychologie cognitive dans le domaine de la programmation. Un des sujets qui a retenu son attention, comme pour bien d'autres chercheurs, est l'enseignement de la programmation (Soloway, 1986). Il suggère une nouvelle approche de l'enseignement de la programmation en informatique, basée sur les découvertes récentes dans les différentes études empiriques publiées. Il soutient qu'on ne peut continuer à former les étudiants en leur enseignant uniquement la syntaxe et la sémantique des langages de programmation. Puisque les études ont démontré que les experts utilisent d'autres formes de connaissances lors des tâches de programmation, Soloway propose de montrer explicitement aux étudiants comment assembler les différentes composantes d'un programme en utilisant les concepts de plan de programmation et de règles de programmation.

2.4.3 Une théorie de la compréhension et de la correction des programmes

Un article de Lukey, publié en 1980, propose une théorie de la compréhension et de la correction des programmes. Celle-ci définit la compréhension d'un programme informatique comme étant la construction de la description du programme. Cette description correspond en fait aux spécifications du programme. À partir de cette

définition et dans une situation où une telle description du programme existe, on peut décrire le processus de correction des erreurs dans un programme comme une tâche faisant en sorte que cette description corresponde exactement à la fonctionnalité réelle du programme. Toute différence représente alors une anomalie (*"fault"* en anglais).

La théorie proposée par Lukey fait ressortir quatre concepts clés pour la compréhension des programmes: la segmentation du programme, qui consiste en la division du programme en morceaux de code qui forment des unités d'analyse, la description du flot de données, qui spécifie de quelle façon les morceaux de code communiquent entre eux, la reconnaissance d'indices indiquant la présence d'anomalies et la description des valeurs des variables. Selon la théorie, ces quatre éléments joueraient un rôle important dans la compréhension.

Pour ce qui est de la correction des erreurs, Lukey entrevoit deux façons d'identifier les anomalies. Premièrement, on peut trouver des anomalies en découvrant des indices dans le code source comme il a été discuté précédemment. Deuxièmement, la comparaison entre les spécifications du programme et le comportement réel de celui-ci peut révéler des anomalies si des différences marquées apparaissent.

Cette théorie n'est basée sur aucune expérimentation mais semble menée par le gros bon sens. Elle n'apporte malheureusement que très peu de réponse à la problématique de la compréhension des programmes informatiques.

2.5 Commentaires sur le contenu du répertoire

Nous croyons que l'ensemble des travaux répertoriés reflète assez bien ce qui a été écrit sur la compréhension des programmes durant les quinze ou vingt dernières années. Les cinq modèles de compréhension des programmes présentés au début du chapitre, surtout celui de Shneidermann et Mayer, sont couramment cités dans les articles traitant de psychologie cognitive dans le domaine de la programmation. Mis à part celui de von Mayrhauser et Vans, ils sont clairement établis et acceptés par la communauté.

Les expérimentations répertoriées ne représentent qu'un sous-ensemble de toutes les expérimentations et études produites en programmation et en compréhension des programmes. Nous n'avons retenu que celles qui touchaient des concepts potentiellement reliés au pseudocode schématique et à certains aspects de la compréhension des programmes. Ceci se traduit par le fait que l'on retrouve un plus grand nombre de travaux dans les colonnes "Schémas de programmation" et "Règles de programmation" du Tableau 2.4. Toutefois, ce tableau fait sensiblement le tour des principaux auteurs reliés au domaine de recherche traité par ce mémoire.

On remarquera que la majorité des études a été effectuée sur des sujets appartenant aux catégories "Étudiants" et "Étudiants avancés", comme le montre le Tableau 2.5 et le Tableau 2.6 à la page 53. Ce sont en fait les sujets les plus proches des chercheurs lorsque ces travaux sont réalisés dans un cadre universitaire. Cependant, le domaine du génie logiciel s'intéresse aux systèmes logiciels développés par des équipes d'ingénieurs, donc des professionnels, et non pas par des étudiants. Nous serions en droit d'espérer, dans ce cas, que plus d'ingénieurs et de professionnels participent à ce genre d'expérimentations si nous voulons obtenir des résultats justes sur lesquels nous pourrons baser des modèles et des théories représentant la réalité industrielle.

Tableau 2.5 - Expérimentations avec des sujets "Étudiants" et "Étudiants avancés"

Expérience des sujets	Thèmes									
	Schémas de programmation	Règles de programmation	Balises	Structures d'un langage de programmation	Stratégies cognitives	Connaissances	Modèles mentaux	Processus mentaux	Complexité cognitive	Styles de programmation
Étudiants	s, t	tr		q, r	c, r			p	l	k, m
Étudiants avancés	b, f, g, i, o, s, t	t	v, w	g, j, r	c, f, r	d, h	x		l, n	m, u

Tableau 2.6 - Expérimentations avec des sujets "Professionnels" et "Experts"

Expérience des sujets	Thèmes									
	Schémas de programmation	Règles de programmation	Balises	Structures d'un langage de programmation	Stratégies cognitives	Connaissances	Modèles mentaux	Processus mentaux	Complexité cognitive	Styles de programmation
Professionnels			w	j, r	a, r	d				m
Experts				j, r	a, c, r		x	e		m

On remarque dans le Tableau 2.6 que plusieurs thèmes n'ont pas été touchés par des expérimentations utilisant des sujets professionnels ou experts. En effet, certaines expérimentations ne comptaient que des étudiants parmi les sujets. Sans douter des résultats, on peut se demander si ceux-ci, ainsi que les conclusions de ces expérimentations, peuvent être étendus à des situations impliquant des professionnels. Il ne fait aucun doute que cela devrait être effectué avec prudence.

Une autre situation soulève des interrogations. Les expérimentations rapportées dans les articles (a) (Adelson et Soloway, 1985), (s) (Soloway et al., 1983) et (t) (Soloway et Ehrlich, 1984) tentent de faire ressortir les différences entre novices et experts, mais n'utilisent soit que des étudiants (étudiants et étudiants avancés) ou que des professionnels (professionnels et experts), selon la classification proposée dans ce mémoire. De plus, les articles (h) (Davies, 1994), (u) (Teasley, 1994) et (v) (Wiedenbeck, 1986) traitent de la même différence entre novices et experts, mais en n'utilisant que des sujets d'une seule catégorie selon notre classification. On peut encore douter de la portée des conclusions de ces expérimentations sur les autres catégories de sujets. Il se peut toutefois que cela indique que la classification proposée dans ce mémoire soit trop rudimentaire pour tenir compte des différences d'expérience entre les sujets, surtout chez les étudiants.

Les quelques sujets traités dans la dernière partie complètent ce qui a été présenté dans les autres sections. On remarque toutefois un grand absent. Aucun des travaux répertoriés ne traite de l'effet des commentaires sur la compréhension des programmes. Pourtant, les commentaires constituent un des éléments essentiels, nécessaires à la compréhension du code source.

Nous aurions pu inclure dans ce chapitre une foule d'autres travaux tous aussi intéressants les uns que les autres comme ceux traitant, par exemple, des aspects de processus cognitifs lors du design d'un logiciel. Ce sujet a été également traité par un bon nombre de chercheurs. Toutefois, nous avons décidé de le mettre de côté pour l'instant afin de nous attarder plus particulièrement à la compréhension des programmes.

2.6 Résumé du chapitre

Dans ce chapitre, nous avons présenté cinq modèles de compréhension en décrivant leur structure et leur fonctionnement. Par la suite, nous avons élaboré un répertoire d'environ vingt-cinq expérimentations portant sur divers aspects de la compréhension des programmes. Ce répertoire était en fait présenté sous forme d'un tableau permettant de classer chacune des expérimentations selon le thème abordé et les attributs de l'expérience. Nous avons par la suite résumé brièvement les résulats les plus intéressants obtenus lors de ces expérimentations. Nous avons terminé en présentant quelques études complémentaires et en formulant quelques remarques sur le contenu du chapitre.

