This article will appear in OzCHI'96, The Sixth Australian Conference on Computer-Human Interaction, Hamilton, New Zealand, 24-27 November 1996.

# Reuse in the Eye of Its Beholder: Cognitive Factors in Software Reuse

Jacob L. Cybulski
Department of Information Systems
The University of Melbourne
j.cybulski@dis.unimelb.edu.au

## Abstract

*In this paper we investigate the impact of human cognition on developers ability to effectively reuse software artefacts. We look at the cognitive impediments to and furtherance of software reuse. We review the computing models of human knowledge and reasoning which may assist in the emulation of our abilities to reuse software. Finally we investigate the possibility of integrating human and machine capabilities to arrive at the efficient method of software reuse.*

## 1. Introduction

As software reuse provides visible gains in productivity, reduced cost, improved reliability and enhanced quality of software products [14, 27], new techniques and methods are created to equip various professional groups with specialised tools to classify, find, select and adopt software artefacts used and produced in the development process. So, we find analysts producing requirements documents with the use of pre-existing domain models [4], designers reusing abstract program components to produce new system designs [16, 21, 30], or programmers assembling code from previously written parametrised functions, templates or object classes [7, 13, 29]. Different developer groups have different tasks to perform and goals to achieve, they use variety of notations, techniques and tools, they deal with distinct types of software artefacts, and apply different classes of knowledge and skills in their problem domains. And yet, they are all able to reuse the results of their own effort, they are capable of retracing and repeating common procedures leading to the production of artefacts within their domain of expertise. Moreover, they apply noticeably similar approaches to the reuse tasks. This means that, perhaps, the secret to the success or failure of software reuse lies not in the type and form of artefacts being produced nor in the methods and techniques required in their construction, but rather, in the nature of expert minds put to work on the reuse process.

People are an essential ingredient of the software development life-cycle. Managers instigate development of software projects; they set goals to achieve and control the progress of software project. Analysts communicate with the users to elicit and specify requirements; designers use their creative skills to convert requirements specifications into software solutions; programmers rely on their knowledge and experience to encode the designs into executable programs; and finally operators interact with the installed software. These acts of decision making, control, communication, creativity, application of knowledge and experience, and man-machine interactions make humans indispensable in the software development process. At the same time, however, it is this human factor that is also regarded as the main source of incompleteness, inconsistency and imprecision infused into the software products.

## 2. Impediments of human cognition to effective reuse

Sommerville [25, ch. 2] identifies several human factors commonly regarded as detrimental to the software development process, i.e.

- *human diversity* which makes it difficult to match employees with their particular work duties,
- *team work* which can be riddled with problems due to group dynamics, inadequate organisation, bad team communication or misperceived loyalties,
- *ergonomics of the workplace* which may lead to reduced productivity of software developers due to the lack of privacy, awareness of the outside world, or personal touch;
- *human cognition* which may hamper development efforts due to limitations of cognitive resources, inadequate modeling abilities or ineffective reasoning.

The issues of human diversity, team organisation and work ergonomics can certainly be addressed by appropriate management practices [14, 27]. The problems of human cognition are more complex to deal with and cannot be completely resolved by improved team structuring, refined work organisation, better planning and monitoring procedures, or the introduction of training.

Human memory is one of the main causes of inadequate reuse performance of software developers [8]. The long-term memories of software artefacts are formed of great many partial descriptions, they are fuzzy and imprecise, and are subject to temporal instability and misinterpretation. The short-term memory is used for temporary storage of problem descriptions, past solutions and methods of their reuse. It is also of a very limited capacity, it can hold only up to seven items of information at any point of time, and, unless it is constantly refreshed, it decays in less than half a minute. At the same time various processes controlling our perception and reasoning constantly compete and interfere with one another due to limited resources for analysing problems and their characteristics to establish the similarity between the new and the previously solved problems [5].

