HAL
open science

La conception et réutilisation de logiciels : l'approche de l'ergonomie cognitive

Françoise Détienne

▶ To cite this version:

Françoise Détienne. La conception et réutilisation de logiciels : l'approche de l'ergonomie cognitive. [Rapport de recherche] RR-2902, INRIA. 1996. inria-00073790

HAL Id: inria-00073790
https://inria.hal.science/inria-00073790v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

La conception et réutilisation de logiciels : l'approche de l'ergonomie cognitive

Françoise Détienne

N° 2902
Mai 1996

THÈME 3

Rapport de recherche

La conception et réutilisation de logiciels : l'approche de l'Ergonomie Cognitive

Françoise Détienne

Thème 3
Projet Psycho-Ergo
Rapport de recherche n° 2902 - Mai 1996 - 24 pages

Résumé : Ce texte a pour objectif de présenter un domaine de recherche appelé "Psychologie de la Programmation" qui a vu son essor croître depuis les années 80. Après un bref historique présentant les évolutions thématiques et méthodologiques de ce domaine de recherche, nous présentons une synthèse des résultats d'études empiriques menées sur l'activité de conception de programmes et sur les processus de réutilisation dans la conception.

Mots-clés : psychologie de la programmation, conception de programmes, réutilisation, processus cognitif, représentation, stratégie.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)
Téléphone : +33 (1) 39 63 55 11 - Télécopie : +33 (1) 39 63 58 92

Software design and reuse activities: the Cognitive Ergonomics approach

Abstract: This paper presents a domain of research called "Psychology of Programming" which has been developed since the 80's. After a brief historical survey highlighting the thematic and methodological evolutions in this domain, we present a state-of-art review of empirical research conducted on the activity of software design and on the reuse processes involved in design.

Keywords: psychology of programming, software design, software reuse, cognitive process, representation, strategy.

1 Introduction

Ce texte a pour objectif de présenter un domaine de recherche appelé "Psychologie de la Programmation" ou "Ergonomie Cognitive de la Programmation" qui a vu son essor croître depuis les années 80. Après un bref historique présentant les évolutions thématiques et méthodologiques de ce domaine de recherche, nous présenterons une synthèse des résultats d'études empiriques menées sur l'activité de conception de programmes et sur les processus de réutilisation dans la conception. L'étude de la conception, qui plus est individuelle, n'est qu'un des thèmes abordés dans ce domaine de recherche. D'autres thèmes concernant d'autres tâches de programmation (Pennington & Grabovski, 1990) sont la compréhension de programmes (voir par exemple : Détienne,1990a ; Pennington, 1987), la détection et correction d'erreurs (voir par exemple : Gilmore 1991), la modification de programmes (voir par exemple : Littman, Pinto, Letovsky & Soloway, 1986), l'apprentissage de la programmation par analogie à des situations familières (voir par exemple : Hoc, 87), les mécanismes de transfert inter-langages (voir par exemple : Chatel, Détienne & Borne, 1992 ; Détienne, 1990b ; Scholtz & Wiedenbeck, 1990), la visualisation de programmes (voir par exemple : Petre, 1995), l'étude des structures notationnelles (voir par exemple : Green, 1990).

2 Historique

Le domaine de Psychologie de la programmation est né dans les années 70. Les recherches menées alors avaient pour objectif principal l'évaluation d'outils informatiques en terme de performance (Shneiderman, 1980). C'étaient des études factorielles visant à analyser les effets de différents facteurs (par exemple, l'indentation, les commentaires...) sur la performance dans différentes tâches de programmation et, ceci, sans modèles cognitifs pour rendre compte de l'activité dans ces différentes tâches. Alors que les études sur la programmation informatique étaient menées alors principalement par des informaticiens, ce domaine a suscité depuis une dizaine d'années un intérêt accru de la part des psychologues et ergonomes qui y ont vu un terrain pour étudier des activités de conception, de compréhension, de résolution de problèmes par l'expert ainsi que le moyen de développer et évaluer des outils d'aide à ces activités. Cette approche consiste à emprunter des modèles théoriques issus de la Psychologie Cognitive et à les enrichir au travers de l'analyse de tâches réelles, plus écologiques, de telles analyses faisant largement défaut à la Psychologie Expérimentale.

Les psychologues ont mis l'accent sur l'analyse des activités de programmation et la construction de modèles de ces activités. L'objectif pratique est, à travers cette analyse, d'améliorer l'activité en adaptant l'outil informatique à son utilisateur. Ces études mettent en valeur certaines causes des difficultés dont les programmeurs font l'expérience dans la réalisation de leurs tâches. Elles tiennent au fait que les outils dont ils disposent ne sont pas adaptés aux processus et représentations cognitives de leurs utilisateurs. La démarche actuellement suivie dans le développement logiciel ne faisant aucune référence à l'activité de l'utilisateur est en partie responsable de cette inadaptation. Les outils, langages, méthodes et environnements de programmation sont développés sur la base de modèles formels de la programmation, par exemple, le modèle hiérarchique. Or, les études psychologiques montrent qu'il y a un écart important entre ces modèles informatiques de la programmation et l'activité réelle des programmeurs.

Le domaine de la Psychologie de la Programmation a connu une double évolution, méthodologique et thématique, pendant cette dernière décennie.

Sur un plan méthodologique, il y a moins d'études de type factorielles qui, tout en essayant de quantifier l'effet de facteurs externes sur l'activité, ignoraient les mécanismes cognitifs responsables de ces effets. Les études récentes sont plus de type clinique, avec des analyses fines de l'activité selon le paradigme des protocoles verbaux. Cela a permis la construction de modèles descriptifs de l'activité.

Sur un plan thématique, il y a eu aussi une évolution. En 1986, deux papiers (Curtis, 1986 ; Soloway, 1986) présentés au 1er workshop sur "Empirical Studies of Programmers" faisaient état des orientations thématiques et des limitations des recherches menées jusqu'alors dans ce domaine : beaucoup d'études sur des tâches proches de l'activité de programmation proprement dite (codage), beaucoup d'études sur des novices en programmation. L'évolution récente a vu l'émergence de recherches sur des activités comme la conception, et notamment sur les mécanismes de raisonnement dans la résolution de problème de conception (plus éloignés de la manipulation de langages de programmation -tout en reconnaissant leur effet sur cette activité-). De plus, ces recherches se sont plus centrées sur l'activité des experts en informatique. Cela a permis d'aborder la modélisation des connaissances et des raisonnements des experts dans un domaine complexe.

D'un point de vue pratique, la recherche en psychologie de la programmation pose le problème de la distance entre les représentations et les traitements humains et les systèmes formels permettant de représenter ces représentations et de manipuler ces représentations de second niveau. De ce point de vue, elle présente un intérêt en génie logiciel ; en effet, l'étude de des activités de programmation permet de construire des modèles de ces activités qui peuvent guider le développement de langages de programmation, de méthodes et d'aides à la résolution de problèmes chez l'expert et le novice.

3 L'activité de conception de logiciels

Après une présentation des caractéristiques des problèmes de conception, nous présenterons deux types d'approches suivies dans les études sur l'activité de conception de logiciel : les approches centrées sur les connaissances et, les approches centrées sur les stratégies. Notre dernier point concernera l'organisation de l'activité de conception, un niveau plus "méta" par rapport aux stratégies.

3.1 Caractéristiques des problèmes de conception

L'étude de l'activité de conception de programmes a été menée dans le cadre des études sur les activités de résolution de problèmes. L'activité de résolution de problèmes est habituellement décrite comme idéalement composée de trois phases successives : compréhension du problème, décomposition du problème en une solution, codage de la solution. En fait, ces phases se chevauchent la plupart du temps.

