Managing Change in Information Systems: Technological Challenges

Dag I.K. Sjøberg
Department of Informatics, University of Oslo
N–0316 Oslo, Norway
dagsj@ifi.uio.no

Abstract: Information systems and other computer-based systems must continuously undergo change in order to reflect change in their environments. The present technology used to implement such systems, including models, methods, tools and languages, does not have an inherent understanding of the nature of evolution. The rigidity of existing systems is a hindrance for user requested enhancements. Propagating changes correctly is a particular problem. It is common to find that necessary changes consequent on some other change have not been made, so that the system is inconsistent and will *eventually* fail to operate correctly. The paper discusses tools for system maintenance and focuses on the issue of automation. A tool that automatically generates and maintains all the information it needs is presented. To provide more information about the form and extent of the evolution in real-world systems, the same tool was instructed to collect change measurements. Information about the evolution of a large health management system was recorded over a period of 18 months. Methods for and problems of automatic change measurements collection are discussed.

1 Introduction

The dominant activity of the large-scale software industry is the production of changes to application systems. Most changes are due to enhancements in functionality (Lientz *et al.* 1978). People do not know in advance or are not able to accurately express all the desired functionality of (say) a large information system. Only experience from using the system in an operational environment will enable the needs and requirements to be properly formulated. The requirements assessment will continuously change during maintenance, and new requirements may be as demanding as those that directed the initial construction (Lientz & Swanson 1981). Other causes of change are accommodation to new environments and error corrections (Swanson 1976, Lientz *et al.* 1978). Some kinds of change may require extensive consequential change in the rest of the application, e.g. changes to data definitions (Marche 1993, Sjøberg 1993a). Figures describing the maintenance proportion of the total lifetime expenditure on a software system vary between 50% and 90% (Zelkowitz 1978, Putnam 1982, Chikofsky & Cross 1990). As application systems live longer and grow in size and complexity, it is likely that this trend will continue (Pfleeger 1987).

The often unexpected and confusing situations that may occur due to changing conditions and new insight during the information systems construction and maintenance process (Lientz and Swanson 1981, Bjerknes 1992) may in turn lead to new user requirements. Many factors may influence change in user requirements: change in market, workforce, skills, economy, legislation, etc. The diversity and possibly conflicting interests among users, problems in programmer-user communication, programmer effectiveness, etc. may complicate the change process. Another crucial factor in system maintenance is the suitability of the underlying technology in coping with changes. Due to rigid structure in existing systems and inadequate methods and tools for change management, implementing necessary changes consequent on new user requirements are often impossible within reasonable costs.

Information systems should be designed and implemented with change in mind, and the organisation should be planned for change (Brooks 1975). It may be difficult to persuade software builders and managers to plan for change since it requires some extra effort during initial construction which may hinder meeting short-term budget and time goals. The short-term thinking discourages designing for maintenance even though it is an investment that will more than pay off in the long run.

The prevalent assumption of stability in current teaching and practice, data modelling, database schema construction, etc. must be breached. New and improved methods and tools for implementing change are required. Tools providing software documentation are one example. Most documentation is notoriously poor and virtually always obsolete. The only reliable, up-to-date program information may be the source code itself or information that is automatically generated from the source code. This paper presents a recording tool that automatically generates and maintains information needed for change and consequential change propagation.

To turn informatics in general and the fields of information systems and software engineering in particular into a more exact science, more measurements on relevant issues should be provided. Quantification of change processes is a means to acquire detailed knowledge about the extent and form of system evolution. The paper reports a study of the evolution in a large health management system and discusses the issue of automatic change data collection.

The paper is organised as follows. Section 2 discusses the problem of change and software maintenance. Section 3 categorises software maintenance tools and presents a tool built by the author. Section 4 reports studies of system evolution and discusses the issue of collecting change statistics. Section 5 concludes.

