A document-centered approach for an open CASE environment framework connected with the World-Wide Web

This paper should be published in "**Software Engineering Notes**" (ACM Publication) in **January 1997** (see "http://www.acm.org/sigsoft/").

Frédéric BLANQUI
ENSIMAG[1]
Concordia University[2] and McGill University[3]
Email: Frederic.Blanqui@ensimag.imag.fr
Web Site: " http://ensimag.imag.fr/OSDS/"

Abstract

Software development suffers from a number of well known difficulties, both technical and managerial [2]. A technical approach that could help to reduce them is the use of Computer-Aided Software Engineering (CASE) technologies. However, current Integrated Project Support Environment (IPSE) frameworks impose many constraints on CASE tool vendors, and this has impeded their adoption [10]. A more incremental strategy should be considered for the transfer and the diffusion of those technologies.

This paper describes a document-centered approach and a simple but extendable system called Open Software Development System (OSDS) based on this approach. First, we describe difficulties inherent in the management of documents involved in software development. Second, we define some requirements to address those difficulties. Finally, we describe the important features of OSDS; these include an extension of Mosaic[4] [3] that enables anyone with network access to connect to an OSDS, browse through it, and automatically integrate some documents into his/her own system.

Keywords: CASE environment framework, documentation, hypertext technology, reuse, World-Wide Web, Mosaic, OSDS.

1. Introduction

Before presenting the contents of the further sections, we are going to present an example of the potential use of OSDS.

1.1 An example scenario with OSDS

In this section, we describe OSDS as it is seen by the user. We consider a team which has to improve some critical part of legacy software, to reach an efficiency equivalent to competitors. Requirements, designs, code, tests, and documentation, are already integrated into an OSDS and linked one another.

The team manager has little time available to complete this project successfully, and, for this reason, he or she has decided to buy and integrate efficient software components instead of rewriting the old ones. Consequently, a designer has undertaken to look for such components in the OSDS of specialized companies, through the World-Wide Web, such that the old software design can be easily adapted.

Once the designer has found suitable components, and the manager has signed an agreement with the vending company, the OSDS is authorized to retrieve the software components. Then it can automatically integrate them within the old software documents, and a developer is undertaken to readjust the interfaces with the help of the system.

Finally, once this is done and checked by the system, a compilation is automatically carried out. After a successful compilation, tests can be generated and run to check the correctness and the efficiency of the new release.

Furthermore, all along that process, changes have been registered and documented, while the system was insuring the consistency of all the links, versions and configurations.

1.2 Overview of the further sections

Section 2 presents difficulties inherent in the management of documents involved in software development: traceability, documentation, modification and restructuring, reuse.

Section 3 recalls some facts about the today's few widespread use of the CASE environment framework technology.

Then, section 4 presents requirements for a CASE environment framework to address the problems described in sections 2 and 3.

Finally, section 5 presents the main features of such a system, called Open Software Development System (OSDS). Especially, it describes an extension of Mosaic [3] to be used to connect an OSDS from the World-Wide Web.

2. Difficulties inherent in the management of documents involved in software development

2.1 Traceability

A manager needs to know the stage of development of a product easily and quickly, in order to be able to plan and to introduce new resources if necessary. But this knowledge is not readily available because dependencies between specification and implementation are not well traced. This problem exists at all steps of the software life-cycle: the lack of traceability might be between requirements and design, or between design and code.

Traceability is not only useful during the initial development phase, but also during maintenance or evolution phases when, for example, managers need to estimate the impact of a requirement change on subsequent documents (such as design, code, and test documents), and to evaluate the cost of the change.

The hypertext technology should greatly help to resolve this

1. Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble (National Graduate School in Computer Science and Applied Mathematics of Grenoble), Grenoble, France. See "http://www.in-pg.fr/" and "http://www.imag.fr/".
2. Montreal, Canada. See "http://www.concordia.ca/".
3. Montreal, Canada. See "http://www.mcgill.ca/".
4. Mosaic is a free Web browser developed by the Software Development Group (SDG) of the National Center for Supercomputing Applications (NCSA) of the University of Illinois at Urbana-Champaign. See "http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html".

problem, but as yet there are only a few tools (including text editors, graphical editors, etc.) that enable the user to include markups in documents, to signal references to other documents, or to put annotations during review processes. For instance, programming languages have no syntactical structures comparable to the links of HTML (HyperText Markup Language) [21]: ... . Consequently, references in program sources must be made inside comments to avoid compilation problems.

