Introduction to Software Reuse

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/

Technical Report TR 96/4

Abstract

Effective reuse of software products is reportedly increasing productivity, saving time, and reducing cost of software development. Historically, software reuse focused on repackaging and reapplying of code modules, data structures or entire applications in the new software projects (Prieto-Diaz 1994). Recently, however, it has been acknowledged as beneficial to redeploy software components across the entire development life-cycle, starting with domain modelling and requirements specification, through software design, coding and testing, to maintenance and operation. There were also attempts to reuse aspects of project organisation and methodology, development processes, and communication structures. However, as the concept of reusing software components is very clear at the code level (whether in source or binary form), the very same concept becomes more fuzzy and difficult to grasp when discussed in the context of reusing specifications and designs (whether in textual or diagrammatical form), or quite incomprehensible when applied to software informal requirements, domain knowledge or human skills and expertise (expressed in natural language, knowledge representation formalism, or existing only in humans). This problem of dealing with reusable software artefacts resulting from the earliest stages of software development, in particular requirements specifications, attracted our particular interest in the reusability technology.

Our work is motivated primarily by the possibility of improving the process of requirements elicitation by methodical reuse of software specifications and their components with the aid of information extracted from user informal requirements documents. The problems and issues that we aim to investigate in this research are best illustrated by the following statement outlining current needs and the goals for the future research in requirements reuse:

- "More research is needed on the advantages and the necessary methods for requirements reuse. For example, what are requirements 'components', what makes them reusable, how can we store and retrieve them, and how do we write a requirements specification that gives us the highest probability of creating or reusing existing requirements components?" (Hsia, Davis et al. 1993).

1. Definitions

To address the issues advanced by Hsia, Davis and Kung, and to avoid any confusion farther in this paper, we need to clearly define some major concepts of software reuse, reusability, reusable artefacts, their possible forms, reusability methods, their major motivators and inhibitors, etc. Hence, we adopt our definitions from Prieto-Diaz (Prieto-Diaz 1989) as follows :-

- *reuse* is the use of previously acquired concepts or objects in a new situation, it involves encoding development information at different levels of abstraction, storing this representation for future reference, matching of new and old situations, duplication of already developed objects and actions, and their adaptation to suit new requirements;
- *reusability* is a measure of the ease with which one can use those previous concepts or objects in the new situations.

2. Reuse Artefacts

The object of reusability, *reusable artefact*, can be any information which a developer may need in the process of creating software (Freeman 1983), this includes any of the following software components :-

- *code fragments*, which come in a form of source code, PDL, or various charts;
- *logical program structures*, such as modules, interfaces, or data structures;
- *functional structures*, e.g. specifications of functions and their collections;
- *domain knowledge*, i.e. scientific laws, models of knowledge domains;
- *knowledge of development process*, in a form of life-cycle models;
- *environment-level information*, e.g. experiential data or users feedback;
- *artefact transformation* during development process (Basili 1990); etc.

A controlled collection of reuse artefacts constitutes a *reuse library*. Such libraries must contain not only reusable components but are also expected to provide certain types of services to their users (Wegner 1989), e.g. storage, searching, inspecting and retrieval of artefacts from different application domains, and of varying granularity and abstraction, loading, linking and invoking of stored artefacts, specifying artefact relationships, etc. The major problems in the utilisation of such reuse libraries are in determining appropriate artefact classification schemes and in the selection of methods to effectively and efficiently search the library. To bypass the problems with reuse libraries, the use of specialised *domain-specific languages* was proposed as an alternative. Such languages use strict syntax and semantics defined in terms of an application domain and its reusable artefacts. While enforcing notational conformance with a predetermined syntax and semantics, the domain-specific languages restrict the number of possible

classification and search mechanisms used in the process of composing a problem solution, e.g. as in DRACO (Neighbors 1989) or GIST (Feather 1989).

3. Artefact Characteristics

Certain classes of software artefacts have been identified as eminently suitable to become part of a reuse library and be, subsequently, utilised as reusable software resources. Such artefacts usually share a number of characteristics, deemed to actively promote reusability (Biggerstaff and Richter 1989; Matsumoto 1989; McClure 1989), those artefact are perceived to be :-