Deficiencies in human cognition are frequently quoted as the source of various problems specifically affecting software reuse. The most prevalent, especially amongst the less experienced developers, include [8] :-

- tendency to force new problems to fit existing solutions;
- difficulty in identifying similarity of concepts, problems and solutions across different problem domains; and
- inclination to the routine use of problem solving formalisms even if they may not be appropriate for the problem at hand.

Inexperienced programmers also tend to :-

- concentrate on the superficial or lexical properties of reuse artefacts while neglecting their semantic features [26];
- elaborate their designs in a depth-first fashion, thus, discounting reuse opportunities at a single level of software abstraction [2, p 257];
- have very poor abilities to develop and memorise new programming constructs, hence, they display difficulties in associating known program solutions with the new problems [2, p 258-259].

Expert programmers, as compared with junior programmers, have a much better memory for programs, they are able to mentally develop program templates and to match them with programming goals [2, p 259]. Experts also tend to develop and use jargon which reminds them of important programming constructs and enables the programmer to more economically represent and think about program plans. Unfortunately :-

- all developers, whether experienced or inexperienced, exhibit mental laziness when it comes to reusing artefacts and attempt copying reusable components rather than to reason about their suitability [26].

It seems that to achieve high quality and performance of the reuse process, the tasks involved cannot be left entirely to human endeavour. Computer-assisted reuse may be the only option to successfully defeat the shortcomings of the human condition.

## 3. Benefits of human cognition in software reuse

For years various studies were conducted to determine the cognitive behaviour of programmers [23], designers [15] and analysts [28]. Only recently, the efforts were also directed towards measuring the performance of software reusers [8, 19, 24, 26]. What transpires through some of these studies, is the realisation that effectiveness of software reuse practices, as performed with modern software development technology, finds its source in the intuition, insight, and inventive abilities of managers, analysts, designers and programmers, rather than in the technical support from specialised reuse tools and environments. In particular, it is recognised [20] that the performance of computer-based reuse systems as applied to certain reuse tasks does not measure up to the results obtained from the performance of human experts, e.g. in problem description, components understanding, their selection, adaptation and generalisation.

To combat both machine and human deficiencies, Maiden and Sutcliffe [20] postulate tight integration of human and machine participation in the reuse process and they propose the construction of tools not only supporting software engineers in their reuse tasks but also utilising humans as an additional source of knowledge and expertise in the process of computer-aided reuse. Unfortunately, few reuse mechanisms furnish such integration of machine and human expertise.

## 4. Cognitive tasks in software reuse

Automating those of the human natural abilities which facilitate successful software reuse requires careful analysis of all mental faculties invoked in the tasks of a typical reuse process. To achieve this understanding, we will extend the earlier work on the psychology of computer programming [23] and compare the tasks invoked in the reuse process with the cognitive phenomena of human memory and reasoning [2, 3, 12, 17, 18, 22]. To this end, we will consider the two components of the software reuse model, as consisting of development-for-reuse (construction of artefact library) and development-by-reuse (use of artefact library) [1, 6, 9], as identical with two complementing cognitive tasks of memorising and remembering information about reusable artefacts.

We will match the tasks of development-for-reuse, i.e. identification, understanding, generalisation, classification and storage of information about reusable components, with the respective cognitive processes of perception, conceptualisation, generalisation, categorisation, and memorising. At the same time, we will try to explain development-by-reuse, i.e. searching, retrieving, understanding, adaptation and integration, with the cognitive phenomena of memory activation, recall, choice, and reasoning. Our comparison (see the summary in table 1) will allow us to consider the methods of representing information about reusable artefacts and to conceive the techniques of manipulating such representations to aid the mental and manual tasks of a software reuser.