L'activité de conception de programmes, comme d'autres activités de conception (par exemple, la conception architecturale), présente comme particularité que les problèmes à résoudre sont "mal définis" (Visser & Hoc, 1990). Un problème est mal défini dans la mesure où des

spécifications du problème manquent et solutionner le problème consiste, en partie, à introduire des contraintes. Une autre particularité de ce domaine est qu'il existe plusieurs solutions acceptables pour le même problème lesquelles solutions ne peuvent être jugées satisfaisantes selon un critère unique.

Un aspect également important de l'activité de conception de programmes est que le concepteur utilise des connaissances dans deux domaines (Brooks, 1977), le domaine d'application et le domaine informatique et qu'il fait une mise en correspondance ou "mapping" entre ces deux domaines. En simplifiant, on peut dire que le programmeur construit au moins deux types de représentations mentales : un modèle du problème et de la solution dans des termes du domaine d'application et un modèle de la solution informatique. Une partie de son activité consiste à transformer et passer d'une représentation à une autre. Selon les caractéristiques de la situation de conception la distance entre ces modèles est plus ou moins grande : on peut supposer que certains paradigmes de programmation, par exemple le paradigme orienté-objet (Rosson & Alpert, 1990), réduisent cette distance dans des domaines d'application particuliers.

3.2 Approches centrées sur les connaissances

De nombreuses recherches menées en Psychologie de la Programmation (Adelson, 1981 ; 1984 ; Black, Kay & Soloway, 1986 ; Détienne, 1990c ; Détienne & Soloway, 1990 ; Rist, 1986 ; 1989 ; 1991 ; Robertson, 1990 ; Robertson & Yu, 1990 ; Soloway, Ehrlich & Bonar, 1982 ; Soloway, Ehrlich, Bonar & Greenspan, 1982 ; Soloway & Ehrlich, 1984) se sont centrées sur l'analyse et la formalisation des connaissances des experts en programmation informatique. Elles se réfèrent généralement à la Théorie des Schémas comme un outil de description des connaissances de l'expert en programmation. C'est une théorie sur l'organisation des connaissances en mémoire et sur les processus de mise en oeuvre de ces connaissances. Un schéma est une structure de données qui représente des concepts génériques stockés en mémoire. Ce concept a été développé en Intelligence Artificielle (Minsky, 1975 ; Schank & Abelson, 1977) et dans des études psychologiques sur la compréhension de textes (Galambos, Abelson & Black, 1986).

Les études sur la programmation (Détienne, 1988 ; 1990a ; Soloway, Ehrlich & Bonar, 1982 ; Soloway & Erhlich, 1984) montrent que l'informaticien expert possède en mémoire des schémas des types suivants :

-des schémas élémentaires représentent des connaissances sur la structure de contrôle et les variables (appelés "control plan" et "variable plan" par Soloway). Par exemple, un schéma de compteur représente les différentes connaissances que l'on a sur le mode d'utilisation d'une variable de type compteur dans un programme et les valeurs qu'elle peut prendre comme valeurs d'initialisation et de mise à jour. Ce schéma peut être formalisé comme une structure de variable de la façon suivante :

But : compte les occurrences d'une action
Initialisation : Compteur:=n
Mise à Jour : Compteur:=Compteur+Increment
Type : entier
Contexte d'utilisation : itération

Remarquons que des valeurs sont plus prototypiques que d'autres comme une initialisation à "0" et un incrément de valeur "1".

-des schémas algorithmiques représentent des connaissances sur la structure d'algorithmes. Par exemple, un programmeur connaît des algorithmes de tri, de recherche, etc. Ces algorithmes sont plus ou moins abstraits et dépendants d'un langage de programmation et ils peuvent se décrire comme, en partie, composés de schémas élémentaires. Par exemple, un schéma de recherche séquentielle est moins abstrait qu'un schéma de recherche et il peut se décrire comme composé, en partie, d'un schéma de compteur.

-des schémas dépendants du domaine d'application représentent des connaissances que l'informaticien a sur certains types de problèmes. Par exemple, il sait, par expérience dans le domaine d'application, qu'un problème de gestion réalise en général des fonctions de type création, modification, suppression.

En ce qui concerne les schémas propres au domaine de la programmation, Soloway et al. (Soloway, Ehrlich & Bonar, 1982 ; Soloway & Erhlich, 1984) font la distinction entre des schémas tactiques ("tactical plans") et stratégiques ("strategic plans") qui sont indépendants du langage de programmation, du moins dans le cadre d'un paradigme de programmation, et les schémas d'implémentation ("implementation plans") qui sont eux dépendants d'un langage particulier.

Il existe différents types de liens entre les schémas : (1) des liens de composition, par exemple, un schéma de recherche peut se décrire comme une composition de plusieurs schémas élémentaires, (2) des liens de spécialisation "is kind of", par exemple, un schéma de recherche séquentielle est un type de schéma de recherche et, (3) des liens de mise en oeuvre, par exemple, un schéma de boucle "For Loop Plan" (en Pascal) est une mise en oeuvre d'un schéma de boucle "Counter Controlled Running Total Loop plan"

Rist (1986) a introduit la notion de rôle ("role"). Un schéma peut être formalisé comme constitué des rôles suivants : "input, calculate, output". Cette structure de rôles est acquise très tôt par le novice en programmation alors que la structure des plans (ou schémas) est construite peu à peu au cours de l'expérience en programmation. Ainsi, alors que le novice doit construire les parties constituantes des plans quand il développe un programme (sans avoir la structure de connaissance correspondante en mémoire), l'expert récupère des schémas en mémoire et les particularisent.

Une autre notion, introduite par Rist (1986, 1991), est la notion de "focus" ou "ligne focale" ("focal line"). Le focus représente la partie de la solution qui réalise directement le but du problème. Ainsi pour un problème de calcul de moyenne, cette partie correspondrait au calcul de la moyenne proprement dit (division du total par la valeur de l'increment). Le focus constitue la partie la plus importante d'un schéma.

Cette partie est certainement celle qui est la plus disponible quand un schéma est évoqué (Galambos, Abelson, & Black, 1986). C'est aussi celle qui, lors de la lecture d'un programme, a le plus de chance de déclencher l'activation d'un schéma. La notion de "beacons", introduite par Brooks (1983) et étudiée notamment par Wiedenbeck (Wiedenbeck, 1986 ; Wiedenbeck & Scholtz, 1989) correspond à cette partie focale du code d'un programme : la reconnaissance de "beacons" déclenche l'activation de schémas et des processus descendants, d'attente dans la

compréhension de programme. Dans la conception, la partie focale peut aussi être celle qui est récupérée et exprimée la première, le code étant construit autour du focus (Davies, 1993 ; Rist, 1989 ; à paraître).

Dans la construction des connaissances en programmation on peut distinguer plusieurs étapes : construction de schémas élémentaires ("simple plans"), construction de schéma complexes ("complex plans"), hiérarchisation de la structure des schémas par l'abstraction de la partie focale (Davies, 1994 ; Rist, 1991).

L'expert possède également des règles du discours ("rules of programming discourse") qui régiraient la construction des programmes et notamment la particularisation des schémas lors de la conception (Soloway, Ehrlich, Bonar & Greenspan, 1982). Par exemple, une de ces règles est "le nom d'une variable doit refléter sa fonction".

