People-oriented Software Reuse: the Very Thought

N.A.M. Maiden & A.G. Sutcliffe

Department of Business Computing, City University
London, UK

Abstract

Most software reuse research has ignored the role of the software engineer. However, software engineers tend to be better reasoners and have more experiences to recall than tool-based reuse mechanisms. This paper argues for integrating software engineers into existing reuse paradigms and providing tool support to assist problem description and component understanding, selection and adaptation. However, empirical studies indicate that these reuse tasks are difficult, even for experienced software engineers. Therefore, guidelines and a high-level architecture for design of tool support are based on reports of behaviour and problems arising during reuse.

1: Introduction

Many technical solutions to software reuse problems have failed to result in widespread reuse. One reason may be that these solutions ignore human involvement. This paper investigates specific problems which arise from excluding software developers from the reuse process and proposes solutions based on human involvement for facilitating reuse.

Human issues in software reuse are poorly understood. Early approaches to reuse exploited low-level code modules during system construction. Modules were treated as black boxes from which systems were composed using parametrisation and modular interconnection languages [25], and the programmer was not required to inspect or modify these modules. Indeed, module adaptation was deemed to lessen the inbuilt quality of the reusable product. However, CASE (Computer-Aided Software Engineering) technology supporting the analysis and design phases of software development necessitates reuse of larger components which require good understanding for effective application and adaptation. Such reuse is a knowledge-intensive task necessitating the involvement of the software engineer, so the processes of problem description and component comprehension, selection and customisation were investigated to inform design of support tools. This investigation is restricted to component rather than transformational reuse (e.g. [5]), although issues will be relevant to other reuse paradigms.

2: Human issues in software reuse

Human involvement during software reuse is needed during five activities shown in Figure 1:

- describing new problems using terms which permit retrieval of reusable components;
- understanding retrieved reusable components;
- selecting the best component from several candidates;
- adapting the selected component to fit the new problem;
- adding extra functionality to the component while ensuring the validity of existing component features.

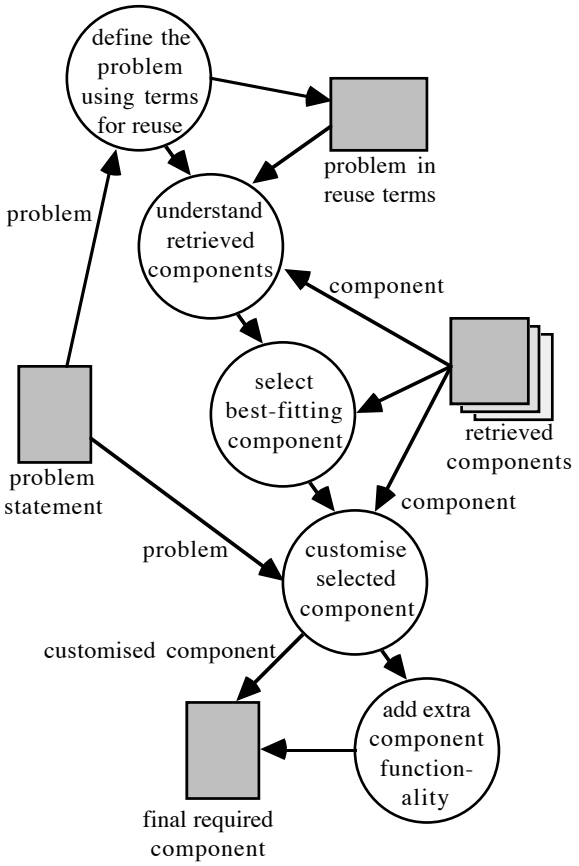

Figure 1 - overview of human tasks in software reuse

In section 2 evidence of problems which occur during these reuse activities are reviewed then functional requirements for support tools are discussed in light of these problems in section 3.

needed for effective description, the effort needed to define and maintain such terms and how well they can used by software engineers to describe a problem. This last problem is demonstrated by the following example.

Prieto-Diaz & Freeman's [36] classification scheme uses

Function	*Objects*	*Medium*	*System type*	*Functional Area*	*Setting*
add	arguments	array	assembler	accounts payable	advertising
append	arrays	buffer	code generation	accounts receivable	appliance repair
close	backspaces	cards	code optimisation	analysis structural	appliance store
compare	blanks	disk	compiler	auditing	association
complement	buffers	file	DB management	batch job control	auto repair
compress	characters	keyboard	expression evaluator	billing	barbershop
create	descriptors	line	file handler	bookkeeping	broadcast station
decode	digits	list	hierarchical DB	budgeting	cable station
delete	directories	mouse	hybrid DB	capacity planning	car dealer
divide	expressions	printer	interpreter	CAD	catalog sales
evaluate	files	screen	lexical analyser	cost accounting	cemetery
exchange	functions	sensor	line editor	cost control	circulation
expand	instructions	stack	network DB	customer information	classified ads
format	integers	table	pattern matcher	DB analysis	cleaning
input	lines	tape	predictive parsing	DB design	clothing store
insert	lists	tree	relation DB	DB management	composition
join	macros	.	retriever	.	computer store
measure	pages	.	scheduler	.	.
modify	.	.	.	.	.
move	.	.	.	.	.

