Evaluating Software Maintenance Technology*

Dag Sjøberg†, Ray Welland‡, Malcolm Atkinson‡, Magne Jørgensen§,
John Petter Martinussen¥ and Arne Maus†

†University of Oslo, Norway, ‡University of Glasgow, UK, §Telenor, Norway ,¥SDS, Norway

Abstract: It is widely documented that the cost of maintenance activities is the dominant part of the total lifetime expenditure on a software system. Still, there is no commonly accepted framework for evaluating technology that is supposed to support maintenance. This paper discusses methods for such evaluation, which include both analytical work, such as development of taxonomies, and empirical work based on benchmarks. Several of the methods have been applied in experiments that we have already finished, in experiments that we are currently running and in planned experiments. What we report is initial work carried out within severe resource limitations. Hence, our work should be regarded as a basis for more extensive work which is required in this area.

Keywords: Maintenance, technology evaluation, research methods, experiments, benchmarks

1 Introduction

Large, long-lived and data-intensive application systems that satisfy a complete area of information-processing requirements, such as most information systems, CAD/CAM systems, CASE environments, etc. must continuously undergo change in order to reflect change in their environments [9, 12, 15]. Figures describing the maintenance proportion of the total lifetime expenditure on a software system vary between 50% and 90% [4, 16, 24]. As application systems live longer and grow in size and complexity, it is likely that this trend will continue.

In this paper we use Boehm's definition of maintenance [3]: “the process of modifying existing operational software while leaving its primary functions intact.” However, we do not intend to attempt to define precisely where the boundary between maintenance and development lies as we feel that this is largely a managerial decision which is organisation dependent.

The purpose of our work is to *evaluate* maintenance methods and the tools that support those methods. The purpose of the evaluation technology is to have a means to test hypotheses to decide whether for example:

- Method(ology) X improves maintainability.
- Tool *A* improves maintainability or tool *A* supports maintenance better than tool *B*.

We are seeking objective measures for evaluation that are founded on empirical work. Our work includes collecting data to identify important factors in maintenance tasks, such as the types and frequency of changes; how effective existing documentation is, the effect of the experience level of the software maintainer on the efficiency of maintenance.

The scope of our work is to consider the *efficiency* of the maintenance process while retaining satisfactory levels of quality and performance of the maintained product. It is necessary to consider the trade-off between the long-term effectiveness of the maintenance process versus the short-term speed of change. For example, a 'quick fix' usually leads to loss of software structure which makes future maintenance more difficult. Moreover, maintenance technology is concerned with making changes effectively when they have been approved via some change control process. We are *not* concerned with change control [6], the commercial (political) problem of controlling which changes are to be made (evaluating cost/benefit, administering the change process, etc.).

Our work focuses on *administrative data-intensive application systems*, which are viewed as consisting of the four components shown in Figure 1. The figure illustrates that a change in one of the user interface, application programs and data model components may mutually imply a change in the other components. A change initiated in the database (extensional data) is possible for performance reasons, but should not imply a change in the schema.

*Appeared at Norwegian Conference in Informatics, Alta, Norway, 18-20 November, 1996, pp. 49-61, TAPIR 1996

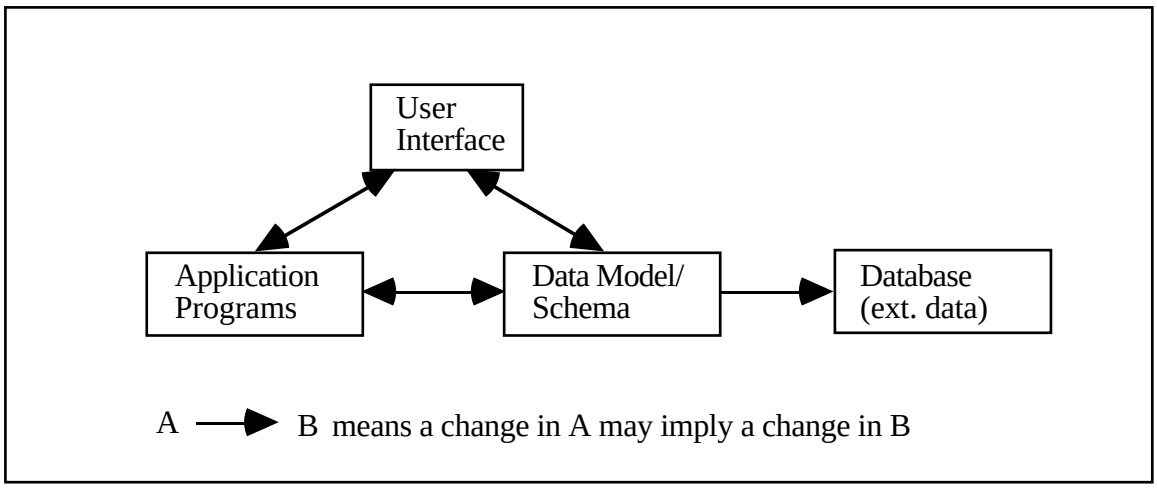

Figure 1: Conceptual model of administrative data-intensive application systems

These systems usually have a long life-time and will constantly evolve as user requirements change therefore they will need to be maintained over a number of years. The following types of systems are beyond the scope of this paper: real-time systems; scientific and numerical programming; and system software.

