# Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation

Amela Karahasanović

Thesis submitted for the degree of Dr.Scient.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo

May 29, 2002

## Abstract

There is a growing number of object-oriented systems within areas such as CAD/CAM, software engineering, office automation and multimedia management where changes are rather a rule than an exception. Application systems in these areas are supposed to be easy to change due to encapsulation and inheritance. On the other hand, the transitivity of inheritance and aggregation structures makes it difficult to detect dependencies between classes, methods and fields of a system, which in turn makes it more difficult to change the system.

The research presented in this thesis demonstrates that identifying and visualising impacts of changes in evolving object-oriented systems is a step towards improving the process of maintaining application consistency in such systems. A technology that identifies and visualises such impacts has been developed and evaluated. This technology helps discover dependencies and thus support maintaining application consistency. It is based on a component-based model of object-oriented systems and an improved version of a transitive closure algorithm. This technology identifies impacts of complex changes, i.e., changes involving several classes like merge and split. Furthermore, a visual language allows displaying relatively large amount of objects, both at a coarse level of granularity (packages, classes and interfaces) and at a fine level of granularity (fields and methods). Several empirical studies focusing on the usability of the proposed technology have been conducted. The results of these studies show that visualising impacts of changes improves the effectiveness of developers when conducting changes. Furthermore, the higher precision of impact analysis achieved by identifying impacts at the finer level of granularity (fields and methods) reduces the number of errors and the time needed to conduct changes. Identifying impacts of complex changes (merge, split) improves effectiveness.

Empirical studies are a prerequisite for usability evaluation of any technology, including those supporting application consistency. Such studies, in turn, require efficient and reliable data collection. A tool for automatic data collection during software engineering experiments has been developed. This tool automatically collects data about the subjects and their interaction with the software technology under study. A ‘think-aloud’ screen was proposed as a means for collecting subjective information. High-level subjective data sources (the think-aloud screen and evaluation screen) are combined with a low-level objective data sources (user command logs). It has been demonstrated that this combination may increase the validity of the conducted studies.

## Acknowledgements

Work on this thesis has been supported by a PhD grant from the Department of Informatics, University of Oslo. Additional funding has been provided by the Simula Research Laboratory.

---

I am indebted to my supervisor Dag Sjøberg for his guidance through my PhD studies. I thank him for the enduring support, rewarding discussions, contributions and encouragement. His outstanding knowledge and enthusiasm have made it a great pleasure to be his student.

I am grateful to the members of the Software Engineering group Bente Anda, Erik Arisholm, Gunnar Carelius, Jo Hannay, Magne Jørgensen, Espen Koren, and Marek Vokác for their comments, discussions and enjoyable time spent together. I thank my MSc student Umair Rasool who conducted the Oracle8i study. I also thank the students of the Department of Informatics at the University of Oslo who participated in the visualisation experiment.

Many people in academia and industry have contributed to the research presented in this thesis. I appreciate the generosity of Samantha Shaw and Paul Callaghan at University of Durham who provided us with the technology for connecting Java applications with the graph visualisation system daVinci. I am also very grateful to Kent-Andre Mardal and Åsmund Ødegård at Simula Research Laboratory for implementing the logging mechanism. I gratefully acknowledge the support from Hans Cristian Benestad and Jon Skandsen at Genera AS who provided the source code used for evaluation of SEMT. I am indebted to Malcolm Atkinson and Ray Welland at University of Glasgow and Brian Lings at University of Exeter for valuable comment and discussions that influenced this work.

This work would not be possible without initial help from Ole-Johan Dahl, Beth Hurlen and Olaf Owe at University of Oslo, Arild Jansen, Mona Johnson, Espen Vaager and other members of the Department of Information Technology at Finnmark University College. They helped me not only to be accepted as a PhD student, but also to start my life in Norway from scratch.

Last but not least, I am grateful to my friends and family for their encouragement and support. Thanks to my husband Amir and my son Anes for their love and patience, as well as practical support during all these years.

## Table of Contents

# 1 Introduction

Object-oriented systems are intended to support non-traditional applications within areas such as CAD/CAM, software engineering, office automation and multimedia management where changes are rather a rule than an exception. Application systems in these areas are supposed to be easy to change due to encapsulation and inheritance. On the other hand, the transitivity of inheritance and aggregation structures makes it difficult to detect dependencies between classes, methods and fields of a system, which in turn makes it more difficult to change the system. The research presented in this thesis demonstrates that impact analysis and visualisation can help discover dependencies and thus support change management.