Comme nous avons pu le constater, il n'existe pas une façon unique de modéliser la compréhension des programmes. Chaque modèle apporte sa contribution afin de résoudre la problématique. Malheureusement, certains modèles n'ont pas fait l'objet d'une vérification expérimentale, essentielle à la validation de ceux-ci. Leur côté abstrait les rend aussi plus difficiles à utiliser lorsqu'on veut en tirer des applications pratiques. Toutefois, les expérimentations répertoriées soulèvent des aspects très intéressants, beaucoup plus concrets et faciles à exploiter. Au chapitre IV, nous tentons de les utiliser en appliquant leurs conclusions à la pratique professionnelle du génie logiciel.

CHAPITRE III -

LE PSEUDOCODE SCHÉMATIQUE

Avant d'examiner de quelle façon le pseudocode schématique peut contribuer à l'intégration des concepts reliés à la compréhension des programmes dans la pratique professionnelle, nous allons d'abord décrire l'origine de cet outil de développement ainsi que sa structure. Nous terminons ce chapitre par un exemple simple, illustrant l'utilisation du pseudocode schématique.

3.1 Introduction au pseudocode schématique

Le pseudocode schématique est apparu à la fin des années 1970, alors que des chercheurs (Robillard et Thalmann, 1980) se sont intéressés à la représentation des structures d'un programme, indépendamment du langage utilisé (Robillard, 1986; Grenier, 1989). Contrairement au pseudocode conventionnel, qui a le fâcheux désavantage de n'avoir aucune règle de construction syntaxique et sémantique, le pseudocode schématique est une notation formelle qui permet d'éviter le phénomène connu du «*Variation in Practice*», créé par la créativité personnelle.

De plus, il apporte des supports à la documentation du code source et à la spécification du flux de contrôle qui ont été identifiés comme des façons d'améliorer l'utilisation des langages de programmation (Robillard, 1986).

Le pseudocode schématique est un outil de développement de logiciels. Il permet de visualiser, à l'aide de symboles graphiques faisant abstraction du langage de

programmation utilisé, les structures de base proposées par Böhm et Jacopini en 1966 et utilisées par les langages de programmation orientée-procédure: la structure séquentielle, la structure conditionnelle et la structure répétitive. Pour être efficace, la représentation graphique ne doit pas seulement être une autre façon d'écrire une boucle do-while. Le vocabulaire utilisé doit être au moins plus petit que celui du langage de programmation et les règles d'assemblage des structures doivent être aussi simples que d'écrire une boucle dans le langage cible.

Le pseudocode schématique intègre aussi le concept de raffinements successifs permettant de décomposer une tâche à réaliser en sous-tâches, qui peuvent également être à nouveau décomposées en d'autres sous-tâches. On obtient alors une représentation arborescente du programme. Cette technique a été originalement proposée par Niklaus Wirth. Schach (1993) en donne une définition: «postpone decisions as to details as late as possible in order to be able to concentrate on the important issues.». Il justifie l'utilité de cette technique avec la Loi de Miller (Miller, 1975) qui dit qu'à chaque instant, un être humain ne peut se concentrer que sur 7 ± 2 éléments d'information.

À chacune des étapes du raffinement, le programmeur doit: 1) tenir compte de l'interdépendance des sous-problèmes; 2) tenter de préserver l'exactitude de la fonctionnalité; 3) reporter aussi loin que possible les décisions concernant les détails d'implantation (Reynolds, Maletic et Porvin, 1992).

3.2 Description des structures

Le Tableau 3.1 résume la grammaire du pseudocode schématique sous la forme BNF. Le symbole "+" indique que l'élément peut être répété une fois et plus. Chacun des éléments est décrit dans les sections suivantes.

Tableau 3.1 - Grammaire du pseudocode schématique

Élément	**contient**
Programme	{ Raffinement }+
Raffinement (0)	{ Structure }+
Structure	Structure séquentielle (3.2.2) ou structure conditionnelle (3.2.3) ou structure répétitive (3.2.4)
Structure séquentielle	Énoncé non structurel du langage ou raffinement ou commentaire narratif

3.2.1 Le raffinement

Un raffinement est un ensemble de lignes de code source ayant un objectif commun. Chaque raffinement est caractérisé par un numéro d'identification et par une ligne de texte décrivant le but visé par les structures qu'il contient. Ce texte est appelé commentaire opérationnel et permet de documenter le code source au fur et à mesure de l'écriture du programme.

Un algorithme en pseudocode schématique est contenu à l'intérieur d'un raffinement de départ ayant le numéro d'identification 000. Un raffinement est représenté par une ligne verticale continue, représentant une structure séquentielle et ayant comme symbole la forme d'un "I" majuscule. On peut placer des structures conditionnelles et répétitives n'importe où sur cette structure séquentielle. On ne peut cependant superposer qu'un seul niveau de structures conditionnelles ou répétitives sur celle-ci, obligeant ainsi l'utilisation des raffinements afin d'imbriquer plusieurs structures.

3.2.2 La structure séquentielle

Trois types de structures séquentielles sont possibles (voir Figure 3.1):

- le commentaire narratif: correspond au commentaire ordinaire du langage cible. Le texte du commentaire est précédé d'un tiret;
- l'énoncé: permet d'effectuer une instruction du langage cible (ex. affectation, appel de fonction). Il occupe une ligne complète.
- le commentaire opérationnel: donne une description du contenu du raffinement en langage naturel. Le texte est précédé par un numéro unique identifiant le raffinement.

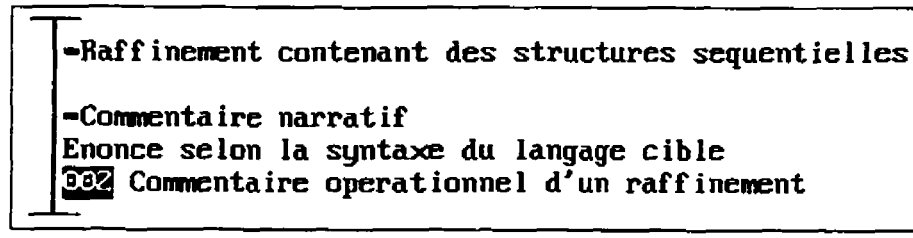


Figure 3.1 - Types de structures séquentielles

3.2.3 La structure conditionnelle

La structure conditionnelle est illustrée graphiquement par une ligne horizontale qui brise la ligne verticale de la séquentielle en détournant le flux de contrôle. comme on peut le voir sur la Figure 3.2. Elle est toujours composée d'au moins une branche "SI" et facultativement d'une ou de plusieurs branches "SINON SI" et d'une seule branche "SINON". On utilise le trait pointillé suivi d'un trait plein dans le cas de la branche "SINON SI" et le trait pointillé seul dans le cas de la branche "SINON".

La structure conditionnelle constitue le seul type de sélection possible dans le pseudocode schématique et ce, indépendamment des langages. On peut donc comprendre facilement l'algorithme sans connaître la syntaxe du langage cible utilisé grâce à cette représentation uniforme. On retrouvera dans chacune des branches de la structure les trois types de structure séquentielle.