The remainder of this paper is organised as follows. Section 2 motivates the need for evaluation of maintenance technology in further detail. Section 3 discusses research approaches to such evaluation. In Section 4 we describe work undertaken at Telenor on the evaluation of CASE tools for maintenance; this work provides valuable empirical results which we can use as the basis for further experiments. This is followed by a description of a number of experiments to evaluate maintenance technology; we discuss some pilot studies that are currently being undertaken and also some possibilities for further experiments using maintenance technology that we have developed. Section 5 concludes.

2 The Importance of Maintenance Evaluation

Software maintenance has traditionally been neglected in teaching and practice compared with initial system construction. This applies to system development methods, data modelling techniques, CASE tools, etc. Nevertheless, a range of methods, techniques, language constructs and tools (all of which are termed *technology* in this paper) are being proposed and developed with the purpose of supporting maintenance. However, to the authors' knowledge, there exists no systematic framework for evaluating such technology. At present, sources of "evaluations" are:

- consultancy reports (e.g. Ovum, Gartner Group)
- trade press
- demonstrations by vendors
- subjective assessment from reference customers
- local "gurus"

None of these provide a systematic method of evaluation. The best that can be achieved is usually a checklist of features which are supported (or not supported). There is no systematic measures of maintenance efficiency and a proper cost/benefit analysis is not possible.

In general, the various sub-disciplines of information systems and software engineering need theoretical foundations that can provide [11]: "(1) criteria for evaluation, (2) means of comparison, (3) theoretical limits and capabilities, (4) means of prediction, and (5) underlying rules, principles, and structure."

There is clearly a need for such foundations. We are often in the situation where we want to evaluate methods, techniques or tools according to some purpose, or we want to compare them with each other. For example, consider the important problem of how to

maintain and redesign large information systems, how can we claim that one method or tool is better than another method or tool? Such questions are pertinent both to research and commerce. Research proposals for new technology should be evaluated and contrasted with existing technology in a well-defined framework. Companies and organisations need a framework for evaluation when they are to buy and start using new products.

The *cause* of changes [23] is not particularly important, except as a motivation for our work: since change is inevitable, there is a need for maintenance technology. Swanson divided maintenance activities into corrective, adaptive and perfective maintenance (preventive maintenance has also been added). These categories describe the *reason* for changes requested by end-users or maintainers. One could envisage a taxonomy of the cause of change, being developed by further refinement of Swanson's categories. This is, however, beyond the scope of this paper.

In order to evaluate maintenance technology we do need to identify the *intent* of a change so that this can be linked to the actual process of making changes. For example, we can automatically log what operations a user is doing but from log data we have no idea what they intended to do and whether they are making effective use of the technology.

3 Methods to Evaluating Maintenance Technology

In this section we propose research methods or approaches to evaluation of maintenance technology. We first give an overview and then discuss taxonomies and experiments in further detail.

3.1 Research Approaches

This paper emphasises empirical approaches to evaluate maintenance technology. However, evaluation of maintenance technology will always include non-empirical (theoretical/analytical) work. It is, for example, meaningless to report observed results if there are no common (or well-described) conceptual model to explain and communicate what is observed.

Measurement of maintenance technology is not as "simple" as measurement of length or weight, because we lack a commonly accepted or well-defined empirical relational system. For this reason, we need more than measures for a meaningful maintenance technology evaluation. Below some requirements and corresponding research approaches are proposed:

i) We need a basic understanding of the maintenance technology and its environment (i.e. we need to develop an understanding of the studied empirical relational system). A problem here is to know how much previous (*a priori*) understanding that will be necessary to develop models, measures and interpretations.

   Possible research approaches: Observation of the use of the maintenance technologies, introspection of one's own impressions and feelings when using the maintenance technologies, observation and analysis of the design and implementation of the maintenance technology, previous understanding of and experience in similar technologies and environment, etc.

ii) We need definitions and classifications of important characteristics of the maintenance technology and its environment, i.e. we need to model our knowledge of the empirical relational system (developing a commonly accepted universe of discourse). Here, it is important to consider how measurement theory restricts the use of statistics and formal treatment according to how the characteristics are modelled.