We think that the hypertext technology, although it can't erase the "invisibility" of software, as described by Brooks in [2] as being an "essential" characteristic of software, is able to greatly reduce it.

2.2 Documentation

The documents involved in software development often suffers from a lack of documentation. Yet, documentation is critical for the successful development of complex software. For instance, during development, modules are developed separately; later, they are integrated together. If the module interfaces or the communication protocols involved are not well documented, problems are bound to arise during integration.

But, above all, documentation is needed during maintenance, reuse, evolution, or enhancement of software, since these tasks are often performed by people who were not involved in the development. Thus managers, designers, developers, and, in fact, everyone who is involved in the process, must maintain a record of their choices, plans, strategies, modifications, and the reasons for all of the decisions involved. Such records enable them to analyze a posteriori their choices, by comparing them with previous and current results. They also enable other people to understand why the developers did what they did (or didn't do), in order to work on or (re)use the things that the documentation describes, to stand in for them, or simply to take advantage of their experience. Thus good documentation can reduce the duration of a learning phase while bad documentation may well increase it.

In [4], Humphrey describes a Personal Software Process (PSP) turn toward planning accuracy, productivity and quality control, through basic statistical methods. At the organization level, Basili and al., in [5], present the experience factory concept "to institutionalize the collective learning of the organization that is at the root of continual improvement and competitive advantage". Here, we want to underline the importance of documenting, throughout the software life-cycle, all useful decisions (e.g. choice of a design structure, choice of an algorithm, choice of a data structure, choice of a testing strategy, etc.).

It is therefore important to seek criteria by which documentation can be judged. We propose some criteria that we consider to be important but we do not try to be exhaustive and precise. For additional material, the reader is referred to the specialized literature in information science [6]. The first task is to define clearly and precisely the purpose of the documentation under consideration. The purpose of a document defines the information it must provide, and the organization of that information.

Qualitative criteria:

- Is the goal of the documentation well defined? For what purposes is the documentation intended?
- Is the documentation coherent? Is information always organized in the same manner? Are there contradictions between different sections?
- Is the documentation clear? Is it well written? Is the structure understandable?
- Is the documentation concise? Is it addressing the essential ideas?
- Is it easy to find a specific information in the scope of the documentation? Is there an index, a table of contents, and so on?

Logical criteria (true or false):

- Is the documentation complete? Does it state requirements and achieve them?
- Is the documentation up to date with the product it deals with?
- Is the documentation correct? Are there any mistake?
- Is the documentation easily identifiable? Does it give information such as authors' names, release date, source, and addresses.

Despite the fact that documentation can't erase the "complexity", the "conformity" obligations, and the "changeability" of software [2], it is an indispensable resource to master these elements.

2.3 Modification and restructuring

Software often grows by little strokes, fixing a bug here, adding a new piece of functionality there, and so on. such that, in the end, the software has lost any clear structure that it might once have had. Furthermore, the code is often changed directly without updating the corresponding design documents. Finally, after several years, the software becomes unmaintainable and unmanageable. This situation is made worse by the fact that people involved in its beginning may have left the company.

Software development environments should enable users to change the structure of software without difficulty (move a function or a submodule from a module to another, split a too big function in smaller components, etc.). Developers should not be required to think in terms of files but rather in terms of logical units of code organized in modules and submodules. But only a programming environment would be able to manage a large number of code units and the many dependencies between them.

Although this is not an attack on the "essential" aspect of software [2], this should greatly simplify the developers' life, by getting rid of those time-consuming worries, thus enabling them to concentrate on what is their actual task. Therefore, that could have an unnegligible impact on productivity.