- *expressive*, i.e. they are of general utility and of adequate level of abstraction, so that they could be used in many different contexts, and be applicable to variety of problem areas;
- *definite*, i.e. they are constructed and documented with a clarity of purpose, their capabilities and limitations are easily identifiable, interfaces, required resources, external dependencies and operational environments are specified, and all other requirements are explicit and well defined;
- *transferable*, i.e. it is possible to easily transfer an artefact to a different environment or problem domain, this usually means that it is self-contained, with few dependencies on implementation-related concepts, it is abstract and well parametrised;
- *additive*, i.e. it should be possible to seamlessly compose existing artefacts into new products or other reusable components, without the need for massive software modifications or causing adverse side effects;
- *formal*, reusable artefacts should, at least at some level of abstraction, be described using a formal or semi-formal notation, such an approach provides means to formally verify an artefact correctness, it enables to predict violation of integrity constraints during artefact composition, or to assess the level of completeness for a product constructed of reusable parts;
- *machine representable*, those of the artefacts which can be described in terms of computationally determined attribute values, which can easily be decomposed into machine representable parts, which can be accessed, analysed, manipulated and possibly modified by computer-based processes, have a clear potential for becoming part of a flexible reuse library; those artefacts can be easily searched for, retrieved, interpreted, altered and finally integrated into larger system;
- *self-contained*, reusable artefacts which embody a single idea are easier to understand, they have less dependencies on external factors, whether environmental or implementational, they have interfaces which are simple to use, they are easier to extend, adapt and maintain;
- *language independent*, no implementation language details should be embedded in reusable artefacts, this also means that most useable artefacts are those which are

described in terms of a specification or design formalism, or those low level solutions which could be used from variety of programming languages on a given implementation platform, either by appropriate macro processors or link editors;

- *able to represent data and procedures*, i.e. reusable artefacts should be able to encapsulate both their data structures and logic, down to a fine grain of detail, such an approach increases artefact cohesion and reduces the possibility of artefact coupling by common data passed via arguments or global variables;
- *verifiable*, as any other software components, reusable artefacts should be easy to test by their maintainers, and, what is even of a greater importance, by their users who embed reusable components into their own systems, and who must have the capability to monitor the components computational context and their interfaces;
- *simple*, minimum and explicit artefact interfaces will encourage developers to use artefacts, simple and easy to understand artefacts can also be easily modified by developers to suit new applications; and
- *easily changeable*, certain type of problems will require artefacts to be adopted to the new specifications, such changes should be localised to the artefact and require minimum of side effects.

4. Reuse in Software Life-Cycle

Computer software can be systematically reused across the entire development life-cycle, i.e. domain analysis, requirements specification, design and implementation, it has its place even in the post-delivery stages of development, e.g. its continuing quality assessment or software maintenance.

Implementation. Early experience with software reuse was limited to reuse of program code in source and binary form. A great emphasis was put on development of programming languages which could support various methods of clustering, packaging, modularisation, parametrisation and sharing of data and code via data types and code blocks (ALGOL), named common blocks (FORTRAN), parametric functions and macros (FORTRAN and LISP), copy libraries (COBOL), information hiding (PASCAL), modules (SIMULA and MODULA), generic packages (ADA), objects and classes (SMALLTALK and C++), etc. The idea of code sharing was further supported by various operating system utilities which allowed independent program compilation, creation of relocatable libraries or link editing (Reed 1983). In those early days, no serious effort on a commercial scale was undertaken to reuse the early life-cycle artefacts, i.e. designs, specifications, requirements or enterprise models. This situation was caused by :-

- the lack of awareness of potential benefits that could be gained from reusing more abstract software artefacts;
- unavailability of commercial methodologies embracing software reuse at their centre-point;
- informal nature of early specification and design documents; and

- shortage of tools capable to represent specifications and designs in a computer-processable form.

At the same time,

- the construction of libraries was known to improve software development productivity, and was practiced in nearly every commercial organisation;
- program code was written according to a formal grammar and it adhered to established semantic rules; and
- the construction of code libraries was supported by editors, compilers, loaders and linkers, which could be freely customised to accommodate various reuse tasks.

Design. Today's development approaches, such as object-oriented methods (Graham 1994) or rapid application development (Martin 1991), vigorously advocate reusing software artefacts at the earliest possible stage of the software life-cycle. Program design methods are now capable of utilising well-defined diagrammatic notations, which allow production of documents which are simpler and more legible than code, which clearly exhibit their conceptual contents, which are well structured and modular, and which allow dealing with problem complexity at various levels of abstraction and granularity. With the advent of CASE tools (McClure 1989) the contemporary design techniques are also supported by specialised software environments capable of capturing design ideas in a form leaning towards further processing by computer-based reuse tools. Today, it is also commonly perceived that reuse of software designs, as opposed to code reuse, is more economic, and cognitively a much more intuitive process.

Requirements Specification. While application of reuse techniques to software design has visible advantages over code reuse, some researchers (Matsumoto 1989) claim further increases in the scope of software reusability when given opportunity to reuse modules at higher levels of abstraction, i.e. software specifications and requirements. Others support this claim, voicing the need to reuse large-scale artefacts going beyond design components and including entire design frameworks and domain resources (Li 1993). Bubenko et. al. (Bubenko, Rolland et al. 1994) further propose to combine design and reuse libraries to accommodate development processes capable of reusing conceptual schemas to support the process of requirements engineering. Such an approach provides users with the library of reusable components that could match their requirements, improves the quality of requirements specifications by making available well-defined conceptual components as early as requirements specification, and improves the productivity of the requirements engineering process by shortening the requirements formalisation effort (Castano and De Antonellis 1994). In the REBOOT system, Morel and Faget (Morel and Faget 1993) aim at extending this approach to the entire software life-cycle. Such advances in requirements and specification reuse were in part facilitated by :-