   Possible research approaches: Modelling work, i.e. definitions, classifications, taxonomies, frameworks, design methods, etc., based on own or others knowledge of the maintenance technology and its environment (the reported work is supposed to extend this knowledge).

iii) We need to develop new or use previously developed measures (i.e. we need to develop the mapping between the empirical and the formal relational system) that

enable analyses and interpretations of the measured values. A maintainability measure may, for example, only be meaningful if the commonly accepted "intuition" of maintainability is preserved into our formal empirical system.

Possible research approaches: Analytical work, i.e. deductions, inductions, consequence analyses, etc., that combines knowledge about the "real world" and knowledge of formal techniques to manipulate the measured data (statistics, calculations, scale theory, etc.).

iv) We need to develop a method for the measurement (i.e. we need to develop a means to evaluate the quality and interpretability of the measured and calculated values).

Possible research approaches: Introspection, case study (one phenomenon measured only once in an uncontrolled environment), repeated observations in an uncontrolled environment, experiment (repeated or not repeated observations in a controlled environment) or combinations of these approaches.

The relationships between the actions described in i) to iv) are complex. In principle, the knowledge achieved through one action may impact all other actions. For example, the result of one maintenance technology evaluation may increase the knowledge of the studied empirical relation system, which may lead to more useful models and better measures, which may lead to better measurement frameworks and so on. Therefore, iterations may be necessary in order to establish useful evaluations/measurements.

3.2 Taxonomies

There is generally a lack of taxonomies in computer science. A discussion of the characteristics and use of some of those that exist can be found in [8]. This section describes preliminary work on a software maintenance taxonomy; a detailed definition is an issue for follow up work. The motivation for such a taxonomy is to provide a means to obtain a better understanding of the potential change processes. In particular, we can inspect each kind of change and consider how it is supported by the maintenance technology being evaluated.

A maintenance task may involve changes in all of the four components of an application system as described in Figure 1: data model/schema, database/extensional data, application programs and user interface. For changes in each of these components we may develop a change taxonomy. We call a collection of such change taxonomies a *maintenance taxonomy*.

Schema evolution taxonomies for object-oriented databases have been developed [2], although they are not complete. Table 1 lists the kind of changes that may occur in a data model described in the Enhanced Entity-Relationship model [5]. A further discussion can be found in [18].

Addition, change and deletion of default values, definitions, etc. are examples of changes at the extensional data level. Developing a change taxonomy for application programs requires changes to be described at a fairly high level. An example of classification in terms of changes in code can be found in Jørgensen's earlier work [9].

Table 1: Data model changes

Attribute Changes	Entity (class) Changes
• add attribute	• add entity
• delete attribute	• delete entity
• change type of attribute	• merge entity
• rename attribute	• split entity
• move attribute	• rename entity
• copy attribute	• move entity
	• add superclass
	• remove superclass

A taxonomy of potential user interface changes will include addition, change and deletion of items such as fields, labels, buttons, menus, screens, etc. Certain kinds of constraint checking and error fixing of user input may also be captured in a user interface change taxonomy.

A maintenance taxonomy is supposed to denote a complete set of changes that can be carried out. Although developing complete change taxonomies may be very hard (for example, they may become very large), our aim is still to enhance the notion of taxonomy to also include annotations of the frequency of each change (based on empirical data) and some indication of how difficult it is to carry out the change. Moreover, we also consider the possibility of including in an enhanced taxonomy the consequences of each change in one of the components on the other components.

3.3 Experiments and Benchmarks

The usability of maintenance technology can be analytically evaluated applying a maintenance taxonomy for software changes. However, to get a better picture of the usability, the analytical evaluation should be complemented with experiments. These may include both case studies and controlled experiments, both of which may use benchmarks. An example of a maintenance benchmark is described in [10].

Whereas a taxonomy is supposed to capture a complete set of cases, a benchmark is supposed to be a selection of a few, but typical cases. Hence, a maintenance benchmark denotes an incomplete set of changes/maintenance tasks in the sense that only a few, but typical tasks are included, enabling maintainers to perform them in practical experiments. To give a flavour of the kind of tasks, the two first tasks of the benchmark are as follows. Task 1: "Development of a new, read-only, user screen with data from an existing database table (or object class) with 5 attribute types." Task 3 is more complex: "Development of a new read-only screen with data from Table X (5 attribute types, X.1 to X.5) and Table Y. Table X has to be created, Table Y exists and has 3 attribute types (Y.1 to Y.3). There is a foreign key from X.4 to Y.1. The new user screen should present the attributes in the following sequences: X.1, X.2, X.3, Y.2, Y.3, X.5."