**Table 1. Development with reuse vs. cognitive tasks involved**

| **Development for reuse vs. *Memorising*** | **Development by reuse vs. *Remembering*** |
|---|---|
| ***Identification vs. Perception:*** | ***Search vs. Activation:*** |
| *Feature Analysis* | *Association* |
| *Pattern Recognition* | *Taxonomies* |
| *Feature Chunking* | *Patterns* |
| *Gestalt Principle* | *Context* |
| *Focus of Attention* | *Reasoning* |
| ***Understanding vs. Conceptualisation:*** | ***Retrieval vs. Recall:*** |
| *Propositions* | *Recognition* |
| *Semantic Networks* | *Recollection* |
| *Schemata & Frames* | *Reconstruction* |
| *Conceptual Graphs* | *Context & Mood* |
| *Production Rules* | *Metamemory* |
| *Predicate Rules* | |
| ***Generalisation vs. Generalisation:*** | ***Selection vs. Choice:*** |
| *Substitution* | *Economy* |
| *Deletion* | *Maximising* |
| *Integration* | *Satisficing* |
| *Abstraction* | *Elimination* |
| | *Compatibility* |
| ***Classification vs. Categorisation:*** | ***Adaptation & Integration vs. Reasoning:*** |
| *Similarity* | *Deduction* |
| *Typicality* | *Induction* |
| *Variability* | *Abduction* |
| *Reasoning* | |
| ***Storage vs. Memorising:*** | *Hybrid Approaches* |
| *Retention* | |
| *Forgetting* | |
| *Elaboration* | |
| *Mnemonics* | |
| *Structuring* | |

## 4.1 Development for reuse

Let us start with the process of software development-for-reuse, in which developer's memory and learning abilities facilitate identification, generalisation, classification and storage of information about reusable software artefacts.

**Identification.** Identification of reusable artefacts in the existing software requires developer's ability of *perception*, which in humans normally involves feature detection and analysis, pattern recognition, feature chunking, Gestalt effects and focus of attention. Such natural abilities may be significantly enhanced with simple techniques allowing visualising, emphasising or hiding certain artefact attributes as required in a given situation. For instance.

- Programmer's perception may be improved by assisting him or her in detecting and analysing artefact *features*. This could be achieved in the reuse software by identifying document structural and surface properties, highlighting keywords, and distinguishing between lexically different types of artefacts with the use of font and colour.
- Automatic detection and the subsequent visualisation of syntactic and semantic *patterns* amongst the collection of observed features, e.g. a simple text parser, may help programmers to classify different classes of observable objects, to identify relationships between them, and to differentiate between the more important and the less relevant information.
- Syntax-based editors or reverse-engineering tools, may further help a programmer to synthesise and integrate collections of observed features into larger and more meaningful feature *chunks* forming complete reusable artefacts. Recognised macro artefacts can then be automatically indexed and stored for future reference and possible reuse.
- On occasions, a programmer may directly associate a complex chunk of observed features with a specific artefact, via a *Gestalt* effect. In such a case, the reuse software may help him or her to decompose a selected artefact into its parts and their attributes to facilitate a better understanding of the artefact characteristics, e.g. by highlighting artefact's attributes or its sub-elements.

- Programmer's ability to effectively identify software artefacts and their properties greatly relies on his or her ability to focus *attention* on specific areas of analysed programs or program components and shifting this attention to the dependent or otherwise related program attributes. Controlling programmer's attention can be achieved by controlling the amount of visible details, by constructing outlines, abstracting certain program features, or by providing hypertext-like navigation facilities between related parts of program text.

**Understanding.** Effective reuse of software artefacts requires their thorough understanding, which in turn can only be achieved by the *conceptualisation* of artefact features, structures, function, its possible uses, and relationships between groups of such artefacts.

- Several existing software products, such as reverse engineering tools, provide programmers with facilities to detect, construct and *visualise* the semantics of software designs, data and programs using standard diagramming techniques.

- The more sophisticated software artefacts, such as feasibility studies or informal requirements specifications, may have to be processed with the use of more complex methods, e.g. those based on natural language understanding, knowledge acquisition and knowledge representation techniques. Cognitive scientists suggest several different types of conceptual structures fit to encode the contents of informal software documents, e.g. in propositional and semantic networks, schemata, frames and conceptual graphs, production and predicate rules, etc. Such rich *representation* can then be used not only to visualise artefacts in iconic form but also to reason about their properties and function.