- Development of the new types of programming languages, such as PROLOG or EIFFEL, which combine elements of program specification and design (via logic and class specification) at the level of code, such an approach promotes interpretation and reuse of abstract program descriptions throughout the life-cycle;

- dissemination of prototyping tools and visual programming environments capable of graphic representation of user requirements and the subsequent generation of code or code skeletons (Vonk 1989; Ambler and Burnett 1990), facilitating effective composition of programs of domain-specific, visual, reuse components;
- introduction of formal requirements and specification languages, such as RML (Greenspan, Mylopoulos et al. 1994), Z (Spivey 1989), VDM (Woodman and Heal 1993) or LARCH (Guttag and Horning 1993), permitting representation, structuring, verification, and reuse of specification components;
- object-oriented technologies integrating various diagrammatic techniques into a single methodology, e.g. Information Engineering (Martin 1993), or unifying elements of conceptual modelling, program specification and design into one consistent notation, e.g. Object-Oriented Conceptual Modelling (Dillon and Tan 1993), such object-oriented development methods allow creation of abstract conceptual schemata which can be readily adapted by instantiation and inheritance to new problem solutions;
- development of full-text databases utilising efficient information retrieval methods (Salton 1989), being introduced as a repository for storing, classification and subsequent retrieval of design and specification texts (Frakes and Nejmeh 1988; Maarek, Berry et al. 1991; Fugini and Faustle 1993); and finally
- application of knowledge-based techniques and intelligent software development assistants in requirements acquisition and specification (Lowry and Duran 1989);

Domain Analysis. The final frontier for software reuse in the development life-cycle is a thorough analysis of a given problem domain. This approach is grounded on the belief that in a real-life situation reusability is not a universal property of program code or processed information but it rather depends on a context of the problem and its solution, which themselves are relatively cohesive and stable (Arango and Prieto-Diaz 1991). The main aim of domain analysis is the construction of a domain model of which components could be reused in solving variety of problems. Such a model will customarily include definition of concepts used in the specification of problems and software systems, definition of typical design decisions, alternatives, trade-offs and justifications, and software implementation plans. Such a model may take variety of different forms, to include (cf. Figure 1) :-

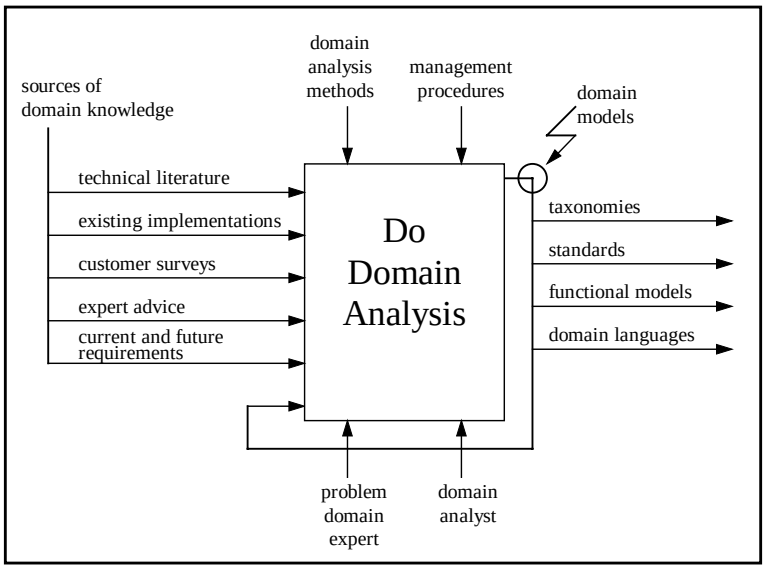

Figure 1 Domain analysis (Arango and Prieto-Diaz 1991)

- *definitional model*, which provides knowledge taxonomies and actonomies describing domain concepts, their structure, semantics, and relationships between them;
- *knowledge representation model*, giving domain semantics and explanation facilities;
- *domain-specific languages*, which when expressed as formal grammars and supported by parsers may provide direct translation of specifications into executable code;
- *instructional models*, indicating the methods of constructing working systems in a given domain, such methods may be described by standards, guidelines, templates, or interface definitions;
- *functional models*, describing how systems work, using representations such as data flow diagrams or program description languages;
- *structural models*, provide means to define architecture of domain systems; etc.

In the process of constructing a domain model, the common knowledge from related systems is generalised, objects and operations common to all systems in a given domain are identified, and a model is defined to describe their inter-relationships. The main problem with this process is that knowledge sources for domain modelling (as found in technical literature, existing implementations, customer surveys, expert advice or current and future requirements) are frequently verbose and informal. Thus, special techniques and tools are needed to deal with it, e.g. knowledge acquisition tools, entity-relationship modelling tools, object-oriented methods, semantic clustering tools, CASE and parsing tools (Agresti and McGarry 1988).