2 The Problem of Change and Software Maintenance

The term *software maintenance* denotes all changes to the software of an information system after its first installation in its operational environment. Since software systems do not physically wear out or break (although a physical copy on any medium may deteriorate), software maintenance differs from general maintenance in that the former is not concerned with rectification to an earlier state. Software does not change on its own, but is changed by people (or possibly by other software such as tools) to adapt to changed requirements, to improve performance, to correct errors, etc. The maintenance activities have been divided into the following categories (Swanson 1976):

- *Corrective maintenance* (detecting and correcting errors - routine debugging)
- *Adaptive maintenance* (accommodation of changes to the environment - including hardware and system software)
- *Perfective maintenance* (user requested enhancements, improved documentation, enhanced performance)

It has been reported that the respective categories count for 17%, 18% and 60% of the total maintenance activities (Lientz *et al.* 1978). Within the third category, two thirds were user requested enhancements. This shows that the majority of changes are not due to errors or other causes that one might believe could be prevented by better requirements analysis, design and implementation techniques. For example, one might argue that the software changes could be reduced by more use of prototyping techniques (Budde *et al.* 1992). Prototyping may be useful during initial construction and may enable end-users to express their needs and requirements more accurately in areas such as screen design and certain aspects of system behaviour. However, since new requirements, changing environments, bug-fixing, etc. are encountered after the system has become operational, it is the operational system itself which has to be changed. The challenge is thus to build large, long-lived, data-intensive systems that can be incrementally modified in compliance with changing user needs. So, reducing the extent of perfective change is not necessarily desirable. It is usual for people carrying out tasks to recognise improved methods and opportunities. Information systems, for example, are therefore most likely to support people well if they facilitate change, and allocating resources to at least perfective change should be regarded as valuable.

The problem of change is closely related to scale. A whole class of problems only show up when a system becomes long-lived (typically involving persistent data) and grows in size, complexity and diversity (variability). This is confirmed by studies showing that software maintenance costs are significantly affected by age, size (Lientz and Swanson 1981) and complexity (Banker *et al.* 1993). Methods, tools and programming languages are also the subject of new and changed requirements in order to cope with increase in scale. For example, they must support *incremental* design, construction and commissioning.

2.1 The Software Development and Maintenance Process

The classical model for describing the software development process is the so-called *waterfall model* (Royce 1970). This analysis-design-implementation-testing model of the software development life cycle, however, does not comply with the way systems are built in the real world. Obvious inadequacies are the lack of recognition of the importance of system changes and its description of systems development as a sequential process. Some of the deficiencies are encompassed in the "spiral model" (Boehm 1988) which adds the notion of risk analysis and allows for iteration of the development tasks.

The problem of software maintenance is not explicitly addressed by any of these models, though it is common to extend the classical model with a separate maintenance phase after testing. In practice, however, the phases of development are repeated during maintenance. New requirements must be determined, the existing software application needs re-design (Braa *et al.* 1992) and re-coding, new tests must be undertaken, etc. This does not mean that the software development and software maintenance life cycles follow the exactly same pattern; at a detailed level the stages and the relative effort applied to the stages may differ (Chapin 1988). Nevertheless, to satisfy new user requirements, we need methods and tools for managing various kinds of software change – independent of whether they occur during initial construction or after the software application has become operational.

2.2 Change Propagation

It is deceptively easy to change software (simple editing). Software is therefore changed much more frequently than tangible products. However, it is not easy to make *consistent* changes; it is easy to cause a mutation, but very hard to generate a viable one, particularly if multiple copies have been shipped, etc. A change in one place may have unintended effects elsewhere; even minor local changes can have global impact. Included in the consequences are new errors (the ripple effect). One study found that more than 50% of all errors were due to previous changes (Collofello & Buck 1987). The challenge is to ensure that *all* consequential changes are dealt with correctly by propagation throughout the system and that no unnecessary changes occur, perturbing working practices and operational software.

The issue of change propagation will be illustrated by an example (Figure 1). Many information systems are centred around a database. User requested enhancements in the functionality of such a system may typically require change in the kind of information provided by the database, which is described by a data model. (The "$\Rightarrow$" in the figure should be read "may imply".) A change to a data model must normally be propagated to the database schema. Changes to database schemata (schema evolution) may in turn have

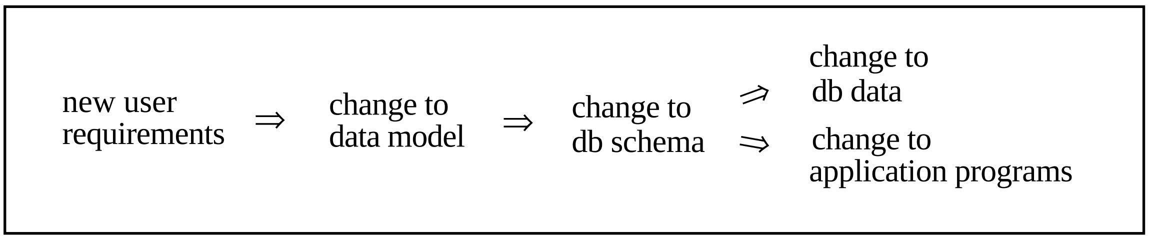

Figure 1: Change propagation

serious impact on other parts of the schema, on extensional data (user data stored in the database) and on application programs (including interfaces for data entry, queries, report generation, etc.).