2.4 Reuse

One solution that is usually considered to increase the productivity, the reliability, and the quality of a software development process, is to reuse parts of products and processes previously carried out [1] [2]. In particular, we have grounds for greater confidence in the quality of components that have been used in other products. It is not only code of some functions or modules that can be reused, but also design, architecture, requirements, standards, processes, and indeed any factor that is involved in software production.

To deduce from specific requirements what could be got back and consistently integrated in one's own environment, how and where to find that, to actually carry out the integration, and then to check that the reuse reaches its goals, is an important process in its own right [7]. One cannot hope to efficiently perform such a reuse process unless CASE technology enables him/her to find and compare

software components, and to quickly and easily integrate them into one's own environment. That's why reuse works well with routine libraries which provide low-level facilities, but less well for higher level facilities.

If a software component doesn't fit the needs of developers, or leads them to spending too much time understanding it to be able to effectively adapt it to their needs, they will prefer to rewrite the component. This applies even if it eventually turns out that it would have been faster and safer to reuse it. *This explains why a component will be reused only if it is well documented and technically easy to integrate into different environments*. On the one hand, key information, such as domain, goal, interface, and language, should be associated with each software component to allow search tools to give useful responses to queries [8]. On the other hand, standards should be adopted to facilitate the integration of a component from one environment to another.

3. Today's CASE environment framework technology

In practice, the most important current problem in the CASE framework technology is the integration of tools and data from multiple vendors [9]. Integration is needed to build a coherent and complete environment which can support the entire software development process. There have been already attempts to build general CASE framework specifications [11] [12].

This first generation of Integrated Project Support Environment (IPSE) was not widely adopted by CASE tool vendors [10], even though construction of the IPSEs was undertaken by large organizations and could therefore take advantage of important resources. Indeed IPSE development has been pushed so far that it has created many difficulties for CASE tool vendors. Yet, the IPSE approach appears theoretically sound, since it potentially introduces a virtual operating system that would provide the basic services required by the tools, an Object Management System (OMS) [15] that could subsume object databases and file systems, and specifications describing the services provided by the tools.

Meanwhile, to answer customer demands for more cooperative tools, CASE tool vendors have had to make trade-offs. So, they have created coalitions to enable their tools to communicate with each other and to share data. But, by working in this way, each vendor reimplements critically important services, such as data and configuration management in ways that are not usable by other vendors. Above all, the fact that there does not exist a central entity, which could manage all the documents, reduces the possibilities of reuse and traceability.

4. Requirements for a CASE environment framework

4.1 Basic principles

At a time when more and more people speak about process definition, process evaluation, and process improvement, especially for software development companies, we should attempt to apply these ideas also to the technological transfer area. The Rogers' social science theory about the diffusion of innovations [13] is widely accepted for a while. Raghavan and Chand, in [14], introduced his work to apply it in software engineering.

The fact that first generation IPSE frameworks are not yet widely adopted by CASE tool vendors [10] should lead us to propose a more incremental strategy to introduce such frameworks. These considerations have led us to adopt some basic principles:

- *Since most today's documents used in software development are implemented by files, the system must rest on the pre-existing file systems, instead of hiding the document implementation to the user*. Indeed, systems, as there are often described ones, which rest on an enormous database where documents are not directly accessible by the user and his/her tools, since they create their own representation of them, can't be quickly and easily installable. (But that doesn't mean that they are bad ones!)

- That implies also that the system can't be responsible for the links between documents, and delegates that to the documents themselves, or more exactly, to the tools which create and manage them. *That leads to a kind of "HTML philosophy" where documents own their links to other documents, thanks to hypertext reference markups amongst their proper information.*

4.2 Documents, document types and document type definitions

Contemporary operating systems work on files with different types or formats (such as ASCII or binary). But users actually reason with documents and document types (such as text, image, data, or source code). A document is a more abstract notion than a file and documents can be implemented by several files with different formats. For instance, the different versions of a document relate to the same entity and are seen as a whole by the user, even when it is necessary to view, reuse, or compare old versions of a document. So, the system should provide users with a more abstract view of their documents than that given by today's operating systems. Nevertheless, it should reuse all the repository management facilities provided, as seen in section 4.1. *But, that supposes to have a metadata base (data about user data) providing a mapping from the documents to the files.*