Reuse Process. In this work, we will view the process of software reuse as comprising three stages of artefact processing (cf. Figure 2), i.e. their analysis, organisation and synthesis.

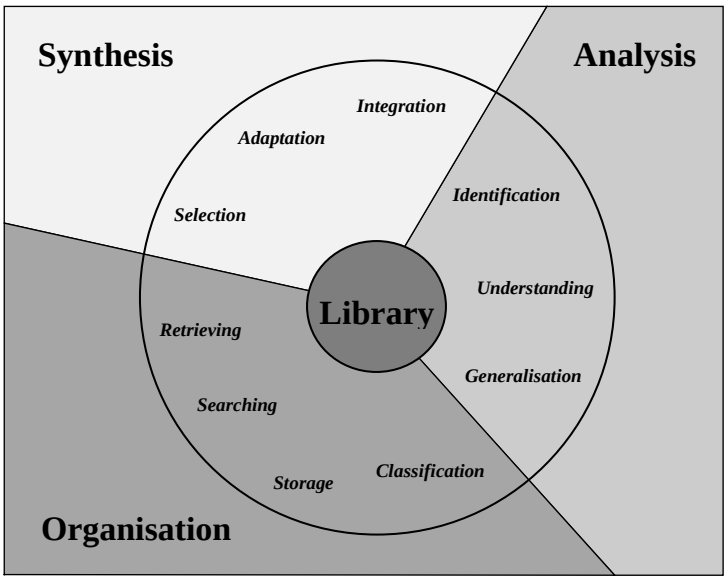

Figure 2 Reuse process and its artefacts

- ***Artefact analysis*** starts with *identification* of artefacts in existing software products (Ning, Engberts et al. 1994) or in a currently analysed domain (Arango and Prieto-Diaz 1991), this is followed by their *understanding* and representation in a suitable formalism to reflect their function and semantics, with possible *generalisation* to widen the scope of their future applications.
- ***Artefact organisation*** includes *classification* and *storage* of artefacts in an appropriate software repository, the subsequent repository *search* and artefacts *retrieval* whenever they are needed in the reuse process.
- ***Artefact synthesis*** consists of artefact *selection* from a number of retrieved candidate artefacts, their *adaptation* to suit the new application, and their *integration* into a completely new software product.

The tasks undertaken in the three stages of artefact processing are also frequently discussed from the perspective of development-for-reuse and development-by-reuse (Bubenko, Rolland et al. 1994).

- ***Development-for-reuse*** is emphasising the construction of the reuse library, involving the *identification*, *understanding*, *generalisation*, and the subsequent *classification* and *storage* of artefacts for later reuse.
- ***Development-by-reuse*** is concerned with the effective utilisation of the reuse library to support new software development, it involves *searching*, *retrieval*, *selection*, *adaptation*, and *integration* of artefacts into the software system under construction.

As reuse is quite independent of any particular development process model, it, thus, could be embedded into a variety of methodologies, to include waterfall model (Hall and Boldyreff 1991), rapid prototyping (Martin 1991), object-oriented design (Meyer 1987), etc. While the inclusion of reuse into a development cycle is of a significant benefit to the entire process, at the same time it may complicate the development process (e.g. see

Figure 3). Also, reuse tasks may significantly overlap with those performed in other development phases, e.g. software integration or maintenance. The separation of concerns lead some researchers (Hall and Boldyreff 1991) into pointing out that reuse must occur across different projects or problem areas, as opposed to those tasks which aim at the change, improvement or refinement of software undertaken within a single project which should not be regarded as reuse, e.g.

- *software porting*, which only aims at adopting existing software product to different hardware or operating system environments;
- *software maintenance*, which strives to correct software erroneous behaviour or to alter the existing program to suit changing requirements; and
- *software reconfiguration*, which provides a method of customising software to be used with different hardware components or making only a subset of its facilities available to the user.

5. Assessing the Reuse Process and its Goals

The value of software reuse cannot be gauged in simple, unambiguous, congruous and canonical fashion. One of the reasons for this difficulty lies in the fact that there is a variety of reusable artefact types and the methods and techniques for their creation, manipulation and maintenance. Another reason can be set in inadequacy of measuring tools to assess the reuse benefit or its hindrance, as it can be measured using variety of incompatible metrics, some of which are based on economic, some on technical, then again others on social or cognitive principles. Finally, it is the numerous software stakeholders who are not likely to agree on the common goals of the reuse process itself, as they will all have distinct and opposing development goals. The contention on thee success or failure of reuse approaches is best reflected in the myths, biases and preconceptions of software developers and management, this section will, thus, summarise such opinions as they are reported in the software engineering literature.

Reuse benefits. Adopting reuse-based software development process attracts a number of well recognised economic and psychological benefits to both the end-users and developers (Tracz 1988b; Hemmann 1992). These include the following.