Table 1 - Partial listing of faceted classification scheme (from Prieto-Diaz & Freeman 1987, p10)

2.1: Describing new problems

Most software reuse paradigms require new systems to be defined using the same terms as reusable components. They assume that software engineers can define system needs using restricted terms, so little support has been provided for this task. However, defining new problems in this way may be problematic. Furnas et al.'s [18] study of vocabulary-driven interaction found large word choices for objects, suggesting inherent problems with lexically-based, keyword retrieval mechanisms. Single access terms provided very poor access for verbs applied by typists to text editing operations and commands entered into a message decoder program. On the other hand, effective retrieval requires many aliases or facets [56]. Furnas claims that an "unlimited aliases" system produced 50-100% hit rates in its first three guesses to untutored queries (p970). No superiority of natural language semantics (e.g. [77]) over formal languages faceted schemes was demonstrated in query correctness or task solution performance during empirical studies of data retrieval [34]. Its only advantage was the conciseness of queries and less time needed to formulate them. Therefore, effective problem description necessitates a restricted set of terms which index reusable components directly. However, a tradeoff exists between the number of terms

6 facets to describe program function, objects, medium, system type, functional area and setting, although later work [54] extends this scheme to define more complex applications. For each of these facets there can be many instances, especially for functional area and setting. Table 1 reveals how understanding the precise meaning of unfamiliar facets and descriptors is difficult since they can be understood by different people in different ways. Terms must be differentiated to be used effectively, especially when similar terms describe the same facet (e.g. append, add and insert for function). One solution is to explain facets, although this increases the amount of data to be assimilated by the software engineer. For instance, inexperienced software engineers were not always able to use abstract terms to describe a domain despite their explanation [40]. Another solution is to train software engineers to use restricted terms, however software engineers are more likely to construct new systems than reuse old ones if they find these terms unnatural. Therefore it may be better to provide a very simple and restricted set of terms for component description. Indeed, Guindon [79] showed that users requested help from a system using a restricted language that is characteristic of language generated under real-time production constraints. These solutions are examined later in this paper.

To sum, describing new problems using restricted terms may be more problematic than assumed in the software reuse literature. Potential solutions in the shape of powerful, generic and understandable problem specification languages are not available. However, human problems also arise during comprehension and customisation of reusable components. A recent IBM study on software maintenance suggested that 50% of all maintenance time was spent understanding code modules to be changed [80]. The following sections investigate problems associated with understanding, selecting and adapting reusable components.

2.2: Component understanding

One reason for the failure of software reuse is that component comprehension is more difficult than assumed. These difficulties were investigated in studies of software reuse, program understanding and program debugging.

Empirical evidence of software reuse behaviour: there have been few empirical studies of software reuse. Lange & Moher [37] studied a single software developer who exhibited a pervasive software reuse strategy while working in an object-oriented programming environment. Of the 99 software components created during the study, 85 were built from preexisting components while only 6 were created from scratch. Prominent strategies for reuse included manually composing the method in a step-by-step transformation of a template, electronically copying the component from its template and editing it, and literal copying employed without modification. Similar copying strategies during design-level reuse by inexperienced software engineers were reported in [69] while expert software engineers have failed to understand reusable components despite their motivation to avoid this copying [42]. This supports Lange & Moher's conclusion that understanding unfamiliar components is difficult and that copying may be employed as one mechanism to compensate for incomplete mental representations.

Other evidence of software reuse behaviour remains anecdotal, for instance Visser & Hoc [73] report the importance of examples and past design reuse exhibited by programmers. However, difficulties in understanding programs were also observed during program debugging. The consensus view is that program understanding is achieved by developing a mental representation which assigns more meaning to code than is present in the program text. Unfortunately, Holt et al. [32] reported that inexperienced programmers were unable to identify meaningful program units from its structure and contents while experts' program understanding was affected by the difficulty of the modification task. Therefore, studies of program comprehension and debugging were examined more closely to identify problems during component reuse.

What people understand about programs: empirical evidence suggests that understanding unfamiliar software is complex and error-prone [70]. The general expectation is that the increased knowledge and qualitatively better mental representations possessed by experts will facilitate the understanding and programming processes. Experts possess deeper, more principled mental representations which permit them to perceive meaningful patterns in domains while novices represent problems superficially. These mental representations, or schemata, define key software engineering abstractions such as stereotypic action sequences in programs (e.g. [58] [65]) consisting of compiled causal knowledge about relationships between parts of programs. They are recalled during program comprehension, composition [16] and reuse of instances of the same schema [15]. Their importance was suggested in studies [30] which identified that a lack of specialised design schemata adversely affected design performance. On the other hand, meaning is assigned to code by mentally reusing abstract plan structures (e.g. [1], [67], [12]). Contradictory evidence about the semantic content of these abstractions has been reported [76] [53] [12]. Inexperienced programmers clustered lines of code on the basis of syntactic similarities between program statements while experts clustered lines by functional units (e.g. [1], [2], [53], [12]) which mapped to their abstractions. Indeed, this mental reuse is the hallmark of professionalism and experience, and effective component reuse depends on the possession of relevant mental abstractions.

Studies revealed that less experienced software engineers must learn key mental abstractions for reuse, however this learning may be difficult. Analogical problem solving in other domains [22] [8] indicates that schema induction only occurs from many problem solving instances. Furthermore, schema induction during complex problem solving may fail due to cognitive overload (e.g. [81], [68]). Therefore, deliberate learning of key mental abstractions in current software engineering environments dominated by project deadlines and shortages in experienced staff is problematic.

Differences in program understanding: variations between programmers' mental abstractions complicate our view of program understanding. For instance, Bhuiyan et al. [3] investigated inexperienced programmers' mental models of recursion. Four different mental models of recursion identified during a single exercise were loop, stack, template and problem reduction. A subject's initial thinking reflected use of a template model, however this model was not used when writing the code. A loop model was abandoned before adopting the template model again throughout the remainder of the exercise. Strong individual differences in programmers' mental abstractions were also identified in [14] [32]. Wu & Anderson [78] speculate that programmers hold adequate mental schemata of programming constructs for different types of

problems. Thus, flexible tool support is needed to overcome inter-individual differences in program understanding, even between programmers with similar experience. These differences imply that effective support tools must be responsive to users' expertise and knowledge [52]. They may require both versatile output to provide adaptive instruction [51] and dynamic representation of the programmers' understanding [52].