Just as a file format is well defined, a document type must also be well defined, and its definition must be accessible, if the document is to be usable. Furthermore, in section 2.1, we have seen the necessity for links between elements of different documents. This is the hypertext technology. But, from the point of view adopted in section 4.1, this need implies that any document type definition should include hypertext reference markup structures, to signal such cross-references to a consulting tool. SGML (Standard Generalized Markup Language) [20] seems a good support for such formal definition, all the more so since it is now widely used thanks to the success of HTML [21] (which is actually an instance of SGML since SGML is a meta-language).

4.3 Internal documentation

Of course users must be able to organize their documents as they want in folders and subfolders. But, as seen in section 2.2, for both documents and folders, it is important they are well documented to be understandable, reengineerable, or reusable. So, for any event, from their creation to any further modifications, one should be able to answer these basic questions:

- What action has been performed?
- Why has this action been performed?

- When?
- By whom?

Example:

- What: creation of a new version of a document.
  Why: to fix a bug in inputs/outputs [hypertext reference to test results].
  When: January 14, 1996, 16:06.
  By whom: William Herbert [hypertext reference to addresses].

Furthermore, in most today's operating systems, documents and folders are identified only by their names. The names are often short, personal abbreviations that may not be intelligible to people other than their owners. The names may be short either because of operating system restrictions, or because users are trying to save time. But short names compromise understandability. The goal of a document or a folder is information that should be made easily available to users as an important complement of its name. That could constitute a great help for someone new, for reengineering, or when browsing to find reusable software components.

Furthermore, as documents or (sub)folders belong to folders, the information would also be inherited. For instance, if a document is added in a folder, the folder would register this event in its own history file.

4.4 Tools, processes, and process monitoring

People use tools to create, consult, edit, analyze, compare and destroy documents. A document type can be used by a single tool (as when a vendor tool uses a proprietary format), or by several tool classes (as when text editor and a compiler operate on the same source text). The use of a certain tool, on a certain document, at a certain time, can be part of an overall process. But here we're going to call process simply the use of a tool.

Some processes can generate new documents: this is the task, for example, of code generators, report generators, test and case generators. The system must be aware of those generated documents to be able to include them automatically into the system. Thus, users must describe the outputs of every process, their document types, and how to recognize them. Filename extensions provide an example of a simple recognition mechanism. A set of document types and processes and how they are related constitutes a process model that can be represented by a Data Flow Diagram (DFD). Figure 1 shows a simplified example of a DFD about LaTeX documents[1].

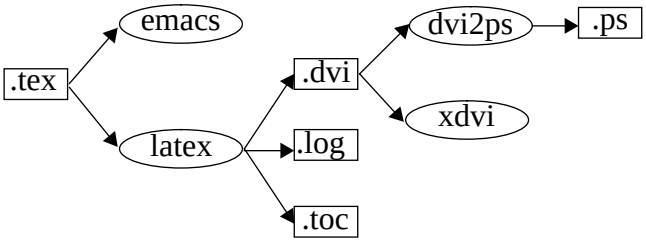

Figure 1

1. *emacs* is a text editor; *latex* is a tex file compiler; *dvi2ps* translates device independent files to PostScript (trademark of Adobe Systems) files; *xdvi* displays dvi files on the screen.

As seen in section 4.1, each tool should offer its user the capability of adding to a document, annotations or references. This should be done by means of uninterpreted character strings, since the consulting tool to run, to display a referenced document, and the protocol to follow, to run it, cannot be known a priori. Indeed, the consulting tool could be changed during the use of the tools themselves. *But, that supposes the existence of an underlying communication means, and a communication protocol, between the tools and a server which would know what to do with those references.*

4.5 Integration with the World-Wide Web

Now that network technologies are well known and are widely used by companies, to present their products or their services, and as a means of communication, we can imagine that they could be also used for exchange of higher structured information than simple text files.