**Generalisation.** Sometimes an artefact can be too specific or its representation may contain constraints limiting some of its future uses. In such cases, a developer may have to mentally *generalise* an artefact or its representation to better reflect its reusable features. Software reuse tools may provide some limited assistance in the programmer's tasks of artefact generalisation.

- Infrequently, software artefacts are produced in atomic, singular, and indivisible form. More often, they involve several sub-components, each of which imposes certain restrictions on the utilisation of the artefact as a whole. One of the simplest methods of making an artefact more general is to *substitute* one of its elements with a more general component. As commonly used taxonomies of artefacts already allow programmers to determine their generality, active assistance in determining artefact suitability for substitution could also enhance the reuse tasks.

- The process of software design frequently generates numerous constraints aimed at defining the type of data to be processed, at limiting the use of certain design or implementation concepts, specifying the user or the operator of the system, or listing certain efficiency or flexibility factors, etc. In the effort to make software more general, some of these constraints sometimes may have to be *deleted* or *relaxed*. This could be achieved by detailed analysis of artefact structure and its interaction with the application domain or by analogy to the previous attempts at generalisation of artefacts.

- Concepts can be made more general by *integrating* them with their subsumption, i.e. ideas which are either implied or redundant, but which may constraint future applications. The reuse tool may suggest hiding some of the artefact details which are not essential or not used in a given application context.

- Ultimately, concepts can be made general by surrounding them with *abstractions* representing entire sets of concepts. A reuse facility could offer hints on the opportunity of generalising or modularising collections of independent routines observed to be coupled via common data, control or sequence.

**Classification.** In the next step in the process of software development for reuse, in his mind, a developer would *categorise*, i.e. classify, cluster and index, selected artefacts to ensure easy recall of their description in the subsequent reuse. Various information retrieval techniques can be utilised to assist artefacts classification based on their *similarity*, *typicality* and *variability* of their features. Some forms of computer-assisted *reasoning* may also be used in this classification process, e.g. by deduction and induction, by analogy or case-based reasoning, all of the methods used successfully in learning programs.

**Storage.** As any other kind of knowledge held by software developers, information about reusable artefacts is *memorised* in the long term memory for future recall, i.e. references to reusable artefacts, their observed features, their structural and functional decomposition, their propositional, associative and schematic representations, artefact generalisations as well as their particular instances, various types of indexes and classifications allowing easier access to artefact description, cases and reasoning rules about artefacts, etc.

- Memories which are exercised and strengthened have better *retention*, memories which are disused or which interferes with the competing facts slowly deteriorate, their associations weaken and they are eventually *forgotten*. It should, hence, be the computer tools that need to maintain all the necessary artefact associations for programmers to follow during the reuse process. Such tools may also provide developers with reuse training facility to refresh and exercise their memories of stored artefacts.

- Human memories of particularly important facts get continually *elaborated* with information about their specific uses, relevance, plausibility and context. Reuse tools must then offer facilities to continually gather and enrich the collection of knowledge about reposited artefacts. They may also provide tools for periodic acquisition of new artefact information about frequently used or modified software components.

- Experiments show that *structured* information is memorised better than that which is unstructured To improve the chance of correct and effective artefact retrieval, reuse tools should suggest methods of structuring artefacts into libraries, packages, abstract data types, modules, database schemata, etc.

- *Mnemonic* strategies, relying on peg-words, loci, rhymes, free imagery and associations, can also be used to provide additional cues for future retrieval. Annotation and personal notes should be stored with artefact formal representation and description to improve future recall.

## 4.2 Development by reuse

Let us in turn focus on the development tasks that take place in the process of software development-by-reuse. Here we find that human unparalleled ability of pattern-matching used in the recognition and recall of memories allows effective recollection of information about reusable artefacts. It is also the reasoning skills and the ability to reconstruct and reorganise human memories that expedite the adaptation and integration of reusable components.

