Information for Authors

The International Journal of Applied Software Technology is a forum for the dissemination of pioneering research work in various areas of software technology. Both original work and tutorial exposition of permanent reference value are welcome. Papers are expected to be technical in nature, and will be reviewed by the advisory editors or competent independent referees.

Submission of a manuscript represents certification by the author that the article is not being considered nor has been published elsewhere. A manuscript largely based upon a conference paper must be so identified.

Format: Three copies of each manuscript should be submitted, including figures and tables. Manuscript should be typed double-spaced on A4 size paper with a minimum 1.25 in. left margin. In general, articles should not exceed 6000 words. Articles should be subdivided into sections and sub-sections using short, meaningful subheads. A title sheet should precede the first page of the article and give the title of the article, name of author(s), institutional mailing address (including department, institution, city, state, zip code, and country), or other affiliation. Include the telephone and fax numbers and e-mail address of the contact author. Number each page.

Disk Format: If the manuscript was prepared on an IBM compatible personal computer, we prefer that a disk be sent with files in both the word processor used and the pure text (ASCII) file formats in addition to a hard copy. The word processor formats of Microsoft Word and Lotus AmiPro are preferred. Receipt of a file will decrease both the chance of error and publication time.

Abstract: An informative, comprehensive abstract of 75 to 200 words must accompany the manuscript. This abstract should succinctly summarize the major points of the paper, and the author's summary and/or conclusions.

Quotations: Quoted material of more than two lines should be set in a narrower width than the remainder of the text but continue to double space. At the close of the quotation, give the complete source including page numbers. Wherever necessary, it is the author's responsibility to obtain written permission from the copyright holders to quote or reproduce material from another publication.

Tables and Figures: All tables and figures must be submitted in camera-ready (publication quality) form. Tables and figures should be prepared on separate sheets at the end of the running text, i.e. Insert Figure 1 here. Number and type captions at the top of each table and bottom of each figure. Photographs are welcome and should be black-and-white glossy prints.

Program Listings: Program listings should be avoided unless they are essential and add significantly to the article. Authors may wish to note in the article that programs are available from the author at a nominal cost.

Credits and Acknowledgements: Financial support for work reported or a grant under which a study was made should be noted just prior to the Acknowledgements. Acknowledgements or appreciation to individuals for assistance with the manuscript or with the material reported should be included in the Acknowledgements at the end of the article.

References: Authors are responsible for checking the accuracy of all references and that all references cited in the text also appear in the Reference section. All references should be in alphabetical order by author (unnumbered), as shown below. Use the following style when referencing a book or an article in a periodical:


Submission: Send three copies of the manuscript to
Prof. Lu Si-Wei (Editor-in-Chief)
Department of Computer Science
Memorial University of Newfoundland
St. John's, Newfoundland
Canada A1C 5S7

Special Issue: Software Reuse

Contents

Structuring Requirement Specifications for Reuse
B Whittle, A Vickers, W Lam and J McDermid

Reuse and Reusability Metrics in an Object Oriented Paradigm
G Succi, S Doubiatt, C Uhrig and P Baruchelli

Industrial Applications of the REBOOT Methodology
E Karlsson and G Sindre

The Development of a Software Clone Detector
N Davey, P Barson, S Field, R Frank and D Tansley

The Role of a Configuration Management System in a Reuse Oriented Framework
G Succi, C Uhrig, M Ronchetti, A Valerio and G Cardino

Causal Connectionist Support to Software Reusability
S Benlarbi and K Zerouni

Structuring Software Components based on Reusability
Y Park and D Ramjisingh

Design and Code Reuse Based on Fuzzy Classification of Components
F Damiani and M Fugini

Volume 1, Number 3/4

International Journal of Applied Software Technology

171

191

203

219

237

249

271

291
The Role of a Configuration Management System in a Reuse Oriented Framework

Giancarlo Succi, Carl Uhrik, Marco Ronchetti, Andrea Valerio, Guido Cardino

LII - DISA, Università di Trento Via Zeni 8, I-38068 Rovereto, Italy
Fax: (+39) 0464-443140,
e-mail: {Giancarlo.Succi, Carl.Uhrik, Marco.Ronchetti, Andrea.Valerio, Guido.Cardino}@lii.unitn.it

Abstract. Software becomes increasingly complex, demanded in higher volume, and with more stringent cost and quality constraints. This precipitates a crisis for the software industry that is being addressed by changes in development practice to promote reuse and reusability. As well, configuration management is seen as a valuable tool for coping with the crisis. This paper describes the role of configuration management in a reuse oriented development framework.