Remarquons que la théorie des schémas permet de rendre compte de certains mécanismes cognitifs mis en oeuvre dans l'activité de conception de programmes mais aussi dans l'apprentissage de programmation ainsi que lors de la compréhension de programmes. L'activité de conception de programmes consiste, en partie, à récupérer des schémas conservés en mémoire, adéquats pour traiter certains problèmes, et à les combiner ensemble pour construire un programme. L'apprentissage de la programmation est caractérisé par la construction progressive de schémas de programmation ; le novice transfère des solutions particulières et des solutions génériques (appelées des "schémas relatifs au problème posé") construites dans le domaine d'application. L'activité de compréhension de programmes consiste en partie en l'activation de schémas stockés en mémoire par des indices extraits du code du programme et l'inférence de certaines informations à partir des schémas évoqués (Détienne, 1988 ; 1990a, 1990c). Une limite de cette approche en terme de schémas est, en se centrant principalement sur les mécanismes d'activation de schémas et les processus d'attente créés par ces activations, de peu rendre compte des processus plus ascendants, constructifs, dans ces différentes activités.

3.3 Approches centrées sur les stratégies

Toutes les études précédemment citées se sont centrées sur l'analyse des connaissances des experts et n'ont que peu analysé les processus de mise en oeuvre de ces connaissances. Ainsi l'aspect stratégique dans la conception de programmes était peu abordé. Or, il semble que l'expert se caractérise non seulement par des connaissances plus abstraites, organisées hiérarchiquement, mais aussi par une panoplie de stratégies plus large et des stratégies plus adaptées que le novice (Gilmore, 1990). Ainsi les experts choisiraient leurs stratégies de conception en fonction de facteurs comme la familiarité de la situation, les caractéristiques de la tâche, les caractéristiques notationnelles des langages. Les novices éprouvent souvent des difficultés non seulement par manque de connaissances adéquates mais aussi par manque de stratégies adéquates pour répondre à une situation particulière.

3.3.1 Dimensions caractérisant les stratégies

De nombreuses études ont porté sur les stratégies de conception de programmes (Brooks, 1977 ; Chatel & Détienne, 1996 ; Détienne, 1995 ; Guindon, 1990 ; Rist, 1996 ; à paraître ; Visser, 1987 ; Visser & Hoc, 1990). Les stratégies de conception peuvent être caractérisées selon plusieurs dimensions :

- la direction ascendante (de moins abstrait vers plus abstrait) versus descendante du développement de la solution,

- le développement prospectif (dans le sens d'exécution de la procédure) versus rétrospectif de la solution,

- le développement en largeur d'abord (tous les éléments de solution sont développés à un même niveau d'abstraction) versus en profondeur d'abord de la solution,

- le guidage procédural de la résolution (c'est un plan de la procédure qui guide la résolution, la recherche est alors basée sur les buts ou procédures) versus déclaratif (c'est un plan de propriétés statiques qui guide la résolution, la recherche est alors basée sur des rôles or des objets)

Des activités de simulation sont aussi mises en oeuvre dans un objectif d'évaluation de la solution. En effet, les informaticiens ont souvent recours à des simulations mentales sur une solution partielle ou totale développée à un niveau d'abstraction plus ou moins grand ou sur des parties de code qu'ils cherchent à comprendre (Adelson & Soloway, 1984 ; Détienne 1984 ; Guindon, Krasner & Curtis, 1987 ; Visser 1987). La simulation permet d'évaluer si une solution répond bien aux buts et d'intégrer des solutions partielles en contrôlant les interactions.

3.3.2 Conditions de déclenchement des stratégies

Une question importante de recherche est de caractériser les conditions déclenchantes des stratégies. Des déterminants externes sont les caractéristiques de la notation (Green, Bellamy, & Parker, 1987) les caractéristiques de l'environnement (Détienne, 1994a) et les types de problèmes (Hoc, 1983 ; Chatel & Détienne, 1996). Des déterminants internes sont liés à l'expertise du sujet dans le domaine informatique, notamment, à la disponibilité de schémas de programmes construits en mémoire (Rist, 1991)

La structure notationnelle des langages de programmation facilite ou non certains traitements. Chaque notation met en valeur des types différents d'information. Donc la réalisation de tâches qui nécessitent de rendre explicites certains types d'information est facilitée par une structure de notation adéquate, c'est-à-dire, qui rend explicite les mêmes types d'information. Des psychologues ont étudié la structure notationnelle des langages de programmation en faisant ressortir différentes caractéristiques qui ont un effet sur l'activité (Green, 1990). Les auteurs ont mis en évidence les caractéristiques suivantes : (1) les pointeurs cachés, (2) l'entremêlement des composants de schémas différents, (3) La propriété d'uniformité syntaxique et lexicale, (4) la redondance, (5) la résistance aux changements, (6) les contraintes d'ordre.

Les stratégies de conception mises en oeuvre par les experts varient selon les structure de la notation (Green, Bellamy & Parker, 1987). Ainsi, par exemple, si la notation présente de fortes caractéristiques de résistance au changement et est donc difficile à modifier car une modification ponctuelle entraîne beaucoup d'autres modifications dans le code (par exemple Pascal en comparaison avec Basic), le concepteur essaye de minimiser les modifications à faire dans le code lors de la conception et le codage. Le programmeur en Pascal met alors en oeuvre une stratégie plutôt descendante par raffinements successifs avec des prises de notes fréquentes. Le programmeur en Basic met en oeuvre une stratégie beaucoup plus linéaire et prospective, ce qui le conduit fréquemment à des retours en arrière quand des ajouts/modifications sont à faire.

Dans ce dernier cas, les modifications sont beaucoup moins coûteuses qu'en Pascal car la notation est beaucoup moins résistante aux changements.

Le type de problème va également influencer le choix de la stratégie mise en oeuvre. Hoc (1981 ; 1983) distingue plusieurs dimensions permettant de caractériser les problèmes : la dimension prospective versus rétrospective et la dimension déclarative versus procédurale. Ainsi, un problème déclaratif se caractérise par une forte structure de données qui va orienter et guider la recherche de la solution. Le sujet utilise alors un plan que nous avons qualifié précédemment de déclaratif. Un problème procédural se caractérise par une forte structure de procédure qui va guider l'activité de conception. Le sujet utilise alors un plan procédural.

Cette typologie a récemment été remise en cause dans le cadre de la conception orientée-objet (Détienne, 1995 ; Chatel & Détienne, 1996). Avec ce paradigme, la dimension déclarative/procédurale ne semble pas pertinente et nous avons suggéré une nouvelle dimension qui caractérise non seulement la structure des objets (ou données) d'une part et la structure de la procédure, d'autre part, mais également la façon dont les deux sont associées. Le plan de conception mis en oeuvre par des experts est déclaratif quand le problème présente une structure de solution hiérarchique avec des communications verticales entre les objets alors que le plan est procédural quand le problème présente une structure de solution plate avec des communications horizontales entre les objets.

Des déterminants internes sont liés à l'expertise du sujet dans le domaine informatique et à la disponibilité de schémas de programmes construits en mémoire (Rist, 1991). Ainsi, l'expert peut suivre une stratégie descendante et prospective pour un problème qui lui est familier et pour lequel il dispose déjà d'un plan (schéma) de procédure abstrait alors que le novice suivra plutôt une stratégie ascendante et rétrospective.

Mais souvent c'est l'interaction entre plusieurs facteurs qui déclenche une stratégie. Ainsi par exemple, l'effet de la notation (Davies, 1991) serait peu important pour des novices ou des experts mais est déterminant dans le déclenchement de stratégies de conception à des niveaux d'expertise intermédiaires.

