Cognitive Requirements for Software Reuse Tools

Technical Report 98/3[1]

Jacob L. Cybulski
Department of Information Systems
The University of Melbourne
Parkville, Vic 3052, Australia
Phone: +613 9344 9244, Fax: +613 9349 4596
Email: j.cybulski@dis.unimelb.edu.au
Internet: http://www.dis.unimelb.edu.au/staff/jacob/

Software reuse is in the eye of beholder.

Abstract

This report is an extension of a paper presented at OzCHI'96. The report investigates the impact of human cognition on the reuse process. It looks at the human factors that are detrimental to software development and others that benefit this process. It discusses cognitive abilities that are relied on by software developers involved in the reuse of different types of software artefacts. Examples of such reuse tasks and mental processes are provided to clarify the main concepts. The article also reviews various computing models of human knowledge and reasoning that may assist in the emulation of human abilities to reuse software. Finally the report investigates the possibility of integrating human and machine capabilities to arrive at the efficient method of software reuse.

[1] This report is an extension of an article that appeared in [15]

1. Impediments of Human Cognition to Effective Reuse

Is human an obstacle to the successful software reuse?

As software reuse gains popularity amongst developers, the 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 [6], designers reusing abstract program components to produce new system designs [32, 86, 49], or programmers assembling code from previously written parametrised functions, templates or object classes [22, 85, 12]. 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 retracting 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, or man-machine interaction make human indispensable in the software development process. At the same time, however, it is this human factor which is also regarded as the main source of incompleteness, inconsistency and imprecision infused into the software products.

1.1 Human factors detrimental to software development

Sommerville [78, ch. 2] identifies several such factors commonly regarded as detrimental to the software development process, i.e.

- *human diversity* makes it very difficult to match employees with their particular work duties, although there is no real evidence that programming ability is related to any particular combination of personality traits, the absence of certain characteristics may mean that some people are not well suited to perform certain software development tasks, e.g.
    - ◊ having no ability to work under stress, or
    - ◊ being unable to quickly adapt to a changing work environment;

- *team work* is riddled with problems of group dynamics, organisation and team communication, e.g.
    - ◊ some team members, having completely different psychological profiles, may find it difficult to work with each other;
    - ◊ loyalties of team members may either be too weak (overly independent individuals) or too strong (individuals loyal to the point of lack of judgement);
    - ◊ organisation of the team may promote or impede group interaction depending on the team size, structure, status and personalities of members and the physical work environment;
- *ergonomics* of the workplace may suffer reduced productivity of software developers due to the lack of :-
    - ◊ privacy,
    - ◊ awareness of the outside world, or
    - ◊ personal touch;
- *human cognition* may hamper development efforts due to :-
    - ◊ limitations of cognitive resources (e.g. short and long term memory),
    - ◊ inadequate modeling abilities (fuzziness, imprecision and incoherence), or
    - ◊ ineffective reasoning (e.g. informal, subjective or incomplete).

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

1.2 Problems due to human involvement in reuse

Human memory is one of the main causes of inadequate reuse performance of software developers [14]. 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 [9].

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 [14] :-

- 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 [81];
- elaborate their designs in a depth-first fashion, thus, discounting reuse opportunities at a single level of software abstraction [4, 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 [4, 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 [4, 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 [81].

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 human condition.

2. Benefits of Human Cognition in Software Reuse

Is human the catalyst of the reuse process?

For years various studies were conducted to determine cognitive behaviour of programmers [71], designers [31] and analysts [84]. Only recently, the efforts turned also into measuring the performance of software reusers [81, 47, 14, 77]. 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 [48] that 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. These tasks include the following :-

- *describing problems* in such a way as to promote future reuse,
- *understanding reusable components* after their retrieval,

- *selecting the best component* from several candidates,
- *adapting the selected component* to fit the new problem, and,
- *extending component functionality* without violating the global integrity constraints, in addition,
- a significant portion of potentially reusable knowledge is neither formal nor recorded, and *exists only in the minds* of software reusers [1].

Thus, to combat both machine and human deficiencies, Maiden and Sutcliffe [48] 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 the humans as an additional source of knowledge and expertise in the process of computer-aided reuse. Such reuse tools must fullfil a number of special requirements.

- They should provide a problem-description notation which facilitates easy retrieval of reusable components.
- The tools should assist software engineers in understanding and assimilation of information manipulated by software components.
- The tools should aid the process of understanding component functionality, structure and boundaries.
- Users should benefit from the tool's guidance in selecting the most appropriate of the retrieved artefacts by assisting the user in identifying of and reasoning about differences between those artefacts.
- The tools should assist in identification of and reasoning about key differences between the selected component and original problem.
- Users should be actively discouraged to reuse by literal copying of artefacts during their adaptation.

Maiden and Sutcliffe further emphasise that a notation used for artefact description should must be tailored specifically to human rather than machine use. The common utilisation of restricted terminology by reuse tools is very convenient for the implementation of automatic indexing and classification schemes, nonetheless, such limited vocabulary proves to be a practical problem in the expression of software specifications. The more effective communication, on the other hand, requires either the power of natural language or the facilities extending the range of available terms by special dictionaries of word aliases and semantic facets. In general, unfortunately, few reuse mechanisms furnish such a notational or terminological support to software engineers.

3. Cognitive Tasks in Software Reuse

What's the essence of the reuse process?

Automating those of the human natural abilities which facilitate the 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 consider the two components of the software reuse model, as consisting of development-for-reuse and development-by-

Development for Reuse - Memorising		**Development by Reuse - Remembering**	
Task	**Cognitive Skills**	**Task**	**Cognitive Skills**
Identification	Perception *and Feature Analysis* *and Pattern Recognition* *and Feature Chunking* *and Gestalt Principle* *and Focus of Attention*	Search	Activation *by Association* *by Taxonomies* *by Patterns* *by Context* *by Reasoning*
Understanding	Conceptualisation *with Propositions* *with Associations* *with Schemata*	Retrieval	Recall *with Recognition* *with Recollection* *with Reconstruction* *with Context & Mood* *with Metamemory*
Generalisation	Generalisation *by Substitution* *by Deletion* *by Integration* *by Abstraction*	Selection	Choice *by Maximising* *by Satisficing* *by Elimination* *by Compatibility*
Classification	Categorisation *by Similarity* *by Typicality* *by Variability* *by Reasoning*	Adaptation & Integration	Reasoning *by Deduction* *by Induction* *by Abduction*
Storage	Memorising and Memory *Retention* *Forgetting* *Elaboration* *Mnemonics* *Structuring*		

Table 3-1 Cognitive skills assisting reuse

reuse, as identical with two complementing cognitive tasks of memorising and remembering information about reusable artefacts. Having this assumption in mind, we will compare the tasks invoked in the reuse process with the cognitive phenomena of human memory and reasoning [for more information on the selected aspects of human cognition see 4, 5, 18, 36, 40, 55].

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 (cf. Table 3-1). Our comparison 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.