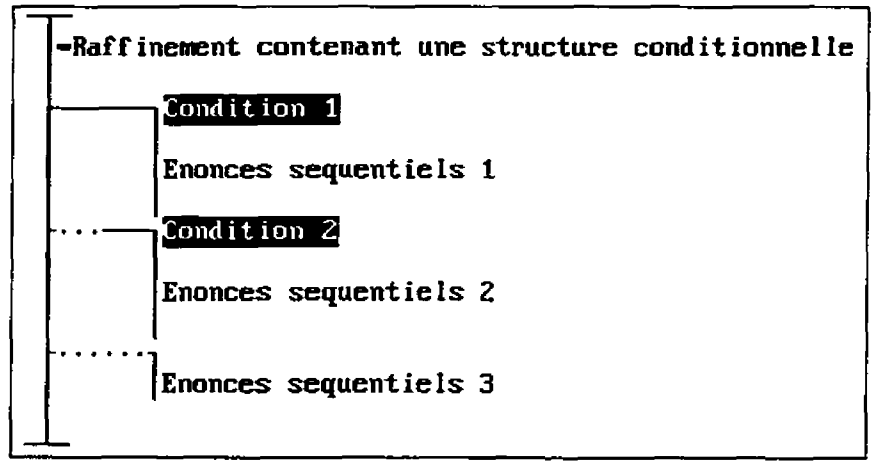


Figure 3.2 - Structure conditionnelle avec branches SI, SINON SI et SINON

3.2.4 La structure répétitive

La structure répétitive constitue la seule façon de répéter un ensemble de structures séquentielles à l'intérieur du pseudocode schématique, indépendamment du fait qu'il existe plusieurs types de boucles dans le langage cible. On représente la structure répétitive à l'aide d'une barre verticale double, comme montrée à la Figure 3.3. Elle doit contenir au moins une condition de sortie de boucle. On place les conditions de sortie n'importe où dans la séquence de la boucle et elles peuvent elles-mêmes contenir des structures séquentielles.

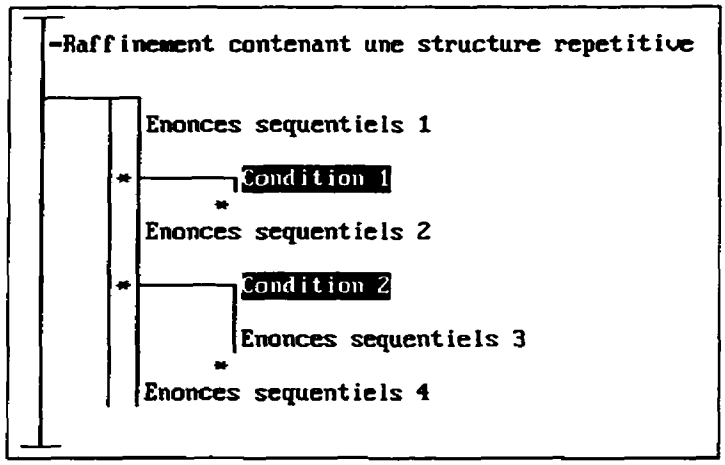


Figure 3.3 - Structure répétitive générale

3.3 Exemple

Afin d'illustrer de quelle façon on utilise le pseudocode schématique dans la pratique, nous avons repris un programme qui a été très souvent utilisé dans des expérimentations sur la compréhension des programmes (Soloway, Bonar et Ehrlich, 1983; Soloway, Ehrlich, Bonar et Greenspan, 1983; Ehrlich et Soloway, 1984; Soloway et Ehrlich, 1984; Soloway, 1986; Gilmore et Green, 1988; Davies, 1990; Davies, 1991; Ebrahimi, 1994). Sa popularité est probablement liée au fait qu'il contient une boucle et une conditionnelle qui sont en fait les deux types de structure de contrôle de base utilisés par les langages procéduraux.

Ce programme, connu sous les appellations *Rainfall* et *Problème de Noah*, a pour but de calculer la moyenne d'une série de nombres entiers qu'on suppose être des quantités de chute de pluie. En fait, il s'agit de lire et d'additionner des nombres entiers, entrés au

clavier par l'utilisateur, jusqu'à ce que la valeur 99999 soit lue. La moyenne de tous les nombres lus, à l'exception de la sentinelle 99999, est alors affichée à l'écran. La Figure 3.4 présente une adaptation, en langage C, du code source en langage Pascal de ce programme (Soloway, Bonar et Ehrlich, 1983). Plusieurs variantes de ce programme existent mais l'idée de base demeure la même.

```
void main(void) {
        int Nombre = 0, Somme = 0, Compteur = 0;
        float Moyenne = 0.0;

/* Lire un nombre */
        scanf( "%d", &Nombre );

        while ( Nombre != 99999 ) { /* Lire un nombre jusqu'a 99999 */
/* Ajouter ce nombre a la somme */
                Somme = Somme + Nombre;
/* Incrementer le compteur de nombres */
                Compteur++;
/* Lire un nombre */
                scanf( "%d", &Nombre );
        }

        if ( Compteur > 0 ) {
/* On a entre au moins un nombre. Calculer la moyenne */
                Moyenne = Somme / Compteur;
/* Afficher la moyenne */
                printf( "%f", Moyenne );
        } else {
/* On n'a entre aucun nombre. Afficher un message d'erreur */
                printf( "Aucun nombre entre\n" );
        }
}
```

Figure 3.4 - Problème de Noah en langage C

À la Figure 3.5, nous pouvons voir le même programme mais, cette fois-ci, représenté à l'aide du pseudocode schématique. On peut remarquer la ligne verticale ayant la forme d'un "I" majuscule, représentant le flux de contrôle, à la gauche de l'illustration. Par souci de clarté et afin de limiter le nombre de figures, nous avons représenté les raffinements sous une forme intégrée. Ainsi, même si le raffinement de base contient trois raffinements, nous avons représenté ces derniers à même le raffinement de base. Ils sont délimités par les flèches pointant vers le bas pour indiquer le début et vers le haut

pour indiquer la fin du raffinement. Les nombres apparaissant à la droite de ces flèches correspondent aux numéros de raffinements. Le commentaire opérationnel accompagnant le raffinement apparaît sur la ligne suivant celle avec la flèche vers le bas.