3.4 Organisation de l'activité : hiérarchique versus opportuniste

Au niveau de l'organisation de l'activité, niveau plus "méta" par rapport à une analyse en termes de stratégies, deux types de modèles s'opposent : (1) le modèle hiérarchique basé sur des modèles normatifs inspirés de méthodes de programmation, et (2) les modèles opportunistes basés sur les résultats d'étude empiriques qui mettent l'accent sur les déviations de l'activité réelle par rapport à un modèle strictement hiérarchique.

Le modèle hiérarchique a été fortement inspiré par la programmation structurée, Adelson et Soloway (1984) ont ainsi caractérisé les processus de décomposition du problème en une solution comme essentiellement descendants et privilégiant une recherche de solution en largeur d'abord. Tous les buts (ou fonctions) de la solution sont identifiés à un certain niveau d'abstraction avant d'être raffinés successivement à des niveaux de plus en plus détaillés. Ce raffinement consiste soit, en la description des éléments qu'un plan de solution représente, soit, en des choix de mises en oeuvre particulières de fonctions plus générales. L'activité prescrite consiste à travailler à un niveau d'abstraction à la fois, en privilégiant une recherche de la

solution en largeur d'abord, avec chaque niveau juste un peu plus détaillé que le précédent, plutôt que de détailler une partie de la solution seulement en privilégiant une recherche en profondeur.

Des études (Guindon, Krasner & Curtis, 1987 ; Guindon, 1990 ; Visser, 1987) mettent l'accent sur les déviations de l'activité des informaticiens par rapport à un modèle hiérarchique. Alors que les concepteurs semblent avoir cette approche comme modèle de leur activité (c'est en effet souvent un modèle didactique), ils se comportent souvent très différemment. L'activité réelle est organisée de façon opportuniste : par exemple, la recherche de solution peut se faire aussi bien de façon descendante qu'ascendante et la solution est recherchée en privilégiant aussi bien la largeur que la profondeur selon la situation. Une structure hiérarchique rend compte du résultat de l'activité mais pas de l'activité elle-même.

Ainsi, ces études témoignent d'une part, de la focalisation sur différents aspects de la solution ou sur différents sous-problèmes au cours de l'activité de conception et, d'autre part, de la mise en oeuvre possible de différentes stratégies de conception pour traiter de ces différents aspects. La solution, de type hiérarchique, est développée en faisant des sauts entre des niveaux d'abstraction variés. Ainsi, les informaticiens raffinent parfois seulement certains aspects de la solution qu'ils estiment critiques (Jeffries, Turner, Polson & Atwood, 1981 ; Visser, 1987). Cela leur permet notamment de détecter des interactions potentielles en développant une partie de la solution qui peut avoir des incidences sur d'autres parties. Ils exploitent parfois des solutions connues analogues ce qui peut entraîner des processus ascendants et aussi des mécanismes de raisonnement par analogie.

Parmi les raisons qui amènent le concepteur à s'écarter d'un plan hiérarchique, Visser (1994a ; 1994b) cite l'utilisation économique des moyens disponibles : si les informations qui permettraient de traiter un certain aspect de la solution ne sont pas disponibles, le sujet met en attente le traitement de cet aspect de la solution ; par contre l'utilisation économique des informations disponibles peut amener à traiter précocement, par rapport à un plan idéal de l'activité, certains aspects de la solution à des niveaux de détails différents. Il y a donc des mises en attente ou des anticipations au cours de l'activité de conception. Les notes prises par les concepteurs lors de leur activité témoignent souvent de ces phénomènes. Notons au passage que cette intense activité de prise de notes, que l'on peut qualifier d'utilisation de mémoire externe, semble être liée à l'expertise et a été analysée par Davies (1996) dans le cadre des modèles dits "display-based problem solving".

Selon les auteurs, la mise en oeuvre d'une activité opportuniste est déclenchée par des facteurs différents. Ainsi pour Davies & Castell (1994) le plan qui guide l'activité est hiérarchique et des épisodes opportunistes locaux sont déclenchés principalement par des défaillances cognitives de la mémoire de travail. Visser (1994a) défend que, bien que les épisodes opportunistes puissent être déclenchés par la limitation des ressources cognitives, les déviations opportunistes sont contrôlées à un niveau méta-cognitif par des critères qui permettent d'évaluer le coût cognitif d'actions alternatives. Ces deux modèles conduisent à des prédictions assez différentes (Détienne, 1994b).

Alors que l'opposition qui a été faite dans ces études entre conception hiérarchique/structurée et opportuniste est aujourd'hui remise en cause (Ball & Ormerod, 1995), ces études ont fait prendre conscience aux concepteurs d'environnements de programmation des contraintes

importantes qu'ils faisaient peser sur les concepteurs en imposant une démarche descendante. Des papiers récents témoignent de ce souci de permettre une conception organisée de façon plus opportuniste (voir par exemple, Bisseret, Burkhardt, Deleuze-Dordron, Détienne & Rouet, 1995 ; Guindon, 1992).

4 La réutilisation dans la conception

Après une discussion des caractéristiques générales de la réutilisation dans la conception, nous présenterons une classification cognitive des situations de réutilisation. Puis nous discuterons de deux types de résultats, apparemment contradictoires, sur les effets de la réutilisation dans la conception : soit, un enrichissement des représentations construites au cours de la conception, soit un abaissement du niveau de contrôle de l'activité. Notre classification cognitive des situations de réutilisation nous permettra d'avancer une explication à ces contradictions. Notre dernier point traitera de l'expertise en réutilisation.

4.1 Caractéristiques générales de la réutilisation dans la conception

Les études sur l'organisation opportuniste de l'activité ont mis en lumière certains phénomènes de réutilisation de solutions ou de raisonnement par analogie, lors de la conception de programmes. Or, beaucoup des études présentées précédemment (notamment celles présentées en 3.2) ont pour limite de peu rendre compte de l'exploitation de solutions antérieures dans l'activité de conception. Ces études se sont centrées sur les raisonnements amenant à l'élaboration de solutions en mettant l'accent sur le rôle de connaissances génériques stockées en mémoire. Or l'élaboration d'une solution se base souvent, non seulement sur l'évocation de connaissances génériques, mais aussi sur la récupération de représentations externes ou internes de solutions particulières développées pour des problèmes analogues (Burkhardt & Détienne, 1995a ; Guindon, 1990 ; Visser & Hoc, 1990).

Parallèlement, la recherche sur la réutilisation de solutions en programmation répond à des préoccupations récentes en génie logiciel. Les développements technologiques et la complexification des logiciels, en matière d'interface par exemple, conduisent à proposer aux concepteurs d'aujourd'hui des composants logiciels (programmes, routines déjà écrites) qui soient "réutilisables", faciles à récupérer et modifier en fonction du nouveau problème que l'on souhaite traiter. L'objectif poursuivi est de minimiser ainsi les coûts de développement tout en augmentant la fiabilité des systèmes conçus.