Let's 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. Let’s use an example of data merging and sorting to illustrate our main points.

3.1 Identification

Identification of reusable artefacts in the existing software requires developer's ability of *perception*. As the majority of software artefacts are of a textual nature, we will assume the perceptual processes to be similar to those occurring in text processing and thus described by cognitive linguists [56].

- First the reuser must be able to perform *feature analysis* of a given software artefact, i.e. to detect its design and implementation characteristics. For our purposes, we’ll assume the process to be similar to that of lexical analysis of text, whereby, the lowest perceptual elements are those related to the surface units of text, i.e. strings of alphabetic and punctuation characters [cf. 26].

  While perusing the listings of a sorting program, developer must be able to identify declarations of data types (e.g. type of data to be sorted) and variables (e.g. data set to be sorted), function headers and function calls (sort header and invocation of the sorting process), implementation of routines and modules (e.g. sorting method), documentation and comments, etc.

- Next, the observed features undergo *pattern recognition*, in which some features are determined to be relevant and considered for further processing, others to be totally irrelevant and henceforth ignored. In the process, analogous to the syntactic analysis of text units, the lower level artefact features are clustered into more complex feature structures.

  In the majority of reuse cases, routine or module interfaces (e.g. sort function header) are more important than their algorithms (e.g. quick-sort algorithm), data types or class declarations (e.g. ordered data set) are more useful than detailed implementation of data structures (e.g. array of numbers), etc. At the same time, in some cases, an abstract sorting algorithm (e.g. merge-sort, quick-sort or heap-sort) may also be considered as reusable, and be re-applied to various types of data (e.g. files, lists or arrays).

- The selected artefact features are subsequently integrated into larger and more meaningful feature *chunks*. Taking the grammarian’s point of view, chunks are related to the syntactic units of text and the feature chunking is equivalent to the bottom-up parsing techniques.

  E.g. definition of several data types can be taken as leading to the specification of a single data structure, declaration of a type together with routines manipulating its instances can be perceived as an abstract data type, program structure can be derived from the routine calls, data flows can be reconstructed from the use of variables, etc.

- On occasions, recognition patterns may associate a complex chunk of features direct, *Gestalt principle*, in

  E.g. a familiar module may be recognised more readily by its name (e.g. sort), rather than by its individual

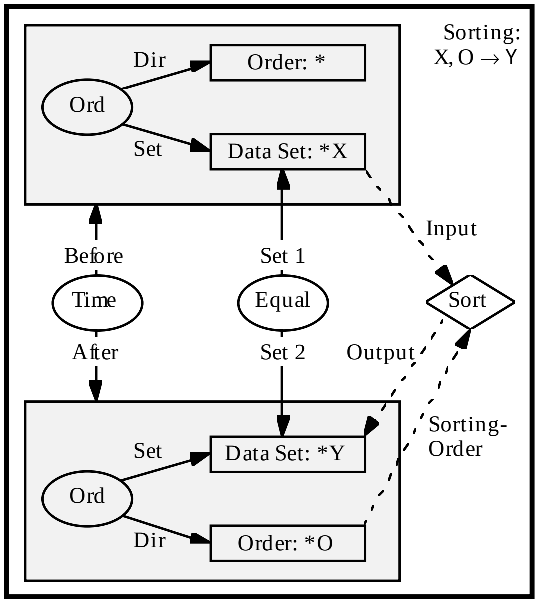

Figure 3-1 Sample propositional net

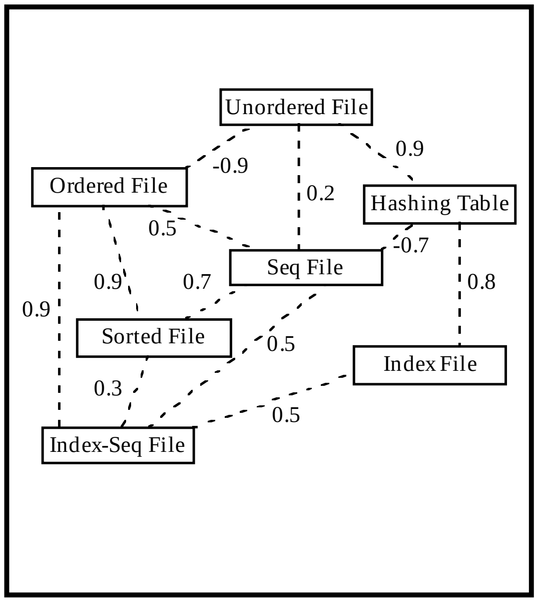

Figure 3-2 Sample associative net

which case, the chunk may have to be broken down into its component features to facilitate better understanding of its characteristics. In linguistics, this phenomenon can be compared to the head-driven parsing [3, p 153, 76, pp 325-330, 20, pp 224-226].

components (e.g. its input, output, and order constraints). To better comprehend such module's aims and methods, it may still be necessary to perform detailed analysis of its member functions, formal arguments and data structures.

- Pattern recognition greatly relies on a developer's ability to focus *attention*. This ability is greatly enhanced by developer's *familiarity* with the problem and the *context* in which an artefact appears. In text processing, some context phenomena can be addressed by the extensive use of memory, such as a chart [33] or context spaces [63]. Familiarity frequently relates to expectations [65], preferences [87, 88] or interestingness [69].

E.g. knowledge of different implementation methods, knowledge of the language used to implement the module, knowledge of requirements, specifications and documentation, the constraints and limitations imposed on the intended use of the program, etc.

3.2 Understanding

Observed artefact feature chunks provide artefact *conceptualisation* which will assist a developer in the full comprehension of artefacts' structure, their attributes, their function and possible uses, and relationships between artefacts.

- Chunks may be represented in memory in terms of semantic

E.g. the concept of sorting may be described by a semantic network which

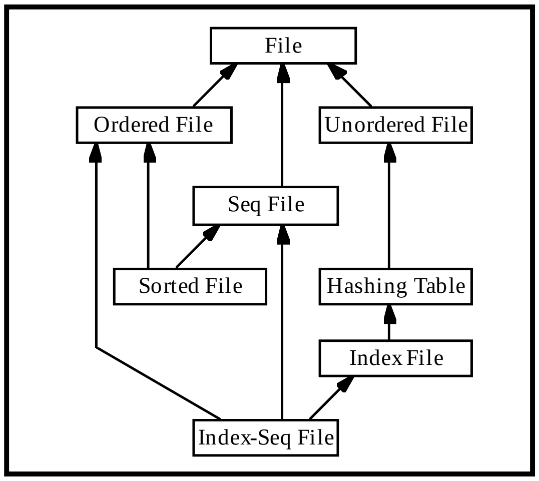

Figure 3-3 Sample inheritance net

structures, such as *propositional* [4, pp 116-121] or semantic networks [62], frames [52] or conceptual graphs [79].

represents relationships (ovals) between unsorted and sorted data sets (rectangles), and the function (diamond) which orders the set according to the selected criteria (cf. Figure 3-1).