Other issues such as language, training and individual experience further complicate the issue of program understanding:

- Sinha & Vessey [64] and Detienne [14] [15] among others claim that mental abstractions are dependent upon language structure and training. Singley & Anderson [63] write that knowledge acquired in the practice of one skill will not transfer to practice of related skills including program comprehension. Therefore component understanding may be specific to training and experience in programming languages and design notations;
- Green & Borning [26] claim that mental abstractions are easier to perceive in 'role-expressive' notations such as Pascal. Therefore, component understanding may be easier with certain programming languages and design notations;
- Gellenbeck & Cook [19] report the importance of key program features such as data structures or operations on program understanding. These 'beacons' were recognised by experienced programmers who used them to confirm hypotheses about unfamiliar programs and suggest new hypotheses.

Understanding design and specification components: similar cognitive processes occur during program, design and specification reuse. Evidence suggests that experienced software engineers possess memory schema representing key domain and design abstractions which they recall during software specification and design [29] [41] [42]. However, some differences are worth noting. Designs and specifications are often represented graphically using notations such as data flow diagrams (e.g. [13]), so they may be more easily understood than program code or formal notations. Indeed, data flow diagrams were developed with ease of comprehension in mind. On the other hand, domain and design abstractions are more complex and may require greater effort and exposure to learn. For instance, program abstractions reported in [16] tend to be simpler than the design abstraction shown in Figure 2. Therefore, design and specification understanding may be more difficult.

Further problems For component comprehension: reported studies indicate that key program and design abstractions must be learned before effective reuse can occur. However, industrial scale reuse is likely to be achieved from larger components, for instance Kruzela [80] identified that NTT reused components with upward of 600 lines of code. Reuse of large components magnifies problems identified in this paper and introduces new problems. For instance, many computer screens do not support multi-windowing facilities which make component browsing easier, so it is difficult to obtain a comprehensive overview of codes' functionality and structure. Solutions to these problems are examined at the end of the paper.

2.3: Component selection

Component selection requires the software engineer to reason about key similarities and differences between components. However, studies of analogical problem solving suggest that similarity-based reasoning is difficult [22]. Recognising analogies often needs syntactic similarities between problems [62] while inducing mental schemata during analogical matching has proven difficult even for expert software engineers [42]. Analogical problem solving may inform the design of selection tools in two directions. First, they reveal difficulties when reasoning analogically between a component and the original problem, with implications for component understanding and adaptation. Second, they indicate the need to reason about similarities between components to select the best fit. However, few reported studies have investigated analog selection, so more empirical investigation is needed.

In reuse, few studies have been undertaken of problems during component selection. One exception is the Cognitive Browser project which aims to assist object browsing and comprehension in a Smalltalk-80 environment [28]. This is achieved by capturing a programmer's knowledge about required systems as a basis for assisting the browsing and selection of relevant reusable objects. However, the general conclusion is that similarity-based reasoning is difficult, so tool support will be needed for component selection. More studies of software selection problems are needed to inform effective tool design.

2.4 Component adaptation

Finally components must be customised to fit the new problem. Empirical studies [9] indicate that component adaptation may be as problematic as understanding and selection. Lange & Moher's [37] study reported that comprehension avoidance is a key reuse strategy which is employed more as a mechanism to compensate for an incomplete mental model than as a shortcut method when the mental model was mature. This tendency to copy rather than understand was found in other domains including physics [9] and mathematical problem solving [50]. Therefore, copying during component adaptation is a key problem to be avoided during software reuse, otherwise cognitive short cuts are likely to result in inferior or incorrect software designs. Studies have also

revealed other difficulties during adaptation of analogical solutions. Novick & Holyoak [49] investigated mathematical problem solving by analogy and observed clear distinctions between understanding and adapting old solutions. Surprisingly, good analogical understanding did not lead to successful transfer. Novick & Holyoak also reported induction of key mental abstractions during analogical transfer. Therefore, people may understand components while adapting them. This blurs the distinction between comprehension and transfer, with implications for tool support. These issues during component adaptation were examined during two empirical studies of design and specification reuse.

Adapting a retrieved software design: the first study investigated software reusers' adaptation and comprehension strategies during design-level reuse [69]. Thirty inexperienced software developers (MSc students in Business Systems Analysis) were asked to develop a scheduling function in which video tapes were allocated to hotels if they met specified requirements. Two reusable designs were developed. A production planning system allocated manufacturing machines to production jobs while a generic scheduling function allocated resources to tasks which had to be fulfilled. The main concept with the concrete and generic designs was the functional requirement to allocate resources within constraints. One of the reusable designs is shown in Figure 2.