At present, schema modifications are often dealt with in an *ad hoc* way. The necessary data conversions and program modifications may be expensive due to factors such as a requirement to shutdown the system, programmer effort, machine resources, etc. However, research on schema evolution is emerging (Banerjee *et al.* 1987, Skarra & Zdonik 1987, Lerner & Habermann 1990, Monk & Sommerville 1993). Work on quantifying the extent and form of schema evolution will be described in Section 4.

3 Software Maintenance Tools

Tools for managing change in information systems can broadly be divided into those supporting project management (Section 3.1) and those supporting implementation (Section 3.2). Tools in the latter category are oriented towards technical issues, and they should as far as possible extract all the information they need automatically. One example of such a tool is the *thesaurus tool* (Section 3.3).

3.1 Change Management at the Project Management Level

Project management is an activity at the overall software life cycle level and involves tasks like scheduling, team management and resource allocation (people, programming languages, tools, operating systems, hardware, etc.). The administration of changes at this level is an important part of project management and is commonly referred to as *change management* (Humphrey 1990) or *change control* (Ferraby 1991). The change process is formalised in that all change requests are evaluated with respect to the need for the change, the impact of the change on the project and system, schedule of necessary activities, etc. During the implementation of a change, information is recorded about who did what when, what is the status, what remains, etc. IBM's Information/Management product is an example of a tool providing support for such change management (IBM 1992):

> The Change Management facility helps you coordinate the various tasks your organization performs to make and test changes in your data processing environment. You can enter data about changes made to any area of your organization's operations: to software and hardware components of the operating system or to procedures, publications, and facilities.

Change management tools at this level are thus support systems that record information and produce corresponding reports.

3.2 Change Management at the Implementation Level

Developing the software of large and long-lived information systems is a complex and time consuming task. Methods and tools should assist in managing this complexity and increase the efficiency and reliability of the development. It is crucial that the software engineers and programmers find it worthwhile to learn and apply the methods and tools. They should not hinder normal working practice, but software builders must understand that they have to invest in setting up and preserving structure if they want an easier maintenance future. The following activities should be supported:

- Predicting change consequences (impact analysis)
- Propagating necessary consequential changes
- Detecting inconsistencies after change or preventing them
- Detecting and recording change (necessary for recompilation, etc.)

A sophisticated maintenance tool should also support activities such as *reverse engineering* (Bachman 1988) and *automatic documentation*. There is a significant amount of “legacy systems” (Brodie 1992) which will still be operational for many years to come. In order to satisfy new requirements, such code is continuously being modified causing deterioration of its structure (Lehman & Belady 1985). One approach to help solve this problem is reverse engineering which is to generate an abstract version of a concrete program and then re-implement the abstract version. Typically, COBOL programs are being re-implemented in COBOL itself or in a more up-to-date programming language (Griswold & Notkin 1992).

A severe problem in the software application industry is obsolete or missing documentation. The major reason for this is that documentation is normally not updated in accordance with modifications to the software. For some sorts of documentation this problem may be alleviated by tools that provide automatic documentation based on static analysis of source code. Such tools may typically generate call graphs, control and data flow charts, cross-reference information, metrics reports, etc.

3.3 The Thesaurus Tool

The Thesaurus tool was built by the author to support software maintenance at the implementation level (Sjøberg *et al.* 1993, Sjøberg *et al.* 1994a). The term *thesaurus* generally denotes “a ‘treasury’ or ‘storehouse’ of knowledge, as a dictionary, encyclopædia, or the like” (Oxford 1961). (The term is not used in the more popular meaning denoting a dictionary of synonyms.) In this context the “knowledge” is information about names and identifiers such as where they are defined and used, what kinds they are, in which contexts they occur, etc. The information captures *all* the software written in *all* the languages of an application. Information about extensional data in a database is also included.