- *Savings in costs and time.* As a developer uses already pre-defined components, hence, the activities associated with components specification, design and implementation are now replaced with finding components, their adaptation to suit new requirements, and their integration. Experience shows (also from other fields, like electronic engineering) that the latter set of activities takes less times and therefore costs less. It should be noted, though, that development of components for reuse will certainly attract additional effort, time and cost. This costs, however, can be offset by savings in a number of different software projects.

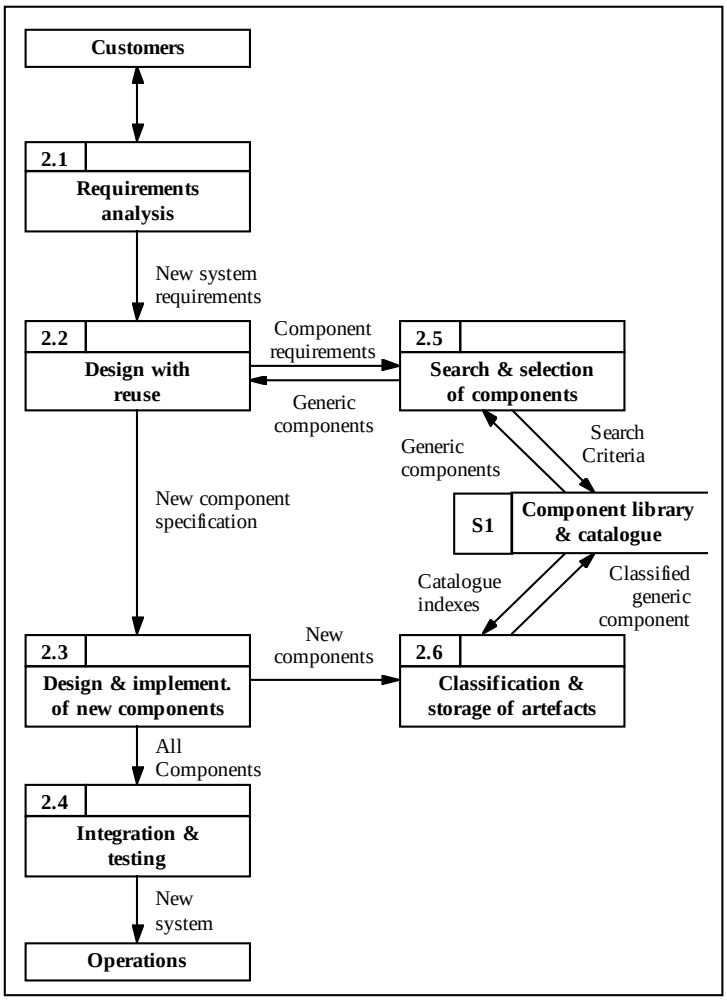

Figure 3 Reuse in Waterfall Model (Hall and Boldyreff 1991)

- *Increase in productivity.* A set of reusable artefacts can frequently be viewed as a high-level language of concepts drawn from a given problem domain. Hence, a developer is given an opportunity to work with more abstract notions related directly to the problem at hand and to ignore low-level, implementation details. It has been shown that working at a higher level of abstraction leads to an increase in development productivity.
- *Increase in reliability.* Reuse library can be viewed as a software product itself, therefore, its development follows a normal cycle of requirements specification, design, implementation, testing, documentation and maintenance. By the very assumption, the user base and a life-span of reuse artefacts is much greater than that of any individual product, thus, the reliability of such artefact is also increased. This also leads to an improved reliability of systems built of reusable components rather than of those built entirely from scratch.
- *Increase in ease of maintenance.* Systems constructed of reusable parts are usually simpler, smaller, and more abstract. Their design is closer to the problem domain and

their reliability is greater. This of course has an very positive impact on the quality of such systems maintenance.

- *Improvement in documentation and testing.* Reusable components are normally accompanied by high quality documentation and by previously developed tests plans and cases. Whenever a new system is created by simple selection and altering of such components, then, their documentation and tests will have to be much easier to develop as well.
- *High speed and low cost replacement of aging systems.* As the reuse-based systems share a very large collection of program logic via the reuse library, thus, they are significantly less complex and much smaller in size than those developed from scratch. Such systems will therefore need less effort during porting or adaptation to new hardware and software environments. It should also be noted that it would normally be the reusable components of the system that is technology intensive, and thus, very expensive to develop, e.g. graphical user interfaces, databases, communications, device control, etc. Sharing that cost across several systems would certainly reduce it when a global replacement of computing resources is called for.