Other differences between a maintenance taxonomy (MT) and a maintenance benchmark (MB) include:

- MB takes an empirical approach and measures change costs in terms of time and possibly the number of code lines changed. MT takes an analytical approach and attempts to evaluate the maintenance technology in terms of a detailed inspection of its support for each given change and change propagation through the given application. For example, for a given schema change, MT should have an idea of necessary consequential changes on application programs, user screens and extensional data (e.g., adding a new attribute requires a sensible default value for the existing entities).
- Consequential changes should also be specified for the user interfaces and application programs. The MB does this implicitly. For example, Task 3 of Jørgensen's MB described above implies a schema change (specified in the task description) and also changes to application programs (not specified in the description).
- In practice, characteristics (size, complexity?) of the given application will influence the support given by the maintenance technology being evaluated. This is implicit in a MB evaluation, but should be explicit parameters when performing an analytical evaluation (see Table 2 below).

There are several approaches to conducting experiments:

i) Real (requested) tasks + actual effort

ii) Real tasks + actual effort + benchmark tasks (undo)

iii) Real tasks + estimated effort + benchmark tasks (undo)

iv) Student experiments + actual effort + benchmark tasks

The purpose of applying benchmark tasks in addition to the real tasks in (ii) and (iii) is to calibrate the experiment. The benchmark tasks must in these cases be undone since they should not have any affect on the operational system.

In addition to the specific measurements of effort collected when conducting the experiments, we should identify general properties such as those listed in Table 2. A particular concern is the major limitations of industrial experiments that the perturbation of the system under study must be minimised and a low risk strategy with regard to the overall production process must be adopted. For example, when carrying out an empirical study of the work of maintenance programmers, there must be minimal disturbance of their work and any experiment must not add to the risk of failing to deliver a modified product. In a university environment, using students as experimental subjects, it is necessary to ensure that any experiment is part of the students' learning experience and that their interest in the subject is not reduced by involvement in boring (and apparently meaningless) experiments.

Table 2: Examples of properties of application system, task, maintainer and experiment

Application System	**Task**
• Application domain • Application size • Application age • Implementation technology • Supporting technology • Team size • Frequency of various kinds of change • Difficulty of making the change (special issues other than time)	• Type {examples?} • Complexity • Language • Size • Code change {class/kind?}
Maintainer	**Experiment**
• Experience • Education • Skill	• Duration • Scale • Control cases • Resources • Procedures • Analysis • Invariants • Perturbation (of industrial systems) caused by experiment – study things that are happening versus building an artificial experiment • Typical versus random tasks

Frequently, empirical data has been collected by the use of questionnaires or manual transcriptions of interviews. We have recently been focusing on how such data collection can be automated. Section 4.3 describes a technique for automatic logging of tool operations and other kinds of information. A long term goal is to supplement such logging by exploiting program and persistent store information recorded in the thesaurus [17]. The thesaurus can be used to collect data on the kind of changes that are occurring by taking snapshots of a system as it evolves and recording deltas between the units of change we want to measure [17].

4 Applying the Methods

This section describes ongoing and planned experiments or pilot studies that apply our evaluation methodology to assess technology that is supposed to support maintenance. The purpose of the experiments is twofold. First, conducting the experiments will demonstrate the potential usefulness of the evaluation methods, and the results will thus feed back into the design of the methods. Second, the evaluation of the maintenance technology itself is also important.

Section 4.1 reports on a finished experiment at Telenor, Section 4.2 and 4.3 experiments in progress and Section 4.4 planned experiments.

4.1 Evaluating Maintenance Efficiency of CASE Tools

Recently an evaluation of development and maintenance efficiency of CASE tools was carried out at Telenor, see [10] for a full description. The three main purposes of this experiment were:

- to develop and validate a method for the evaluation of how CASE tools impact software development and maintenance efficiency.
- to study the impact of a few selected CASE tools on software development and maintenance efficiency in Telenor.
- to provide a "database" of efficiency values to be used for estimation purposes.

The development efficiency measures were based on the MkII Function Point (MkII FP) measure, together with the measures Effort (E), Time (T), and Lines Of Code (LOC). The variables are described below:

- *MkII Function Point* (MkII FP) is a measure of the size of the applications in terms of its user required functionality.
- *Effort* (development) is a measure of the total effort spent on a project, from the requirement phase to the installation phase, including user participation in the requirement and test phases.
- *Effort* (maintenance) is a measure of the estimated effort (average case) to solve a specified maintenance task (a benchmark task). The effort estimate should include requirement analysis, design, implementation, inspections and tests.
- *Time* is a measure of the total calendar time on a development project, from the requirement phase to the installation phase.
- *Lines of Code* is a measure of the number of *physical* lines of source code *written* (i.e. not generated) by a developer. Comments were not counted. Reused code was only counted if there were changes on the reused code. All writing of source code was counted, inclusive the number of physical lines of code written in CASE tool forms (tool screens). Menu selections, navigation, etc. were not counted.