**Search.** In developing new systems, expert developers identify reuse opportunities by relying on their experience with previously developed programs, their knowledge of common program components and the way they interact with each other, and the use of development procedures which enhance the reuse practices. Novices, however, find relevant software components by mentally matching their descriptions with the specification of the new software. Knowledge of all the matching software artefacts will become *activated* for the subsequent recall.

- Having the initial problem description, expert reusers utilise problem-domain concepts and the technical jargon as a guide to *associate* and subsequently recall information about the necessary software components. The strength of association between the concepts depends on *contiguity* of concepts and the *frequency* of their simultaneous use. Keeping track of development history and context may help developers to recall artefact associations based on the previous simultaneous uses of related artefacts.

- Novices who lack the knowledge and experience in both development and reuse, have to laboriously scan artefact collections, trying to identify software appropriate for reuse. Evidently, those of the reusers who cannot instantly associate appropriate concepts will greatly benefit from various visual representation of artefact relationships and dependencies, e.g. through the use of *lists*, *maps* or *taxonomies*.

- Both types of developers will use their innate ability to form complex *patterns* of required software features to match the artefacts considered for reuse. With experience, reusers can fall back on *context* and *reasoning* to promote or reject likely reuse candidates. Reuse tools may apply complex reasoning to assist in searching for required artefacts.

**Retrieval.** Once the search of the reuse repository is completed, e.g. through the use of indices or classes of artefacts, a developer may then have to fetch or *recall* the details of the best candidate artefacts and to make a full assessment of their suitability for reuse.

- The simplest method of memory recall is by *recognising* previously learnt patterns of stimulus-response pairs, e.g. being able to determine whether a given fact constitutes previously memorised or new information, or being able to select the correct/known pattern from a given list of alternatives. Hence, reuse tools should provide artefact browsing and previewing capabilities.

- Small amounts of information may be fully *recollected* exactly as they were learnt. Such recollections may be totally free, stimulated only by the problem at hand, or they can be triggered by additional memory cues, i.e. information associated with the learnt facts. Query facilities will assist the developer to retrieve artefact associated with keyword or conceptual cues.

- Remembering larger units of information usually involves recollecting partial memories, using them to *reconstruct* the originally learnt facts, and subsequently recognising re-generated information as valid. Information retrieval techniques could be used to map a partial artefact description into a set of matching artefacts.

- Factors frequently quoted as assisting memory retrieval are those recall cues which go beyond the learnt information contents, i.e. *context, mood*, *state of mind*, etc. The benefits of contextual cues can be felt as long as the retrieval cues are processed in the same way as during memorising. Personal artefact annotations could be used in artefact indexing and classification, so that retrieval queries could be formulated not only in terms of artefact features but also in terms of contextual cues.

- Finally, the knowledge of your own memory mechanisms, known as *metamemory*, may provide additional retrieval cues. Reuse tools should be capable of formulating queries against a development history, so that past retrieval successes and failures could be used to guide processing of the new artefact searches.

**Selection.** In the development-for-reuse, artefacts are memorised in terms of conceptual representations which are subsequently stored as part of our knowledge of an artefact, they are used as cues and associations in the process of finding and retrieving artefacts, and they can also be used to understand the similarities and the differences between retrieved candidates to select the most appropriate artefact. Thus the onus of the understanding process in the development-by-reuse is on *choosing* an artefact, and with it on associated decision making.