## 1.1 Managing Change — What, Why and How

Database and software systems undergo considerable changes after they have become operational. This is expressed by Lehman's first law of software evolution (Lehman, 1974). It states that software must be continually adapted and changed if it is to remain satisfactory in use. Although the reported figures differ, most researchers on software maintenance agree that more then 50% of programming efforts are due to changes of the system after the implementation (Zelkowitz, 1978; Lientz, 1983; Lehman and Belady, 1985; Nosek and Prashant, 1990; Coleman *et al.*, 1994). It has been reported that the maintenance proportion increased from 35–40% in 1970 to 70–80% in 1980 (Pfleeger, 1987) and that the maintenance proportion was significantly higher as we were approaching the year 2000 (Holgeid *et al.*, 2000). Changes to database structures (Sjøberg, 1993), class libraries (Johnson and Opdyke, 1993; Nakatani, 2001) and structure oriented environments with abstract-syntax trees (Garlan *et al.*, 1994) after a system has become operational may be quite frequent.

Both the database and software-engineering communities have devoted significant research efforts to different aspects of change (Banerjee, 1987; Atkinson, 2000; Bertino, 1992; Bohner and Arnold; 1996; Briand, 2001; Casais, 1991; Kung *et al.,* 1994). However, there is a need for better communication between the sub-communities interested in the problem of change:

> *" ... while there are a number of fields that have to deal with change, each field has, to date, tended to develop its own conceptual framework and to deal with the problem of change separately. For example, changes in process modelling, data modelling, spatio-temporal modelling and the structural aspects of databases each have their own, largely independent literature sources (as evidenced by largely non-overlapping citation trees in the workshop papers).*
>
> (Roddick *et al.*, 2000)

Change related issues can be considered around *what*, *why* and *how* questions. This may enable better exchange of the ideas and solutions within the software change management (Roddick *et al.*, 2000) and to identify best practice (Lehman and Ramil, 2001), Research around *what* questions focuses on the subjects of a change. The subjects of a change can be, for example, data, constraints, database schemata, models and meta-models, documentation, software architecture and class libraries. While the subject of the change may be quite different, many of the approaches for dealing with the change address similar problems such as considering a model for time, the granularity of phenomena and the links between cause and effect, and can thus use the same principles for the solution (Roddick *et al.*, 2000).

Research around *why* questions focuses on the causes of a change. Changes of a subject can be caused by changes in the universe of discourse, changes to the interpretation of facts about the universe of discourse, correction of errors, performance enhancements, improvements to user interfaces and new or changed functionality. Studies have been conducted focusing on types and frequency of changes (Lientz *et al.*, 1978; Lehman and Belady, 1985; Pfleeger, 1987; Sjøberg, 1993; Jørgensen, 1995a). This research has been called *what* and *why* of evolution (Lehman, 2000). Such knowledge may be used to influence work on new models, methods and tools in databases and software engineering.

Research around *how* questions focuses on improving the process of change (Lehman and Ramil, 2001). Numerous solutions have been proposed:

- New paradigms, such as the object-oriented paradigm including object-oriented programming languages (Simula, C++, Java, etc.) and object-oriented design (Booch, 1991).
- Programming guidelines, such as those found in step-wise refinement (Wirth, 1971), structured programming (Dahl *et al.*, 1972) and modularization (Parnas, 1972).
- Improved development processes, such as prototyping (Pomberger, 1991) and incremental development (Boehm and Papaccio, 1988).
- Tools supporting different aspects of change, such as cross-reference tools, test environments, versioning and configuration management systems and CASE tools.

### 1.2 Application Consistency in Evolving Object-Oriented Application Systems

The term *system* can be used to denote a set of programs, hardware-software combinations, specification and design documents and other artefacts involved in the software life cycle (Schach, 1997), software architectures, and systems-of-systems (Mittermeier, 2001). Systems can be described by the paradigm in which the system is built (e.g. object-oriented), its size and complexity. Size can be expressed by the number of lines of code or by the number of groups involved in their development and maintenance (Lehman and Ramil, 2001).

Maintaining *consistency* between different parts of an a system when it undergoes changes is a major issue of the change related research. *Inconsistency* in software engineering can be used to denote any situation in which a set of descriptions does not obey some relationship that should hold among the descriptions (Nuseibeh *et al.*,

1994). Change in one place of a system may propagate throughout the system and introduce inconsistencies (the ripple effect). When inconsistencies are identified, they can be handled in several ways, including resolving the inconsistency immediately, ignoring it completely, or tolerating it for a while (Nuseibeh *et al.*, 2000). Selecting adequate ways of handling inconsistencies is important to reduce risks. The Ariane-5 rocket, for example, reused much of the software from Ariane-4 (Nuseibeh, 1997). An inconsistency that was acceptable in the Ariane-4 software caused the explosion of Ariane-5.