The thesaurus is a meta-database that focuses on names which are central to system builders' thinking and thus influence the way software is organised. Meaningful names are important for problem solving, understanding of semantic structure and retention (Barnard *et al.* 1982, Weiser & Shneiderman 1987, Anand 1988). The choice of names for identifiers is crucial for the readability of programs and is particularly important when trying to administer and manage change. The meanings attached to names are relatively stable when dealing with concepts at an abstract level (even though the detailed semantics and interpretation may vary between people and between contexts). This contrasts with all changes in physical software implementations. Therefore, there is potential for tools that help manage the evolution while preserving the use of names. A focus on names may encourage people to be more conscious of what the names are supposed to refer to, though the semantic relation between names and what they refer to is a classical, largely unresolved problem (Nelson 1992). Choosing names carefully would also prevent name ambiguity.

The comprehensiveness of the thesaurus is in contrast to most commercially available tools which focus either on the source code only (source code analysers) or on database-specific information (data dictionaries). A few data dictionary tools also include source code information, but relationships between names and identifiers in the software written in the various languages are not recorded automatically. All the contents of the thesaurus are automatically maintained. The whole application system is analysed, and the thesaurus updated, regularly at times specified by the user, for example daily at 02:00. A full analysis and update can also be initiated at any time.

The author has built two thesaurus tools. The HMS thesaurus tool was developed for a health management system (HMS) in an industrial (C, C^{++}, X Window System and relational database) environment (Sjøberg 1993a). Another thesaurus tool was thereafter built in the research context of the strongly typed, persistent programming language Napier88 (Morrison *et al.* 1989). (The concept of persistence tackles the mismatch between database systems and programming languages (Atkinson 1978); a uniform model for representations and operations on persistent and transient data is provided.) Some of the features of the thesaurus tools are:

- structure and dependency visualisation,
- impact analysis, and
- automatic build management, including smart recompilation. (In large application systems, recompilations represent a significant part of the maintenance costs and may thus be a hindrance for required system evolution.)

Moreover, well-structured software is a requirement for easy maintenance in the future (Lehman and Belady 1985, Gibson & Senn 1989). To ensure correctness and prevent deteriorating structure, a set of application independent constraints to which each suite of application software should adhere, have been defined (Sjøberg 1993b). Another thesaurus-based tool automatically verifies these constraints.

When introducing a tool that automatically checks the quality of software, one should ask: Who should use the tool? How should the working process be organised to benefit as much as possible from the tool? How should the project management motivate and encourage active use of the tool? It is particularly important that inexperienced and immature programmers find bugs and inconsistencies by themselves before the software is released. The only purpose of the tool should be to improve the quality of the software; a negative attitude may be created if it is felt that the tool is used for individual monitoring purposes, e.g. by the project management.

The present tools focus on the implementation phase (initial construction and maintenance). There is a trade-off between automation and to what extent also other phases of the life cycle can be supported. If design structures, data model specifications, etc. become more well-structured, thesauri may extend their scope of information and form a basis for tools supporting other phases of the life cycle as well.

4 Change Statistics

In order to more accurately identify the requirements of methods and tools for change management, relevant information about the extent and kind of change should be provided. For example, Lehman has proposed five "laws" concerning software evolution (Lehman and Belady 1985), which are based on long experience and quantitative studies of several systems, mostly operating systems. The first two follow:

- A program must continuously undergo change in order to reflect change in its environment. If not, the program will become less and less useful.
- As a large program is continuously changed, its complexity increases, which reflects deteriorating structure, unless work is done to maintain or reduce it.

Changes to data models and database schemata are a kind of change that is particularly serious concerning the impact on the rest of the system (Section 2.2). As a step further in the direction of quantifying such change, the HMS thesaurus tool (Section 3.3) was instructed to collect change measurements in a large health management system over a period of 18 months, both during initial construction and after the system became operational (Sjøberg 1993a). The results were:

- 140% increase in the number of relations
- 270% increase in the number of relations
- every relation was changed
- more additions than deletions, but still a significant number of deletions

The extent of schema evolution and the considerable consequential changes to code confirm the need for supporting tools.

Another study reports the evolution of the data models in seven traditional applications: “Sales and payments”, “Property inventory”, etc. (Marche 1993). Approximately 60% of the entity attributes changed during the period of investigation (6 to 80 months depending on the application).