- The economic theory of choice relies on decision makers to construct a formal and ideally universal model of choice, in which all available options are listed, ranked in various dimensions and compared. A measure of semantic closeness between reusable artefacts may be used to rank their utility in a given situation.
- The theory of risky choice, deals with decision in the face of uncertainty, it predicts that decision makers will attempt *to maximise the expected utility*, i.e. long-run expected gains, of the selected option. Frequency of artefact uses could be used to determine the probability of artefact expected usefulness in a new situation.
- The theory of bounded rationality aims at reaching a level of satisfaction (*satisficing*) of decision makers rather than maximisation of benefits. In this approach optimal artefact could actually be missed as the first satisfactory match would be used.
- *Elimination by aspects* model of choice relies on the selection of decision dimensions, finding the best option within each dimension, and elimination of all choices which are not close to the selected one. Based on this model, selection of artefacts would trigger artefact pruning until no more candidates are matched against eliminating constraints.
- Another general mechanism that might support the response-mode effects is the *compatibility principle*, in which, the weight of any input component of a stimulus is enhanced by its compatibility with the output (response). The rationale for this principle is that any incompatibility requires additional mental transformation, which increases the effort and error, thus, reducing the impact of stimulus. A reuse tool which considers the imperfect artefacts compatible with the requirements of utmost importance though violating the less important constraints.

**Adaptation and Integration.** Integration of the selected software artefacts into the necessary problem solution may require developers to parametrise, alter, and in some cases drastically restructure the available artefacts. In the process developers depend on their reasoning abilities, their knowledge and experience, their intuition and other subconscious processes. On occasions, they try to use brute force to experiment with the problem and to test its boundaries. More often, however, they use sophisticated reasoning methods, such as induction, deduction and abduction. Expert problems solvers will intertwine several paradigms to arrive at a hybrid but at the same time very effective problem solution. Various computer-based reasoning techniques, implemented in a form of expert systems and knowledge based systems, have been suggested as effective in a limited domain of application.

## 5. Applications to requirements reuse

A prototype software reuse tool, RARE IDIOM/SoDA offering many of the above-mentioned cognitive attributes has been designed and subsequently developed (see the summary in table 2). RARE IDIOM/SoDA [10, 11] assists software requirements engineers in the process of analysis, refinement and reuse of informal software requirements documents, which are predominantly in natural language text (e.g. English). The tool automates many of the reuse tasks (identification, understanding, classification, storage, searching and retrieval), but it also provides its users with an opportunity to volunteer their reuse skills in those activities regarded as exclusively in the human domain of expertise (generalisation, selection, adaptation and integration). Hence, RARE IDIOM/SoDA extends requirement engineers natural abilities to analyse and reuse textual software artefacts and it embeds the human in its technological cycle.

RARE IDIOM/SoDA analyses the requirements texts, identifying in them the features of known software artefacts described in the ever-expanding domain vocabulary. Such features are subsequently used as the basis for defining hypertext links between requirements (informal) and specification (formal) documents stored in the system repository. The restricted-natural-language parser unravels natural language phrases into patterns of artefact-related concepts, their attributes and relationships. This process is further assisted by the customised text editor used in the collection of software requirements, which assures the text entered into a document template to be structured into topical and cohesive chunks of requirements texts. Hypertext navigation and document outlines are used to direct users attention to specific documents or their components.

We have decided to represent individual software requirements in a semantic network. The concepts are presented to the user as hypertext buttons, their attributes and

**Table 2. Cognitive considerations in the design of a software-requirements-reuse tool**

| ***Reuse Phase*** | **Techniques** |
|---|---|
| *identification* | features - domain vocabulary<br>patterns - natural language parser<br>chunks - customised text editor<br>attention - hypertext & outlines |
| *understanding* | presentation - hypertext buttons<br>representation - semantic networks |
| *generalisation* | abstraction - artefact taxonomy |
| *storage & retrieval* | repository - relational database<br>organisation - facts & rules<br>mnemonics - annotations |
| *classification & search* | association - facets<br>taxonomy - concept hierarchy<br>recollection - menus & dictionary<br>recognition - hypertext links |
| *selection* | economic - generated alternatives<br>satistificing - volunteered selection |
| *integration adaptation* | copying - text replication<br>referencing - hyper-linking<br>specialisation - template filling<br>transformation - history replay |

relationships are recalled with the aid of menus and hypertext links navigable between potentially distant text fragments. Although RARE IDIOM/SoDA does not attempt to automatically generalise software requirements, it provides the user with the facility to use hypertext as the means of inspecting associative and taxonomic relationships between previously analysed artefact abstractions.