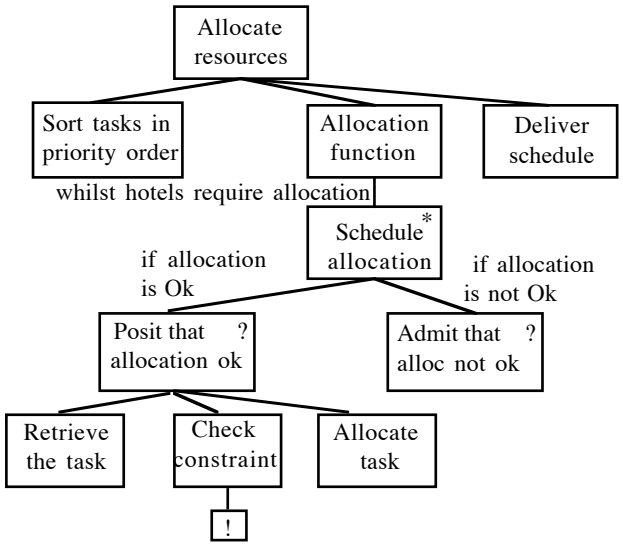

Figure 2 - reusable designs for the video hiring problem

Three groups of ten software engineers reused: (i) the concrete design; (ii) the generic design and; (iii) designed the video hiring function without any help. Results revealed that reuse significantly improved design completeness but not validity. Reusers were able to recognise and exploit the reusable designs despite evidence of poor understanding. Copying occurred in both reuse groups, for instance nine software engineers reused the JSD backtracking concept correctly although only three understood its meaning. Furthermore, errors in the designs of 11/13 successful reusers were consistent with poor comprehension of the reusable designs. Software engineers exploited salient similarities between components in the target and reusable domains as a basis for reuse. Therefore, this study revealed evidence of mental laziness and copying during software reuse, suggesting that it did indeed provide a shortcut for software engineers [37].

Adapting a retrieved software specification: requirements specification reuse was examined in two studies, the first of reuse by five inexperienced software engineers [41] and the second by ten expert software engineers [42]. Software engineers were required to specify an air traffic control system by reusing an analogical flexible manufacturing system specification, part of which is shown in Figure 3. Protocol analysis (i.e. speaking out loud while thinking) elicited software engineers' reasoning while video cameras captured other behaviour.

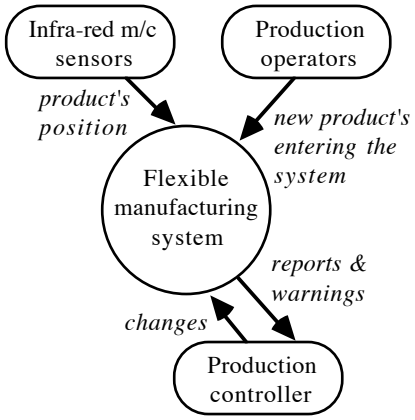

Figure 3 - reusable context DFD of flexible manufacturing system

Results indicated that inexperienced software engineers copied the reusable specification. They only understood objects or concepts which were similar or prominent in the reusable specification. On the other hand, expert software engineers avoided copying and exhibited strategies which maximised transfer of the specification and exploited all reusable components. They tended to understand the specification while adapting it [49]. However, understanding the reusable specification was difficult despite the best efforts of the experts. They also only understood components which were similar and prominent in the reusable specification.

Study conclusions: these two studies revealed that complex component understanding and adaptation is difficult. The emphasis on analogy in the second study

demonstrates additional complexity for reuse which does not arise during design, debugging or maintenance tasks. The observed behaviour concurs with observations in [9], [37] and [49]. Solutions to these problems are examined in the remainder of this paper.

2.5: A summary of human problems during software reuse

This review can be interpreted either as a pessimistic conclusion for software reuse or as a challenge for tools supporting reuse. They identify key problems to be supported by tools. Major functional requirements of these tools are:

- to support system definition using terminology which retrieves reusable components;
- to assist assimilation and understanding of information from many software components;
- to assist understanding of the functionality, structure and boundaries of each component;
- to assist identification and reasoning about key differences between retrieved components to select the best fit;
- to assist identification and reasoning about key differences between the selected component and original problem;
- to discourage mental laziness manifest as copying during component adaptation.

Some of these requirements have been identified in the software reuse literature, others have been ignored or down-played. They are elaborated in the remainder of this paper.

3: Cooperative support for software reuse

Before considering the implications from the reported work, existing cooperative support tools for reuse are reviewed.

Biggerstaff [4] proposed hypermedia tools to aid retrieval and understanding of software components. Such techniques are necessary, Biggerstaff argues, if large scale software reuse is to be achieved. Searching for components becomes less important than understanding the design of large scale components. He demonstrates the role of hypermedia in two systems called PlaneText and PlaneFig. These systems support component browsing, retrieval and presentation.

Gerhard Fischer developed tools called CODEFINDER and CODEEXPLAINER to support component location, comprehension and modification. Fischer et al. [17] reports that well-articulated queries for component retrieval are unlikely. Rather, queries must be constructed incrementally and support is needed for comprehending what is retrieved. Therefore, information access methods need support for query construction and relevance evaluation as an integral part of the location process. CODEFINDER uses an associative form of spreading activation based on a psychological model of human memory. Spreading activation represents items and keywords as network nodes linked by association which have activation values assigned to them during querying. This permits varying degrees of query matching. In studies, subjects found CODEFINDER's query construction methods to be quite natural and capable of retrieving solutions [17]. However, CODEFINDER's underlying model of human memory is unlikely to inform design of effective support tools due to the complexity and variations in component understanding, selection and adaptation reported in this paper.

AIR is a cooperative reuse toolset [43] to aid analogical specification reuse. Specifications are retrieved by matching a problem description to domain abstractions and specifications which instantiate these abstractions. Retrieval involves iterative problem description and analogical matching to refine the description and narrow the search space incrementally. Dialogue with AIR is guided to retrieve key domain facts. This paradigm was found to be effective in user studies with inexperienced requirements engineers [44]. AIR also assists understanding and customisation of retrieved specifications [69], [41], [42]. Its design is founded on empirical investigations reported in this paper.

Finally Linn & Clancey [39] reported an investigation of the effectiveness on program teaching of case studies which correspond to reusable components. Case studies provide an 'expert commentary' on the complex problem-solving skills used in programming worked out solutions to programming problems. They emphasise eight principles to teach templates and design skills used by experts, including:

- the recycling principle which encourages template reuse by illustrating how experts recall mental abstractions when solving similar programming problems. Templates are introduced as pseudocode so they can be readily recognised in new contexts;
- the multiple representation principle to encourage a robust understanding of programming templates by linking the verbal description, pseudocode, pictorial descriptions, dynamic illustrations and code examples for each template. Multiple representations aid template retrieval and understanding;
- the alternative paths principle which encourages generation and evaluation of alternative programming designs by describing processes for finding alternatives and criteria for selecting them;
- the divide and conquer principle, which encourages students to identify problem parts to be solved individually.