Reuse drawbacks. At the same time, in practice, radical gains in productivity and quality cannot be achieved due to some preconceptions held by developers and their management (Tracz 1988b; Hemmann 1992). The claims commonly put forward by programmers include :-

- reusing code, as compared with development of entirely new systems, is boring;
- locally developed code is better than that developed elsewhere (NIH factor);
- it is easier to rewrite complex programs from scratch rather than to maintain it;
- there are no tools to assist programmers in finding reusable artefacts;
- in majority of cases, developed programs are too specialised for reuse;
- adopted software development methodology does not support software reuse;
- reuse is often ad-hoc and is unplanned;
- there is no formal training in reusing code and designs effectively;
- useful reusable artefacts are not supported on the preferred development platform;
- the reuse process is too slow;
- interfaces of reusable artefacts are too awkward to use;
- code with reusable components is often too big or too inefficient;
- programs built of reusable components are not readily transportable;
- reusable components do not conform to adopted standards;
- reuse techniques do not scale up to large software projects;
- there are no incentives to reuse software.

Myth	*Correction*
Software reuse is a technical problem, so let's wait until proper tools are introduced into the organisation first.	Software reuse is both a technical and non-technical problem, frequently reuse gains may be achieved by changing work practices rather than introducing new development tools.
Special tools are needed to store, search and retrieve software components.	No special tools are needed, available data base technology can be applied to help organise and retrieve software in large repositories
Reusing code results in huge increases in productivity.	No order of magnitude increase in productivity or quality is possible by introducing software reuse only.
Intelligent tools will solve the reuse problem. Artificial intelligence is our saviour.	Artificial intelligence may play some role in improving reuse tools, e.g. in information retrieval or domain modelling. But majority of reuse effort will go into changing management and development practices.
The Japanese have solved the reuse problem.	Japanese firms were first to reuse code on a large scale but most of the savings come from reusing code. There is still scope to improve on the Japanese experience.
Ada has solved the reuse problem.	No single language will solve the problem of software reuse. Reuse of specifications and designs, as well as, reuse of domain knowledge, goes well beyond the capability of programming languages.
Designing software from reusable parts is like designing hardware using integrated circuits.	It is certainly worthwhile to study hardware reuse while developing the model for software reuse. Nevertheless, reuse of software and hardware are quite distinct activities, each having their own requirements and peculiarities.
Reused software is the same as reusable software.	Reusing software planned to be reused is easier to reuse than that which wasn't specially constructed, but, it costs more to develop software for reuse than to develop software from scratch.
Software reuse will just happen.	A lot of planning, training, and cost must be spent before some gains from software reuse will be visible.

Table 1 Myths of software reuse (Tracz 1988a)

Meanwhile, management also raises objections based on the following arguments :-

- it takes too much effort and time to introduce reuse in workplace;
- perceived productivity gains will result in cuts to the project man-power;
- customers may expect reusable artefacts to be delivered with their product;
- it may be difficult to prevent plagiarism of reusable artefacts;
- reuse of code may lead to legal responsibility in case of software failure;
- the cost of maintaining reusable libraries is prohibitive;

- management is not trained in software development methods with reuse;
- there is no coordination between software project partners to introduce reuse.

Such problems of perception often result from irrational, nevertheless, deeply rooted myths about reusability and the reuse process. A selection of such myths (cf. Table 1) were reported and subsequently demistified by Tracz (1988a).

Reuse motivators. While the common prejudice, miconceptions and outright myths among developers and management prevent companies to effectively introduce reuse into their mainstream development, Frakes and Fox (1995) show in their survey that only few factors listed above have any real impact on the success or failure of software reuse, i.e.

- *the type of application domain* - althought the reasons for this phenomenon are not known, it seems that certain types of industries show significantly higher levels of reuse (e.g. telecommunication companies) in certain areas of the life-cycle than others (e.g. aerospace industries);
- *perceived economic feasibility* - in those organisations where management convinced its software developers that reuse is desirable and economically viable had a much higher success in the introduction of reuse into those organisations;
- *high quality and functional relevance of reuse assets* increases the likelihood of the assets to be reused;
- *common software process* - although developers themselves do not regard a common software process as promoting reuse, there is a strong correlation between the gains in the process maturity and the gains in the level of software reuse; and finally,
- *reuse education* - education about reuse, both in school and at work, improves reuse and is a necessary part of a reuse program, however, since the issues of software reuse are rarely discussed in the academic curriculum, it is necessary for management to bear the responsibility to provide reuse-specific training to its employees.

The same study also showed that other factors, widely perceived as reuse motivators or inhibitors, have only a minimal effect on the reuse process, e.g.