Identifying and handling inconsistencies are challenging. Tools for expressing the relationships that should hold among the software engineering artefacts, check consistency with respect to these relationships, and visualise the detected inconsistencies are envisaged (Finkelstein, 2000).

### 1.2.1 What is an Object-Oriented Application System?

Large *applications systems* are often built around databases. Such systems consist of a *database* where user data is stored, a *database schema* describing the data in the database and *applications* that operate on the data (Figure 1.1). Classes are used for modelling applications in object-oriented systems and a *class hierarchy*[1] plays a central role (Pomberger and Blaschek, 1996). A *class hierarchy* is a collection of classes describing a particular application domain (Casais, 1991). A *database schema* in object-oriented databases can also be defined as a class collection describing an application domain. Both a class hierarchy of an object-oriented application system and a database schema of an object-oriented database application system are collection of classes. The difference between them is that database schema classes are *persistent capable*[2].

Our research is conducted in an object-oriented context. In this context an *application system* means a collection of classes describing an application domain (a class hierarchy or a database schema), applications built around them, and, in the case of persistent capable classes, a database.

[1] Multiple inheritance is not considered here.

[2] ODMG 3.0 (Cattell and Barry, 2000) defines persistent-capable classes as classes with the property that their instances (objects) can be made persistent.

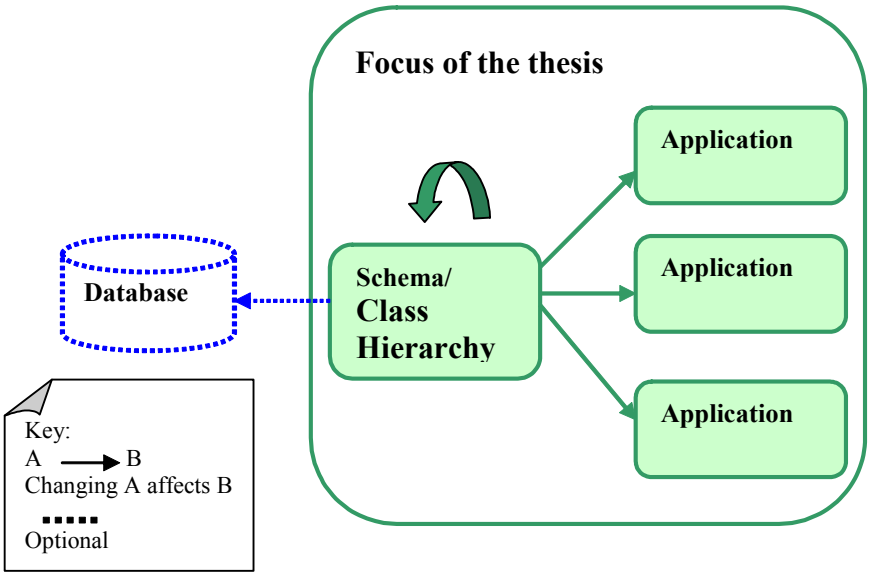



**Fig. 1.1.** Typical system components and relationships between them

### 1.2.2 What is Application Consistency?

*Schema evolution* deals with problems arising when a database schema evolves. Changes to a database schema after a system has become operational will typically affect other parts of the schema, the data in the database and the applications (Sjøberg, 1993). When a database schema is changed, three types of consistency should be maintained:

- *Schema/hierarchy consistency.* The *invariant* properties (Banerjee *et al.*, 1987; Barbedette, 1991; Zikari, 1991) of the schema/class hierarchy should be preserved.
- *Consistency with data in the database,* also called structural consistency (Zicari and Ferrandina, 1997). For instance, if the type of an attribute were modified, it would no longer be consistent with data in the database.
- *Applications consistency*, also called behavioural consistency (Zicari and Ferrandina, 1997). It concerns the dynamic aspects of objects. For instance, if an attribute were deleted or modified, a method referring to that attribute would no longer be consistent with the schema.

*Class evolution* deals with problems arising when a class hierarchy evolves. Class evolution differs from schema evolution in that the former does not consider

consequences of changes at the instance level (structural consistency).[3] Only schema/hierarchy consistency and application consistency should be preserved.