The effectiveness of these principles was shown during experimental studies undertaken in [39]. Although not related directly to software reuse, these studies have implications for reuse support tools.

3.1: Implications for design of support tools

An architecture of support tools which cooperate during reuse is shown in Figure 4. The functional requirements and strategies of support tools are discussed by component.

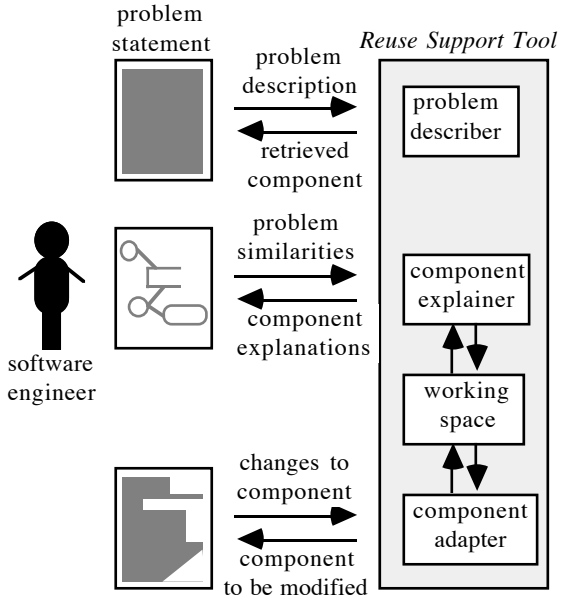

Figure 4 - a general architecture of a cooperative support tool for software reuse

The problem describer: problems must be described using terms which permit component retrieval. Two forms of support are envisaged:

- reuse within a single domain may be achieved by extensive classification of domain facets. Software engineers' facet choices may be large and variable, so extensive classification schemes permit effective problem description, assisted by simple facet explanations which can be browsed;
- classification schemes are ineffective for inter-domain reuse. One solution is to describe problems by example, for instance problem X is like problem Y except for difference Z. This is intuitively appealing since people understand in terms of known examples [59]. It avoids the need to understand complex abstract terminology, however examples may be less effective for defining problem details due to its low descriptive power. This suggests that combining example- and term-based problem description may be the most effective strategy.

The component explainer: the explainer assists understanding by teaching key mental abstractions for each component. Strategies which encourage induction of mental abstractions include presenting many concrete examples of these abstractions [8] [22], visualising the abstraction and its examples [21] and guided exposure to components to aid mental model formation [40]. However, more empirical research is needed for a complete and correct set of program, design and domain abstractions [33]. On the other hand, strategies for effective component understanding can inform how tools should promote component learning [14] [16] [38]:

- reestablishing the original author's intentions [35] was investigated empirically in [74] to reveal that expert programmers strove for the original author's intention prior to program debugging. This may be achieved by starting reading at the beginning of the program using a bottom-up strategy to identify the program goals [68]. Once goals were recognised programmers went into a predictive, top-down mode, dividing the problem into parts before solving them. This emulation of expert strategies can inform tool design. For instance, the tool can encourage software engineers to identify program goals bottom-up then assimilate program details top down;
- experts recall mental abstractions by attempting to understand entire programs while novices only comprehend isolated fragments (e.g. [36] [48] [60] [68]). Therefore tool support should also encourage understanding of entire components before presenting details;
- reestablishing rationale behind components (e.g. [46] [47]) has assisted program understanding [39]. Tool support should encourage generation and evaluation of alternative designs.

Graphic representations can also assist program understanding:

- fundamental aspects of computer programs were understood more quickly and accurately when represented graphically (e.g. [11]);
- indentation and colour coding can aid program comprehension and debugging [23]. Indeed, Van Laar [71] reported empirical studies which revealed that both colour and indentation improved program comprehension.

Therefore, visual programming languages and graphic representation of programs, designs and domains can aid component understanding, although [27] reports from studies of comprehension with visual programming languages that '*what a programmer sees is largely a matter of training*'. To overcome these problems, multiple representations of components, including text, graphics, dynamic illustration and examples may all be necessary. Highlighting key component features such as data structures and operations in programs to act as beacons for program understanding may also be needed [19].

Other empirically-based strategies which inform design of tool support are:

- program documentation improves code understanding. Gellenbeck & Cook [20] reported empirically founded guidelines to aid comprehension, including header

comments in source code which act like preview statements and advance organisers, mnemonic module names and and linking module names to header comments (p96). Studies of expert programmer documentation [57] also suggest that communicating semantic domain knowledge about the function and behaviour of programs aids comprehension;
- electronic work spaces [29], [31] can aid understanding and capture facts about components which permit more adaptive assistance through error diagnosis and topic focus during component explanation;
- responsive and flexible tools are needed to accommodate individual differences in component understanding and training. Adaptive instruction and dynamic representation of programmers' understanding are required [82] to inform design of adaptive tools.

The component adapter: the adapter achieves effective component adaptation through:
- controlled reuse to discourage copying and lead to effective comprehension before allowing component adaptation. Strategies for effective adaptation and understanding can be derived empirically by observing successful expert reusers, although they are likely to be language- and design-specific [43];
- information hiding can focus attention on key features of the component. Gradual exposure combined with explanation dialogue can assist understanding. Partial exposure to functionality has proven successful in other domains [7].