- *use of specific programming languages and paradigms* - it is often perceived that structured, modular, object-oriented, or high-level languages improve the prospects of successful software reuse, the collected statistics, however, show no such correlation;
- *utilisation of software support environments and CASE* - although development tools are frequently marketted as greatly enhancing software reusability, some studies show that the current employed CASE tools are not particularly effective in promoting reuse of life-cycle objects across projects in an organisation;
- *employment of staff experienced in software engineering* - it seems to be evident that experienced software development practitioners are potentially better reusers than those who have no formal training in software engineering, however contrary to this belief, it can be shown that experience and knowledge of software development principles is not a substitute for training in methods and techniques specific to reuse activities;

- *provision of recognition rewards* as an incentive to promote reuse practices in the organisation - it is likely that only monetary rewards are a more effective motivator for implementing reuse practices;
- *existence of perceived legal impediments to the utilisation of reusable software* - as majority of reuse efforts concentrates on the in-house development of reusable artefacts, thus, the legal issues are of less concern;
- *existence of reuse repositories* - many organisations consider such repositories as central to their reuse efforts, practice, however, shows that those organisations which do not use sophisticated computer-aided tools assisting the classification and retrieval of software artefacts achieve similar levels of reuse as those who are active proponents and users of such automated repositories;
- *the size of an organisation conducting a software development project* - the project or development team size is often used as an argument against the introduction of a formal reuse process, small companies believe the narrow scope of their application domain will limit the potential benefit of reuse, while the big companies fear the necessary investment of resources and money to properly implement systematic reuse, the apprehension in both of these cases in unwarranted and the likelyhood of a success or failure of reuse efforts is independent of the company or project size;
- *considerations of software and process quality* - majority of surveyed developers had generally positive experience in reusing various software assets developed outside their home companies, overall, the quality concerns had little impact on the level of software reuse, the situation would probably be very different if the quality of reused assets were to deteriorate;
- *reuse measurements* - in majority of companies measurement of reuse levels, software quality, and software productivity are not done, however, those organisation which measure software reusability are not getting any significant higher reuse levels than those which fail to monitor their successes or failures in reusing software artefacts, thus in practice, measuring software reuse has very little effect on the whole of the reuse process.

Finally, Krueger (1989) provides four tenets of the successful software reuse, the tenets based on the technical and cognitive factors which he believes will ultimately translate into variety of development goals to achieve an effective policy on software reusability, i.e.

- reuse must reduce the cognitive effort of software development;
- constructing systems of reusable components must be easier than to building them from scratch;
- finding reusable artefacts must be more efficient than building them;
- understanding artefacts is fundamental to their effective selection.

6. Summary

This paper defined the concepts of software reuse, reusability, reuse artefact and reuse library. It listed those attributes of software artefacts which increase a chance of them being reused, e.g. they have to be expressive, definite, transferable, additive, formal, machine representable, self-contained, language independent, able to represent data and procedures, verifiable, simple, and easily changeable. Then the paper gave an overview of major reuse efforts in the life-cycle, starting with coding and design, and then going through specification and requirements capture, and finally covering domain analysis and modelling. Two forms of reuse-based development were discussed, i.e. development-for-reuse, aiming at the construction of reuse library, and consisting of artefact identification, generalisation, classification and storage; and the second, development-by-reuse, aiming at the construction of a new software product with the use of reuse library, and including the tasks of searching for reusable artefacts, their understanding, adaptation to new requirements, and their integration into a new system. The stages of artefacts processing include their analysis, organisation and synthesis. Finally the paper analyses the benefits and the perceived disadvantages of software reusability, focusing in particular on the myths and misconceptions held by developers and their managers. Four preconditions for reusability success were given as reduction in cognitive complexity, ease of implementation, ability to understanding of artefact structure and function, and finally, economy of reuse.

7. Bibliography

Agresti, W. W. and F. E. McGarry (1988). The Minnowbrook Workshop on Software Reuse: A summary report. Software Reuse: Emerging Technology. W. Tracz. Washington, D.C., Computer Society Press**:** 33-40.

Ambler, A. L. and M. M. Burnett (1990). Influence of visual technology on the evolution of language environments. Visual Programming Environments: Paradigms and Systems. P. G. Ephraim. Los Alamitos, California, IEEE Computer Society Press**:** 19-32.

Arango, G. and R. Prieto-Diaz (1991). Part1: Introduction and Overview, Domain Analysis and Research Directions. Domain Analysis and Software Systems Modeling. P.-D. Ruben and A. Guillermo. Los Alamitos, California, IEEE Computer Society Press**:** 9-32.

Basili, V. R. (1990). Viewing maintenance as reuse-oriented software development. IEEE Software**:** 19-25.

Biggerstaff, T. J. and C. Richter (1989). Reusability framework, assessment, and directions. Software Reusability: Concepts and Models. J. B. Ted and J. P. Alan. New York, New York, ACM Addison Wesley Publishing Company. **1:** 1-18.

Bubenko, J., C. Rolland, et al. (1994). Facilitating "Fuzzy to Formal" requirements modelling. The First International Conference on Requirements Engineering, Colorado Springs, Colorado, IEEE Computer Society Press.

Castano, S. and V. De Antonellis (1994). “The F3 Reuse Environment for Requirements Engineering.” ACM SIGSOFT Software Engineering Notes **19**(3): 62-65.

Dillon, T. S. and P. L. Tan (1993). Object-Oriented Conceptual Modeling. Sydney, Prentice-Hall.