- Related concepts can be *associated* via sets of semantic relationships or feature values. The strength of these associations may subsequently be used to identify similar concepts and to provide a mechanism to activate or inhibit relevant concepts during recall.

E.g. Concepts of sorted and index-sequential files are closely related to ordered data sets, whereas hashing tables are strongly associated with unordered data sets (cf. Figure 3-2).

- In particular, an artefact may become linked to a description of a more general *schematic* artefact, which could then be used for knowledge inheritance, i.e. filling in missing or default reuse information.

E.g. we may distinguish several classes of files, i.e. ordered, unordered and sequential, some sequential files may also be ordered when sorted, hashing tables may be considered to be unordered, but index-sequential files take properties or ordered, sequential and index files (Figure 3-3).

3.3 Generalisation

Artefacts or their descriptions may have to be *generalised* to broaden their scope of applicability. [See 83 for a discussion of generalisation methods].

- 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.

E.g."merge-sorting of numeric data" may have to be redesigned and reformulated into a more general concept of sorting arbitrary data sets.

- Software artefacts are infrequently produced in atomic, singular, and indivisible form. More often, they involve several sub-components, each of which may impose certain restrictions on the utilisation of the artefact as a whole. The process *substituting* selected artefact components with the more general ones, will also make the whole artefact more general as well.

  E.g. in the routine of "number merge-sorting", the substitution of "merge-sort" with an arbitrary "sort", will make the routine of "number sorting" more independent of the methods in use, and thus, more general.

- 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*.

  E.g. in the routine of "number sorting", the routine's makes an assumption that the record contains the key only, and that the key is numeric. Removing both of these constraints will allow the routine to sort records of any structured type, and having no restriction on the type of the key, thus, making it more general.

- 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.

  E.g. the concept of "sorting" necessarily subsumes the activities of data "input", "order" and "output", plus certain optional activities, like "assume ascending order". Specifying the fact that "sort inputs data to be ordered and then outputs the sorted data" will not have any impact on the program generality. However, declaring "sorting with ascending order" will limit the concept generality. Such optional specification statements should be removed by integrating them into the main sort requirements.

- Ultimately, concepts can be made general by surrounding them with *abstractions* representing entire sets of concepts.

  E.g. the concept of "file input", "order", "file output" and "ascending/descending order" may certainly be used to effectively maintain the order of data files. However, abstracting these activities into an independent concept of "sort" will allow generalisation of data input and output via mediating processes, data sorting in partial order, or adding additional steps into the *sorting process.*

Any of these techniques leads to more generic artefact descriptions thus making them more applicable to future reuse tasks.

3.4 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. [See 75 for a discussion of memory categorisation].

- In this task, an artefact is checked for the *similarity* of its features to those of other artefacts.

  E.g. various sorting routines, i.e. "merge-sort", "quick-sort", "bubble-sort", may be found to be similar in their functional requirements, but different in the methods used.

- It is also important to consider artefact *closeness* to or *typicality* of the selected artefact categories.

  E.g. although both "sorting" and "merging" result in an ordered data set on their output, the input to "sorting" is a single and unordered data set, while that of "merging" consists of two ordered data sets. Hence, based on the closeness of the matching process between "merge-sort" and the generic classes of "sorting" and "merging", "merge-sort" can be classified as a better representative of "sorting" than that of "merging".

- The *variability* in artefact features within a category may determine whether the new artefact can be adequately characterised as a member of that category.

  E.g. the variation in file access will not prevent a file from being sorted (whether is sequential, index-sequential or random), however, the requirement for the records to be serialised in any key order (no variations) will prevent a hashed file to be considered for sorting.

- In some cases an artefact is also analysed by various forms of *reasoning* to discover its reuse or applicability potential, i.e. reasoning by deduction and induction [80], and in particular by analogy [21] and cases [35].

  E.g. since an index-sequential file is already ordered then it does not have to be sorted by its primary key.

3.5 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.

- As one artefact is used more frequently than others, memories of its features are exercised and strengthened, thus, leading to improved memory *retention* and thus increased potential of its reuse.

  E.g. as quick-sort or merge-sort are used routinely in many programs, the use of other sorting algorithms may be neglected, even though they may be more applicable in some cases, e.g. the use of heap-sort in memory-limited applications.

- Whenever an artefact is disused or its reuse interferes with the use of other, possibly competing, artefacts, its access slowly deteriorates, its associations weakened and it is eventually *forgotten*.

  E.g. bubble-sort may be superseded by improved quick-sort, generic sort will be preferred over a more specific quick-sort, and the "Shell's diminishing increment insertion sorting method" may not be used due to its long and threatening name.

- Memories of some particularly important artefacts get continually *elaborated* with information about their specific uses, relevance, plausibility and context.

  E.g. the concept of sorting may be elaborated with knowledge of key-order (numeric or alphabetic, ascending or descending, partial), sorting-methods (bubble-sort, quick-sort, heap-sort) and their space/time efficiency, data structures which maintain order (ordered lists and trees), or programming constructs which provide sorting facilities (Cobol, Lisp or Unix sort), etc.

- *Mnemonic* strategies, relying on peg-words, loci, rhymes, free imagery and associations, can also be used to provide additional cues for future retrieval.

  Naming complex concepts with peg-words denoting familiar notions or imagery facilitating easier recall, e.g. ordered tree balancing may be compared to "rotations", sorting by element exchange to "bubbling", or array counting to "histogramming", finding a maximum of a function to "hill-climbing", etc.

- Experiments show that *structured* information is memorised better than that which is unstructured, thus, to improve the chance of correct and effective artefact reuse developers should structure their artefacts into libraries, packages, abstract data types, modules, database schemata, etc.

  E.g. we may want to bundle a merge-sort into a package manipulating sequential files, quick-sort into lists, heap-sort into arrays, and a generic sort into a package manipulating arbitrary data collections. Such structuring of reuse information will increase the chance of reusing the sort most suited to a given problem.

Let's in turn focus on the development tasks that take place in the process of software development-by-reuse. Here we find that human unparallel 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.

3.6 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 [58].

E.g. when facing the task of printing customer records in order of their names, the experts will recognise the need for record sequencing and ordering, indicating the use of either index-sequential files or sequential files with sort.

- Novices who lack the knowledge and experience in both development and reuse, have to laboriously scan the *lists, maps* or *taxonomies* of modules and data structure descriptions, trying to identify the software appropriate for reuse. Evidently, those of the experts who cannot instantly associate appropriate concepts will also utilise mental maps and taxonomies while searching for the relevant reuse artefacts.

Facing a similar task, novices may have to start with preliminary specification or design of the problem and its solution, then having a better insight into the problem, search the manuals of the target programming language, its extensions, standard libraries, operating system utilities, or data dictionaries of previously developed software, leading to the likely candidates for reuse. In this process a clearly defined taxonomy of concepts will guide the developer into a given area of interest in an organised fashion.

- Both types of developers will use their innate ability to form complex *patterns* of required software features to match the artefacts considered for reuse.