4: Conclusions

Human issues in software reuse have been ignored in favour of technical solutions and changes in management practice. However, this paper shows the need to involve software engineers in problem description and component understanding, selection and adaptation. Unfortunately software engineers are not perfect, so support tools must be designed to avoid likely errors which arise and overcome cognitive limitations in component understanding and transfer. This paper reviewed problems for reusers from empirical studies in other domains to focus attention on the functional requirements of support tools. Some of these requirements have been implemented in AIR [43].

Reported findings have several implications for future research. First, more empirical investigations of all reuse phases are needed to extend the few existing studies [15] [37] [42] [69]. Error models must inform design of diagnostic tool support while process, task and reasoning models can inform how to achieve effective reuse. Second, reuse of program, design and specification components must be investigated to promote reuse during the key early stages of software development. Theoretically, these stages provide the greatest payoff for reuse, however they have received least attention. Finally, this paper has demonstrated the importance of people during software reuse in all but the most well-defined domains. The next generation of support tools must recognise and accommodate people as critical for effective reuse, otherwise the acclaimed benefits in software productivity and quality may never be achieved.

References

[1] Adelson B., 1991, 'Problem Solving and the Development of abstract Categories in Programming Languages', *Journal of Experimental Psychology: Learning, Memory and Cognition* **9(4)** , 422-433.

[2] 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.

[3] Bhuiyan S.H., Greer J.E. & McCalla G.I., 1988, 'Mental Models of Recursion and Their Use in the SCENT Programming Advisor', *Proceedings of ITS-88*, June 1-3 1988, Montreal, Canada, 135-144.

[4] Biggerstaff T.J., 1987, 'Hypermedia as a Tool to Aid Large Scale Reuse', MCC Technical Report STP-202-87, Software Technology Program, MCC, Austin TX.

[5] Biggerstaff T.J. & Richter C., 1987, 'Reusability Framework, Assessment, and Directions', *IEEE Software*, **March 1987**, 41-49.

[6] Byrne E.J., 1991, 'Software Reverse Engineering: A Case Study', *Software - Practice and Experience* **21(12)**, 1349-1364.

[7] Carroll J.M., Smith-Kerker P.L., Ford J.L. & Mazur-Rimetz S.A., 1988, 'The Minimal Manual', *Human-Computer Interaction* **3**, 123-153.

[8] Cheng P.W. & Holyoak K.J., 1985, 'Pragmatic Reasoning Schemas', *Cognitive Psychology* **17**, 391-416.

[9] Chi M.T.H., Bassok M., Lewis M.W. et al., 1989, 'Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems', *Cognitive Science* **13**, 145-182.

[10] Chi M.T.H., Glaser R. & Rees E., 1982, 'Expertise in Problem Solving', *Advances in the Psychology of Human Intelligence*, ed. R. Sternberg, Lawrence Erlbaum Associates, 7-75.

[11] Cunniff N. & Taylor R.P., 1987, 'Graphical vs Textual Representation: An Empirical Study of Novices' Program Comprehension', in *Empirical Studies of Programmers, Second Workshop*, eds G. Olsen, S. Sheppard and E. Soloway, Ablex, Norwood NJ, 114-131.

[12] Davies S.P., 1989, 'Skill Levels and Strategic Differences in Plan Comprehension and Implementation in Programming', *Proceedings of HCI'89*, ed. A. Sutcliffe & L. Macaulay, Cambridge University Press, 487-502.

[13] De Marco T., 1978, '*Structured Systems Analysis and Specification*', Prentice-Hall International.

[14] Detienne F., 1992, 'Acquiring Experience in Object-Oriented Programming: Effect on Design Strategies', in *Cognitive Models and Intelligent Environments for Learning Programming*, ed. E. Lemut, B. du Boulay and G. Dettori, Springer-Verlag.

[15] Detienne F., 1991, 'Reasoning from a Schema and from an Analog in Software Code Reuse', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S.P. Robertson, Ablex, Norwood NJ, 5-22.

[16] Detienne F. & Soloway E., 1990, 'An Empirically-derived Control Structure for the Process of Program Understanding', *International Journal of Man-Machine Studies* **33**, 323-342.

[17] Fischer G., Henninger S. & Redmiles D., 1991, 'Cognitive Tools for Locating and Comprehending Software Objects for Reuse', Proceedings of 13th International Conference on Software Engineering at Austin Texas, May 1991.

[18] Furnas G.W., Landauer T.K., Gomez L.M. & Dumais S.T., 1987, 'The Vocabulary Problem in Human-System Communication', *Communications of the ACM* **30(11)**, 964-971.

[19] Gellenbeck E.M. & Cook C.R., 1991, 'An Investigation of Procedure and Variable Names as Beacons during Program Comprehension', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S.P. Robertson, Ablex, Norwood NJ, 65-81.

[20] Gellenbeck E.M. & Cook C.R., 1991, 'Does Signalling Help Professional Programmers Read and Understand Computer Programs', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S.P. Robertson, Ablex, Norwood NJ, 82-98.

[21] Gick M.L., 1989, 'Two Functions of Diagrams in Problem Solving by Analogy', in *Knowledge Acquisition from Text and Pictures*, eds H. Mandi and J.R. Levin, Elsevier Science Publishers B.V. (North-Holland), 215-231.

[22] Gick M.L. & Holyoak K.J., 1983, 'Schema Induction & Analogical Transfer', *Cognitive Psychology* **15**, 1-38.

[23] Gilmore D.J. & Green T.R.G., 1988, 'Programming Plans and Programming Experience', *Quarterly Journal of Experimental Psychology* **40A**, 423-442.

[24] Glaser R. & Chi M.T.H., 1988, Overview, *The Nature of Expertise*, ed. M.T.H. Chi, R. Glaser & M.J. Farr, LEA Hillsdale NJ.

[25] Goguen J.A., 1986, 'Reusing and Interconnecting Software Components', *IEEE Computer* **19**, 16-28.