1. Introduction

More so than in the past, the pace of software development has become frenetic. Some sectors of the software industry are more on the verge of an approaching crisis than others. For example in the telecommunications sector, radical changes force telecom software developers to constantly update network software to cope with rapid technological evolution, global market forces, competition, and growing user demands for quality, services and bandwidth.

In today’s business arena, time to market can determine the success or brutal death of a software product. Increasingly, software products must be sequenced through their life cycle in a more ordered, controlled, stream-lined and efficient development process so as to be turned out in less time at less cost. At the same time, software users have become more voracious in their demands for squeezing new functionality out of software - and rightly so since computers should do more work, and people, less - thus contributing to the demand for more and more complex systems. Yet these complex systems need to be reliable systems with increasingly high quality guarantees for present and future efforts. This means the software must be easily testable to prove its reliability and easily understandable for subsequent low cost maintenance. Moreover, to enhance the market value of these sophisticated products, they must become
even more sophisticated - being highly portable across various hardware platforms and
highly flexible/open for subsequent modifiability to add new functions, or better yet,
built with reusability in mind so that their components can be re-synthesized into new
products to meet the needs of future software markets.

Thus a confluence of demands for more and more software, with more and more
functionality/complexity, developed in less and less time, at lower and lower cost, with
higher and higher quality, describes the essence of the problem. Contrary to popular
belief, the solution is not simple but manifold. Some of the solution lies in revising
the software development process to promote better software engineering practices -
such as design with reuse and design for reusability. Some of the solution lies in revising
the software development process to include more sophisticated mechanisms and tools
for developing and managing software. In fact, configuration management (CM) is
one such mechanism/tool of strategic importance in controlling growing costs and
ensuring product quality in a development process which must deal with increasingly
complex software ensembles. Any company which seriously aims to comply with the
SEI Capability Maturity Model at level 2 (which includes CM) must address various
CM issues, and any enterprise that would aim for ISO 9001 certification (or achieving
high marks in various other software development process assessment mechanisms
such as TickIt or SPICE) must seriously consider their CM status as well. CM not
only assists the developer in maintaining internal consistency of its own production
line, but it guarantees a customer that he really has a knowable and maintainable
version of a piece of software so that anything be found amiss, he/she can stand a good
chance of quickly having the problem resolved by the developer.

2. Configuration Management Systems

Conceptually, every change introduced into an evolving software development cre-
ates a different version of a software product. To avoid the impending chaos that could
result without the care, configuration management (CM) concerns initiating, eval-
uating and implementing changes while maintaining product integrity. Hence, changes
to the software product must be subject to a systematic mechanism of checks and
procedures in order to ensure that integrity and traceability be preserved throughout
the software's life cycle. Thus, the purpose of software configuration management is to
establish and maintain the integrity of the products of a software project throughout
the project's software life cycle despite a serious of changes that must inevitably be
made during and after development.

Software configuration management involves identifying the configuration of the
software as a set of software products and any appurtenant artifacts (e.g., documenta-
tion or requirements or design notes) at various times and subjecting their revision
to controls. Items under control include end-user final deliverables as well as the
tools and inputs required to generate them. The items operate as a set since changes
to one item often necessitate changes to another item. For instance, if errors are
discovered in a requirements document, many changes introduced in the requirements
must be propagated to the functional specification, design, code, testing and end user
documentation. Yet, not all items need to be controlled. Depending on the software
product, the controlled items, referred to as configuration items, will vary. The criteria
depends strongly on the projected plans to maintain and enhance the product. These
could include such items as requirements, specifications, design and analysis docu-
ments, source codes, object libraries, test plans and data, user and sysadmin manuals,
programmer reference manuals, memory maps, databases, graphics images, and pro-
cedural or policy documents. As indicated, the specific items under control will vary
according to the project, as certain items may be suspended from control, especially
to allow for the finite resources available to accomplish this control. In general, it is
infeasible to control everything.

Eventually, a version of the software object will be reached that is not intended to
be changed. This immutable released version or baseline will be designated as such by
some particular person or plan. Configuration management helps to establish these
baselines by placing items under configuration control. Often the only identifiable
high-water mark that a project has moved from one phase to another is that the end
of the previous phase has produced a critical document or deliverable artifact. Above
all these critical documents should be subject to CM controls.