*The research in this thesis focuses on the support for application consistency, i.e., on the support for maintaining consistency between object-oriented database schemata/ class hierarchies and applications built around them (Figure 1.1).*

Several approaches have been developed to support application consistency at the source code level:

- *Tailoring* implies slightly modifying class definitions (by renaming, for example) when the desired changes cannot be achieved by subclassing.
- *Direct schema evolution*[4] (Banerjee *et al.*, 1987; Penney and Stein, 1987; Nguyen and Rieu, 1989; Lerner and Habermann, 1990; Barbedette, 1991; Tresch and Scholl, 1992; Zikari, 1991; Connor *et al.*, 1994; Zicari and Ferrandina, 1997; Dmitriev and Atkinson, 1999; Atkinson *et al.*, 2000) is supported when schema changes are allowed without loss of existing data. The old schema and the old state of the schema extension are inaccessible after the change took place.
- *Schema and class versioning* (Skarra and Zdonik, 1987; Kim and Chou, 1988; Bjørnerstedt and Hulten, 1989; Andany *et al.*, 1991; Monk and Sommerville, 1992; Bratsberg, 1992; Clamen, 1994; Odberg, 1994a; Odberg, 1994b; Fornari *et al.*, 1995; Lautemann, 1996; Lautemann *et al.*, 1997; Lautemann, 1997; Brazile and Dongil, 1995) and schema *evolution by views* (Bertino, 1992; Tresch and Scholl, 1993; Breche *et al.*, 1995; Bellahsene, 1996; Young-Gook and Rundensteiner, 1997). Schema changes produce a new schema version or view. Old application programs can be used with the schema version or the view for which they were designed.
- *Impact analysis* identifies potential impacts of changes on applications before they are carried out.

When tailoring is applied, old applications can be used without modifications. However, the set of possible changes is quite limited. Allowing old applications to work on top of the particular version or the view is very useful when there are insufficient resources to change the applications, or if the source code is unavailable.

The purpose of introducing a change is to use the new schema and the applications should therefore be adapted to the new schema version when possible. With change comes a need to estimate the scope (size and complexity) of the changes (Bohner and Arnold, 1996). Doing this manually is very difficult for large systems. A case study of an industrial object-oriented project showed that the number of classes predicted to be changed was largely underestimated; only one third of the set of changed classes were

[3] The term class evolution is used in the literature to denote evolution of classes without persistent instances. We follow this convention.

[4] This approach is also denoted as *schema evolution* (Roddick, 1995; Jensen and Dyreson, 1998) and *adaptational approach* to schema changes (Ferrandina and Lautemann, 1996). As the term *schema evolution* is sometimes used in the literature to denote both *direct schema evolution* and *schema versioning*, we consider the term *direct schema evolution* to be more precise.

identified (Lindvall, 1997). A major goal of impact analysis is to identify parts of a program (or other software artefacts) affected by a change. Applying impact analysis to support application consistency raises the challenge of how to present the relevant information to the developers.

### 1.3 Goals

The discussion above gives the motivation behind our research. It aims to improve the process of maintaining application consistency in an object-oriented context from the perspective of schema or application developers. One aspect of this process is the difficulty of identifying and presenting impacts of a proposed change. There is a high demand for development of technologies that identify and present impacts of changes. This, in turn leads to a demand for usability evaluation of the proposed technologies. Thus, the goals of this research are:

- to develop a technology that identifies impacts of changes to an object-oriented system and presents this information in a way that improves efficiency, accuracy and user satisfaction of developers when conducting change tasks, and
- to establish a framework for evaluation of technologies supporting application consistency (including empirical studies and related technologies) and to evaluate the existing schema evolution tools within it.

To achieve these goals several research questions have to be addressed:

**RQ1** How do the existing schema evolution tools support application consistency?

**RQ2** To what extent can former work within the field of *software change impact analysis in programming languages* contribute to identify impacts of schema/class changes on the applications?

**RQ3** What is the trade-off between a textual and visual presentation of the impacts of schema/class changes?

**RQ4** How should schema and class evolution technologies be evaluated regarding their support for application consistency?