Deux remarques générales concernent l'adaptation aux utilisateurs des outils de réutilisation développés en génie logiciel. Premièrement, comme pour les méthodes prescrites de conception de type hiérarchique, les outils/modèles de réutilisation préconisent la plupart du temps une approche descendante. Or, les études empiriques montrent que la réutilisation entraîne des processus descendants mais aussi ascendants (Burkhardt et Détienne, 1995b). Deuxièmement, une idée généralement admise dans la communauté informatique est qu'il faut toujours organiser et présenter les composants réutilisables de façon formelle et abstraite. Or, les études empiriques montrent l'importance du recours aux exemples, fortement contextualisés (Burkhardt & Détienne, 1995b ; Rosson & Carroll, 1993 ; Rouet, Deleuze-Dordron & Bisseret, 1995). Par exemple, les études de Rosson et Carroll (1993) sur la réutilisation en Smalltalk soulignent ce fréquent recours à des exemples sous la terminologie "reuse of uses". Dans une tâche de

modification de programmes orientés-objets où les concepteurs peuvent réutiliser des classes existantes, ils observent que les sujets ne réutilisent pas la classe elle-même mais plutôt l'application-exemple ("example application") utilisant cette classe. Cette application-exemple leur fournit des spécifications implicites pour la réutilisation de la classe, notamment beaucoup d'informations contextuelles.

La réutilisation de solutions en conception de programmes met en oeuvre, entre autres, des mécanismes classiquement étudiés sous le thème du raisonnement par analogie ; compréhension de la situation cible, récupération d'une situation source, mise en correspondance entre source et cible. Toutefois, les études sur le raisonnement analogique ont souvent cherché à caractériser principalement les mécanismes d'utilisation de sources, sources construits par l'expérimentateur et fournis aux sujets, dans la résolution de problèmes cibles.

Un aspect particulier à la situation de réutilisation en programmation est que les sources ont été construits préalablement par le concepteur lui-même ou d'autres concepteurs experts du domaine et que, dans une situation naturelle de conception, elles ne sont pas fournies d'emblée au concepteur mais doivent être récupérées par le concepteur. Un autre aspect particulier à cette situation est qu'il s'agit de problèmes de conception pour lesquels il existe différentes solutions alternatives et pour lesquels les contraintes du problème sont mal définies : ce n'est pas le cas de la plupart des situations de résolution de problèmes utilisées dans les études sur le raisonnement par analogie.

4.2 Classification des situations de réutilisation

Burkhardt et Détienne (1994 ; 1995a) ont cherché à intégrer dans une classification des situations de réutilisation, les caractéristiques liées au type d'éléments réutilisés, le type d'épisode de réutilisation et le statut cognitif du composant réutilisé.

Une classification est couramment opérée en Génie Logiciel sur la base du type d'élément réutilisé, auquel est associé un type préférentiel d'activité. Elle distingue l'extraction de code d'une application existante (par exemple, lignes de code ou procédures), la spécialisation de composants en général extraits d'une librairie, et enfin l'héritage/composition de classes. Pour chacune de ces situations, la réutilisation est associée à l'opération de modifications/adaptations à des niveaux différents.

Une autre classification peut être faite selon le type d'épisode de réutilisation, en particulier, selon que l'épisode de réutilisation commence ou non dès l'élaboration du source. Détienne (1991) distingue ainsi des épisodes de "new code reuse" et de "old code reuse".

L'épisode de réutilisation appelé "new code reuse" commence dès l'élaboration du source. Au cours de la conception d'un même programme, le concepteur élabore une solution à un sous-problème (qui aura le statut de source), et envisage de la réutiliser pour solutionner d'autres sous-problèmes à venir (qui auront le statut de cibles). Un processus cognitif particulier à ce type de situation est le processus d'anticipation mis en oeuvre au cours du développement du source. Le concepteur anticipe les adaptations futures du source pour construire les solutions-cibles et construit une opération opérative du source ainsi que des procédures d'adaptation du source en cibles. Dans les situations observées, la source et les cibles étaient des

particularisations ("instances") d'un même schéma mais on peut envisager des situations où il y a une simple relation d'analogie entre solutions particulières sans schéma commun.

Dans la deuxième situation appelée "old code reuse", le concepteur récupère et adapte une solution antérieurement développée pour un problème analogue à celui traité. Pour Détienne (1991) ce type de situation implique que les mécanismes mis en oeuvre concernent alors plutôt la récupération et l'approfondissement de la relation entre solution source et cible, afin de développer la nouvelle solution. Dans cette situation, il semble pertinent de distinguer un premier cas où le concepteur réutilise un élément qu'il a lui même élaboré dans le passé, et un deuxième cas correspondant à la réutilisation d'un élément ayant été élaboré par un autre concepteur.

Enfin, on peut distinguer les situations de réutilisation selon le statut cognitif de l'élément réutilisé qui est différent selon que le concepteur est engagé dans la phase d'analyse du problème, de recherche de solution ou d'implémentation de solution.

Dans la phase d'analyse du problème, l'évocation de problèmes-solutions analogues peut permettre d'inférer de nouvelles spécifications pour le problème cible. Dans la phase de recherche de solution, un composant réutilisable peut servir de modèle qui guide la résolution de problème. Il peut s'ensuivre la remise en cause de solutions choisies précédemment pour le problème-cible. Enfin, dans la phase d'implémentation où la solution est déjà choisie et doit être implémentée, la réutilisation d'un composant va permettre d'économiser la réécriture du code. Dans ce cas, le concepteur cherche un composant intégrable ("pluggable component") moyennant quelques modifications.

Ces distinctions nous semblent rejoindre celles faite par Maiden (1991) qui distingue trois domaines pour lesquels la mise en correspondance des situations source et cible peut se faire : les domaines de problème source et cible ; entre les domaines des objectifs/buts des systèmes conçus ; entre les connaissances sur les deux solutions informatiques.

Cette classification permet d'enrichir le modèle de réutilisation qui prévaut actuellement dans le développement d'outils d'assistance à la réutilisation et qui formalise la tâche de réutilisation comme une boucle faisant intervenir trois sous-tâches (1) localiser un composant ; (2) comprendre le composant ; (3) adapter/modifier le composant en fonction du problème à traiter.

Ainsi, par exemple, selon le type d'épisode de réutilisation, il semble qu'une assistance différente soit requise : assistance à la construction d'une représentation opérative qui pourra être le support d'une automatisation de la modification des solutions-sources en solutions-cibles pour les épisodes de "new code reuse" et assistance à la récupération et à la compréhension pour les épisodes de "old code reuse".

4.3 Enrichissement de la représentation versus abaissement du niveau de contrôle de l'activité

Il y a deux types de résultats, apparemment contradictoires, dans la littérature sur l'activité de réutilisation. D'une part, certaines études montrent que la réutilisation de composants permet d'enrichir des représentations construites au cours de l'activité de conception, par exemple, par

l'ajout de contraintes à la représentation du problème ou l'évocation de solutions alternatives pour résoudre certains sous-problèmes. D'autre part, des études mettent l'accent sur l'utilisation de stratégies par essais et erreurs lors de la réutilisation avec évitement de compréhension du composant réutilisé. Dans ce cas, il n'y pas d'enrichissement des représentations construites au cours l'activité de conception mais il y a plutôt un abaissement du niveau de contrôle de l'activité.

Notre classification cognitive des situations de réutilisation nous permet d'avancer une explication à ces contradictions. Il nous semble que ces deux types de résultats concernent en fait des situations de réutilisation tout à fait différentes : l'enrichissement des représentations construites concerne la réutilisation au cours des phases d'analyse du problème et de recherche de solution ; l'abaissement du niveau de contrôle de l'activité se situe plutôt dans la réutilisation au cours de la phase d'implémentation de solution. Bien qu'il puisse y avoir des interactions entre ces différentes phases, les traiter de façon séparée permet de bien mettre en valeur les processus de différentes natures qui y sont mis en oeuvre.