Naturally, there are many different sorts of changes to the items under control, and
CM should permit those changes to be treated differently. The two principle sorts of
changes are discrepancies and requested changes.

Discrepancies may be reported by anyone in the development process, marketing
personnel or by customers subsequent to delivery of the product. Such reports must
be logged, scheduled for consideration by a configuration control board, and then pro-
cessed. Quite sophisticated links in the CM process can be established in some CM
tools with respect to the simple logging of such discrepancy reports - e.g., preventing
disallowing suspect software from being linked into other software once an error report
has been logged against it. Discrepancies can be further categorized as requirements
errors, development errors and standards violations. Although requirements errors
account for only 11% of software errors, the effects are propagated extensively causing
the cost of changes from this source to be ultimately very high. Either the customer or
marketing did not fully understand the needs or incorrect information was introduced.
On the other hand, downstream errors account for 89% of errors in the development
process. This means that a correct requirement was incorrectly implemented. Devel-
opment errors occur between the time the requirements are baselined and the time the
product is released to marketing or the customer. Finally, a discrepancy can occur in
the form of a standards violation, but in the interest of process quality preservation
one does not want to allow these sorts of discrepancies to be treated too lightly.

At a high level, requested change requests should be treated in much the same
way as discrepancy reports, but the sources of the request are different: unimple-
mented requirements, enhancements, improvements. Unimplemented requirements
are requirements that did not get into the set of implemented features due to resource
constraint, but at a later time, the evaluation of the situation can change - e.g., after
a spaghetti code has been cleaned up, additional room may exist in the system for
other functions. Enhancements, on the other hand, are completely new functionalities
not previously considered, and hence they represent new requirements. And finally,
improvements are changes requested that will improve the product, NOT for function-
ality or performance, but for QUALITY - portability, testability, understandability,
portability or flexibility to future change requests.

In addition to educating software developers in CM techniques and giving them
tools for CM, a human based control infrastructure needs to be established, in the
form of configuration control boards which serve as the authority for the evaluation
and implementation of changes to parts of the software product under its jurisdiction. The boards must be serious, prompt acting and responsive to the developers’ requests for action or explanation. In the same way the software development structure is hierarchical, the connectivity between different developers in a large project and different configuration control boards (and even among control boards) is hierarchical. For instance, a subsystem control board’s decisions are subject to review from the system level control board. In this way, there is a means to make an appeal to a higher authority and a system of checks and balances. The boards are typically sharing personnel with the development process. For instance, in order to enforce decisions of the board, the highest level board always includes the project manager, and a software control board always has the principle software architect. The board should evaluate reports submitted to it for changes.

Various factors enter into the board’s decisions: complexity and size of changes, criticality, relation to other changes already in progress, test requirements, resource needed to effect the changes (skills + hardware + system level), processing and memory impact, politics (internal + customer or marketing desires), maturity of the change (was it frequently requested in past), alternatives. If the CM board is evaluating a discrepancy item, it is important to realize that not all discrepancies need be fixed - for instance, it may be uneconomical to repair some errors, they will be marked as “waived” items. If the CM board is considering change or enhancement requests, additional time and cost evaluations are required since changes and enhancements generally should be considered as carefully and within the same evaluation criteria as the decision process for the original design of the software product.

Version control is closely akin to the simultaneous update problem when handling file I/O in a computer operating system, and therefore the same sort of semaphore control mechanisms enter into it. Especially when multiple programmers are working on a large project, there is a chance that any two of them will risk to attempt simultaneous modifications to the same piece of code. Naturally, it is undesirable because one person’s changes will cancel or distort the effect of the other person’s changes, thus resulting in a new set of bugs or maladies (possibly worse then either of the two were intending to address with their changes). As in a public library, one must check out and check in books to enable their use, also in version control, one must check out and check in other software artifact (documentation, test plans, etc.) before it can be used and wait for some one else to check it in if it is already in use by another. Complications arise in the natural spawning off of diverse versions from the same parent system, which can be registered with each set of changes introduced. Sometimes these changes have major significance resulting in limited new releases of the system and in alternate development paths for the system.

Many automated tools have been devised to assist in version control: SCCS, RCS, CMS, Cedar, Domain, ClearCase, only to name a few (Tichy 1985, Tichy 1988, Rothkind 1975, Leblond 1985, DEC 1985). Often these tools are closely coupled/integrated to system description languages, which allows repeating the process of building a version of a system through a series of incremental transformations/modifications that will result in a desired system automatically based on input parameters and states of other input elements (e.g., files). This is in some sense an expansion of the UNIX MAKE tool.