[26] Green T.R.G. & Borning A., 1990, 'The Generalised Unification Parser: Modelling the Parsing of Notations', *Proceedings of INTERACT'90*, ed. D. Diaper, D. Gilmore, G. Cockton & B. Shackel, Elsevier Science Publishers B.V. (North-Holland), 951-957.

[27] Green T.R.G., Petre M. & Bellamy R.K.E., 1991, 'Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Mismatch' Conjecture', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S. Robertson, Ablex, Norwood NJ, 121-146.

[28] Green T.R.G., Gilmore D.J., Blumenthal B.B., Davies S. & Winder R., 1992, 'Towards a Cognitive Browser for OOPS', *International Journal on Human-Computer Interaction* **4(1)**, 1-34.

[29] Guindon R., 1990, 'Designing the Design Process: Exploiting Opportunistic Thoughts', *Human-Computer Interaction* **5**, 305-344.

[30] Guindon R., Krasner H. & Curtis B., 1987, 'Breakdowns and Processes During the Early Activities of Software Design by Professionals', in *Empirical Studies of Programmers, Second Workshop*, eds G. Olsen, S. Sheppard and E. Soloway, Ablex, Norwood NJ, 65-82.

[31] Haddley N. & Sommerville I., 1990, 'Integrated Support for Systems Design', *Software Engineering Journal* **5(6)**, 331-338.

[32] Holt R.W., Boehm-Davis D.A., Schultz A.C., 1987, 'Mental Representations of Programs for Student and Professional Programmers', in *Empirical Studies of Programmers, Second Workshop*, eds G. Olsen, S. Sheppard and E. Soloway, Ablex, Norwood NJ, 33-46.

[33] Jarke M., Rolland, Sutcliffe A.G. & Vassiliou Y., 1993, 'Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis', Proceedings of IEEE Symposium on RE, IEEE Computer Society Press.

[34] Jarke M., Turner J.A., Stohr E.A., Vassiliou Y., White N.H. & Michielsen K., 1985, 'A Field Evaluation of Natural Language for Data Retrieval', *IEEE Transactions on Software Engineering* **11(1)**, 97-114.

[35] Johnson W.L., 1990, 'Understanding and Debugging Novice Programs', *Artificial Intelligence* **42**, 51-97.

[36] Katz I.R. & Anderson J.R., 1988, 'Debugging: An Analysis of Bug-Location Strategies', *Human-Computer Interaction* **3**, 351-399.

[37] Langer B.M. & Moher, T.G., 1989, 'Some Strategies of Reuse in an Object-Oriented Programming Environment,' Proceedings of CHI'89, ed. by K. Bice & C. Lewis, ACM Press, 69-73.

[38] Letovsky S., 1987, 'Cognitive Processes in Program Comprehension', *Journal of Systems and Software* **7**, 325-339.

[39] Linn M.C. & Clancey M.J., 1992, 'Can Experts' Explanations Help Students Develop Program Design Skills', *International Journal of Man-Machine Studies* **36**, 511-551.

[40] Maiden N.A.M., 1992, 'Analogical Specification Reuse during Requirements Analysis', Ph.D. Thesis, Dept Business Computing, City University, London.

[41] Maiden N.A.M. & Sutcliffe A.G., 'The Abuse of Reuse: Why Software Reuse must be Taken into Care', in prep.

[42] Maiden, N.A.M. & Sutcliffe, A.G., 1992, 'Analogically-based Reusability', *Behavioural Information and Technology* **11(2)**, 79-98.

[43] Maiden N.A.M. & Sutcliffe A.G., 1992, 'Exploiting Reusable Specifications Through Analogy', *Communications of the ACM* **34(5)**, 55-64.

[44] Maiden N.A.M. & Sutcliffe A.G., 1993, 'Requirements Engineering by Example: an Empirical Study', Proceedings of IEEE Symposium on RE, IEEE Computer Society Press.

[45] Maiden N.A.M. & Sutcliffe A.G., 1991, 'Analogical Matching for Specification Reuse', *Proceedings of 6th Knowledge-Based Software Engineering Conference*, IEEE Computer Society Press, 101-112.

[46] McLean A., Young R.M., Bellotti V.M.E. & Moran T.P., 1991, *Human-Computer Interaction* **6(3&4)**.

[47] Mostow J., 1989, 'Design by Derivational Analogy: Issues in the Automated Replay of Design Plans', *Artificial Intelligence* **40**, 119-184.

[48] Nanja M. & Cook R.C., 1987, 'An Analysis of the Online Debugging Process', in *Empirical Studies of*

Programmers, Second Workshop, (eds G. Olsen, S. Sheppard & E. Soloway), Ablex, Norwood NJ, 172-184.

[49] Novick L.R. & Holyoak K.J., 1991, 'Mathematical Problem Solving by Analogy', *Journal of Experimental Psychology: Learning, Memory, and Cognition* **17(3)**, 398-415.

[50] Novick L.R., 1988, 'Analogical Transfer, Problem Similarity, and Expertise', *Journal of Experimental Psychology: Learning, Memory and Cognition* **14(3)**, 510-520.

[51] Ohlsson S., 1986, 'Some Principles of Intelligent Tutoring Systems', *Instructional Science* **14**, 293-326.

[52] Payne S., 1988, 'Methods and Mental Models in Theories of Cognitive Skill', *Artificial Intelligence and Human Learning (Intelligent Computer-Aided Instruction)*, ed. J.A. Self, Chapman & Hall, 69-87.

[53] Pennington N., 1987, 'Comprehension Strategies in Programming', in *Empirical Studies of Programmers, Second Workshop*, ed. G. Olson, S. Sheppard and E. Soloway, Ablex, 100 - 113.