The patterns may be represented as rules to remind a developer of certain categories of software, e.g. if the printing is mentioned then consider I/O libraries and report generators, if order is needed then look into ordered collections of data or sorting unordered data, if records are *mentioned then take into account databases and files, etc.*

- With experience, reusers can fall back on *context* and *reasoning* to promote or reject likely reuse candidates.

  In the context of application requirements, e.g. the need to maintain the collection of customer records, handling of on-line queries, producing periodical and ad-hoc reports, it may be much more apparent that a database or an index-sequential filing system may be a more appropriate representation for the data collection than an unordered file of records to be sorted for the purpose of report generation.

3.7 Retrieval

The search of a reuse repository culminating in the superficial assessment of several reusable artefacts and their features is usually insufficient to set a developer's mind as to their applicability to the problem at hand. The developer may then have to fetch or *recall* the details of the best candidate artefacts and to make a full assessment of their suitability. [cf. 36, pp 141-152, 5, ch 8, 40, ch 9].

- 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. Frequently the contextual or changing criteria modify the strength of association and thus the outcome of recognition/selection.

  E.g. when a software designer is presented with a number of alternative candidate artefacts (say selection-sort, bubble-sort and quick-sort), he or she is capable to immediately recognise their usefulness and to select the best of the available alternative (e.g. quick-sort is the most efficient).

- 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.

  Details of simple artefacts can be fully recalled on demand, e.g. the names of functions, modules or methods. Short algorithms (e.g. selection sort) and unsophisticated data structures (e.g. a list or a tree) can also be recollected by the more experienced developers.

- Remembering larger units of information usually involves recollecting partial memories, using them to *reconstruct* the originally learnt facts, and subsequently recognising re-generated information

  It may be very hard to remember the exact details of a complex software artefact, e.g. quick-sort algorithm. A skilled programmer, however, will reproduce the algorithm in any programming language he or she is as valid. In memory reconstruction, people may infer and recall information that wasn't actually studied, this on occasions may result in *fabrications*.

familiar with (e.g. C on Unix), even though it was originally written, studied and learnt in a different language and development environment (e.g. Pascal in Microsoft Windows).

- 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.

Algorithm recollection may depend on the overall application context, e.g. in high quality programs, when you need speed use quick-sort, when you are short of memory use heap-sort, but when you have no time to worry about sorting or high quality is not required, use the quick-and-dirty bubble-sort.

- Finally, the knowledge of your own memory mechanisms, known as *metamemory*, may provide additional retrieval cues.

E.g. when I fail to remember an artefact, I scan the letters of the alphabet for the first letter of its name, when I want to recall a process I imagine going through the shopping mall for cues, when I recall the program architecture I know that it frequently mirrors the data structure it manipulates, so I may try to remember it first.

3.8 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 issues of choice behaviour are discussed at length in 73, 40 pp 564-582, 5 pp 136-151].

- 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. Decision makers are thus assumed to be:
  - ◊ well informed,
  - ◊ infinitely sensitive to alternatives,
  - ◊ rational

  in the sense that they can rank

E.g. consider choosing between two algorithms (e.g. quick-sort and heap-sort). We could create a choice model by listing their comparable dimensions (e.g. speed, memory-use, ease of implementation, program availability) and then ranking them in each dimension (out of -5 to +5):

Sort	*Speed*	*Mem*	*Ease*	*Avail*
Quick	+5	-1	-2	+5
Heap	+3	+5	-3	+3

The selection could then be made based alternatives so as to *maximise the utility* of the selected option.

on the overall score (heap-sort is a winner 8:7) or the number of winning attributes (quick-sort is a winner 3:1).

- 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.

  In this model it is important to take into account the probability of events under consideration. To arrive at the value of expected utility of the selection, utilities of all the available options must be weighted by the probability of their occurrence.

  *E.g. consider the selection of a sorting algorithm for the real-time application. We know that on average quick-sort betters heap-sort by taking 2N ln N comparisons vs 2N lg N. However, in real-time applications we will also need to analyse the worst-case performance, which for quick-sort could degrade to as low as N*N, whereas it is constant for heap-sort. Considering the probability of the real-time application failing due to the extremely slow sorting of quick-sort, we may decide to use a slower but more predictable algorithm , i.e. heap-sort.*

- 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 alternatives may actually be missed.

  In this approach the decision maker sets the cutoff for the value of an alternative in each dimension, then considers all alternatives one at the time in the order in which they occur, while rejecting all alternatives which do not pass the set criteria until the first satisfactory alternative is found.

  *Consider a PC owner facing a once-off task of sorting 1000 records. He has access to the book with a number of sorting algorithms, i.e. bubble-sort, insertion-sort, quick-sort and heap-sort. He imposes a limit of 1.5 hours for the program to complete (length of a video movie). Assuming 0.02 sec per record comparison, sorting 1000 records with a N*N complexity algorithm will require less than 5.6 hours of computation on his PC. Bubble-sort will take half that time - rejected, insertion-sort a quarter - good enough. Better sorting algorithms are not even considered!*

- *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. The process is repeated for each consecutive dimension until we are either left with a single decision option, or the number of alternatives is sufficiently small to use another method of selection.

  E.g. assume that we need to select the best algorithm to sort sequential files. The artefacts to choose from include quick-sort, heap-sort, merge-sort and poly-phase merge-sort. The dimensions to be considered, in order of importance, are type of record access, the number of external files available, CPU and memory efficiency, etc. As both quick-sort and heap-sort need random-access to records, they are rejected outright. Next we determine that the plain p-way merge-sort requires far greater number of external files than poly-phase merge-sort, thus it is eliminated. The reuser will be left with the clear choice before other *dimensions are even considered.*

- Another general mechanism that might underlie 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.

  E.g. assume that two artefacts (quick-sort and shell-sort) are classified in terms of two sets of attributes (quick-sort is fast when tuned but memory-hungry whereas shell-sort is slower but simple to implement). The compatibility principle predicts that reusers will consider the common dimension (speed) as more important than the unique dimensions (memory use and development time), because its implications for choice are more directly apparent.

It should be noted that all the models outlined above assume stable preferences of decision makers. What some of the studies show, however, that certain preferences are quite labile and may depend on the way the problem of choice have been posed, questions are phrased, and responses are elicited. Managing preferences in a clear and explicit fashion is as important part of decision making as the actual methods of choice selection.

3.9 Adaptation and Integration

Integration of the selected software artefacts into the required problem solution may necessitate 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 the 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. [Issues of reasoning are covered in 5 ch 9, 27, 21, 35].

- Every day problems are commonly solved by *induction*, i.e. by composition, restructuring, generalisation and proceduralisation of previously acquired knowledge. In the process people rely on various heuristics, analogy and cases. Certain heuristics, however, are known to be notorious leading to bias, distortion of facts and invalid conclusions, e.g. availability, anchoring-adjustment, or hindsight heuristics.