Feather, M. S. (1989). Reuse in the context of a transformation-based methodology. Software Reusability: Concepts and Models. J. B. Ted and J. P. Alan. New York, New York, ACM Addison Wesley Publishing Company. **1:** 337-359.

Frakes, W. B. and C. J. Fox (1995). Sixteen questions about software reuse. Communications of the ACM. **38:** 75-87,112.

Frakes, W. B. and B. A. Nejmeh (1988). An information system for software reuse. Tutorial on Software Reuse: Emerging Technology. W. Tracz. Washington, D.C., IEEE Computer Society Press**:** 142-151.

Freeman, P. (1983). Reusable software engineering: concepts and research directions. Tutorial on Software Design Techniques. F. Peter and I. W. Anthony. Los Angeles, California, IEEE Computer Society Press**:** 63-76.

Fugini, M. G. and S. Faustle (1993). Retrieval of reusable components in a development information system. Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability, Lucca, Italy, IEEE Computer Society Press.

Graham, I. (1994). Object Oriented Methods. Wokingham, England, Addison-Wesley Pub. Co.

Greenspan, S., J. Mylopoulos, et al. (1994). On formal requirements modeling languages: RML revisited. 16th International Conference on Software Engineering, Sorrento, Italy, IEEE Computer Society Press.

Guttag, J. V. and J. J. Horning (1993). Larch: Languages and Tools for Formal Specifications. New York, Springer-Verlag.

Hall, P. and C. Boldyreff (1991). Software reuse. Software Engineer's Reference Book. A. M. John. Oxford, U.K., Butterworth-Heinemann Ltd**:** 41/1-12.

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

Hsia, P., A. Davis, et al. (1993). Status Report: Requirements Engineering. IEEE Software**:** 75-79.

Krueger, C. W. (1989). Models of Reuse in Software Engineering, CMU-CS-89-188, , School of Computer Science , Carnegie Mellon University, Pittsburgh.

Li, H. (1993). Reuse-in-the-large: modeling, specification and management. Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability, Lucca, Italy, IEEE Computer Society Press.

Lowry, M. and R. Duran (1989). Knowledge-based software engineering. The Handbook of Artificial Intelligence. P. R. C. Avron Barr and A. F. Edward. Readings, Massachusetts, Addison-Wesley Publishing Company, Inc. **IV:** 241-322.

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

Martin, J. (1991). Rapid Application Development. Singapore, Maxwell MacMillan International.

Martin, J. (1993). Principles of Object-Oriented Analysis and Design. Englewood Cliffs, New Jersey, Prentice Hall.

Matsumoto, Y. (1989). Some experiences in promoting reusable software: presentation in higher abstract levels. Software Reusability: Concepts and Models. J. B. Ted and J. P. Alan. New York, New York, ACM Addison Wesley Publishing Company. **2:** 157-185.

McClure, C. (1989). CASE is Software Automation. Englewood Cliffs, New Jersey, Prentice Hall.

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

Morel, J.-M. and J. Faget (1993). The REBOOT environment. Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability, Lucca, Italy, IEEE Computer Society Press.

Neighbors, J. M. (1989). Draco: A method for engineering reusable software systems. Software Reusability: Concepts and Models. J. B. Ted and J. P. Alan. New York, New York, ACM Addison Wesley Publishing Company. **1:** 295-320.

Ning, J. Q., A. Engberts, et al. (1994). "Legacy code understanding." Communications of the ACM **37**(5): 50-57.

Prieto-Diaz, R. (1989). Classification of reusable modules. Software Reusability: Concepts and Models. J. B. Ted and J. P. Alan. New York, NY, Addison-Wesley Pub. Co. **1:** 99-123.

Prieto-Diaz, R. (1994). Historical Overview. Software Reusability. R. P.-D. Wilhelm Schafer and M. Masao. London, Great Britain, Ellis Horwood**:** 1-16.

Reed, K. (1983). Factors Influencing the Design, Implementation, Performance and Use of Linking Loaders. Msc Thesis, , Clayton, Monash University.

Salton, G. (1989). Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Readings, Massachusetts, Addison-Wesley Pub. Co.

Spivey, J. M. (1989). "An introduction to Z and formal specifications." Software Engineering Journal **4**(1): 40-50.

Tracz, W. (1988a). Software reuse myths. Software Reuse: Emerging Technology. W. Tracz. Washington, D.C., Computer Society Press**:** 18-22.

Tracz, W. (1988b). Software reuse: motivators and inhibitors. Software Reuse: Emerging Technology. W. Tracz. Washington, D.C., Computer Society Press**:** 62-67.

Vonk, R. (1989). Prototyping: The Effective Use of CASE Technology. Englewood Cliffs, N.J., Prentice-Hall Int.

Wegner, P. (1989). Capital-intensive software technology. Software Reusability: Concepts and Models. J. B. Ted and J. P. Alan. New York, New York, ACM Addison Wesley Publishing Company. **1:** 43-97.

Woodman, M. and B. Heal (1993). Introduction to VDM. London, McGraw-Hill.