Knowledge of accumulated software requirements is organised into a collection of PROLOG facts and rules, which are stored into a relational database for efficient retrieval. Artefact mnemonics, in the form of annotations that could potentially be used as an additional source of concept associations, are processed and stored in exactly the same fashion as the requirements text themselves.

In its processing cycle, IDIOM/SoDA relies on the human expert to select, integrate and adapt requirements artefacts. In the process of requirements analysis the system follows the economic model of choice, suggesting to the user a selection of the semantically closest refinements of analysed requirements. At the same time, RARE IDIOM/SoDA will alter its artefacts representation to allow for the user to volunteer even the most unlikely though satisfactory choice of requirements interpretation. Once a partial requirement specification is generated, it can be further adapted and elaborated by the user with the built-in text editor, hypertext linker and template editor. Further work will also include the facility to automatically alter new requirements documents by replaying previously observed text transformations.

## 6. Summary and conclusions

This paper considered various cognitive factors in software development for-reuse and by-reuse which are both thought to be vital for the effective software construction. We have matched a collection of typical software reuse activities performed by software developers with a range of cognitive tasks invoked in the process of their completion. Several software technologies were then briefly assessed for their suitability to support the identified cognitive tasks. The result of our analysis can subsequently be used to design a practical software tool capable of integrating both machine and human capabilities to act in-concert in the reuse process. In general, such a reuse tool should exhibit the following attributes:

- it should automatically identify and manipulate artefact features, their patterns and groups;
- it should provide mechanisms for the visualisation and representation of artefact semantics;
- it should offer assistance in the generalisation of artefact features by component substitution, deletion and relaxation, integration and abstraction;
- it should assist developers in the process of artefact classification by similarity, typicality and variability, possibly with the use of sophisticated reasoning;
- it should deliver effective tools for browsing and searching artefact lists, maps and taxonomies;
- it should embed programming and training aids for artefact recollection and their retrieval from software libraries;
- it should help software reusers to select the best from amongst of retrieved candidate artefacts;
- it may offer some limited assistance in artefact adaptation and integration.

Right now and in the near future, practical reuse will only be successful with the human in its cycle. We should not strive to supplant human natural abilities with still inferior technology but rather to enhance it for better performance. Reuse is only in the eye of its beholder!

## 7. References

[1] Agresti, W.W. and F.E. McGarry, "The Minnowbrook Workshop on Software Reuse: A summary report". In *Software Reuse: Emerging Technology*, W. Tracz, Editor. Computer Society Press: Washington, D.C. 1988, p. 33-40.

[2] Anderson, J.R., *Cognitive Psychology and Its Implications*. Second Edition ed. New York: W.H. Freeman and Co., 1985.

[3] Anderson, J.R., *Learning and Memory: An Integrated Approach*. New York: John Wiley & Sons, Inc., 1995.

[4] Arango, G., "Domain analysis methods". In *Software Reusability*, W. Schafer, R. Prieto-Diaz, and M. Matsumoto, Editors. Ellis Horwood: London, Great Britain. 1994, p. 17-49.

[5] Bobrow, D.G. and D.A. Norman, "Some principles of memory schemata". In *Representation and Understanding*, D.G. Bobrow and A. Collins, Editors. Academic Press, Inc.: New York, NY. 1975, p. 131-149.

[6] Bubenko, J., *et al.*, "Facilitating "Fuzzy to Formal" requirements modelling". In *The First International Conference on Requirements Engineering*, Colorado Springs, Colorado: IEEE Computer Society Press. 1994, p. 154-157.

[7] Cheng, J., "Reusability-based software development environment", *ACM SIGSOFT Software Engineering Notes* **19**(2), 1994: p. 57-62.