If some standards were widely adopted by the software development community for software document repositories, some Internet servers could let people browse through (or just their "public" part), and even obtain documents (freely or not). As seen in section 2.4, this capability would greatly increase the reuse of software documents through the whole software community, since their integration could be carried out automatically.

The availability of such a system would create a virtual world wide software component market easily accessible. That was one of the Brooks' "Silver Bullet" [2] to fight the "essential" difficulties of software development: buying instead of building.

5. OSDS

5.1 User object classes

The previous discussion brings to light various entities, such as documents and processes, and the relationships between them. These ideas have led us to well define basic object classes which would be manipulated by users. We have considered two different approaches.

The first approach is to allow users to define their own classes and their relationships. This is the approach adopted in the Object Management Systems (OMS) [15]. The second approach is to predefine the classes: this is the solution we have adopted.

Our choice might seem restrictive, but there are two important reasons for it. OMS are not yet very efficient, although this might be only a question of time. But, above all, the adoption of the OMS approach could create compatibility problems between OSDS, and then avoid reuse possibilities.

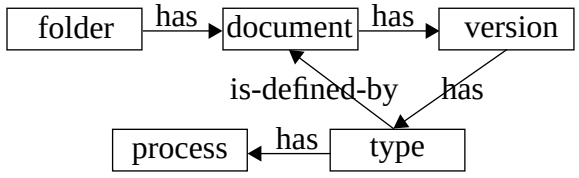

Figure 2

Figure 2 describes the main object classes of OSDS and their relationships. *Documents* are organized in *folders*, *subfolders*, etc. A *document* may have several *versions*. Each *version* belongs to a *document type*; thus, a *document* can have *versions* with different

document types. Notice that what can be consulted is actually a *version*, not a *document*. *Processes* are tied to a *document type*, as methods to a class in the object paradigm. In the current release, a *process* is simply a command that a tool can ask the underlying operating system to execute, but further improvements can be easily imagined, what would lead to a process-driven environment. Finally, a *document type* can be defined (through a *document*...) more or less formally, although a formal definition (e.g. SGML definition [20]) could be used by tools.

Our approach can be compared with that taken by contemporary operating systems like UNIX[1] [16]. The basic classes of an operating system are also predefined: file, directory, pipe, socket, etc. It is not only their simplicity but also their generality that gives them their power and their suppleness. So, OSDS can be seen as a layer on top of an operating system, just as an operating system is a layer on top of a processor.

An important feature of OSDS is that each user object owns a unique identifier or "surrogate" (term first introduced by Hall, Owlett and Todd in [17]), visible to the user, conversely to most today's database management systems. So even if names are used by the user for ergonomic reasons, they must be further replaced by their identifiers. This allows us to separate the location of an object from its identification, contrary to current operating systems, in which an object is identified by its location (its path) and its name. There are many advantages to the use of surrogates. For instance, references to a document are not affected by moving the document from one folder to another, or by changing the name of the document.

5.2 Tool-system and system-system communication

The classes defined above led us to build our system around a database of user objects. But the sections 4.4 and 4.5 identify the needs for a means of and protocol for communication, on the one hand, locally between a tool and the database, and on the other hand, through the World-Wide Web, between two OSDS databases. Consequently, we added to the database manager the capacity to receive requests or messages, and to broadcast in turn messages to tools which would be affected by those requests. A similar message service is provided by the Hewlett Packard's Softbench environment [18], but without these database and database management facilities.

Furthermore, a program able to manage a simplified interface with the OSDS message service, will be also provided to encapsulate the applications, with which the user is familiar, but which have not been designed to take advantage of the OSDS facilities. This should enable the system to get back some control on those applications.

5.3 Code, code dependencies and compilation

Until now, we have spoken about documents without distinguishing them, but in software industry there are a special kind of document: the program source, which is the source of all the problems discussed here!