La réutilisation au cours des phases d'analyse du problème et de recherche de solution a pour effet l'enrichissement des représentations construites. Au cours de la phase d'analyse, la récupération de composant réutilisables, notamment d'exemples, permet, d'une part, l'ajout de nouvelles contraintes à la représentation du problème, et, d'autre part, les contraintes déjà connues sont exprimées à un niveau significativement plus abstrait (Burkhardt & Détienne, 1995a ; 1995b ; De Vries, 1993). Rosson et Carroll (1993) observent que des concepteurs infèrent, à partir d'un exemple analogue, de nouvelles spécifications pour leur situation-problème courante.

Dans la phase de recherche de solution, un composant réutilisable peut servir de modèle qui guide la résolution de problème par l'évocation de solutions alternatives, la construction de plan de résolution ou l'évocation de critères pour l'évaluation de solutions alternatives potentielles (Burkhardt & Détienne, 1994). L'évocation de composants réutilisables amène aussi le concepteur à évoquer des critères de validité liés au contexte dans lequel le composant-source avait été précédemment mis en oeuvre, ce contexte étant jugé similaire au contexte cible. Cette information récupérée est utilisée afin d'évaluer la validité du composant pour la solution courante, entraînant éventuellement la remise en cause totale ou partielle de la solution.

La réutilisation au cours de l'activité d'implémentation de solution a pour effet d'abaisser le niveau de contrôle de l'activité. Le concepteur a choisi une solution qu'il cherche à implémenter en réutilisant un composant intégrable dans sa solution moyennant quelques modifications. Dans cette phase, la réutilisation de composants entraîne un abaissement du niveau de contrôle de l'activité : soit le concepteur utilise des stratégies par essais et erreurs en privilégiant l'utilisation d'outils de deboggage et en minimisant l'effort cognitif nécessaire à la compréhension du composant réutilisable ; soit il/elle se base sur une représentation opérative de la source dans des épisodes de "new code reuse".

Une stratégie de réutilisation qui a souvent été mise en évidence par les études empiriques sur la programmation est une stratégie par essais et erreurs qui consiste à copier et modifier le code (copy/edit style). Cette stratégie a été observée dans le cadre du paradigme procédural mais aussi dans le cadre du paradigme de programmation orientée-objet qui pourtant encourage à un autre style de réutilisation basé sur la propriété d'héritage. Les concepteurs tendent à réutiliser

du code en le copiant et en faisant des modifications qu'ils jugent les plus probables. Ils cherchent à éviter la compréhension du code source et utilise une stratégie dite de "comprehension avoidance", utilisant des indices de surface pour faire des hypothèses sur les fonctionnalités du code source. Ils s'appuient alors plutôt sur les outils de tests et deboggage mis à leur disposition pour modifer et corriger le code pour l'adapter à la situation cible (Lange & Moher, 1989 ; Rosson & Carroll, 1993). Remarquons que cette stratégie est certainement encouragée sinon déterminée par les outils disponibles en phase de codage : les outils de deboggage sont actuellement bien plus développées que les outils d'aide à la compréhension (documentation adéquate, au bon niveau).

Le code réutilisé peut provenir d'autres programmes écrits ou non par le programmeur lui-même ou encore du programme même que le programmeur est en train de développer (Détienne, 1991 ; Rosson & Carroll, 1993). Dans ce dernier cas, Détienne décrit des épisodes de "New code reuse" où la construction de la cible se base sur une représentation opérative de la source mettant l'accent sur les parties à modifier et sur l'utilisation d'une procédure de modification de la solution-source en solutions-cibles. Le développement de plusieurs cibles à partir d'un même source est de préférence fait d'affilée ("reuse in a row") mais peut aussi être interrompu par d'autres activités ("scaterred reuse"). Dans ce dernier cas, il peut y avoir des erreurs par oubli de certaines modifications. L'abaissement du niveau de contrôle de l'activité a pour conséquence la propagation d'erreurs, le programmeur ne recourant pas à une activité d'évaluation de ses solutions-cibles.

Selon le type d'activité dans lequel le concepteur est engagé et le type d'information qu'il recherche dans l'élément réutilisable (contraintes pour la spécification, modèle pour la conception, élément intégrable), l'assistance ne semble pas être du même type. Sur ce dernier point, il semble qu'offrir plusieurs types de documentation d'un même composant qui soient accessibles de façon distincte permettrait d'assister ces différents processus.

4.4 Expertise en réutilisation

Il nous semble important d'insister sur le fait que l'activité de réutilisation nécessite une expertise particulière, distincte de l'expertise en programmation ou dans le domaine d'application. Une étude de terrain sur l'utilisation d'un environnement de réutilisation (Rouet, Deleuze-Dordron & Bisseret, 1995) a ainsi montré clairement que des programmeurs expérimentés, même experts dans le domaine d'application, éprouvent des difficultés importantes quand ils se familiarisent avec l'utilisation d'un tel outil. Ils manquent alors de méthode pour récupérer ou sélectionner des composants, déterminer le bon niveau d'abstraction d'un composant réutilisable, etc. De même, Woodfield, Embley et Scott (1987) montrent que des concepteurs peu expérimentés en réutilisation estiment difficilement la réutilisabilité d'un composant qu'ils n'ont pas conçu, en regard des spécifications d'un problème cible à traiter. Ils basent en effet leur choix sur des caractéristiques non pertinentes des composants, par exemple leur taille, plutôt que pertinentes, par exemple l'estimation de la quantité de modifications à apporter. Il semble donc nécessaire d'avoir des méthodes et une formation à la réutilisation en plus d'outil d'assistance à cette activité.

5 Conclusion

Toutes ses recherches sur l'activité de conception et sur la réutilisation dans la conception ont de nombreuses implications pour la conception des environnements de programmation, outils d'aide à la conception chez l'expert, les aides à la réutilisation... En conclusion, nous souhaiterions étendre notre propos à la conception, non plus comme une activité individuelle, mais comme une activité collective.

D'une part, nous pensons que l'étude de la conception individuelle est une étape nécessaire et indispensable au développement des connaissances/modèles sur la conception.

D'autre part, il semble que l'activité de développement de logiciel soit souvent une activité collective. Bien qu'on puisse alors la considérer comme un ensèmble d'activité individuelles, notamment avec la spécialisation des tâches d'analyse, de conception, et de codage, elle est alors bien plus que cela si l'on considère tous les aspects d'interaction entre les différents partenaires d'un projet logiciel. Cela amène à introduire d'autres thèmes de recherche, propres aux activités collectives, qui sont les thèmes de communication, coordination, et les aspects organisationnels (voir par exemple : Herbsleb, Klein, Olson, Brunner, Olson & Harding, 1995 ; Krasner, Curtis & Iscoe, 1987 ; Visser, 1993).

Remerciements

Nous remercions Jean-Marie Burkhardt et Willemien Visser pour leur lecture critique d'une version intiale de ce papier.

Références

Adelson, B. (1981) Problem solving and the development of abstract categories in programming languages. Memory and cognition, 9 (4), 422-433.

Adelson, B. (1984) When novices surpass experts: The difficulty of a task may increase with expertise. Journal of Experimental Psychology: Learning, Memory and Cognition, 10 (3), 483-495.

Adelson, B. & Soloway, E. (1984) A model of Software Design. Rapport de Recherche 342, Yale University, New Haven.

Ball, L. J., & Ormerod, T. C. (1995) Structured and opportunistic processing in design: a critical discussion. International Journal of Human-Computer Studies, 43, 131-151.

Bisseret, A., Burkhardt, J-M., Deleuze-Dordron, C., Détienne, F., Rouet, J-F. (1995) The role of software structuration and documentation formats in software reuse: result of advanced studies. SCALE European project. Deliverable 2.3.2-3.