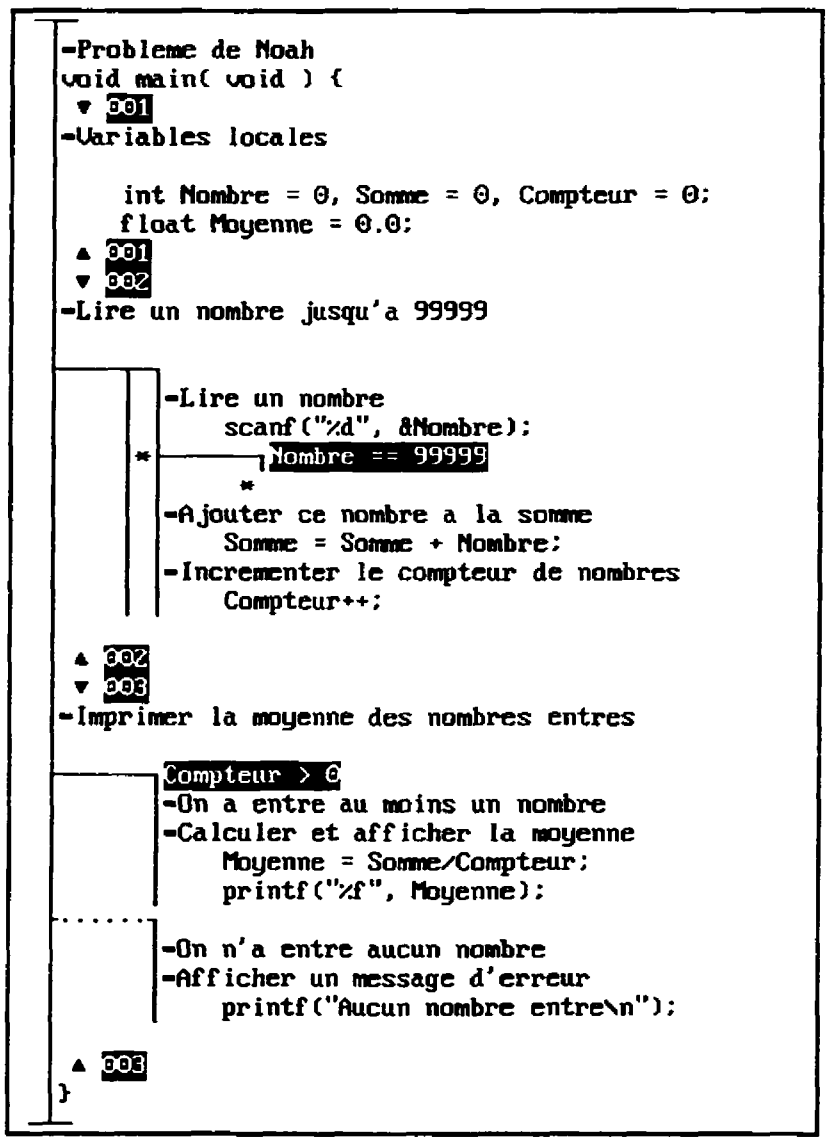


Figure 3.5 - Problème de Noah représenté sous forme de pseudocode schématique

Sur la première ligne, on peut voir le commentaire opérationnel du raffinement de base (Problème de Noah). Ensuite, il y a un énoncé qui marque le début de la fonction selon la syntaxe du langage C. Le raffinement 001 débute sur la ligne suivante. Celui-ci contient les déclarations des variables locales, toujours dans la syntaxe du langage cible.

Sur la ligne suivante, on rencontre un autre raffinement, qui contient cette fois une structure répétitive qui nous permettra de lire plusieurs nombres. L'énoncé apparaissant au début de la structure permet de lire une valeur au clavier (*scanf("%d", &Nombre)*). La condition de sortie, placée au milieu de la boucle, permet de sortir de celle-ci lorsque la valeur 99999 est entrée au clavier. On prend soin ensuite d'additionner le nombre lu et d'incrémenter le compteur si la condition de sortie n'est pas respectée.

Lorsque la condition de sortie est vraie, on doit terminer l'exécution de la boucle et revenir à la séquence d'énoncé suivant celle-ci. On arrive donc à la structure conditionnelle qui est contenue dans le raffinement 003. La structure conditionnelle placée dans ce raffinement vérifie si l'utilisateur a entré au moins un nombre. La première condition rencontrée est "Compteur > 0". Si celle-ci est vraie, c'est que l'usager a entré au moins un nombre, et le calcul ainsi que l'affichage de la moyenne s'effectuent. Sinon, on évite la division par zéro et on saute à la partie SINON de la structure qui affiche un message d'erreur.

On peut remarquer que dans la représentation sous forme de pseudocode schématique, on ne lit qu'à un seul et unique endroit un nombre au clavier, alors que dans la version écrite en langage C, l'énoncé de lecture apparaît deux fois. Cela dépend en fait de la stratégie de boucle qui a été utilisée (Soloway, Bonar et Ehrlich, 1983). Dans la version en C, on a recours à une stratégie de type traiter/lire, ce qui nous oblige à effectuer une lecture avant le début de la boucle. Toutefois, dans la version en pseudocode schématique, on utilise plutôt une stratégie lire/traiter, ce qui évite de doubler cet énoncé

de lecture. Cela est possible grâce à la condition de sortie qui peut apparaître au milieu de la boucle, comme le montre notre exemple.

De plus, les commentaires narratifs nous ont permis de documenter adéquatement le programme sans alourdir la lecture de celui-ci.

3.4 Résumé du chapitre

Dans ce chapitre, nous avons présenté le pseudocode schématique. Nous en avons expliqué la provenance et les idées justifiant son développement ainsi que sa structure. Enfin, nous avons illustré son fonctionnement à l'aide d'un exemple abondamment utilisé dans les expérimentations en compréhension des programmes.

Le pseudocode schématique est la seule représentation connue permettant de réaliser un design détaillé et d'y inclure le code source par la suite. Sa structure fait en sorte qu'il est applicable à la majorité des langages procéduraux puisqu'on se restreint aux structures de haut niveau comme les boucles et les conditionnelles en évitant de s'attarder à la syntaxe. Nous y voyons une façon de mettre en pratique les conclusions de plusieurs travaux sur la compréhension des programmes informatiques.

CHAPITRE IV -

APPLICATION PRATIQUE DES RÉSULTATS

Nous avons déjà défini, dans un premier temps, les principaux concepts reliés à la psychologie cognitive et à la compréhension des programmes et, dans un deuxième temps, ceux reliés au pseudocode schématique. Nous en sommes donc rendu, dans ce chapitre, à étudier comment l'application des conclusions des différents travaux sur la psychologie cognitive, conjointement avec le pseudocode schématique, peut avoir un impact sur le développement de logiciels. Est-ce que l'on peut bâtir des programmes informatiques en tenant compte des conclusions de ces travaux ?

4.1 Thèmes applicables au génie logiciel

Les modèles de compréhension des programmes et les expérimentations répertoriées au chapitre II forment une sorte de dualité. D'un côté, on a l'approche conceptuelle des chercheurs qui ont tenté de modéliser les processus de compréhension dans le but d'en arriver à faire des prédictions, ce que peu ont réussi. De l'autre, on a l'approche, plus expérimentale et concrète, des chercheurs qui expérimentent divers aspects de la programmation en se basant quelquefois sur ces modèles. Notre revue de la littérature nous a fait constater que les concepts entourant ces modèles (schéma, modèles mentaux, etc.) sont encore mal définis et mal cernés bien que de nombreux chercheurs s'y intéressent. Nous préférons donc nous en tenir aux aspects plus pratiques soulevés par les différentes expérimentations.

On doit d'abord se poser la question suivante: quelles conclusions expérimentales devrait-on tenter d'appliquer à la pratique du génie logiciel ? Sur la panoplie de travaux publiés dans les périodiques spécialisés, une bonne partie touche à des thèmes trop abstraits pour être appliqué directement et concrètement. On pense alors à des sujets comme les stratégies cognitives, les processus mentaux et la complexité cognitive. Ceux-ci requièrent plus de travail avant d'en arriver à une application concrète. Toutefois, les travaux touchant les schémas et règles de programmation, les balises, les structures des langages et les styles de programmation sont plus pratiques, car souvent à caractère expérimental, et nous suggèrent plus facilement des applications.

Il faut toutefois garder un esprit très critique envers ces études. Comme dans tout domaine, certaines seront plus sérieuses et surtout mieux menées.

4.2 Buts visés par l'application dans la pratique du génie logiciel

L'intégration au génie logiciel, et particulièrement à la compréhension des programmes, de concepts venus de la psychologie représente une toute nouvelle direction de recherche. Au fil des années, les gens ont constaté qu'un élément essentiel était constamment oublié dans l'élaboration des nouvelles techniques pour le développement des logiciels. Ce ne sont pas tous ces outils que l'on voit annoncés à pleine page dans les revues qui assurent qu'un logiciel sera de qualité. Bien que cela peut avoir une certaine influence, la matière première pour l'élaboration d'un logiciel provient de l'esprit d'un professionnel capable de construire un design, d'analyser son implantation et de le traduire en code source.

On a longtemps oublié que l'esprit humain avait un grand rôle à jouer dans la réalisation de logiciels. On laisse encore beaucoup de crédit aux compilateurs, éditeurs et autres

dans la réussite d'un projet logiciel. Pourtant, ce ne sont que des supports aux diverses activités mentales qu'un professionnel accomplit.

L'application des principes de la psychologie cognitive au génie du logiciel vise à prendre conscience des possibilités mais aussi des limites de l'esprit humain dans le développement d'un logiciel. Nous devons en venir à tirer profit des capacités d'abstraction et d'analyse du programmeur afin de produire des logiciels de meilleure qualité et dont la maintenance sera facilitée.