Based on the basic measures we defined a set of *development efficiency measures*, which are described in [10]. Both the basic and the development efficiency measures were exploited in the simple, yet powerful, technique we developed for determining the maintenance efficiency, the Software Maintenance Benchmark Task Technique:

1. *Determine the common characteristics of the applications maintained applying the CASE tools.* In our study, all of the applications were connected to a database and had screens for user input and output.
2. *Develop typical maintenance tasks (the benchmark maintenance tasks) which are meaningful to carry out on all the applications,* i.e. maintenance tasks based on the common application characteristics from step 1.
3. *For each of the selected applications, ask experienced maintainers to estimate the on-average case effort needed and the on-average case LOC to be written to solve the task.* It may be even better to ask the maintainers to carry out the maintenance tasks and then measure the effort and LOC. However, this may not be possible in a professional maintenance environment.
4. *Compare and analyse the estimates.*

Although this approach is simple we have not seen similar approaches to evaluate maintenance efficiency. As opposed to traditional "maintainability metrics" our technique is based on the assumption that maintainability is not a characteristic of the application *alone*, but of the application, the tools and the maintenance tasks to be carried out. Our main indicators of maintenance efficiency are the sum of estimated effort and the sum of the estimated size for the maintenance tasks on an application.

4.2 Investigating Documentation and Maintenance Practices and Needs

We have been carrying out a case study at a major Norwegian software house to investigate hypotheses on documentation and maintenance practices. Information from approximately 40 maintenance tasks conducted by 25 maintainers have been collected. Examples of hypotheses being tested are:

- "Understanding the existing system counts for the majority of the time spent on maintenance work."
- "The majority of documentation is produced to satisfy *quality assurance* requirements and is not directly used for development or maintenance activities."
- "Written documentation is used less than personal communication."

Some of the hypotheses testing is a replication of some of Jørgensen's work in another context [9]. Comparing results from two different companies is a significant part of the experiment. The analysis is not yet finished, but we have preliminary results confirming some of Jørgensen's results and other results that indicate differences. The complete experiment is described in [13].

4.3 Maintaining a Napier88 Application

This section describes an experiment in a controlled environment whose results are currently being analysed. The activities of programmers in the Glasgow Napier88 [14] community were automatically logged during the experiment. We will describe the logging technique, the kind of data analysis we can produce and the kind of tasks carried out in the experiment.

Napier88 programs can be manipulated within a program development environment called the *Persistent Workshop* [22]. A Persistent Workshop log records, for each operation executed on the current WorkItem (the units on which the Workshop users are working. e.g. a source text):

```
username  timestamp  tool  operation  workitem
```

Such a log gives us a detailed picture of who is doing what and when. We have just started analysing the log entries recorded for 11 users in a period of about 6 weeks (totalling about 9400 entries). Examples of analysis include use frequency of tools and operations (cf. Sjøberg's analysis of use of Napier88 language constructs [20]). Table 3 shows an example from which we can identify the relative usage of the various tools in the Workshop. Popular tools and operations may be the subject of optimisation. Moreover, we may ask questions about why a particular tool is heavily used or, alternatively, why it is not used as much as we would expect. For example, the importer is a tool which imports text (program or data) from Unix files into the persistent store – why is it used so heavily? One possibility is that users are creating and editing source files outside the persistent store and importing them for processing. We have anecdotal evidence that this is at least part of the reason for the high usage of the importer, because the editors available in the Unix environment (such as *emacs*) are much more powerful than the Workshop editor.

The groupExporter provides a mechanism for exporting groups of WorkItems from the Workshop into Unix files. The very low usage of this tool, especially when contrasted with the high usage of the importer, suggests that the users did not know about it or perhaps it was unreliable! Of course, we also need to consider the interaction between the maintenance tasks being undertaken and the tool usage profile. Some of the tasks may be unlikely to require much use of certain tools. We could attempt to calibrate this by getting experienced Napier88 programmers to carry out the same tasks and study their tool usage profiles.

Table 3: Frequency of use of the tools in the Persistent Workshop

Tool	Frequency	Percent
importer	2232	35.1
linker	1840	28.9
compiler	1178	18.5
editor	732	11.5
exporter	251	3.9
duplicator	85	1.3
remover	23	0.4
groupExporter	13	0.2
editorDeclSet	2	0.0
Total	6356	100.0

The relative usage of certain groups of tools is also interesting. Again, looking at Table 3 – why is the compiler used more often than the editor and the linker significantly more than the compiler, when we might expect users to carry out an edit-compile-link cycle of operations? We can only really begin to answer this type of question by studying sequences of operations and we show some preliminary results of pairs of successive operations in Table 4. This table shows which operations (columns 2-7) immediately follow a given operation (column 1).