Black, J. B., Kay, D. S. & Soloway, E. (1986) Goal and Plan Knowledge Representations: From Stories to Text Editors and Programs. In J.M. Carroll (Ed): Interfacing Thought: Cognitive Aspects of Human-Computer Interaction. Cambridge, Mass: MIT Press.

Brooks, R. (1977) Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 9, 737-751.

Brooks, R. (1983) Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, 18, 543-554.

Burkhardt, J-M. & Détienne, F. (1994) La réutilisation en Génie Logiciel : une définition d'un cadre de recherche en Ergonomie Cognitive, ERGO-IA 94, Biarritz, 26-28 Octobre.

Burkhardt, J-M. & Détienne, F. (1995a) La réutilisation de solutions en conception de programmes informatiques. Psychologie Française, numéro spécial sur l'Ergonomie Cognitive, 40-1, 85-98.

Burkhardt, J-M. & Détienne, F. (1995b) An empirical study of software reuse by experts in object-oriented design. In K. Nordby, P. H. Helmersen, D. J. Gilmore & S. A. Arnesen (Eds): Proceedings of INTERACT'95. Chapman & Hall. 133-138.

Chatel, S. & Détienne. F. (1996) Strategies in object-oriented design. Acta Psychologica, special issue on Cognitive Ergonomics, 1996.

Chatel, S., Détienne, F. & Borne, I (1992) Transfer among Programming Languages: An Assessment of Various Indicators. Proceedings of the Fifth Workshop of the Psychology of Programming Interest Group. Paris, December 9-12.

Curtis, B. (1986), "By the Way, Did Anyone Study Any Real Programmers?" In Emprirical Studies of Programmers, First workshop, E Soloway & S Iyengar [Eds], Ablex, pp.256-262.

Davies, S. P. (1991) The role of notation and knowledge reprresentation in the determination of programming strategy: a framework for integrating models of programming behavior. cognitive Science, 15, p 547-572.

Davies, S. P. (1993) Models and theories of programming strategy. International Journal of Man-Machine Studies, 39, p 237-267.

Davies, S. P. (1994) Knowledge restructuring and the acquisition of programming expertise. International Journal of Human-Computer Studies, 40, p 703-726.

Davies, S. P. (1996) Display-based problem solving strategies in computer programming. In W. D. Gray, & D. A. Boehm-Davis (Eds): Empirical Studies of Programmers, Sixth Workshop. Ablex Publishing Corporation. Norwood: NJ.

Davies, S. P., & Castell, A. M. (1994) From individual to groups through artifacts: the changing semantics of design in software development. In D. Gilmore, R. Winder et F. Détienne (Eds): User Centred Requirements for Software Engineering Environments. Springer Verlag, NATO ASI Series. p 11-23.

Détienne, F. (1984) Analyse exploratoire de l'activité de compréhension de programmes informatiques. Actes du séminaire AFCET "Approches quantitatives en génie logiciel". Sophia-Antipolis, 7-8 Juin.

Détienne, F. (1988) Une Application de la Théorie des Schémas à la Compréhension de Programmes, Le Travail Humain, Numéro Spécial "Psychologie Ergonomique de la Programmation". 51, 4, 335-350.

Détienne, F. (1990a) Program Understanding and Knowledge Organization: the Influence of Acquired Schemas. In P. Falzon (Ed): Cognitive Ergonomics: understanding, learning and designing Human-Computer Interaction, Academic Press, London. 245-256.

Détienne, F. (1990b) Difficulties in Designing with an Object-Oriented Language: An Empirical Study. In D. Diaper, D. Gilmore, G. Cockton & B. Shackel (Eds): Human Computer Interaction, proceedings of INTERACT'90, Cambridge, UK, August 27-31, p 971-976, North Holland.

Détienne, F. (1990c) Expert Programming Knowledge: A Schema-Based Approach. In J-M. Hoc, T.R.G. Green, R. Samurçay, D. Gilmore (Eds): Psychology of programming, Academic Press, People and Computer Series, 205-222.

Détienne, F. (1991) Reasoning from a schema and from an analog in software code reuse. In J. Koenemann-Belliveau, T.G. Moher and S.P. Robertson (Eds): Empirical studies of programmers, Fourth Workshop. Ablex, Norwood, NJ. 5-22.

Détienne, F. (1994a) Constraints on design: language, environment, code representation. In D. Gilmore, R. Winder and F. Détienne (Eds): User centred requirements for Software Engineering environments. Springer Verlag, NATO ASI Series. p 69-80.

Détienne, F. (1994b) Design Activities and Representations for Design: An Introduction. In D. Gilmore, R. Winder et F. Détienne (Eds): User Centred Requirements for Software Engineering Environments. Springer Verlag, NATO ASI Series. p 7-9.

Détienne, F. (1995) Design Strategies and Knowledge in Object-oriented design: Effects of experience. HCI Journal, vol 10 (2 & 3), 129-170.

Détienne, F. & Soloway, E. (1990) An Empirically-Derived Control Structure for the Process of Program Understanding. In R. Brooks (Ed): special issue "what programmers know", International Journal of Man-Machine Studies, 33 (3), p 323-342.

De Vries, E. (1993). The role of case-based reasoning in architectural design : Stretching the design problem space. In W. Visser (Ed.), proceedings of the Workshop of the Thirteenth International Joint Conference on Artificial Intelligence "Reuse of designs : an interdisciplinary cognitive approach", (pp. B1-B13). Chambery August 29, 1993: INRIA Rocquencourt.

Galambos, J. A., Abelson, R. P., & Black, J. B. (1986) Knowledge Structures. Laurence Erlbaum Associates, Hillsdale: NJ.

Gilmore, D. J. (1990) Expert programming knowledge: a strategic approach. In J-M. Hoc, T.R.G. Green, R. Samurçay, D. Gilmore (Eds): Psychology of programming, Academic Press, People and Computer Series, 223-234.

Gilmore, D. (1991) Models of debugging. Acta Psychologica, 78, 151-172.

Green, T. R. G. (1990) Programming languages as information structures. Dans J-M Hoc, T.R.G. Green, R. Samurçay & D. Gilmore: Psychology of Programming. Série "Cognitive Ergonomics and Cognitive engineering", Wiley. p 117-138.

Green, T. R. G., Bellamy, R. & Parker, M. (1987) Parsing and Gnisrap: A Model of Device Use. Dans G. M. Olson, S. Sheppard & E. Soloway: Empirical Studies of programmers: second workshop, Ablex

Guindon, R. (1990) Designing the design process: exploiting opportunistic thoughts. Human-Computer Interaction, 5, 305-344.

Guindon, R. (1992) Requirements and design of Design Vision, an object-oriented graphical interface to an intelligent software design assistant. In Proceedings of CHI'92, ACM Press, p 499-506.

Guindon, R. Krasner, H. & Curtis, B. (1987) Breakdowns and Processes during the Early Activities of Software Design by Professionals. Dans G. M. Olson, S. Sheppard & E. Soloway: Empirical Studies of programmers: second workshop, Ablex.

Herbsleb, J. D., Klein, H., Olson, G. M., Brunner, H., Olson, J. S. & Harding, J. (1995) Object-oriented analysis and design in software project teams. Human-Computer Interaction, 10 (2&3), 249-292.

Hoc, J-M. (1981) Planning and Direction of Problem-Solving in Structured Programming: an Empirical Comparaison between two methods. International Journal of Man-Machine Studies, 15(4), 363-383.

Hoc, J-M. (1983) Une méthode de classification préalable des problèmes d'un domaine pour l'analyse des stratégies de résolution: la programmation informatique chez des professionnels. Le Travail Humain, 46 (3), 205-217.