The studies of Marche and the author are the only examples of data model or database schema evolution measurements found in the literature. In spite of the changes reported these studies, it is often claimed that there is less need for change in traditional systems since they are simpler and their functionality and behaviour better understood (Banerjee *et al.* 1987). However, it could also be the case that the traditional systems are so rigid, and the consequences of change so extensive, that due to lack of appropriate methods and tools, user requested change is simply rejected. An example is the Norwegian census database – a CODASYL network database containing 5 gigabytes of data about 5 million persons. In spite of changed user needs, the schema has not been changed during the last decade due to extreme costs – typically measured in units of person-years; any (minor) schema change would imply database reorganisation and application code modification, the needed work amounting to at least half a person-year per minor schema change (Gløersen 1993).

To acquire more general knowledge about the extent and form of change, applications systems in various application domains should be measured. One may then be able to identify properties related to change consequences that are independent of application domain, data model and implemented system.

4.1 Automatic Measurement Collection

It may be impractical to collect change data manually in a large, real-world application system. There are two reasons for automating the process:

- *Reliability* In large systems with frequent changes manual collection is error-prone.
- *Economy* It is very hard to persuade people on a project to spend time and effort on keeping track of the change history.

The latter is one major reason for the lack of measurements in this area.

Automating the process requires simple detection of change. For example, it may be hard for a tool to distinguish between a rename and a deletion followed by an addition. Moreover, it is often semantically difficult to tell what kind of change has occurred. For example, assuming a type Person has been renamed to Patient and at the same had several attributes removed and added. Has Person been changed, or has Person been removed and a new type Patient been created? The more sophisticated categories of change we create, the harder is automation.

In the health management study reported in the previous section, the automatic change data collection was made complicated by the significant change to also other parts of the system structure and to the development environments. Due to changes to directory structures and file naming conventions, changes to the support software (e.g. operating system, DBMS, version control systems), etc., tools collecting change statistics need to be subject to the same change control mechanisms as the rest of the system under study. Completely automated collection of change data seems thus impossible. Therefore, in order to collect reliable measurements of a real-world system, the application development people on the site must have the time and interest in co-operating with the experiment. One problem is to convince them that the data collection is worth the investment.

5 Conclusions

Adapting large, long-lived information systems to their changing circumstances and requirements, remedying errors and improving existing functions are the technologically most challenging issues for software engineers responsible for the implementation of such systems. Methods and tools for systems development should have an inherent understanding of the nature of evolution in large information systems. Hence, they should provide adequate means for managing change, including necessary consequential change.

Thesauri that collect and correlate information about all names used in the *whole* processing environment of an application system form a useful platform for software development environments. This approach has proved computationally feasible and extremely useful both in an open, industrial environment and in a closed, research environment. The focus on names is justified by the observation that within a context (e.g. an application) people tend to use the names to have a consistent intended meaning. Thus names are interesting markers when trying to administer and manage change.

A whole class of tools could utilise the thesaurus information to support change and build management, incremental schema design, visualisation, schema evolution, etc. can be envisaged in a software engineering environment. Future research will emphasise change management. Supporting tools can operate at two levels (Atkinson *et al.* 1993). First, informative systems like the thesaurus interfaces and parts of EnvMake provide application developers and maintainers with data about the existing system, its present representation and some of its dependencies. Second, more challenging to build are automatic systems that directly implement some of the steps necessary to deal with the consequences of change. Further work on automation requires more knowledge about which changes should be propagated and which absorbed. Notations to describe propagation requirements should be developed.

For reliability and efficiency reasons, the information required by change management tools should be automatically maintained. (Information that relies on manual update is usually out of date.) A consequence of this requirement, at least at present, is that most of the thesaurus information relates to the implementation and operational phases since information related to earlier phases is harder to collect and analyse automatically. Ultimately, however, information related to all phases of the life cycle should be collected, e.g. the analysis tool could scan design structures.

To turn informatics into a more exact science, more measurements should be obtained provided they are relevant. Claimed problems and proposed solutions should be quantified. The reported study of the evolution in a large health management system is one step further in this direction. Other areas also need quantification. For example, as a supplement to anecdotal description of user experiences, attempts should be made to quantify the potential benefits of new and enhanced methods and tools (Basili & Reiter 1981, Law & Naeem 1992). This may be achieved by measuring and analysing application systems before and after the methods have been adhered to and the supporting tools applied (Sjøberg *et al.* 1994b).