CM planning needs to be project specific. Due to cost and resource constraints not all items and not all phases will be subject to controls, but this is a project-by-project compromise. Many organizations provide their own prescriptions about CM planning, but IEEE/ANSI Std 828 may be taken as a generic standard.

All of the above introduction to Software Configuration Management is based upon the Software Engineering Institute’s prescriptions. However, it provides no particular consideration for REUSE. In the next section, a rather complex network oriented reuse repository, oriented software design and development paradigm will be put forward which even more so emphasizes the role of CM in the future.

3. Integrating Configuration Management into a Reuse Oriented Environment

As software systems become increasingly complex, it is widely recognized in the industry and in academia that there is the need of a radical paradigm shift from the past approaches to design, development and maintenance of software largely due to the consequence of factors needs aforementioned in the introduction. A number of object oriented technologies have emerged to address this paradigm shift - such as the 4th generation languages (SmallTalk, C++, Objective C) and client-server architectural extensions (CORBA, OSCA, PCTE), and as well various tool integration and interoperability mechanisms (DLL, Tooltalk, Softbench). Methodologically, a host of oriented design techniques are available (OOP/BD (Schalsale and Mellor 1992), OOD (Wirsf-Brock, Wikerson and Wiener 1990), OOA (Coad and Yourdon 1991), BON Nerson 1992), OMG (Rumbaugh 1991)), all evolving out of the ideas of domain analysis (Neighbors 1987). These all represent better means of organizing and reorganizing functional pieces of application-domain concepts into software which will ultimately have less cost and development overheads. One of the principal driving objectives behind these developments is software reuse (or more properly, design with reuse) and reusability (design for reuse) which pushes development teams to be more efficient, cut development time and design with more understandable software components.

The case for software reuse is simple. If one considers a typical development group, there are often similarities among the new projects and products undertaken and the past ones which have reuse potential. Similarly, the problems of one department or division in a firm are frequently similar to those seen in another department or division, again suggesting some utility in providing a shared pool of programming objects, accessible to all and easily tailored to special needs. Finally, assuming no competitive or proprietary interests for a set of such sharable objects, one may easily extend the notion to that of a structural unification of all software and information resources across a multi-organizational network environment, not just including code objects, but almost any sort of object in a software development process, or even more generally, any sort of information resource, as a target for reusability.

To facilitate reuse and reusability, information resource entities must be organized in a convenient way to enable users to classify, search, for, access/use and maintain them. It is desirable to have a means for different users of a single or multiple organizations to be connected to a wide-area network and to properly structure items from all phases of the software life cycle development process - entities that constitute a heterogeneous mix of objects: requirements documents, designs and architectures, chunks of executable programs, source code and object libraries, classes and class li-
libraries, both end user oriented and programmer oriented documentation, pictures and animation clips, sounds and audio tracks, databases, test plans and test data, documents in various electronic forms, any form of information resources or even physical objects such as books. Beyond this, the environment is heterogeneous in the sense that eventually it should include more than just programmers: customers, suppliers, users, developers, managers, advertisers are a non complete list of participants to such an environment; an heterogeneous environment targeted to software production is by his own nature distributed in time and space, multi-organizational, multi-systems and heterogeneous in terms of the machines and the operating systems working on it. In this way, all users can identify, locate and utilize whatever they have a use for from a distributed computing environment, provided they have the rights to do so. This is the realization of the "reuse dream": to allow different users of distributed software repositories to identify and locate the entities for which they might find labor saving uses.

Two are the main ways an implementor may register information when depositing an item in the networked repository or a reuser can employ when trying to identify a desired entity:

- "attributes" of various types are used for recording useful data about the artifacts as well as for classification and retrieval;
- *intrinsic attributes* pertain to the items ontology (e.g., the last modification or usage date, the owner, the name, the size);
- *external attributes* are human or computer annotations which may be openly defined (e.g., representing enumerated subject categories, keywords, free text, facets in a faceted classification system (Frakes and Pole 1994) characterizing the entities: the quality/maturity level of a product, a description of the behavior of a piece of code, a local comment for a remote file are all examples of annotations.
- "relations" which can be any kind of association that can be drawn between two entities, such as being one a formally recorded requirement specification for the other or one being the next version of the other 1.