Hoc, J-M. (1987) L'apprentissage de l'utilisation des dispositifs informatiques par analogie à des situations familières. Psychologie Française, Numéro spécial "Les langages informatiques dans l'enseignement. 32, 4, 217-226.

Jeffries, R., Turner, A. A., Polson, P., & Atwood, M. E. (1981) The Processes Involved in Designing Software. Dans J.R. Anderson (Ed): Cognitive Skills and their Acquisition. Hilldale, NJ: Erlbaum.

Krasner, H., Curtis, B., & Iscoe, N. (1987) Communication breakdowns and boundary spanning activities on large programming projects. In G. M. Olson, S. Sheppard & E. Soloway: Empirical Studies of programmers: second workshop, Ablex. p 47-64.

Lange, B.M. and Moher T.G. (1989) Some strategies of reuse in an object-oriented programming environment. In K. Bice and C. Lewis (Eds.), Proceedings of CHI'89 Conference on Human Factors in Computing Systems. ACM Press, 69-73.

Littman, D. C., Pinto, J., Letovsky, S. & Soloway, E. (1986) Mental Models and Software Maintenance. In E. Soloway & S. Iyengar (Eds.): Empirical Studies of Programmers. Proceedings of the first workshop on Empirical Studies of Programmers. Norwood, N.J.: Ablex Publishing Corporation.

Maiden, N. (1991). Analogy as a paradigm for specification reuse. Software Engineering Journal, 3-15.

Minsky, M. (1975) A framework for representing knowledge. In P. Winston (Ed.): The psychology of computer vision. New York: MacGraw-Hill. 211-277.

Pennington, N. (1987) Stimulus structures and mental representation in expert comprehension of computer programs. Cognitive psychology, 295-341.

Pennington, N., & Grabovski, B. (1990) The tasks of programming. Dans J-M Hoc, T.R.G. Green, R. Samurçay & D. Gilmore: Psychology of Programming. Série "Cognitive Ergonomics and Cognitive engineering", Wiley. p 45-62.

Petre, M. (1995) Why looking isn't always seeing: readership skills and graphical programming. Communication of the ACM, June 1995, 33-44

Rist, R. S. (1986) Plans in Programming: definition, demonstration and development. Dans E. Soloway & S. Iyengar (Eds): Empirical Studies of Programmers, Norwood, NJ: Ablex.

Rist, R. S. (1989) Schema creation in programming. Cognitive Science, 13, 389-414.

Rist, R. S. (1991) Knowledge creation and retrieval in program design: a comparison of novice and experienced programmers. Human-Computer Interaction, 6, 1-46.

Rist, R. S. (1996) System structure and design. In W. D. Gray & D. A. Boehm-Davis (Eds): Empirical Studies of Programmers, Sixth Workshop. Ablex Publishing Corporation, Norwood: NJ. 163-194.

Rist, R. S. (à paraître) Program structure and design. Cognitive Science.

Robertson, S. P. (1990) Knowledge Representations used by Computer Programmers. Journal of the Washington Academy of Sciences, 80 (3), p 116-137.

Robertson, S.R. & Yu, C-C (1990) Common cognitive representations of program code across tasks and languages. International Journal of Man-Machine Studies, 33, p 343-360.

Rosson, M.B. & Alpert, S.R. (1990) The cognitive consequences of object-oriented design. Human-Computer Interaction, 5, 345, 379.

Rosson, M.B. and Carroll, J.M. (1993) Active programming strategies in reuse. Proceedings of ECOOP'93, Object-Oriented Programming. Berlin: Springer-Verlag. 4-18.

Rouet, J-F., Deleuze-Dordron, C., & Bisseret, A. (1995) Documentation as part of design: exploratory field studies. In K. Nordby, P. H. Helmersen, D. J. Gilmore & S. A. Arnesen (Eds): Proceedings of INTERACT'95. Chapman & Hall. 213-216.

Shneiderman, B. (1980) Software Psychology, Human Factors in Computer and Information Systems. Cambridge, Mass: Winthrop Publishers.

Schank, R. & Abelson, R. (1977) Scripts-Plans-Goals and Understanding. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Scholtz, J. & Wiedenbeck, S. (1990) Learning second and subsequent programming languages: a problem of transfer. International Journal of Human-Computer Interaction, 2(1), 51-57.

Soloway, E. (1986), "What to Do Next: Meeting the Challenge of Programming-in-the-Large" In Emprirical Studies of Programmers, First workshop, E Soloway & S Iyengar [Eds], Ablex, pp.263-268.

Soloway, E., Ehrlich, K. & Bonar, J. (1982) Tapping into Tacit Programming Knowledge. Human Factors in Computer System, 15-17, 1982, 52-57.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. (1982) What do novices know about programming? In A. Badre & B. Shneiderman (Eds.): Directions in human computer interaction. Norwood, N.J.: Ablex Publishing corporation.

Soloway, E. & Ehrlich, K. (1984) Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering, SE-10 (5), 1984, 595-609.

Visser, W. (1987) Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer. In G. M. Olson, S. Sheppard & E. Soloway: Empirical Studies of programmers: second workshop, Ablex. p 217-230.

Visser, W. (1993) Collective design: a cognitive analysis of cooperation in practice. In N. F. M. Roozenburg (Ed.): Proceedings of ICED 93, 9th International Conference on Engineering Design (Volume 1), Zürich: HEURISTA.

Visser, W. (1994a) Planning and organization in expert design activities. In D. Gilmore, R. Winder et F. Détienne (Eds): User Centred Requirements for Software Engineering Environments. Springer Verlag, NATO ASI Series. p 25-40.

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

Visser, W. & Hoc, J-M. (1990) Expert Software Design Strategies. Dans J-M Hoc, T.R.G. Green, R. Samurçay & D. Gilmore: Psychology of Programming. Série "Cognitive Ergonomics and Cognitive engineering", Wiley.

Wiedenbeck, S. (1986) Beacons in computer program comprehension. International Journal of Man-Machine Studies, 25, 697-709.

Wiedenbeck, S., & Scholtz, J. (1989) Beacons: a knowledge structure in program comprehension. In G. Salvendy & M.J. Smith (Eds): Designing and using human-computer interfaces and knowledge-based systems. Amsterdam: Elsevier.

Woodfield, S. N., Embley, D. W., & Scott, D. T. (1987) Can programmers reuse Software? IEEE Software, July 1987, 52-59.

Table des matières

1 INTRODUCTION 6

2 HISTORIQUE 6

3 L'ACTIVITÉ DE CONCEPTION DE LOGICIELS 7

3.1 Caractéristiques des problèmes de conception 7
3.2 Approches centrées sur les connaissances 8
3.3 Approches centrées sur les stratégies 10
3.3.1 Dimensions caractérisant les stratégies 10
3.3.2 Conditions de déclenchement des stratégies 11
3.4 Organisation de l'activité : hiérarchique versus opportuniste 12

4 LA RÉUTILISATION DANS LA CONCEPTION 14

4.1 Caractéristiques générales de la réutilisation dans la conception 14
4.2 Classification des situations de réutilisation 15
4.3 Enrichissement de la représentation versus abaissement du niveau de contrôle de l'activité 16
4.4 Expertise en réutilisation 18

5 CONCLUSION 19

REMERCIEMENTS 19

RÉFÉRENCES 19

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine - Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes - IRISA, Campus universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 Grenoble Cedex 1 (France)
Unité de recherche INRIA Sophia Antipolis - 2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

ISSN 0249 - 6399

RR_ 2902