### 1.4 Contributions

Improving the process of maintaining application consistency requires a systematic understanding of schema and class evolution processes and developing adequate supporting technologies. Means for evaluation of such technologies are also needed. The main contributions of this thesis are:

- It has been demonstrated that identifying and visualising impacts of changes in evolving object-oriented systems is a step towards improving the process of maintaining application consistency of such systems from the perspective of schema or application developers. We have developed the Schema Evolution

Management Tool (SEMT), which supports application consistency in evolving object-oriented systems by identifying and visualising impacts of changes. This technology is based on a component-based model of object-oriented systems and an improved version of a transitive closure algorithm. This allows identifying impacts of changes to changes to classes and changes to the structure of a class hierarchy. The main difference between SEMT and similar tools is that SEMT identifies impacts of complex changes, i.e., changes involving several classes like merge and split. Furthermore, the visual language of SEMT allows displaying relatively large amount of objects, both at a coarse level of granularity of (packages, classes and interfaces) and at a fine level of granularity (fields and methods). Several empirical studies focusing on the usability of SEMT have been conducted. The results of these studies show that visualising impacts of changes improves effectiveness of developers when conducting changes. Furthermore, the higher precision of impact analysis achieved by identifying impacts at the finer level of granularity (fields and methods) reduces the number of errors and the time needed to conduct changes. Identifying impacts of complex changes improves effectiveness.

- Research in schema and class evolution has primarily been driven by the needs of schema and application developers. This thesis proposes a framework for evaluating technologies supporting consistency in object-oriented applications, that is a basis for empirical evaluation of schema and class evolution technologies. The framework focuses on the attributes of the technologies that are directly relevant for the users, namely functionality and usability. Usability can be expressed in terms of certain measures (*e.g.,* the time needed to conduct a schema change). The need for support for complex changes was identified. Furthermore, the various kinds for presenting change impacts were envisaged.

- To help conduct empirical studies on usability of technologies supporting application consistency, we have developed a tool for automatic data collection during software engineering experiments. This tool automatically collects data about the subjects and the interaction between the subjects and the software technology under study. A ‘think-aloud’ screen is proposed as a means for collecting subjective information. High-level subjective data sources (think-aloud screen and evaluation screen) are combined with a low-level objective data sources (user command logs). It has been demonstrated that this combination may increase the validity of the conducted studies.

There is a limited number of empirical studies focusing on the usability of schema and class evolution tools. We believe that the results of the usability studies described in this thesis contribute to increase the understanding of use of schema and class evolution technologies and to provide feedback on their development. Achieving generality and triangulation across research disciplines in usability studies is difficult (Karat *et al.*, 1998). However, we believe that our results can also be of interest for other communities dealing with visualisation in, for example, software restructuring and reverse engineering.

## 1.5 Thesis Statement

Maintaining application consistency during schema and class evolution is a non-trivial task. Identifying and visualising impacts of changes in evolving object-oriented application systems is a step towards improving the process of maintaining application consistency in such systems. A proposed technology based on impact analysis may improve this process by visualising the impacts. Presenting impacts of changes as a graph is more effective and leads to higher user satisfaction than does presenting the impacts in a textual form. Identifying and presenting impacts of complex changes improves effectiveness. Presenting change impacts at a fine granularity level (fields and methods) reduces the time needed to conduct changes and leads to fewer errors than presenting impacts at a coarse granularity level (classes).

Empirical studies are prerequisite for usability evaluation of any technology including those supporting application consistency. Such studies, in turn require efficient and reliable data collection. Technologies that automatically collect data about subjects, their interaction with and evaluation of the technology under study simplify data collection and analysis during the studies and increase their validity.

## 1.6 Thesis Outline

This thesis is organised in seven chapters. In addition, there are four appendices giving details on the conducted studies. Table 1.1 gives an outline of the thesis.

**Table 1.1. Thesis outline**

| Chapter | Content |
|---|---|
| Chapter 1 | **Introduction –** presents the motivation and the goals of the research presented in this thesis. |
| Chapter 2 | **Evolution in Object-Oriented Systems –** gives an overview of concepts of database technology and software engineering with emphasis on those relevant for evolving object-oriented systems and describes information visualisation, which can be used to support evolution. This chapter also proposes a framework for validation of technologies regarding their support for application consistency. |
| Chapter 3 | **Research Methods –** describes research methods used in software engineering that are also used in this thesis. This chapter also describes a tool we have developed for automatic data collection during empirical studies in software engineering. |
| Chapter 4 | **Supporting Application Consistency – Examples from Current Practice –** describes the results of a survey of current practice in maintaining application consistency and a study of one commercial system. |
| Chapter 5 | **SEMT: Schema Evolution Management Tool** – describes the Schema Evolution Management Tool (SEMT), a technology that supports application consistency in evolving object-oriented systems by identifying and visualising the impacts of potential changes. |
| Chapter 6 | **Evaluation of SEMT** – describes the results of studies conducted to evaluate the proposed technology. |
| Chapter 7 | **Conclusions and Future Work** – summarises the results and contributions of the thesis. Future work related to the main contributions is outlined. |
| Appendix A | This appendix describes questionnaires of the logging tool (Section 3.4.2). |
| Appendix B | This appendix describes experimental material from the state of practice survey (Section 4.1). |
| Appendix C | This appendix describes the application used in the study of a commercial system (Section 4.2). |
| Appendix D | This appendix describes the experimental material and raw data from the visualisation experiment (Section 6.2). |