For the reuse purposes, attributes serve to facilitate searching the repository of reusable items and relations serve to facilitate browsing it (which is particularly necessary because often one find no exact match for the item sought and some further investigation of close matches or best matches is useful - in this case, relations serve as bridges to additional candidates which may better satisfy the needs than the close match). Thus, relations impose a lattice or web-work over the reusable repository which may be traversed independently of the attribute searches.

At this point, it should begin to be clear where CM can enter into the reuse repository system management. Configuration management extends over the life cycle of every software product and every deliverable artifact arising therefrom, it and

1 Note that the definition of what is an entity and what is a relation is not sharp: relations can rise to the range of entities when they have a self standing relevance e.g., a baseline is a relation among several modules of code identifying a working product, therefore it is also a well defined entity. That is relations can represent sets or tuples that aggregate other entities.

since tools, techniques and standards exist targeting its correct implementation, it has classically been considered a stand-alone function, though closely related to the concerns of the development process. With the holistic view that every artifact of the design, development, and maintenance process is reusable and a traceable element in the web of software development (indeed, the relationships explicitly recorded in the reuse repository outline this web), CM is a logical part of that web reflected the relationships between artifacts, whether the relations represent a virtual script for a system description language or the paths an early version of a piece of code had to pass through before it got established in a baseline.

An obvious role of CM in a reuse repository system like the one abovementioned is to store references to controlled entities and establish the links between controlled entities. In this way, the repository can follow links, yet the CM process can be made relatively independent. As a side effect track, this capability provides tracking of the production process of a software entity: beginning from the expectations (the informal description of the problem to be solved), formalizing a set of requirements producing a requirement specification, together with a clear structuring of all the single subrequirements involved, defining the relationships among them, providing a design/architecture of the tool requested, satisfying the requirements just produced, helping the developers to build the project, designating the single entities to implement and their interconnections, subsequently implementing every single entity and then connecting them together, searching for reusable modules in the repository which realizes the functions required thereby possibly avoiding the phase of testing and debugging, even recording links to potentially multiple versions of the source code implemented, and last but not least, completing the documentation for the tool produced.

Often in inserting legacy code into a reuse repository, there is the need to reengineer or reverse engineer some piece of existing software (for example, to make it more maintainable or make it manageable for future evolutions). Certainly an obvious step that involves CM is to create and maintain a distinctly new version of the item which will go into the repository that can be backwards related to the original version. However, the availability of the simple source code may be insufficient for the task. In this case, it is very important to collect all the available information regarding the considered component, but very often all the related documentation and the requirements are buried in a lot of other things referring to other parts of the same project. It is not clear that these relations to be created are unique to the specific case of reengineering or reverse engineering. The starting entity can be the source code that implements some function and the related requirements can be found following the relation "Implementation of", while the existing documentation will be linked to the source code by the relation "Is documented by". Because it can be very difficult to find the documentation related to a specific piece of code out of all the information existing, a somewhat artificial CM history may need to be reconstructed from "archeological researches". One of the most important of these CM artifacts will certainly be the system description or a meta-file to enable the new versions of the code to be more systematically constructed.

Software requirements are the specification of the "conditions or capabilities that must be met or possessed by a system or a system component to satisfy a contract, standard, specification, or other formally imposed documents" (IEEE 1983). As such,
they "form the basis for subsequent development of the system or system component" (IEEE 1983). "Requirements Management" is therefore a Key Area in the SEI Capability Maturity Model, at Level 2 of the Model (Repeatable) (Paulk, Curtis, Chrissis, Weber 1993); to reach this Level there is a minimum set of "key practices" that must be performed at a minimum level of rating. A good reuse repository would desirably support this activity since it is well known that in order to gain the maximum advantage from software reuse/reusability one should be able to identify common patterns at the earliest possible stage of the software life cycle, and for this reason it is essential to catch similarities at the Requirement Analysis stage, so as to be able to inherit them in all the following stages (from general design to code).

Since existing relations can be traversed forwards and backwards, it is possible to go from requirements to later stages of the software life cycle, or to find which requirements are at the origin of a piece of code for example, or for instance, also the testing phase can be helped by tracing the software to be tested back to the requirements. Version control plays an important role: a new version of the requirements document can be compared with the previous version, and all the requirements which were added, deleted or modified can be identified. By following the existing relations, one can find all artifacts at later stages in the life cycle which need to be updated to reflect the changes in the requirements.