  Assume, we have a routine for sorting numbers and another for sorting strings. Since both numbers and strings have an order operator, we can draw a hypothesis that it is relatively easy for the existing code to be adapted and integrated into a program sorting any type of ordered data. By analogy, we could also extend this kind of thinking into use of a data key, composite or computed keys, sorting lines of text or files of records, etc. Clearly none of these analogues could be used in the calculation of the greatest common divisor for strings!

- Similarly to induction, *abduction* uses experience to yield new knowledge

  Assume, we observed a reuser removing code redundancies (disease) by adapting from observations. However, while inductive reasoning results in the generalisation of observations, then, abduction results in identifying possible explanations of observable facts. Abduction is routinely used by medical practitioners to identify the causes of a illness, given only its symptoms.

a fragment of a program to write a new parametric function (treatment), and subsequently replacing several instances of repeated code (symptoms) with calls to the newly created function. When in future we witness a developer to replicate and alter a sequence of program statements, we may abduct the need for the new functional abstraction which could replace redundant code.

- The most formal model of reasoning is *deduction*, i.e. proving conclusions from given evidence, by formulating goals and plans and closely following various inference rules. As deduction preserves truth, so it strengthens confidence in the derived hypotheses. However, as the premises may be false, thus, the conclusions may also not be valid either. Although, deductive reasoning seems the only sound methods of scientific reasoning, there is ample evidence [36] to suggest that in abstract problem solving tasks, demands are so great that logic is short-circuited and simplified. On familiar tasks, subjects just remember just remember the correct answer, rather than arrive at it by reasoning.

A deductive model of reasoning in software reuse may include a number of reuse rules which when followed will suggest improvement of code. E.g.

if code is literally duplicated then
suggest the use of macro

if code is duplicated and changed then
suggest the use of a function

if code structure is duplicated then
suggest the use of code templates

if function is very long then
suggest splitting it into sub-functions

if data relates several functions then
suggest the use of abstract data types

if code is derived hierarchically then
suggest the use of classes and objects

4. Technological Assistance to the Reuse Process

Is technology the saviour of the reuse effort?

As it has already been established earlier, some of the steps taken in the cycle of software reuse will greatly benefit from a developer’s direct involvement in a given reuse task, e.g. in understanding, generalisation, selection and adaption of software artefacts. Others will, unfortunately, suffer severely from the human element and will, thus, require services of some automatic procedures assisting a developer, e.g. in identification, classification, search, storage and retrieval of artefacts. The technological bridge is, hence, required to offer the assistance to the developer and be assisted by the developer whenever it is required in the process of software reuse. Developer’s cognitive tasks involved in the reuse process can be supported by such technological means in several different ways [41], i.e. by :-

- *CASE tools* providing tools managing artefacts creation and maintenance;

- *object-oriented methods* structuring artefacts into an elaborate taxonomy of reusable concepts and ready-to-use artefacts;
- *formal methods* imposing strict artefact description formalism to reduce the amount of human intervention needed in the subsequent artefact manipulation;
- *knowledge-based systems* capturing human skills, experience and knowledge into a reuse-specialist expert system.

Let's briefly review the benefits of taking any one of the possible four paths to enhancing human cognition in the reuse process.

4.1 Enhancing reuse with CASE

A growing number of modern CASE [Computer-Aided Software Engineering - 67, 66] environments go beyond the simple tasks of software documentation and book-keeping, and provide the facilities which significantly enhance the reuse tasks by reducing or completely eliminating human intervention in some of the steps of the software development process, e.g.

- artefact creation with *tool support* facilitates computer-based representation of various artefacts, thus, ensuring that the developed artefacts are correct, consistent and complete (e.g. design diagrams, database schemata and queries, report or screen layouts);
- *software repository* expands developer's memory by providing means of describing, storing, searching and retrieving various forms of software artefacts, whether textual (requirements, specifications or programs), graphical (user interfaces or diagrams) or binary (bitmaps, data or programs);
- direct support of *domain analysis* further extends repository facilities by making reuse artefacts applicable beyond the boundaries of a single problem area;
- *reverse engineering* of existing code into diagrammatical representation allows developers to extract the conceptual design of unstructured or poorly structured artefacts (such as program code or database schema) and to visualise such designs to better understand the reusable features of legacy software;
- *code generation* eliminates the need for code derivation from its design, thus allowing developers to more effectively reuse early life-cycle artefacts;
- *requirements* and *design tracking* of software artefacts permits developers to track, record, and subsequently re-trace and comprehend the features of low-level artefacts (such as programs or user-interfaces) in terms of their design and specification;
- some of the research environments also aim at directly assisting developers in the *selection of reuse artefacts* during design tasks, by means of hyper-navigation [as in HyperCASE 16] or intelligent assistants [e.g. ASPIS 61].

Some of the more recent CASE tools and environments make reuse the crux of the entire development process, e.g. SEA/1, IPSEN or GTE environment [68].

4.2 Enhancing reuse with object-oriented systems

Object-oriented analysis, design and programming are claimed to drastically improve software reuse [51]. Such an improvement is possible by forcing developers to perceive, describe and solve problems in a highly structured and modular form, so that when the systems are constructed in adherence to the planned solution, their components are readily available for the subsequent reuse. Specially designed object-oriented programming languages, design methods and tools [24] further support this approach by providing the following features :-

- through the concept of a *class*, object-oriented languages blur the distinction between design and implementation of programming concepts, this allows better understanding of artefacts prior to their reuse and facilitates easier maintenance of artefact libraries;
- classes can be readily used as a vehicle for the implementation of abstract data types, thus, allowing *encapsulation* of related data and routines into highly cohesive reusable artefacts;
- classes can be easily adapted to suit new software solutions by means of *inheritance*, which allows preserving parts of the exiting class functionality and to easily add new or alter the existing functionality in a consistent fashion;
- *multiple inheritance* facilitates effortless fusion of reusable features belonging to several existing classes into an entirely new class;
- development-for-reuse is further enhanced by allowing to plan some of the class functionality but to *defer* the implementation of their details to the descendant classes;
- artefacts can be simply replicated by *instantiating* (or deriving) them from existing classes, such class instances are usually referred to as objects;
- objects can communicate with each other by message passing, this promotes *information hiding* as well as objects' *loose coupling*, both of which are regarded as beneficial for reuse.

Classes, objects, encapsulation, message passing, inheritance and instantiation have the pivotal role in the successful reuse of object-oriented systems components. The best examples of such systems are graphical user interface toolkits (e.g. X/Motif or Microsoft Foundation Classes), database toolkits [e.g. ObjectStore, Ontos and Versant - 24] and the more recent CASE repositories [e.g. PCTE+ - 25].