Le facteur humain a déjà fait sa marque dans d'autres domaines du génie. Son intégration avec le génie industriel a produit une nouvelle discipline: l'ergonomie. De plus, on tente maintenant d'appliquer les concepts développés en ergonomie au génie du logiciel, ce qui nous conduit à l'ergonomie du logiciel. Cette discipline vise à étudier les relations entre l'humain et le logiciel, surtout en terme d'interfaces.

4.3 Acteurs dans l'application des concepts

Les professionnels du développement de logiciels devraient être les premiers à tenter d'intégrer, dans leur travail quotidien, certaines pratiques guidées par les travaux en compréhension des programmes. Toutefois, cela risque de se faire indirectement par l'application de normes, de processus de développement et d'outils qui eux-mêmes sont issus de l'application de ces concepts.

De plus, les étudiants en informatique et en génie informatique devraient être sensibilisés tôt à l'importance des facteurs humains dans la compréhension des programmes par l'intégration de ces concepts dans les différents cours durant leur formation.

4.4 Exemples d'application

Le but de cette section consiste à suggérer des applications possibles des travaux en compréhension des programmes et de leurs résultats à la pratique du génie logiciel. Plusieurs de ces applications sont déjà utilisées dans la pratique. Toutefois, nous désirons justifier et promouvoir leur emploi en effectuant des liens avec les travaux étudiés précédemment.

4.4.1 Application des schémas de programmation

Nous avons déjà présenté, à la section 1.2, la théorie des schémas, et un résumé d'expérimentations sur les schémas de programmation a été présenté à la section 2.3.3.1. Afin d'appliquer ce concept à la pratique, on doit en arriver à encapsuler une certaine fonctionnalité dans une entité générique qui devra être capable de s'adapter aux besoins du programmeur.

L'idée de raffinement contenue dans le pseudocode schématique peut être vue comme une bonne manière d'appliquer le concept de schéma de programmation. Toutefois, la notion de schéma de programmation est plus large que celle du raffinement. Un schéma peut aussi bien être implanté par un seul ou bien par un ensemble de raffinements reliés par des structures de contrôle. Ces raffinements sont associés de telle sorte qu'ils forment la fonctionnalité globale du schéma.

Les structures d'itération et de sélection, telles que définies dans le pseudocode schématique, peuvent aussi être considérées comme des schémas de programmation. En effet, la structure conserve sa forme, peu importe le genre de boucles et de conditionnelles que le programmeur désire implanter, en lui laissant la responsabilité de

remplir les espaces vides par l'ajout d'expressions booléennes adéquates et de structures séquentielles de façon à instancier une nouvelle structure se comportant selon des besoins particuliers. Cela correspond à la définition que Robertson et Yu (1990) donnent des plans globaux.

Pour être avantageux, un outil supportant le pseudocode schématique doit permettre de visualiser la structure et la hiérarchie des raffinements. De plus, il doit permettre de manipuler ceux-ci par des mécanismes de sélection, de déplacement et de copier/coller communs aux éditeurs de texte. Enfin, il doit permettre d'effectuer des recherches basées sur des mots clés ou encore sur des balises. En effet, Robertson et Yu (1990) affirment à la fin de leur article:

> « An explicit coding scheme in the basic-level language of programming i.e. plans, should be available to readers of code. Similarly, an interface that explicitly shows the content and structure of a program's plan hierarchy (or any other psychologically valid representation scheme), and that allows this structure to be utilized for search and modification of programs, should be developed. »

De plus, cet outil devrait permettre à l'utilisateur de définir des schémas génériques qu'il pourra réutiliser au besoin. Par exemple, il arrive très souvent qu'un programmeur ait à construire une structure de boucle pour parcourir un tableau. Celle-ci se compose toujours d'une structure d'itération ayant une condition de sortie placée au début de la boucle afin d'éviter un débordement de tableau. De plus, celle-ci débutera toujours par l'initialisation d'un compteur et se terminera par l'incrémentation de ce compteur. Il n'est pas nécessaire de réécrire à chaque fois la même chose. On devrait être capable d'utiliser simplement des schémas génériques qui ont été écrits correctement une seule fois. Cette pratique éviterait des erreurs et des différences d'écriture entre les programmeurs et permettrait d'augmenter la compréhensibilité puisque ces structures seraient bien connues de tous.

La réutilisation d'algorithmes provenant de différents ouvrages constitue une autre possibilité d'application des schémas de programmation. En parcourant la littérature traitant d'algorithmie, on se rend vite compte que ce sont toujours les mêmes exemples qui reviennent. En réutilisant des algorithmes connus, le programmeur augmente les chances que les gens qui auront à lire et comprendre son travail connaissent déjà la logique utilisée. Toutefois, cela peut être très variable selon l'expérience des gens.

4.4.2 Application des règles de programmation

L'implantation pratique des règles de programmation peut s'effectuer tout simplement en élaborant un guide de programmation contenant des conventions que les programmeurs doivent respecter. On vérifie habituellement le respect de ces conventions par des inspections formelles du code source.

Ces conventions peuvent touchées plusieurs niveaux du code source. On peut les utiliser pour restreindre l'utilisation de certaines structures d'un langage ou encore pour guider l'organisation du code source dans un fichier. On se sert souvent aussi du guide de programmation pour imposer une nomenclature pour les noms de variables. Bien que Teasley (1994) conclu qu'une telle nomenclature n'a que peu d'effet sur la compréhension du code source par les programmeurs expérimentés; celle-ci permet d'éviter une trop grande diversité dans le choix des identificateurs en restreignant leur longueur ainsi que la langue utilisée. Il faut remarquer que Teasley n'a utilisé que des étudiants avancés comme sujets pour son expérimentation. De plus, la taille du programme considéré peut avoir une incidence importante sur les résultats obtenus. En effet, si le programme ne contenait que quelques dizaines de lignes, il est normal que l'utilisation d'une nomenclature n'ait que peu d'effet. Toutefois, comme la taille des programmes considérés en génie logiciel est plutôt de dizaine de milliers, voire même de

millions de lignes, il est logique de penser qu'une telle nomenclature aura assurément des effets bénéfiques sur la compréhension du code source.

Nous proposons donc dans le Tableau 4.1, le Tableau 4.2 et le Tableau 4.3, l'exemple d'une nomenclature adaptée au langage orienté-objet C++. Il est assez simple de la modifier de façon à supporter les structures particulières à d'autres langages procéduraux ou orientés-objet. Une convention du même genre, appelée "*Hungarian naming*" a été proposée par Charles Simonyi dans les années 1970.

Tableau 4.1 - Règles générales de la nomenclature des identificateurs

• Chaque identificateur est composé de trois parties: le préfixe, le mot principal et le mot secondaire. Le mot principal décrit de façon générale l'identificateur, tandis que le mot secondaire en précise le sens.
• Tous les mots utilisés doivent être en français. Aucune abréviation n'est permise.
• Chacun des mots doit débuter par une lettre majuscule suivie des autres lettres en minuscules.
• Le qualificatif peut être un nom, un nom composé ou un adjectif. Il s'accordera en genre et en nombre avec le mot principal, selon le sens.
• On ajoute la lettre "s" au préfixe lorsqu'il s'agit d'une variable locale, globale ou d'une constante statique et lorsqu'il s'agit d'une méthode ou d'une fonction statique.