4. CM and Reuse Oriented Process Reengineering

To a large extent, reuse oriented an effective implementation of reuse involves a considerable amount of process reengineering. A notable change in the traditional software development process is needed to take fullest advantage of the benefits of reuse and design for reusability. Here a brief outline of some of these activities is given, and it will be made apparent what the role of CM could be in promoting such process innovation.

One may consider both reuse in the small, which is reuse of code on the level of components (functions, subroutines, classes, libraries), or reuse in the large, which means reuse of whole subsystems or systems, or their generalizations into shells and middleware to be used in generating whole families of product lines. Naturally, reuse in the large has a much bigger pay off for developers than reuse in the small. It largely depends upon a proper mechanism for decomposing designs, specifications, requirements and the other early software life cycle artifacts in order to make a good reengineering of these objects into new contexts (this explains the emphasis at the end of the last section on well structuring requirements and putting them under CM control). Also, to a large extent, object oriented software has some of the desirable early life cycle properties that reuse in the large demands.

To be more explicit, the concern for the design decomposition should be more at the level of domain analysis (Neighbors 1987), whereby related sectors of business activities that require similar or related but different tools may establish some vertical reuse or sharing of subsystems for doing accounting/billing functions across various applications for a business customer. Whole sets of generic operating system level primitives may also be reusable across domains, thus achieving horizontal reuse of convenient tools for building interfaces. Another route to high level reuse in the large may come from the decision to take a particular first generation product (or set of early products) in a future planned product line and generalize the elemental subsystems to create a shell or system description language or program generator that will be used to (semi)automatically synthesize a number of future generations of the system. These instances of paths to reuse in the large are ambitious, but a simpler case of reuse may simply arise from the realization that a whole subsystem's requirements for a new project 90% overlap the requirements of a particular subsystem of a product implemented 2 years ago.

One can imagine various alternative ways to encourage the sorts of reuse in the large. At the beginning of a project, an architectural style (Prieto Diaz 1989, Karlsson 1991, REBOOT 1994) is usually assumed for a project, set partly by choice and partly by the cultural experiences of the senior designers in the project. Various examples of innovative approaches to reuse in the large can enter in at this point, but needless to say, there is always a perceived risk. There will be a tendency to desire changing the architectural style, and the style or general design strategy itself is an item that can benefit from CM control as the top-most design choice in the project. Thus, CM serves the role of organizing and binding the top level architecture to the project. Moreover, since this set of decisions is to be reuse oriented more and more in the future, CM becomes a reuse policy enforcement instrument, effectively representing a breakwater against the waves that recalcitrant designers and engineers will try to raise against the desired reuse and design for reusability efforts for reuse in the small, but more importantly, especially for reuse in the large which at times may seem like too large a paradigm shift from past cultural experiences of the senior designers.

5. Conclusions

Almost all phases and roles of development are involved in the initiating, evaluating and controlling of changes to software products during development activities: managers, analysts, designers. Each role is involved in CM from a different point of view, depending on what has to be controlled, when, how, and why. Thus, all aspects of maintaining product integrity, change management, version control, establishing baselines and control board activities need to be addressed.

The reuse concept presented here is that of a multiorganizational networked environment to support a broad community of different users of distributed reusable software repositories to identify and locate the entities to which they might desire access. The entities are a heterogeneous mix of objects such as programs, documents in various electronic forms, information resources or even physical objects such as books, coming from any phase of the software life cycle. This reuse paradigm integrates well with CM and in fact must have a sophisticated CM component if it is to succeed.

The overall the activities provided for in the CM integrated into the reuse repository system are:

- support the use of the tools, training, making repairs, etc.
- monitoring and collecting statistics to characterize usage of the system and process,
- advise users of the system on procedures and providing heuristics,
- adjudicating and reviewing decisions taking to control against capricious or unwarranted behavior on the part of users.
With the opening horizons of the information highway, even the most mundane software developments can get world-wide exposure overnight. Moreover, the potential reuse of any software asset or an amplification effect - propagating the worth (and flaws) of the reused software into many downstream applications. When multiple components will have been reused across many many generations of many many diverse applications, there will be an explosive potential for vastly propagated chaos. There is the need to systematize ever minute components’ configuration in order to know why a system built upon other systems or components would have suddenly become dysfunctional. Also, it is worth recalling the extremely severe constraints arising in the software world: CM provides a mechanistic and rationalized framework in which to deal with the demands of an irrational world composed of users demands and resource constraints.

References