[8] Curtis, B., "Cognitive issues in reusing software artifacts". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 269-287.

[9] Cybulski, J.L., *Sharing and reuse in the development of information systems*. Research Report 96/4, The University of Melbourne, Department of Information Systems: Melbourne. 1996.

[10] Cybulski, J.L. and K. Reed, "A hypertext-based software engineering environment", *IEEE Software* **9**(2), March 1992: p. 62-68.

[11] Cybulski, J.L. and K. Reed, *The Use of Templates and Restricted English in Structuring and Analysis of Informal Requirements Specifications*. Research Report TR024, Amdahl Australian Intelligent Tools Programme, La Trobe University: Bundoora. 1993.

[12] Eysenck, M.W., *Principles of Cognitive Psychology*. Hove, UK: Lawrence Erlbaum Assoc., Pub., 1993.

[13] Goguen, J.A., "Principles of parametrised programming". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 159-225.

[14] Hemmann, T., *Reuse in Software and Knowledge Engineering* , German National Research Center for Computer Science (GDM), Artificial Intelligence Research Division. 1992.

[15] Kant, E. and A. Newell, "Problem solving techniques for the design of algorithms", *Information Processing and Management* **28**(1), 1984: p. 97-118.

[16] Katz, S., C.A. Richter, and K.-S. The, "PARIS: A system for reusing partially interpreted schemas". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 257-274.

[17] Leahey, T.H. and R.J. Harris, *Human Learning*. Second ed. Englewood Cliffs, New Jersey: Prentice Hall, 1989.

[18] Lindsay, P.H. and D.A. Norman, *Human Information Processing: An Introduction to Psychology*. Second ed. San Diego: Harcourt Brace Jovanovich, Pub., 1977.

[19] Maiden, N.A. and A.G. Sutcliffe, "Exploiting reusable specifications through analogy", *Communications of the ACM* **35**(4), 1992: p. 55-64.

[20] Maiden, N.A.M. and A.G. Sutcliffe, "People-oriented software reuse: the very thought". In *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability,* Lucca, Italy: IEEE Computer Society Press. 1993, p. 176-185.

[21] Marques, D., *et al.*, "Easy programming: empowering people to build their own applications", *IEEE Expert* **7**(3), June 1992: p. 16-29.

[22] Norman, D.A., *Memory and Attention: An Introduction to Human Information Processing*. Second ed. New York: John Wiley & Sons, Inc., 1976.

[23] Shneiderman, B., *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, MA: Winthrop Publishers, 1980.

[24] Soloway, E. and K. Ehrich, "Empirical studies of programming knowledge". In *Software Reusability: Applications and Experience*, T.J. Biggerstaff and A.J. Perlis, Editors. Addison-Wesley Pub. Co.: Readings, Massachusetts. 1989, p. 235-267.

[25] Sommerville, I., *Software Engineering*. 4 ed. Wokingham, England: Addison-Wesley Pub. Co., 1992.

[26] Sutcliffe, A. and N. Maiden, "Specification reusability: why tutorial support is necessary". In *Software Engineering 90*, Brighton, U.K.: Cambridge University Press. 1990, p. 489-509.

[27] Tracz, W., "Software reuse: motivators and inhibitors". In *Software Reuse: Emerging Technology*, W. Tracz, Editor. Computer Society Press: Washington, D.C. 1988, p. 62-67.

[28] Vitalari, N.P. and G.W. Dickson, "Problem solving for effective systems analysis: an experimental exploration", *Communications of ACM* **26**(11), 1983: p. 948-956.

[29] Volpano, D.M. and R.B. Kieburtz, "The template approach to software reuse". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York. 1989, p. 247-256.

[30] Waters, R.C. and Y.M. Tan, "Toward a design apprentice: Supporting reuse and evolution in software design", *ACM Software Engineering Notes* **16**(2), 1991: p. 33-44.