Identifying what is interesting to measure and carrying out experiments yielding reliable results are non-trivial tasks. For example, many human properties that are crucial for change management in large-scale information systems (people's efficiency, skill in management, ability to communicate, etc.) are difficult to measure. We are certain, however, that much more than is the case at present could and should be measured within the fields of information systems and software engineering.

Acknowledgements

Numerous stimulating discussions with Malcolm Atkinson and Ray Welland were essential for the ideas and work reported in this paper. The author was supported by the Research Council of Norway.

References

Anand, N. (1988). "Clarify Function!". *ACM SIGPLAN Notices*, Vol. 23, No. 6, pp. 69–79.

Atkinson, M.P. (1978). "Programming Languages and Databases". In: *Fourth International Conference on Very Large Data Bases*, Berlin, West Germany, 13th–15th September 1978, IEEE and ACM, pp. 408–419.

Atkinson, M.P., Sjøberg, D.I.K. and Morrison, R. (1993). "Managing Change in Persistent Object Systems". In: *First JSSST International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, 4th—6th November 1993, Lecture Notes in Computer Science 742, Springer-Verlag, pp. 315–338.

Bachman, C. (1988). "A CASE for Reverse Engineering". Datamation, Vol. 34, No. 13, pp. 49–56.

Banerjee, J., Kim, W., Kim, H.–J. and Korth, H.F. (1987). “Semantics and Implementation of Schema Evolution in Object-Oriented Databases”. In: *ACM SIGMOD 1987 Conference on the Management of Data*, San Francisco, CA, 27th–29th May 1987, pp. 311–322.

Banker, R.D., Datar, S.M., Kemerer, C.F. and Zweig, D. (1993). “Software Complexity and Maintenance Costs”. *Communications of the ACM*, Vol. 36, No. 11, pp. 81–94.

Barnard, P., Hammond, N.V., MacLean, A. and Morton, J. (1982). “Learning and Remembering Interactive Commands”. In: *Conference on Human Factors in Computer Systems*, ACM Washington, CD 1982.

Basili, V.R. & Reiter, R.W. (1981). “A Controlled Experiment Quantitatively Comparing Software Development Approaches”. *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 3, pp. 299–320.

Bjerknes, G. (1992). “Dialectical Reflection in Information Systems Development”. *Scandinavian Journal of Information Systems*, Vol. 4, pp. 55–77.

Boehm, B.W. (1988). “A Spiral Model of Software Development and Enhancement”. *IEEE Computer*, Vol. 21, No. 5.

Brodie, M. (1992). “The Promise of Distributed Computing and the Challenges of Legacy Systems, Invited paper”. In: *Tenth British National Conference on Databases,* Aberdeen, Scotland, 6th–8th July 1992, Lecture Notes in Computer Science 618, Springer-Verlag, pp. 1–28.

Brooks, F.P. (1975). *The Mythical Man-Month*. Addison Wesley.

Braa, K., Bratteteig, T. and Øgrim, L. (1992). “Redesign Process in System Development”. In: *15th IRIS,* Larkollen, Norway, August 1992, pp. 112–126.

Budde, R., Kautz, K., Kuhlenkamp, K. and Züllighoven, H. (1992). *Prototyping. An Approach to Evolutionary System Development*. Berlin, Springer-Verlag.

Chapin, N. (1988). “Software Maintenance Life Cycle, Proceedings”. In: *Conference on Software Maintenance,* Phoenix, AR, USA, 24th–27th October 1988, IEEE Computer Society Press, pp. 6–13.

Chikofsky, E. & Cross, J. (1990). “Reverse Engineering and Design Recovery: A Taxonomy”. *IEEE Software*, Vol. 7, No. 1, pp. 13–17.

Collofello, J.S. & Buck, J.J. (1987). “Software Quality Assurance for Maintenance”. *IEEE Software, September 1987*, pp. 46–51.

Ferraby, L. (1991). *Change Control During Computer Systems Development*. Prentice-Hall (UK).

Gibson, V.R. & Senn, J.A. (1989). “System Structure and Software Maintenance Performance”. *Communications of the ACM*, Vol. 32, No. 3, pp. 347–358.

Gløersen, R. (1993). Private Communication. Statistics Norway, Oslo, Norway.

Griswold, W.G. & Notkin, D. (1992). “Computer-Aided vs. Manual Program Restructuring”. *ACM Software Engineering Notes,* Vol. 17, No. 1, pp. 33–41.

Humphrey, W.S. (1990). *Managing the Software Process*. Reading, Massachusetts, Addison-Wesley.