Tableau 4.2 - Nomenclature des identificateurs en langage C++

		Préfixe	Mot principal	Mot secondaire
Types	Classe	c	Nom au singulier	Qualificatif
	Classe gabarit	cg		
	Structure	s		
	Énumération	e		
	Valeur d'une énumération	v		
	Type défini	t		
Variables	Globale	g	Nom au singulier	Qualificatif
	Locale	l		
	Paramètre	p		
	Paramètre d'une classe gabarit	pg		
	Attribut d'une structure ou d'une classe	a		
	Constante	k		
Macro de précompilation	Nom de la macro	d	Nom au singulier	Qualificatif
	Paramètre	p		
Actions	Méthode	m	Verbe à l'infinitif	Nom au singulier
	Méthode virtuelle	mv		
	Fonction	f		

Tableau 4.3 - Complément à la nomenclature pour les méthodes

Opération	Paramètre	Valeur de retour	Préfixe	Mot Principal
Obtenir la valeur d'un attribut	void	Type de l'attribut	m	Nom de l'attribut (sans le préfixe a)
Modifier la valeur d'un attribut	Type de l'attribut	void	m	Nom de l'attribut (sans le préfixe a)

Le nom des méthodes servant à obtenir ou à modifier la valeur d'un attribut d'une classe devra respecter la convention présentée dans le Tableau 4.3. Celle-ci utilise avantageusement la surcharge des noms de méthodes permise dans le langage C++.

On devra aussi ajouter certaines règles concernant les expressions conditionnelles à l'intérieur d'un tel guide de programmation. C'est ce que suggère la conclusion des travaux de Iselin (1988). Il semble qu'une formulation positive des énoncés conditionnels facilite la compréhension. Autrement dit, il sera plus facile de saisir le sens d'un énoncé vérifiant qu'une variable contient une certaine valeur qu'un autre énoncé qui vérifie qu'elle ne contient pas cette valeur. On peut en déduire aussi que les associations avec des ET et des OU logiques ont avantage à être évitées. La compréhension peut être difficile lorsque le programmeur a besoin de se construire un tableau pour vérifier toutes les combinaisons produites par ces associations.

4.4.3 Application des balises

Nous avons déjà discuté, à la section 2.3.3.2, que les balises peuvent contribuer efficacement au processus de compréhension des programmes si elles sont placées aux bons endroits. Un commentaire ou un segment d'algorithmie peut constituer une telle balise. Les balises, telles qu'étudiées par Wiedenbeck (1986, 1991), peuvent être qualifiées d'accidentelles et d'implicites. Elles contribuent à faciliter la compréhension mais elles se trouvent là par hasard. Il serait intéressant de voir si on peut placer ces balises de façon intentionnelle et quelle forme elles pourraient alors prendre.

Les commentaires dans le code source représentent sûrement la forme de balises la plus simple. Toutefois, il faut que le commentaire soit juste et formulé de la bonne façon. Un commentaire résumant un groupe de lignes en terme du langage de programmation utilisé n'est pas très efficace. Un commentaire doit être formulé de manière à ce qu'il

renseigne le lecteur en le plaçant dans le contexte du domaine d'application. Le commentaire opérationnel du pseudocode schématique peut être une bonne façon de placer une balise puisqu'il résume en quelques mots la fonctionnalité d'un ensemble d'énoncés. Un lecteur désirant naviguer dans le code source jusqu'à un point bien précis n'a pas à suivre le code ligne par ligne. Il suffit de se guider en consultant le texte des commentaires opérationnels, car une balise doit aussi aider à parcourir le code source. Leur identification visuelle doit être évidente pour que cela soit efficace. Une organisation uniforme du contenu d'un fichier en termes de fonctions, prototypes, classes et autres aide le lecteur à se positionner et constitue une forme plus discrète de balises.

4.4.4 Impact sur les structures d'un langage et le style de programmation

Au cours des années, les langages de programmation se sont multipliés, chacun ayant sa syntaxe et sa grammaire propre. Bien que tous les langages permettent d'effectuer des itérations et de sélectionner l'action à accomplir selon une certaine condition, la syntaxe et la structure, pour y arriver, peuvent différer passablement et peuvent causer des problèmes à plusieurs programmeurs. C'est dans ce sens que l'expérimentation menée par Soloway, Bonar et Ehrlich (1983) est très révélatrice. Pour chacun des langages, le programmeur doit réapprendre comment effectuer une itération. Pourquoi ne cherchons-nous pas une manière d'uniformiser l'écriture des structures de base tout en laissant la liberté d'utiliser les particularités reliées à chacun des langages ? L'approche symbolique des structures de contrôle du pseudocode schématique constitue un élément de réponse à cette section. En effet, au lieu de multiplier les énoncés permettant d'effectuer une boucle, il propose une façon unique d'exécuter cette action en simplifiant l'écriture (voir section 3.2.4). Il en est de même pour les structures conditionnelles.

De plus, la sortie en milieu de boucle permise par le pseudocode schématique nous permet d'implanter simplement une stratégie lire/traiter, plus naturelle selon Soloway,

Bonar et Ehrlich (1986), sans doubler les structures ou encore sans utiliser des variables de contrôle.

Il n'est pas souhaitable qu'un programmeur ait besoin de tracer des lignes entre ses structures de contrôles pour connaître leur portée comme le rapporte Miara, Musselman, Navarro et Shneiderman (1983). C'est pourquoi plusieurs ont pris l'habitude d'indenter les structures de contrôle de façon à bien identifier leur début et leur fin. Toutefois, cela a tendance à limiter le nombre d'imbrications pour éviter une trop grande indentation vers la droite. Le pseudocode schématique nous affranchit de l'utilisation de l'indentation en montrant bien la portée des structures de contrôle (comme illustré à la Figure 3.2).

4.5 Résumé du chapitre

Toutes les techniques présentées dans ce chapitre sont bien simples à appliquer en pratique et trouvent un fondement dans les études en compréhension des programmes. La nomenclature proposée pour les identificateurs utilisés dans les programmes en est un bon exemple. Son application sous forme de guides ou de normes peut sans aucun doute améliorer la qualité des programmes et faciliter la maintenance de ceux-ci.

Le pseudocode schématique semble, pour sa part, posséder des caractéristiques qui intégrent bien certains concepts de la compréhension des programmes et de la psychologie cognitive. L'utilisation de ce formalisme peut sûrement être un avantage afin de produire du code source de meilleure qualité et ayant une maintenance plus facile. Son intégration à l'intérieur d'un outil de design et d'édition de code source incluant des fonctionnalités qui facilitent sa manipulation représente un bon point de départ dans le rapprochement des domaines de la psychologie cognitive et du génie logiciel.

CONCLUSION

Nous avons tenté, tout au court de ce mémoire, de synthétiser les résultats des travaux des quinze dernières années des chercheurs dans le domaine de la compréhension des programmes. Les différentes études et expérimentations présentées au chapitre II couvrent bien l'ensemble des recherches effectuées dans ce domaine.

Après avoir présenté au chapitre I quelques-uns des concepts de base de la psychologie cognitive, des modèles de compréhension des programmes et des expérimentations dans ce domaine, le chapitre II nous a permis d'avoir une vue d'ensemble des travaux rapportés dans les revues spécialisées. Cette rétrospective nous a permis de constater certaines faiblesses au niveau des concepts reliés à la compréhension des programmes. La définition de ce qu'est une balise demeure assez vague et laisse place à beaucoup d'interprétation. Il en est de même pour les schémas de programmation. Chaque auteur a sa façon de les définir. On en conclut que plusieurs concepts sont encore mal cernés et mal compris.

De plus, nous avons proposé un ensemble de catégories afin de classer les sujets utilisés dans les expérimentations selon leur expérience, après avoir constaté une faiblesse à ce niveau. En effet, chacun des auteurs avaient sa propre évaluation de ce que représente un professionnel. Cette catégorisation uniformisant la taxonomie s'avérait nécessaire et aurait avantage à être utilisée afin de mieux comparer les expérimentations entre elles.

Le chapitre III se voulait une brève introduction au pseudocode schématique. Nous avons donc présenté sa structure, sa sémantique ainsi que les idées qui ont mené à son élaboration. Afin de bien démontrer son application, un exemple simple, adapté aux spécifications d'un problème courant dans les expérimentations sur la compréhension

des programmes, a été présenté. Cet exemple illustrait bien la souplesse et la rigueur des structures de contrôle contenues dans le pseudocode schématique.