The operation change WorkItem ('changeWI') is an implicit operation generated when the user changes the current WorkItem. That means, entries in the changeWI row show the first operation of a new sequence of operations on the selected WorkItem. The combination of changeWI and save indicates that the user has changed the current WorkItem, edited the newly selected WorkItem and saved it.

The full table is too large and detailed to fit into the paper. Thus we have displayed several combinations of operations which are of interest and collected the remaining operations in the 'other' row and column. Most of the operations in the 'other' row and column relate to import and export of WorkItems (1259 of the 1355 in the 'other' row and 1278 of the 1374 in the 'other' column). As we discussed earlier, we believe these large numbers are related to the lack of the sophistication of the Workshop editor.

Identification of repeated operational sequences may call for a command that includes several operations (e.g. compile & link). Moreover, the log information may identify the need for user education ('improving the user'!). Identification of common errors may also call for user education, but may also indicate required improvements in the programming environment.

Table 4: Frequency of operations succeeding a given operation

Prevop\op	changeWI	compile	link	remove	save	'other'	Total
changeWI	0	1148	1240	13	311	394	3106
compile	197	61	949	2	282	461	1952
link	1663	17	60	0	25	442	2207
remove	20	2	0	0	0	1	23
save	163	429	0	0	83	46	721
'other'	972	303	12	8	30	30	1355
Total	3015	1960	2261	23	731	1374	9364

The Napier88 logging technology and the Persistent Workshop have both been developed in Glasgow and so we have control over the technology being used in the experiment described below. We will emphasise that this experiment is a pilot study that aims to improve the evaluation technology; the results of this study may suggest alternative

logging mechanisms or modifications to the Workshop structure, which may make future experiments more meaningful.

The logs in their present form are too low level and cannot be linked to the higher level (more abstract) changes that are being implemented. We could attempt to mark the log at the beginning and end of a change task, but this would involve either having an observer sitting with the programmer or relying on the programmer's discipline to record what they are doing. The first of these options is infeasible because it is too labour intensive (cf. 'think-aloud' protocols in HCI research). Experience with other tools, such as RCS, suggests that programmers cannot be relied upon to record what they are doing or why they are doing it!

Munros – Maintenance Tasks

The experiment was carried out with five groups of students, four groups of two and one group of three. The students were given a large application written in Napier88 which had been built as a demonstrator for an ESPRIT project ($FIDE_2$). This application had been written by several different programmers and consists of about 6000 lines of code in 100 modules. Before starting the experiment the code was tidied up to try to remove gross anomalies in the structure.

The application concerns the storage and display of information about *Munros*, Scottish mountains which are over 3,000 feet in height, which were originally listed by the Reverend Munro in a famous set of tables. Some hillwalkers and mountaineers in Scotland set themselves the target of climbing all the Munros and this application records details of which mountains have been climbed by a particular climber and with whom. Details of 288 Munros are stored and displayed by this application. The data also includes photographs of some of the Munros which are stored as images in the database. The data is bulk loaded from a spreadsheet giving details of all the Munros and their current status; the images are added separately after the basic data has been loaded.

The data is displayed using a *starfield viewer* [1] which allows various queries to be formulated using sliders. The main display representation is an outline map of Scotland with the locations of the Munros depicted by small icons. The icons vary in size to show the relative heights of the mountains and are colour coded to show whether they have been climbed or not. The data displayed can be modified by zooming the map viewing area or by using a variety of sliders, for example: with whom climbed or when climbed. The details of any individual mountain can be displayed in a separate dialogue box.

The students were given a set of five tasks to make changes to the interface; each task was progressively more difficult than the previous one but they could all be carried out independently. Students were required to complete a task and demonstrate the completed changes before moving onto the next task. While the tasks were being carried out, the students' use of software tools in the Persistent Workshop was being automatically logged. The final state of their software was also captured so that the changes made could be analysed. The five tasks were:

- Change the check boxes *done* & *not done* to be labelled *climbed* & *not climbed*, respectively. Required a simple change of text labels in the interface.
- Icons for mountains are currently differentiated by colour, blue for climbed and red for unclimbed. Change these icons to use *shape* encoding. Required a change of presentation of database state involving the modification of icons.
- Add further encoding to the icons to indicate which of the unselected mountains have associated photographs in the persistent store. This is useful if when building up a slide show of mountains. Required a change of representation using additional information from the persistent store combined with information about the current state of the interface.
- Add a new dialogue to the Munro details window which allows the user to enter the date of ascent, and with whom climbed. Commit the change if details are confirmed by the user and the date is acceptable. Required a new dialogue to update an existing data structure. The date and with whom climbed data is currently stored in the persistent store but cannot be updated interactively.