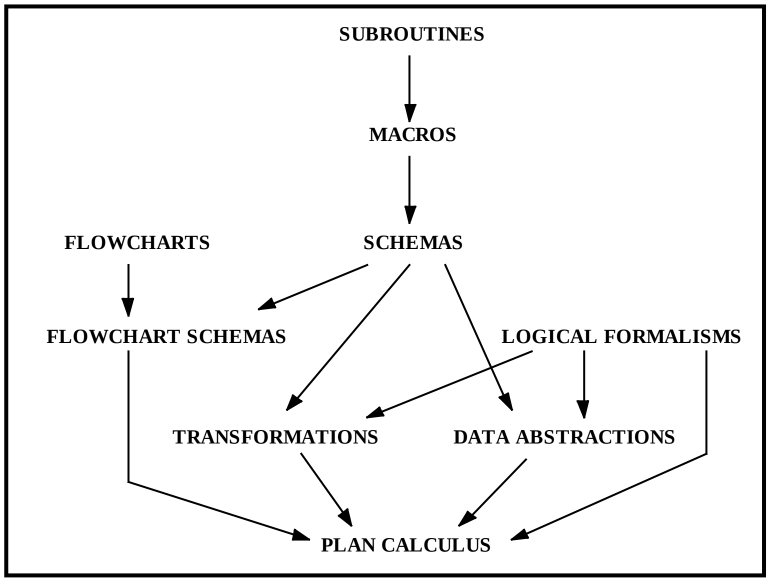

Figure 4-4 Approaches to formalisation in reuse [64]

4.3 Enhancing reuse with formal methods

A number of recent developments in software reuse, such as ISAR and a π-language [13], dispose of any informal, thus user-oriented, artefact notations and problem solving methods, thus instituting the application of completely formal, rigid and methodical approaches to software construction. Goguen and Moriconi [23] suggest that formalisation of software artefacts, in general, contributes to the development of better software development tools, reusable tool generators, semantic extensions of existing tools, and even entirely new tools which reduce human involvement in the development process. They claim that formalisation also helps eliminating misunderstanding between tool users and tool developers, and it diminishes commonly encountered differences between implementations of the same tool.

Tracz [82, pp A73-74] reports from the discussion of the Formal Methods for Reuse working group at IWSR2 workshop that formal specifications bring a number of distinct benefits to software reuse, i.e. they :-

- help *capturing design*,
- assist *retrieval of components*,
- help *evaluating the suitability of a component*,
- aid in *improved maintenance*,
- aid *increase acceptance* of reusable artefacts,
- help in *avoiding misuse* of reusable components,
- serve as valuable *documentation*,
- aid in *reuse education*, and
- provide a foundation for certain *module composition languages*.

In view of these benefits, some of the new approaches to reuse tend to abandon the less formal artefact descriptions such as subroutines, macros, schemata or flowcharts to give way to the increasingly formal notations, such as mathematical logic, flowchart schemata, transformations, data abstractions or the plan calculus (cf. Figure 4-4). Rich and Waters [64], however, do not restrict the formalism for encoding reuse information to any particular method, instead they proclaim that in order to gain the anticipated benefits of formal approaches to reuse, the notation employed must conform to the following five desiderata :-

- *expressiveness* - the formalism must be capable of expressing many different kinds of components;
- *convenient combinability* - the method of combining components must be easy to implement and the properties of combinations should be evident from the properties of their parts;
- *semantic soundness* - the formalism must be based on a mathematical foundation that allows correctness conditions to be stated for the library of components;
- *machine manipulability* - it must be possible to manipulate the formalism effectively using computer tools;
- *programming language independence* - the formalism should not be dependent on the syntax of any particular programming language.

Sharing similar views on the introduction of formal approaches to reuse, Biggerstaff and Richter [8] eliminate natural language and most of the block and connector styles of graphical representations of software artefacts, instead they are looking for the notations having the following representational properties, they must have ability to :-

- represent knowledge about implementation in *factored form* which would allow to separate different design concerns, e.g. data from algorithms;
- create *partial specifications* of design information that can be incrementally extended - they argue that you cannot specify the details too precisely or else you reduce the reuse potential, at the same time too much fuzziness will impede implementation, so a mixture of the two needs to be controlled in an intelligent way;
- allow *flexible couplings* between instances of designs and the various interpretations they can have by means of viewpoint management and flexible naming conventions; and
- express *controlled degrees* of abstraction and precision (i.e. degree of ambiguity).

The formalism that could potentially embrace the four requirements is the *semantic binding* of components, i.e. inter-object references must be based on the component meaning rather than its name or its syntactic structure, as done in the majority of currently used reuse assistants. Few of the existing reuse systems seems to be able to accommodate this requirement (with the exception of some of the analogical reasoning systems).

4.4 Enhancing reuse with knowledge-based systems

In the extremal view of some of the artificial intelligence researchers, human agents are only needed to provide the essence of their skills, knowledge and expertise. However, once the critical human abilities are captured and codified in the computer-based expert system, then, they could be efficiently applied to the problem at hand, without the human interference known to be detrimental to the whole process. Several attempts have been made to take such a knowledge-based approach to software development and software reuse in particular, e.g. supporting reuse via knowledge-based domain modeling [28, 42], reuse of software requirements [30], sharing and reuse of knowledge [54], reusing description of purpose in design [19], construction of large-scale reusable knowledge bases [38, 39], etc.

Knowledge-based reuse systems (KBRS) have a number of features which will, in the long term, eliminate the need for human involvement in the reuse cycle, and will, in the interim, provide assistance to developers without specific training and skills in software reuse, e.g. they may :-

- *capture reuser's skills and experience* characterising reuse domains [50, 34, 6, 53, 57, 60, 70];
- *represent knowledge* of reusable artefacts and their uses in elaborate knowledge structures, such as rules and semantic networks [72], frames [17, 2] or schemata [43];
- apply different inference methods to *reason about software artefacts* to discover their characteristics and reuse potential, e.g. proving correctness of reuse components [11] or using fuzzy logic to find relevant artefacts [59, 45];
- use machine learning techniques to *find, generate or adapt reusable artefacts* from previously seen examples of reuse, e.g. by deduction from formal specifications [74], by subsumption to match, replace and adapt reusable artefacts [29], by analogy [47, 46, 37] and case-based reasoning [45]; or
- *enrich knowledge about software artefacts* by interpreting artefact text documents, e.g. for the purpose of their classification and retrieval [44], querying reuse libraries [10] or better representation of reusable artefacts [7].

Although knowledge-based technology is still in its infancy, some of the approaches to intelligent management of reuse information offer promising results. The human reuser still cannot be fully replaced by the automatic system, however, various activities performed in the reuse cycle may be assisted by the computer-based processing.