Dans le chapitre IV, nous avons mis l'accent sur l'application des concepts traités auparavant à la pratique du génie logiciel. Il faut bien se rappeler que notre but n'était pas de modéliser la compréhension des programmes ou encore d'en arriver à une théorie expliquant le comportement du programmeur dans un processus de compréhension d'un programme informatique. C'est pourquoi nous avons mis de côté l'approche conceptuelle afin de nous attarder aux aspects plus pratiques et concrets de la compréhension des programmes.

Les possibilités d'application sont vastes car presque encore inexplorées. Les exemples fournis dans le chapitre IV, simples à implanter, démontrent bien la possibilité de tenir compte des facteurs humains et peuvent rapporter des bénéfices à très court terme.

La poursuite de nos recherches devrait aboutir à la conduite d'expérimentations utilisant, entre autres, le pseudocode schématique et un guide de programmation afin de vérifier si l'implantation des pratiques proposées rapportent les dividendes escomptés. L'élaboration d'autres applications serait aussi souhaitable. Enfin, il sera nécessaire de garder un oeil sur l'avancement des recherches dans ce domaine en plein essor.

BIBLIOGRAPHIE

ADELSON, B. et SOLOWAY, E. (1985). The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, SE-11 (11), 1351-1360.

ADELSON, B. et SOLOWAY, E. (1988). A Model of Software Design. *The Nature of Expertise*, M.T.H. Chi, R. Glaser, M.J. Farr, Lawrence Erlbaum Associates, 185-208.

ANDERSON, R.J., HEATH, C.C., LUFF, P. et MORAN, T.P. (1993). The social and the cognitive in human-computer interaction. *Int. J. Man-Machine Studies*, 38, 999-1016.

BÖHM, C. et JACOPINI, G. (1966). Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules. *Communications of the ACM*, 9, 366-371.

BROOKS, R. (1977). Towards a theory of the cognitive processes in computer programming. *Int. J. Man-Machine Studies*, 9, 737-751.

BROOKS, R. (1983). Towards a theory of the comprehension of computer programs. *Int. J. Man-Machine Studies*, 18, 543-554.

CAÑAS, J.J., BAJO, M.T. et GONZALVO, P. (1994). Mental models and computer programming. *Int. J. Human-Computer Studies*, 40, 795-811.

CHEN, H.G. et VECCHIO, R.P. (1992). Nested IF-THEN-ELSE constructs in end-user computing: personality and aptitude as predictors of programming ability. *Int. J. Man-Machine Studies*, 36, 843-859.

CLIFTON, M.H. (1978). A technique for making structured programs more readable. *ACM SIGPLAN Notices*, 4, 58-63.

CURTIS, B. (1984). Fifteen years of psychology in software engineering: individual differences and cognitive science. *Proceeding of the Seventh Conference on Software Engineering, Orlando, Florida (USA)*, 97-106.

CURTIS, B. (1989). Cognitive issues in reusing software artifacts. *Software Reusability Vol. II - Applications and Experience*, ACM Press, 269-287.

DAVIES, S.P. (1990). The nature and development of programming plans. *Int. J. Man-Machine Studies*, 32, 461-481.

DAVIES, S.P. (1991). The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior. *Cognitive Science*, 15, 547-572.

DAVIES, S.P. (1993). Models and theories of programming strategies. *Int. J. Man-Machine Studies*, 39, 237-267.

DAVIES, S.P. (1994). Knowledge restructuring and the acquisition of programming expertise. *Int. J. Human-Computer Studies*, 40, 703-726.

DAVIS, J.S. (1995). A guessing measure of program comprehension. *Int. J. Human-Computer Studies*, 42, 245-263.

DÉTIENNE, F. (1988). Une application de la théorie des schémas à la compréhension de programmes. *Le Travail humain*, 51 (4), 335-350.

DÉTIENNE, F. (1989). Une revue des études psychologiques sur la compréhension des programmes informatiques. *Technique et Science Informatiques*, 8, 5-20.

DÉTIENNE, F. et RIST, R. (1995). Introduction of This Special Issue on Empirical Studies of Object-Oriented Design. *Human-Computer Interaction*, 10, 121-128.

DÉTIENNE, F. et SOLOWAY, E. (1990). An empirically-derived control structure for the process of program understanding. *Int. J. Man-Machine Studies*, 33, 323-342.

EBRAHIMI, A. (1994). Novice programmer error: language constructs and plan composition. *Int. J. Human-Computer Studies*, 41, 457-480.

EHRLICH, K., et SOLOWAY, E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. *Human Factors in Computer Systems*, Ablex, Norwood, 113-133.

GILMORE, D.J. et GREEN, T.R.G. (1988). Programming Plans and Programming Expertise. *The Quaterly Journal of Experimental Psychology*, 40A (3), 423-442.

GREENO, J.G. (1973). The Structure of Memory and the Process of Problem Solving. *Contemporary Issues in Cognitive Psychology*, Winston, Washington.

GRENIER, A. (1989). *Méthode de construction de traducteurs de pseudocode schématique*. Mémoire M.Sc.A., Université de Montréal, Montréal (Québec).

GUINDON, R. (1990). Designing the Design Process: Exploiting Opportunistic Thoughts. *Human-Computer Interaction*, 5, 305-344.

GUINDON, R. (1990). Knowledge exploited by experts during software system design. *Int. J. Man-Machine Studies*, 33, 279-304.

HALSTEAD, M.H. (1977). *Elements of software science*. Elsevier, New York.

HARRISON, W. et COOK, C. (1986). Are Deeply Nested Conditionals Less Readable ? *The Journal of Systems and Software*, 6, 335-341.

ISELIN, E.R. (1988). Conditional statements, looping constructs, and program comprehension: an experimental study. *Int. J. Man-Machine Studies*, 28, 45-66.

KESLER, T.E., URAM, R.B., MAGAREH-ABED, F., FRITZSCHE, A., AMPORT, C. et DUNSMORE, H.E. (1984). The effect of indentation on program comprehension. *Int. J. Man-Machine Studies*, 21, 415-428.

KHALIL, O.E.M. et CLARK, J.D. (1989). The influence of programmers' cognitive complexity on program comprehension and modification. *Int. J. Man-Machine Studies*, 31, 219-236.

KORBI, T.A. (1989). Program understanding: challenge for the 1990s. *IBM Systems Journal*, 28, 294-306.

LARKIN, J.H. et SIMON, H.A. (1987). Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, 11, 65-99.

LETOVSKY, S. (1987). Cognitive Processes in Program Comprehension. *Journal of Systems and Software*, 7, 325-339.

LUKEY, F.J. (1980). Understanding and debugging programs. *Int. J. Man-Machine Studies*, 12, 189-202.

MIARA, R.J., MUSSELMAN, J.A., NAVARRO, J.A. et SHNEIDERMAN, B. (1983). Program Indentation and Comprehensibility. *Communications of the ACM*, 26 (11), 861-867.

MILLER, G.A. (1956). The magical Number Seven Plus or Minus Two: some Limits on our Capacity for Processing Information. *Psychological Review*, 63, 81-97.

MILLER, G.A. (1975). The magic number seven after fifteen years. *Studies in Long Term Memory*. A.Kennedy ed., John Wiley & Sons.

MYNATT, B.T. (1984). The effect of semantic complexity on the comprehension of program modules. *Int. J. Man-Machine Studies*, 21, 91-103.

NEWELL, A., SIMON, H.A. (1972). *Human Problem Solving*. Prentice-Hall.

PARNAS, D.L., MADEY, J. et IGLEWSKI, M. (1994). Precise Documentation of Well-Structured Programs. *IEEE Transactions on Software Engineering*, 20 (12), 948-976.

PENNINGTON, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19, 295-341.

PETRE, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38 (6), 33-44.