- The input data used to build the persistent data structure includes a list of maps (Ordnance Survey *sheets*) which show a particular Munro. The current application only stores and displays the first sheet on this list. Enhance the application to read, store and display the list of all relevant sheets for a particular Munro. Required a modification to the persistent storage structure, the bulk load routine and the display interface.

The tasks were chosen to represent similar types of changes to those proposed by Jørgensen [10]. Although the environments are different, it is possible to map these Napier88 tasks onto similar maintenance tasks identified in a commercial environment. After carrying out a full analysis of the logs and code from this year's experiment, and having rectified any weaknesses in our measurement technology, we intend to design and run a similar experiment during the next academic year.

4.4 Maintenance Technology to be Evaluated in Future Experiments

This section reports on technology supposed to support maintenance that the authors have been and are currently working on. For the work that is still going on, it is at present too early to conduct experiments to evaluate the potential maintenance benefits, but we will briefly describe the technology and outline how it may be evaluated.

We have developed a tool, the Builder, which supports rebuilding activities such as recompilation and linking [21]. The Builder is implemented in Napier88 and provides build management, including incremental linking, for applications written in Napier88. It exploits features such as strong typing, run-time linguistic reflection, and referential integrity provided by the language processing technology. The Builder operates over both programs *and* (complex) data, which is in contrast to conventional language-centred tools.

We have also been involved in work on supporting maintenance by defining a coherent set of (optional) rules and conventions that should be adhered to in a given programming environment and that which are explicit and checkable [19]. A particular such a set is the Structured Persistent Application System Model (SPASM), which is tailored to the programming environment of Napier88. A requirement is that such constraints are automatically checkable, and we have implemented a tool called SPASMCheck. We plan to conduct experiments similar to the one outlined in the previous section, aiming to evaluate the effect of using the Builder and SPASMCheck, and thus indicate how they can be improved.

Another project is ZEST (Zoned Evolvable System Technology), which is concerned with the architecture of large persistent application systems (PASs). Our basic premise is that many PASs are composed of zones which are the units of evolution of the application system. A zone is usually related to the structure of the organisation using the PAS; a zone may be implemented using heterogeneous technology and is continually evolving internally. However, each zone has clearly defined interfaces to other zones and these interfaces change comparatively infrequently, normally by negotiation between the owners of inter-connected pairs of zones. The main objective of ZEST is to define the inter-zone architecture of a PAS and to study the use of zones as the units of software evolution. The work on the maintenance project reported in this paper is of interest for two reasons: we can get useful information on the types of changes which take place at the interfaces between zones; and we expect other types of maintenance technology to be used within the zones. ZEST is funded by the UK EPSRC, for three years from 1 January 1996, and more details about the project can be found on WWW [7].

The Builder, described above, uses *dependency graphs*, automatically inferred from an application's structure, to systematically rebuild the application. However, at present, the dependency graph is not presented to the user in a graphical form. We believe that providing a graphical representation of dependency graphs and associated tools will be a useful aid to the programmer. As build management is an important component of maintenance, we hypothesise that providing visualisations of dependency graphs will improve the efficiency of maintenance. Therefore, one of our long-term aims is to evaluate the effectiveness of these visualisations as an aid to maintenance, which would fit within the framework of the present project.

The visualisation of dependency graphs will provide a challenging project for Master's student on an Advanced Information Systems course at Glasgow. For a realistic application the dependency graph will be large and have many connections. There are some interesting problems in presenting such large graphs including: how to scale or partition the graph; providing different views of the graph (for example, using 'fish-eye' views); using colour to highlight certain features of the graph or to show the impact of changes. It is also possible to analyse the dependency graph looking at the connectivity of the graph or how components are clustered; measures of this type might give is some measure of the 'maintainability' of the application. We may also be able to identify partitions of the graph which allow maintenance activities to be carried out safely in parallel on different parts of an application.

A typical application will use many standard library components and the visualisation of an application will be simplified if we can 'prune' the graph to remove immutable components which are provided and not modifiable by the programmer. (This is connected to some other work we are doing on copying parts of applications between stores and how we can safely cut the links to standard components.)

5 Conclusions

The work reported in this paper is a result of a project on evaluation of software maintenance technology. We first discussed approaches to solving this problem, including analytical work on developing taxonomies of changes in various components of the kind of systems in focus, namely administrative, data-intensive systems. The emphasis of our work, however, has been on empirical studies. The use of a benchmark of maintenance tasks proved useful in an industrial experiment on evaluation of CASE tools. Another study is currently being undertaken in a major Norwegian software house to test hypotheses on maintenance. Yet another experiment, also based on a set of predefined maintenance tasks, were carried out in a research and teaching context of the persistent programming language Napier88. As part of this experiment, we developed a new technique for automatic logging of commands. Finally, we outlined several experiments aiming to evaluate tools that potentially support maintenance.