IBM (1992). The Information Management Library: Problem, Change, and Configuration Management, User's Guide. SC34-4328-00, IBM.

Law, D. & Naeem, T. (1992). “DESMET – Determining an Evaluation Methodology for Software Methods and Tools”. In: *CASE, Current Practice, Future Prospects*. Spurr, K. and Layzells, P. (editors), J. Wiley & Sons, Chichester, England, pp. 167–181.

Lehman, M.M. & Belady, L. (1985). *Program Evolution, Processes of Software Change*. A.P.I.C. Studies in Data Processing No. 27, London, Academic Press.

Lerner, B.S. & Habermann, A.N. (1990). “Beyond Schema Evolution to Database Reorganisation”. In: *Conference on Object-Oriented Programming Systems, Languages and Applications*, 1990, pp. 67–76.

Lientz, B.P. & Swanson, E.B. (1981). “Problems in Application Software Maintenance”. *Communications of the ACM*, Vol. 24, No. 11, pp. 763–769.

Lientz, B.P., Swanson, E.B. and Tompkins, G.E. (1978). “Characteristics of Application Software Maintenance”. *Communications of the ACM*, Vol. 21, No. 6, pp. 466–471.

Marche, S. (1993). “Measuring the Stability of Data Models”. *European Journal on Information Systems*, Vol. 2, No. 1, pp. 37–47.

Monk, S. & Sommerville, I. (1993). “Schema Evolution in OODBs Using Class Versioning”. *SIGMOD Record*, Vol. 22, No. 3, pp. 16–22.

Morrison, R., Brown, F., Connor, R. and Dearle, A. (1989). The Napier88 Reference Manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews.

Nelson, R.J. (1992). *Naming and Reference*. The Problems of Philosophy, Routledge, London.

Oxford (1961). *The Oxford English Dictionary*. Oxford University Press, London.

Pfleeger, S.L. (1987). *Software Engineering – The Production of Quality Software*. Macmillan.

Putnam, L.H. (1982). “Software Cost Estimating and Life Cycle Control”. *IEEE Catalog*.

Royce, W.W. (1970). “Managing the Development of Large Software Systems”. In: *IEEE WESCON,* 1970.

Sjøberg, D.I.K. (1993a). “Quantifying Schema Evolution”. *Information and Software Technology*, Vol. 35, No. 1, pp. 35–44.

Sjøberg, D.I.K. (1993b). Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems. Ph.D. Thesis, Department of Computing Science, University of Glasgow.

Sjøberg, D.I.K., Atkinson, M.P., Lopes, J. and Trinder, P. (1993). “Building an Integrated Persistent Application”. In: *Fourth International Workshop on Database Programming Languages – Object Models and Languages,* Manhattan, New York City, USA, 30th August – 1st September 1993, Springer-Verlag and British Computer Society, pp. 359–375.

Sjøberg, D.I.K., Atkinson, M.P. and Welland, R. (1994a). “Thesaurus-Based Software Environments”. In: *Workshop on Research Issues in the Intersection between Software Engineering and Databases, 16th International Conference on Software Engineering,* Sorrento, Italy, 16th–17th May 1994.

Sjøberg, D.I.K., Cutts, Q., Welland, R. and Atkinson, M.P. (1994b). “Analysing Persistent Language Applications”. In: *Sixth International Workshop on Persistent Object Systems,* Tarascon, Provence, France, 5th – 9th September 1994.

Skarra, A.H. & Zdonik, S.B. (1987). “Type Evolution in an Object-Oriented Database”. In: *Research Directions in Object-Oriented Programming*. Shriver, B.S. and Wegner, P. (editors), MITP, Cambridge, MA, Computer Systems, pp. 393–415.

Swanson, E.B. (1976). “The Dimensions of Maintenance”. In: *Second International Conference on Software Engineering*, 13–15 October, San Francisco, California, Long Beach, CA, 1976, IEEE Computer Society. IEEE Catalog No 76CH1125-4 C, pp. 492–497.

Weiser, M. & Shneiderman, B. (1987). “Human Factors of Computer Programming”. In: *Handbook of Human Factors*. Salvendys, G. (editor), John Wiley & Sons, pp. 1398–1415.

Zelkowitz, M.V. (1978). “Perspectives on Software Engineering”. *ACM Computing Surveys*, Vol. 10, No. 2, pp. 197–216.