PREECE, J. et ROMBACH, H.D. (1994). A taxonomy for combining software engineering and human-computer interaction measurement approaches: towards a common framework. *Int. J. Human-Computer Studies*, 41, 553-583.

REYNOLDS, R.G., MALETIC, J.I. et PORVIN, S.E. (1992). Stepwise Refinement and Problem Solving. *IEEE Software*, septembre 1992, 79-88.

RIST, R.S. (1989). Schema Creation in Programming. *Cognitive Science*, 13, 389-414.

ROBERTSON, S.P., et YU, C.C. (1990). Common cognitive representations of program code across tasks and languages. *Int. J. Man-Machine Studies*, 33, 343-360.

ROBILLARD, P.N. (1986). Schematic pseudocode for program constructs and its computer automation by Schemacode. *Communications of the ACM*, 29 (11), 1072-1089.

ROBILLARD, P.N. et THALMANN, D. (1980). *Complex problem solving using schematic pseudocode (SPC).* Publication #373, Université de Montréal, Montréal (Québec).

SCHACH, S.R. (1993). *Software Engineering*. Second Edition, Aksen Associates.

SCHANK, R.C. et ABELSON, R.P. (1977). *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum Associates.

SHEIL, B.A. (1981). The Psychological Study of Programming. *ACM Computing Surveys*, 13 (1), 101-120.

SHEPPARD, S.B., CURTIS, B., MILLIMAN, P. et LOVE, T. (1979). Modern Coding Practices and Programmer Performance. *IEEE Computer*, Décembre 1979, 41-49.

SHNEIDERMANN, B. (1976). Exploratory Experiments in Programmer Behavior. *International Journal of Computer and Information Sciences*, 5 (2), 123-143.

SHNEIDERMAN, B. (1977). Measuring computer program quality and comprehension. *Int. J. Man-Machine Studies*, 9, 465-478.

SHNEIDERMAN, B. (1981). *Software psychology*. Winthrop Publishers.

SHNEIDERMAN, B. et MAYER, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences*, 8 (3), 219-238.

SHNEIDERMAN, B., MAYER, R., McKAY, D. et HELLER, P. (1977). Experimental Investigations of the Utility of Detailed Flowcharts in Programming. *Communications of the ACM*, 20 (6), 373-381.

SIME, M.E., GREEN, T.R.G. et GUEST, D.J. (1973). Psychological Evaluation of Two Conditional Constructions Used in Computer Languages. *Int. J. Man-Machine Studies*, 5, 105-113.

SIME, M.E., GREEN, T.R.G. et GUEST, D.J. (1977). Scope marking in computer conditionals - a psychological evaluation. *Int. J. Man-Machine Studies*, 9, 107-118.

SKUCE, D. (1995). Knowledge management in software design: a tool and a trial. *Software Engineering Journal*, Septembre 1995, 183-193.

SOLOWAY, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29 (6), 850-858.

SOLOWAY, E., ADELSON, B. et EHRLICH, K. (1988). Knowledge and Processes in The Comprehension of Computer Programs. *The Nature of Expertise*, M.T.H. Chi, R. Glaser, M.J. Farr, Lawrence Erlbaum Associates, 129-152.

SOLOWAY, E., BONAR, J. et EHRLICH, K. (1983). Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, 26 (11), 853-860.

SOLOWAY, E., et EHRLICH, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10 (5), 595-609.

SOLOWAY, E., EHRLICH, K., BONAR, J. et GREENSPAN, J. (1983). What Do Novices Know About Programming ?. *Directions in human-computer interactions*, Norwood: Ablex, 27-54.

SOLOWAY, E., WOOLF, B., RUBIN, E. et BARTH, P. (1981). Meno-II: An intelligent tutoring system for novice programmers. *Proceedings of IJCAI-81 Vancouver*, 975-977.

SPOHRER, J.C. et SOLOWAY, E. (1986). Novice mistakes: are the folk wisdoms correct ?. *Communications of the ACM*, 29 (7), 624-632.

SUTCLIFFE, A.G. et MAIDEN, N.A.M. (1992). Analysing the novice analyst: cognitive models in software engineering. *Int. J. Man-Machine Studies*, 36, 719-740.

TEASLEY, B.E. (1994). The effects of naming style and expertise on program comprehension. *Int. J. Human-Computer Studies*, 40, 757-770.

VISSER, W. (1992). Designers' activities examined at three levels: organization, strategies and problem-solving processes. *Knowledge-Based Systems*, 5 (1), 92-104.

VISSER, W. (1994). Organisation of design activities: opportunistic, with hierarchical episodes. *Interacting with Computers*, 6 (3), 239-274.

VON MAYRHAUSER, A. et VANS, A.M. (1995). Program Understanding: Models and Experiments. *Advances in computers*, 40, 1-38.

VON MAYRHAUSER, A. et VANS, A.M. (1995). Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28, 44-55.

VON MAYRHAUSER, A. et VANS, A.M. (1995). Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, 10, 171-182.

WEINBERG, G.M. (1971). *The psychology of computer programming*. Litton Educational Publishing.

WEISSMAN, L. (1974). Psychological complexity of computer programs: an experimental methodology. *ACM SIGPLAN Notices*, 6, 25-36.

WIEDENBECK, S. (1986). Beacons in computer program comprehension. *Int. J. Man-Machine Studies*, 25, 697-709.

WIEDENBECK, S. (1991). The initial stage of program comprehension. *Int. J. Man-Machine Studies*, 35, 517-540.

WIEDENBECK, S., FIX, V. et SCHOLTZ, J. (1993). Characteristics of the mental representations of novice and expert programmers: an empirical study. *Int. J. Man-Machine Studies*, 39, 793-812.

WOODFIELD, S.N., DUNSMORE, H.E. et SHEN, V.Y. (1981). The Effect of Modularization and Comments on Program Comprehension. *Proceedings of the 5th International Conference on Software Engineering*, IEEE Computer Society Press, 215-223.

ANNEXE I : Sortie du programme présenté en introduction

On the first day of Christmas my true love gave to me
a partridge in pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in pear tree.

On the fourth day of Christmas my true love gave to me
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

On the fifth day of Christmas my true love gave to me
five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

On the sixth day of Christmas my true love gave to me
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

On the seventh day of Christmas my true love gave to me
seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

On the eigth day of Christmas my true love gave to me
eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

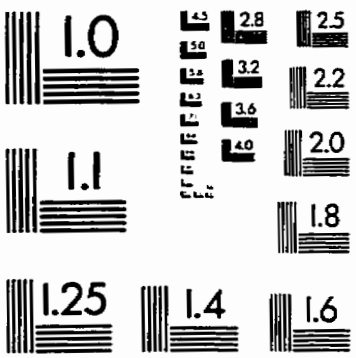
On the ninth day of Christmas my true love gave to me
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

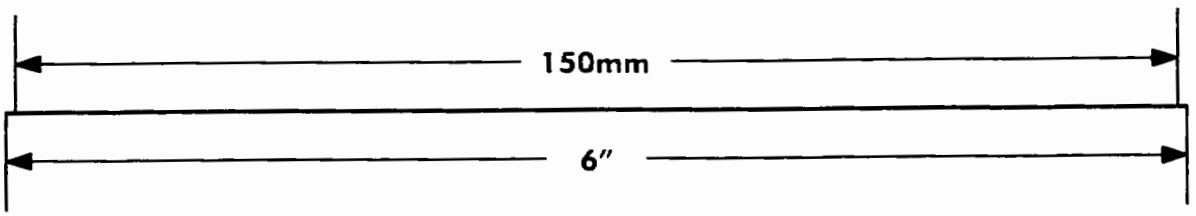
On the tenth day of Christmas my true love gave to me
ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

On the eleventh day of Christmas my true love gave to me
eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

On the twelfth day of Christmas my true love gave to me
twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in pear tree.

IMAGE EVALUATION
TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989