Within the present project there were severe resource limitations. It was only possible to carry out pilot studies of limited size and short duration, and it was only possible to work with available systems and tools that may not be the ideal choices for such studies. Moreover, we were constrained to work with the subjects who were available, regardless of their experience. However, in spite of these limitations we are going on with experiments, for example, assessing the impact of people on the maintenance process, the efficiency of experienced maintainers versus novices, and we hope to produce further interesting results.

Acknowledgements

The authors benefited from a collaboration project between Norway and UK funded by the Research Council of Norway and British Council.

References

[1] Ahlberg, C. and Shneiderman, B. “Visual Information Seeking: tight coupling of dynamic query filters with starfield displays”. In: *ACM CHI'94 (Boston, MA, 24–28 April, 1994)*, pp. 313–317.

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

[3] Boehm, B.W. *Software Engineering Economics.* Englewood Cliffs, N.J. Prentice-Hall, 1981.

[4] Chikofsky, E. and Cross, J. “Reverse Engineering and Design Recovery: A Taxonomy”. *IEEE Software*, Vol. 7, No. 1, pp. 13-17, January 1990.

[5] Elmasri, R. and Navathe, S.B. *Fundamental of Database Systems*. Benjamin/Cummings, 1989.

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

[7] http://www.dcs.gla.ac.uk/zest/. Persistence and Distribution Group, Department of Computing Science, 1996.

[8] Glass, R.L. and Vessey, I. “Contemporary Application-Domain Taxonomies”. *IEEE Software*, pp. 63–76, July 1995.

[9] Jørgensen, M. “An Empirical Study of Software Maintenance Tasks”. *Journal of Software Maintenance*, Vol. 7, No. 1, pp. 27–48, January–February 1995.

[10] Jørgensen, M., Bygdås, S.S. and Lunde, T. Efficiency Evaluation of CASE Tools – Methods and Results. Scientific Report TF R 38/95, Telenor FoU, August 1995.

[11] Leveson, N.G. “High-Pressure Steam Engines and Computer Software”. *IEEE Computer*, pp. 65–73, October 1994.

[12] Lientz, B.P. and Swanson, E.B. *Software Maintenance Management: A Study of the Maintenance in 487 Data Processing Organizations*. Addison-Wesley Publishing Company, Reading, Mass., 1980.

[13] Martinussen, J.P. A CASE Study of Software Maintenance in a large Norwegian Software House. M.Sc. Thesis, Department of Informatics, University of Oslo, 1996.

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

[15] Nosek, J.T. and Palvia, P. “Software Maintenance Management: Changes in the last Decade”. *Journal of Software Maintenance*, Vol. 2, No. 3, pp. 157–174, 1990.

[16] Putnam, L.H. “Software Cost Estimating and Life Cycle Control”. *IEEE Catalog*, 1982.

[17] Sjøberg, D.I.K. “Quantifying Schema Evolution”. *Information and Software Technology*, Vol. 35, No. 1, pp. 35–44, January 1993.

[18] Sjøberg, D.I.K. “Towards a Methodology for Evaluating Software Maintenance Technology”. In: *18th Information systems Research seminar In Scandinavia (IRIS18) (Gjern, Denmark, 11–13 August, 1995)*, Dahlbom, B. *et al.* (editors), pp. 653–663, Gothenburg Studies in Informatics, Report 7, June 1995.

[19] Sjøberg, D.I.K., Atkinson, M.P. and Welland, R. Maintaining Consistency Using Software Constraints. Preprint Report 1996 No. 1, Department of Informatics, University of Oslo, February 1996.

[20] Sjøberg, D.I.K., Cutts, Q., Welland, R. and Atkinson, M.P. “Analysing Persistent Language Applications”. In: *Sixth International Workshop on Persistent Object Systems (Tarascon, Provence, France, 5–9 September, 1994)*, Atkinson, M.P., Benzaken, V. and Maier, D. (editors), pp. 235–255, Springer-Verlag and British Computer Society, 1994.

[21] Sjøberg, D.I.K., Welland, R., Atkinson, M.P., Philbrow, P. and Waite, C. Exploiting Persistence in Build Management. Software – Practice and Experience, 1997. (to appear)

[22] Sjøberg, D.I.K., Welland, R., Atkinson, M.P., Philbrow, P., Waite, C. and MacNeill, S. “The Persistent Workshop – a Programming Environment for Napier88”. In: *7th Nordic Workshop on Programming Environment Research (Aalborg, Denmark, 29–31 May, 1996)*, Bendix, L., Nørmark, K. and Østerbye, K. (editors), pp. 37–52.

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

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