[54] Prieto-Diaz R., 1991, 'Implementing Faceted Classification for Software Reuse', *Communications of the ACM* **34(5)**, 88-97.

[55] Prieto-Diaz R., 1990, 'Domain Analysis: An Introduction', *ACM SIGSOFT Software Engineering Notes* **15(2)**, April 1990, 47-54.

[56] Prieto-Diaz R. & Freeman P., 1987, 'Classifying Software for Reusability', *IEEE Software*, **January 1987**, 6-16.

[57] Riecken R.D., Koenemann-Belliveau J & Robertson S.P., 1991, 'What Do Expert Programmers Communicate by Means of Descriptive Commenting', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S.P. Robertson, Ablex, Norwood NJ, 177-195.

[58] Rich C., 1981, 'Inspection Methods in Programming', Technical Report TR-604, Cambridge MA, Artificial Intelligence Laboratory, MIT.

[59] Riesbeck C.K. & Schank R.C., 1989, 'Inside Case-based Reasoning', Lawrence Erlbaum Associates, Hillsdale NJ.

[60] Robertson S.P., Davis E.F., Okabe K. & Fitz-Randolf D., 1990, 'Program Comprehension Beyond the Line', *Proceedings of INTERACT'90*, ed. D. Diaper, D. Gilmore, G. Cockton & B. Shackel, Elsevier Science Publishers B.V. (North-Holland), 959-963.

[61] Ross B.H., 1989, 'Distinguishing Types of Superficial Similarities: Different Effects on the Access and Use of Earlier Problems', *Journal of Experimental Psychology: Learning, Memory and Cognition* **15(3)**, 456-468.

[62] Ross B.H., 1987, 'This is Like That: The Use of Earlier Problems and the Separation of Similarity Effects', *Journal of Experimental Psychology: Learning, Memory and Cognition* **13(4)**, 629-639.

[63] Singley M.K. & Anderson J.R., 1989, '*The Transfer of Cognitive Skill*', Harvard University Press.

[64] Sinha A.P. & Vessey I., 1992, 'Cognitive Fit: An Empirical Study of Recursion and Iteration', *IEEE Transactions on Software Engineering* **18(5)**, 368-379.

[65] Soloway E. & Enrlich K., 1984, 'Empirical Studies of Programming Knowledge', *IEEE Transactions on Software Engineering* **10(5)**, 595-609.

[66] Soloway E., Pinto J., Letovsky S., Littman D. & Lampert D., 1988, 'Designing Documentation to Compensate for Delocalised Plans', *Communications of the ACM* **31**, 1259-1267.

[67] Soloway E., Enrlich K., Bonar J. & Greenspan J., 1982, 'What Do Novices Know About Programming ?', in *Directions in Human-Computer Interaction*, ed. B. Scneiderman & A. Badre, Ablex, Norwood NJ.

[68] Spohrer J.C. & Soloway E., 1989, 'Novice Mistakes: Are the Folk Wisdoms Correct ?', *Studying the Novice Programmer*, ed. E. Soloway & J.C. Spohrer, LEA.

[69] Sutcliffe A.G. & Maiden N.A.M., 1990, 'How Specification Reuse can Support Requirements Analysis', *Proceedings of Software Engineering'90*, ed. P. Hall, Brighton UK, July 24-27 1990, Cambridge University Press, 489-509.

[70] Thompson R. & Huff K.E., 1991, 'Supporting Understanding and Adaptation in Software Reuse', *Proceedings 1st Intl Workshop on Software Reusability*, Dortmund 3-5th July 1991, 45-50.

[71] Van Laar D. 1989, 'Evaluating a Colour Coding Programming Support Tool', *Proceedings of HCI'89*, ed. A. Sutcliffe & L. Macaulay, Cambridge University Press, 217-230.

[72] VanLehn K., 1988, 'Student Modeling', in *Foundations of Intelligent Tutoring Systems*, ed. M.C. Polson & J.J. Richardson, Lawrence Erlbaum Associates, 55-77.

[73] Visser W. & Hoc J-M., 1990, 'Expert Software Design Strategies', *Psychology of Programming*, ed. J.M. Hoc, T. Green, R. Samurcay & Gilmore D., Academic Press.

[74] Waddington R. & Henry R., 1990, 'Expert Programmers Re-establish Intentions When Debugging Another Programmer's Program', *Proceedings of INTERACT'90*, ed. D. Diaper, D. Gilmore, G. Cockton & B. Shackel, Elsevier Science Publishers (North-Holland), 965-970.

[75] Watt D.A., Wichmann B.A. & Findlay W., 1987, *Ada Language and Methodology*, Prentice-Hall (UK) Ltd.

[76] Weiser M., 1982, 'Programmers Use Slices when Debugging', *Communications of ACM* **25**, 446-452.

[77] Wood M. & Sommerville I., 1988, 'An Information Retrieval System for Software Components', *Software Engineering Journal* **3(3)**, 198-207.

[78] Wu Q. & Anderson J.R., 1991, 'Strategy Selection and Change in PASCAL Programming', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S.P. Robertson, Ablex, Norwood NJ, 227-238.

[79] Guindon R, 1991, 'Users Request Help From Advisory Systems With Simple and Restricted Language: Effects of Real-Time Constraints and Limited Shared Context', *Human-Computer Interaction* **6**, 47-75.

[80] Kruzela I., 1991, 'Successful Management Structures for Reuse', Proceedings UNICOM Seminar *Integrated Software Engineering With Reuse*, December 3-4 1991, Heathrow UK, 40-49.

[81] Sweller J., 1988, 'Cognitive Loading During Problem Solving: Effects on Learning', *Cognitive Science* **12**, 257-285.

[82] Polson M.C. & Richardson J.J., 1988, 'Foundations of Intelligent Tutoring Systems', LEA.