Code has a special status since it has to be compiled into an executable program. The modification and restructuring problems discussed above in section 2.3 suggest the adoption of one document for each code unit definition (function, class specification, etc.). Our system, alone, cannot impose such a use on developers. However, the size and the complexity of the dependencies generated by so many documents must clearly be handled by software.

For this reason, we have built a tool, called Code Integrator (CI), able to recognize the syntactical structures of a document (for a specified programming language), determine the dependencies on other code units (except when there are unresolvable ambiguities), and separate them into several documents. Then, developers can write several code units in the same file, but the system will create separate documents when the file will be incorporated into the system.

The Code Integrator frees users from the responsibility of specifying dependencies in their programs ("extern" and "#include" in C/C++, "with" in ADA, etc.), and also in their makefiles [19], since the dependencies are generated automatically. So, if the user defines once all the information needed to carry out a compilation, then the system will be able to do it automatically. For instance, these information could be: which compiler to use for every language, and which linker to use to get the executables, all that depending on the target platforms and other compilation options.

Finally, code units are conceptually organized in an architecture of modules or classes. In OSDS, modules or classes are simply seen as special folders grouping several code units. Furthermore, the class or module interfaces (indeed, there could be several ones, depending on their purpose: use, development, etc.) can be automatically generated. Thus, that greatly simplifies the restructuring difficulties described in section 2.3.

5.4 Integration with the World-Wide Web

In order to be able to access an OSDS through the World-Wide Web, or to include hypertext references to documents or folders of an OSDS into any HTML document, we have defined a new data transfer protocol called ODTP (OSDS Data Transfer Protocol), based on HTTP (HyperText Transfer Protocol) and FTP (File Transfer Protocol). So as we make references to HTML documents by, for instance:

<a href="http[2]://www.unknown.com[3]:1000[4]/public/my_document.html[5]">,

we can do:

<a href="odtp[2]://www.unknown.com[3]:1000[4]/v1721[5]">.

From a user point of view, there is only one difference between the ODTP and HTTP protocols: OSDS objects are identified by an unique identifier instead of a path and a name, as we explained in Section 5.1. The communication port number provides for the identification of several OSDS on a same machine.

If no document identifier is given, the URL (Uniform Resource Locator) refers to the whole OSDS through which one can browse to find and retrieve documents. In this order, we have developed an extension of Mosaic [3]. A mechanism for opening and closing folders allows users to easily browse through the entire system, keeping only the parts of interest in view. Furthermore, comments related to folders or documents, which are given to the system at their creation (see section 4.3), are displayed automatically to help understanding the role of the documents.

1. Trademark of Bell Laboratories.

2. protocol name
3. host machine address
4. communication port number (optional)
5. document identifier

5.5 Installation

As we have seen, OSDS is composed of a user object database and its manager which supports remote connections, but the documents themselves are managed by the underlying operating system. This organization makes OSDS easy to install and use, and also simplifies the task of introducing new documents and using existing tools.

Of course, it may take a certain time to create the database if there are many already existing documents, because it is necessary to define the document type of each document. But this process can be partly automated by the use of tools that use file extensions and user indications.

6. Conclusion

Tool and data integration are often discussed either too abstractly or by focusing on some specific mechanism. Here we have presented a more concrete view centered on the documents involved all along the software life cycle, the references between elements of different documents, and the processes associated to the document types.

A review of some difficulties inherent in the management of those documents has allowed us to define some of the requirements that a CASE environment should meet. In particular, we think that such an environment should support different integration level, since most today's CASE tools are not able of a high level of integration. Furthermore, to have a chance of being widely used, an environment framework should be easy to install, improvable, and extendible, thanks to an open architecture based on the underlying operating system.

Furthermore, we think that the access to the public part of such an environment, from the World Wide Web, should be a great help for increasing the reuse habits through the software development community. Indeed, it would give more precise and complete information about the software components and their related documents than the today's Reusable Software Libraries (RSL), and it could support an automatic integration from a system to another, thus providing an important cost and time saving.