5. Summary

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. Such a reuse tool should exhibit the following attributes:

- it should automatically identify and manipulate artefact features, their patterns and groups;

Reuse Task	**Cognitive Abilities**	**Assisting Methods and Techniques**
Identification	*Features*	Typographic analysis
		Lexical analysis
	Patterns	Syntactic analysis
		Semantic analysis
		Statistical analysis
	Chunks & Gestalt	Text clustering
	Attention	Information highlighting and hiding
		Iterative refinement and elaboration
		Hypertext linking
Understanding	*Associations*	Associative networks
		Neural networks
	Propositions	Predicate logic
		Propositional networks
	Schemata	Frames
		Conceptual graphs
Classification	*Similarity*	Faceted classification
	Typicality	Taxonomic classification
	Reasoning	Case-based classification
		Analogical classification
Search	*Activation*	Menus
		Hypertext navigation
		Query and natural language systems
		Deductive databases
		Information retrieval
Storage	*Memorising*	Full text databases &
Retrieval	*Remembering*	Knowledge based systems
Generalisation	*Induction*	Parametrisation
		Classes, schemas and clichés
		Templates and outlines
Adaptation	*Induction /*	Analogical reasoning
	Deduction	Case-based reasoning
		Transformations
Integration	*Deduction*	Software bus
		Interconnection systems

Table 5-2 Technological assistance in software reuse

- 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.

6. 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. 1988, Computer Society Press: Washington, D.C. p. 33-40.
2. Allen, B.P. and S.D. Lee. "A knowledge-based environment for the development of software parts composition systems". in *11th International Conference on Software Engineering*. 1989. Pittsburgh, Pennsylvania: IEEE Computer Society Press, p. 104-112.
3. Allen, J., *Natural Language Understanding*. 1987, Menlo Park, CA: The Benjamin/Cummings Pub. Co., Inc.
4. Anderson, J.R., *Cognitive Psychology and Its Implications*. Second Edition ed. 1985, New York: W.H. Freeman and Co.
5. Anderson, J.R., *Learning and Memory: An Integrated Approach*. 1995, New York: John Wiley & Sons, Inc.
6. Arango, G., "Domain analysis methods", in *Software Reusability*, W. Schafer, R. Prieto-Diaz, and M. Matsumoto, Editors. 1994, Ellis Horwood: London, Great Britain. p. 17-49.
7. Biebow, B. and S. Szulman. "Enrichment of semantic network for requirements expressed in natural language". in *Information Processing'89*. 1989. San Francisco, California: North-Holland, p. 693-698.
8. Biggerstaff, T.J. and C. Richter, "Reusability framework, assessment, and directions", in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. 1989, ACM Addison Wesley Publishing Company: New York, New York. p. 1-18.
9. Bobrow, D.G. and D.A. Norman, "Some principles of memory schemata", in *Representation and Understanding*, D.G. Bobrow and A. Collins, Editors. 1975, Academic Press, Inc.: New York, NY. p. 131-149.
10. Burton, B.A., *et al.*, "The reusable software library". *IEEE Software,* 1987. **4**(4): p. 25-33.
11. Caplan, J.E. and M.T. Harandi, "A logical framework for software proof reuse". *Software Engineering Notes*, 1995 (August): p. 106-113.
12. Cheng, J., "Reusability-based software development environment". *ACM SIGSOFT Software Engineering Notes*, 1994. **19**(2): p. 57-62.

13. Cramer, J., E.-E. Doberkat, and M. Goedicke, "Formal methods", in *Software Reusability*, W.S.R. Prieto-Diaz and M. Matsumoto, Editors. 1994, Ellis Horwood: London, Great Britain. p. 79-111.

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

15. Cybulski, J.L. "Reuse in the eye of its beholder: cognitive factors in software reuse". in *OzCHI'96*. 1996. Hamilton, New Zealand: IEEE Press, p. 228-235.

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

17. Devanbu, P., *et al.*, "LaSSIE: A knowledge-based software information System". *Communications of ACM*, 1991. **34**(5): p. 34-49.

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

19. Franke, D.W., "Deriving and using descriptions of purpose". *IEEE Expert*, 1991. **6**(2): p. 41-47.

20. Gazdar, G. and C. Mellish, *Natural Language Processing in PROLOG*. 1989, Wokingham, England: Addison-Wesley Pub. Co.

21. Gentner, D., "The mechanisms of analogical learning", in *Similarity and Analogical Reasoning*, S. Vosniadou and A. Ortony, Editors. 1989, Cambridge University Press: Cambridge, U.K.

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

23. Goguen, J.A. and M. Moriconi, "Formalization in programming environments". *IEEE Computer*, 1987. **20**(11): p. 55-64.

24. Graham, I., *Object Oriented Methods*. 1994, Wokingham, England: Addison-Wesley Pub. Co.

25. Group, I.E.P. *Introducing PCTE+*. 1989. IEPG TA-13, GIE Emeraude.

26. Hobbs, J. and L. Rau. "Tutorial T13: Text Interpretation". in *International Joint Conference on Artificial Intelligence*. 1991. Sydney, Australia.

27. Holyoak, K.J., "Problem solving", in *An Invitation to Cognitive Science: Thinking*, D.N. Osherson and E.E. Smith, Editors. 1990, The MIT Press: Cambridge, Massachusetts. p. 117-146.

28. Iscoe, N., "Domain-specific reuse: an object-oriented and knowledge-based approach", in *Software Reuse: Emerging Technology*, W. Tracz, Editor. 1988, IEEE Computer Society Press. p. 299-308.

29. Jeng, J.-J. and B.H.C. Cheng. "A formal approach to reusing more general components". in *KBSE'94, The Ninth Knowledge-Based Software Engineering Conference*. 1994. Monterey, California: IEEE Computer Society Press, p. 90-97.

30. Johnson, W.L. and D.R. Harris. "Sharing and reuse of requirements knowledge". in *6th Annual Knowledge-Based Software Engineering Conference*. 1991. Syracuse, New York, USA: IEEE Computer Society Press, p. 57-66.

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

32. 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. 1989, ACM Addison Wesley Publishing Company: New York, New York. p. 257-274.

33. Kay, M. *Algorithm schemata and data structures in syntactic processing*. 1980. CSL-80-12, Xerox Palo Alto Research Center.

34. Klinker, G., S. Benetet, and J. McDermott, "Knowledge acquisition for evaluation systems". *International Journal of Man-Machine Studies*, 1988. **29**(6): p. 715-732.

35. Kolodner, J.L., "Improving human decision making through case-based decision aiding". *AI Magazine*, 1991. **12**(2): p. 52-68.

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

37. Lee, H.-Y. and M.T. Harandi. "An analogy-based retrieval mechanism for software design reuse". in *KBSE'93, The Eighth Knowledge-Based Software Engineering Conference*. 1993. Chicago, Illinois: IEEE Computer Society Press, p. 152-159.

38. Lenat, D., M. Prakash, and M. Shepherd, "CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks". *The AI Magazine*, 1986. **6**(4): p. 65-85.

39. Lenat, D.B., *et al.*, "CYC: Toward programs with common sense". *Communications of the ACM*, 1990. **33**(8): p. 30-49.

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

41. Lowry, M.R., "Software engineering in the twenty-first century", in *Automatic Software Design*, M.R. Lowry and R.D. McCartney, Editors. 1991, AAAI Press / The MIT Press: Menlo Park, California. p. 628-654.