Finally, we have described some important features of a CASE environment framework called Open Software Development System (OSDS), which answers the defined requirements, and thus, help resolve the reviewed problems. A first documented version of OSDS will be released at the end of 1996 as a freeware. Some part of it are based on the NCSA WWW browser Mosaic [3]. We expect to develop metrics and to conduct experiments to prove the benefits brought by this environment. So we will be happy to convince people to adopt it.

7. Acknowledgments

I would like to thank very much **Peter Grogono**, from Concordia University (Montreal, Canada), for his support, his advice and his great help for the writing of this paper. I would like also thank **Nazim Madhavji**, from McGill University (Montreal, Canada), for his knowledge in the software process field.

8. References

[1] *Improving Software Productivity*; B.W. Boehm; IEEE Computer, 1987, 20(9):43-57.

[2] *No Silver Bullet: Essence and Accidents of Software Engineering*; Frederick P. Brooks; Information Processing'86, Elsevier Science Publishers B.V., North Holland, 1986, pp. 1069-1076; or Computer, April 1987, pp. 10-19.

[3] *The World-Wide Web*; Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielen, Arthur Secret; Communications of the ACM, August 1994, vol. 37, no. 8, pp. 76-82.

[4] *A Discipline of Software Engineering*; Watts Humphrey; Addison Wesley, 1995.

[5] *The Software Engineering Laboratory: An Operational Software Experience Factory*; V.R. Basili, G. Caldiera, F. McGarry, R. Pajersky, G. Page, S. Waligora; Proceedings 14th International Conference on Software Engineering, Melbourne, Australia, May 11-15, 1992; IEEE Computer Society Press, May 1992, pp. 370-381.

[6] *An Introduction to Information Science*; Roger R. Flynn; Marcel Dekker Inc., 1987.

[7] *Support for Comprehensive Reuse*; V.R. Basili, H.D. Rombach; Software Engineering Journal, September 1991, vol. 6, no. 5, pp. 303-316.

[8] *Classifying Software for Reusability*; R. Prieto-Diaz, Ruben and Peter Freeman; IEEE Software, January 1987, pp. 6-16.

[9] *Past and Future Models of CASE Integration*; Alan W. Brown, Peter H. Feiler, Kurt C. Wallnau; Proceedings 5th International Workshop on Computer-Aided Software Engineering, IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 36-45.

[10] *Software Engineering Environments, Automated Support for Software Engineering*; Alan W. Brown, Anthony N. Earl, John A. McDermid; The McGraw-Hill International Series in Software Engineering, London, 1992, pp. 296-305.

[11] *Portable Common Tool Environment (PCTE)*; Technical Report ECMA-149, European Computer Manufacturers Association (ECMA), Geneva, Switzerland, 1990.

[12] *CASE Interface Service Base Document*; Technical Report CIS v1.0, Digital Equipment Corporation, 1990.

[13] *The Diffusion of Innovations*; E.M. Rogers; Free Press, New York, 1983.

[14] *Diffusing Software-Engineering Methods*; S.A. Raghavan, D.R. Chand; IEEE Software, 1989, pp. 81-90.

[15] *Object-Oriented Databases: Applications in Software Engineering*; Alan W. Brown; McGraw-Hill International Series in Software Engineering, 1991.

[16] *The UNIX Time-sharing System*; D.M. Ritchie, K. Thompson; Bell Systems Journal, 1975, 57(6):1905-1929.

[17] *Relations and Entities*; P. Hall, J. Owlett, S. Todd; Modelling in Database Management Systems, North Holland, 1976, pp. 1-20.

[18] *The H.P. Softbench Environment: An Architecture for a New Generation of Software Tools*; Hewlett Packard Journal 41, 3, June 1990.

[19] *Make: A Program for Maintaining Computer Programs*; S.I. Feldman; Bell Laboratories Computing Science Technical Report #57, 1977; or Software Practice and Experience, 1979, 9:255-265.

[20] *SGML Theory and Practice*; Gilbert S.K. Wu; British Library Research and Development Department, London, 1989.

[21] *The HTML Sourcebook: A Complete Guide to HTML 3.0*; Ian S. Graham; John Wiley & Sons, New York, 1996.