42. Lubars, M.D., "Domain analysis and domain engineering in IDeA", in *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Editors. 1991, IEEE Computer Society Press: Los Alamitos, California. p. 163-178.

43. Lubars, M.D. and M.T. Harandi, "Addressing software reuse through knowledge-based design", in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. 1989, ACM Addison Wesley Publishing Company: New York, New York. p. 345-377.

44. Maarek, Y.S., D.M. Berry, and G.E. Kaiser, "An information retrieval approach for automatically constructing software libraries". *IEEE Transactions on Software Engineering*, 1991. **17**(8): p. 800-813.

45. MacKellar, B.K. and F. Maryanski. "Knowledge base for code reuse by similarity". in *Proc. COMPSAC 89*. 1989. Orlando, FL, USA: IEEE, IEEE Service Center, Piscataway, NJ, USA, p. 634-641.

46. Maiden, N. and A. Sutcliffe. "Analogical matching for specification reuse". in *6th Annual Knowledge-Based Software Engineering Conference*. 1991. Syracuse, New York, USA: IEEE Computer Society Press, p. 108-116.

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

48. 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*, P.-D. Ruben and B.F. William, Editors. 1993, IEEE Computer Society Press: Los Alamitos, California. p. 176-185.

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

50. McGraw, K.L. and K. Harbison-Briggs, *Knowledge Acquisition: Principles and Guidelines*. 1989, London, UK: Prentice-Hall International.

51. Meyer, B., "Reusability: the case for object-oriented design". *IEEE Software*, 1987 (March): p. 50-64.

52. Minsky, M., "A framework for representing knowledge", in *The Psychology of Computer Vision*, P. Winston, Editor. 1975, McGraw-Hill: New York. p. 211-280.

53. Moore, J.M., "Domain analysis: framework for reuse", in *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Editors. 1991, IEEE Computer Society Press: Los Alamitos, California. p. 179-203.

54. Neches, R., *et al.*, "Enabling technology for knowledge sharing". *AI Magazine*, 1991. **12**(3): p. 36-56.

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

56. Paivio, A. and I. Begg, *Psychology of Language*. 1981, Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

57. Patel, J. "On the road to automatic knowledge engineering". in *International Joint Conference on Artificial Intelligence*. 1989. Detroit, Michigan, p. 628-632.

58. Potter, M.C., "Remembering", in *An Invitation to Cognitive Science: Thinking*, D.N. Osherson and E.E. Smith, Editors. 1990, The MIT Press: Cambridge, Massachusetts. p. 3-32.

59. Prieto-Diaz, R., "Classification of reusable modules", in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. 1989, Addison-Wesley Pub. Co.: New York, NY. p. 99-123.

60. Prieto-Diaz, R., "Domain analysis: an introduction". *ACM Sigsoft Software Engineering Notes*, 1990. **15**(2): p. 47-54.

61. Puncello, P.P., *et al.*, "ASPIS: a knowledge-based CASE environment". *IEEE Software*, 1988 (March): p. 58-65.

62. Quillian, M.R., "Semantic memory", in *Semantic Information Processing*, M. Minsky, Editor. 1968, MIT Press: Cambridge, MA. p. 227-270.

63. Reichman, R., *Getting Computers to Talk Like You and Me*. 1985, Cambridge, Massachusetts: The MIT Press.

64. Rich, C. and R.C. Waters, "Formalising reusable software components in the Programmer's Apprentice", in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. 1989, ACM Addison Wesley Publishing Company: New York, New York. p. 313-343.

65. Riesbeck, C.K., "Expectation-driven parsing", in *Encyclopedia of Artificial Intelligence*, S.C. Shapiro, Editor. 1987, John Wiley & Sons: New York. p. 696-701.

66. Rock-Evans, R., *CASE Analyst Workbenches: A Detailed Product Evaluation*. 1989, London, England: Ovum Ltd.

67. Rock-Evans, R. and B. Engelien, *Analysis Techniques for CASE: A Detailed Evaluation*. 1989, London, England: Ovum Ltd.

68. Rombach, H.D. and W. Schafer, "Tools and environments", in *Software Reusability*, W. Schafer, R. Prieto-Diaz, and M. Matsumoto, Editors. 1994, Ellis Horwood: London, Great Britain. p. 113-152.

69. Schank, R.C., "Interestigness: controlling inferences". *Artificial Intelligence*, 1979. **12**: p. 273-297.

70. Schoen, E. "Active assistance for domain modeling". in *6th Annual Knowledge-Based Software Engineering Conference*. 1991. Syracuse, New York: IEEE Computer Society Press, p. 26-35.

71. Shneiderman, B. and R. Mayer, "Syntactic/semantic interactions in programmer behaviour: a model and experimental results". *International Journal of Computer and Information Sciences*, 1979. **8**(3): p. 219-238.

72. Simos, M.A., "The growing of organon: a hybrid knowledge-based technology and methodology for software reuse", in *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Editors. 1991, IEEE Computer Society Press: Los Alamitos, California. p. 204-221.

73. Slovic, P., "Choice", in *An Invitation to Cognitive Science: Thinking*, D.N. Osherson and E.E. Smith, Editors. 1990, The MIT Press: Cambridge, Massachusetts. p. 89-116.

74. Smith, D.R., "KIDS - A knowledge-based software development system", in *Automatic Software Design*, M.R. Lowry and R.D. McCartney, Editors. 1991, AAAI Press / The MIT Press: Menlo Park, California. p. 483-514.

75. Smith, E.E., "Categorization", in *An Invitation to Cognitive Science: Thinking*, O.D. N. and S.E. E., Editors. 1990, The MIT Press: Cambridge, Massachusetts. p. 33-53.

76. Smith, G.W., *Computers and Human Language*. 1991, New York: Oxford University Press.

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

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

79. Sowa, J.F., *Conceptual Structures: Information Processing in Mind and Machine*. 1984, Readings, Massachusetts: Addison-Wesley Pub. Co.

80. Sternberg, R.J., "Reasoning, problem solving, and intelligence", in *Handbook of Human Intelligence*, R.J. Sternberg, Editor. 1982, Cambridge University Press: Cambridge, U.K. p. 225-307.

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

82. Tracz, W., "Second International Workshop on Software Reusability - IWSR2: Summary". *ACM SIGSOFT, Software Engineering Notes*, 1993. **18**(3): p. A73-77.

83. van Dijk, T.A., "Semantic macro-structures and knowledge frames in discourse comprehension", in *Cognitive Processes in Comprehension*, A.M. Just and P.A. Carpenter, Editors. 1977, Lawrence-Elrbaum Assoc., Pub.: Hillsdale, New Jersey. p. 3-32.

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

85. 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. 1989, ACM Addison Wesley Publishing Company: New York, New York. p. 247-256.

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

87. Wilks, Y., "An inteligent analyzer and understander of English". *Communications of the ACM*, 1975. **18**(5): p. 264-274.

88. Wilks, Y. and D. Fass, "The preference semantics family". *Computers Math. Applic.*, 1992. **23**: p. 205-221.