## 2 Evolution in Object-Oriented Systems

Supporting application consistency in evolving systems necessarily employs knowledge from several research areas. This chapter describes concepts of database technology and software engineering with emphasis on those relevant for evolving object-oriented systems. These are *schema/class evolution*, and *software change impact analysis*. The chapter also describes *information visualisation*, which can be used to support evolution.

*Database technology* comprises the concepts, systems methods, and tools that support the construction and operation of databases, including data modelling, persistence, data integrity, querying, concurrency control, security, secondary storage management, optimisation and heterogeneity (Dittrich *et al.*, 2000). *Schema evolution* is a specific field of database application development dealing with problems arising when a database schema evolves.

*Software engineering* is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software (IEEE, 1990). In the area of software engineering, evolution of software systems has been studied from different viewpoints. Software change *impact analysis* deals with consequences of changes, whereas *class evolution* deals with evolving class hierarchies.

*Information visualisation* deals with how to depict an extract of an information space, how to visually communicate important features of data (Preim, 1998). Figure 2.1 illustrates how these research areas are related.

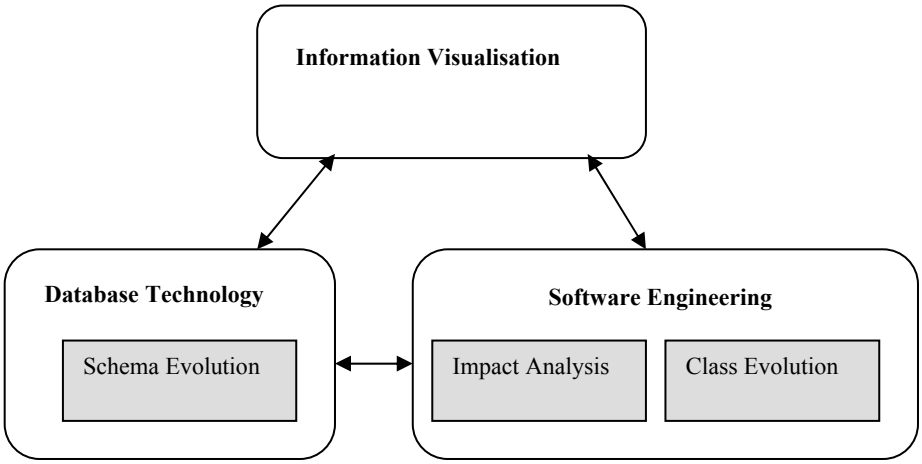



**Fig. 2.1.** Relationships among information visualisation, database technology and software engineering

The remainder of this chapter is organised as follows. Section 2.1 discusses schema and class evolution. Section 2.2 proposes a framework for evaluation technologies that support application consistency. Section 2.3 describes concepts in the field of software change impact analysis. Section 2.4 presents the concepts in the field of information visualisation. Section 2.5 summarises.

## 2.1 Class and Schema Evolution

Classes are used for modelling applications in object-oriented systems. A *class hierarchy* is a collection of classes describing a particular application domain (Casais, 1991). A *database schema* in object-oriented databases can also be defined as a class collection describing an application domain. An overview of definitions of database schema is given in Table 2.1. A formal definition of a database schema corresponding to a model of object-oriented database application system that we propose is given in Section 5.3.

**Table 2.1.** Definitions of database schema

| | |
|---|---|
| (Elmasri and Navathe, 1994) | A database schema (the meta-data) is a description of a database. |
| (Odberg, 1995) | A database schema is a collection of class and relation[5] definitions, with the classes organised in a hierarchy. The schema reflects a description of a particular domain, and is the basis for storing information about this domain in a database. |
| (Li, 1999) | A database schema is a set of classes with a set of relationships existing between classes. |
| (Riccardi, 2001) | A schema is a precise description of the structure of some collection of similar objects. |

*Class evolution* deals with problems arising when a class hierarchy evolves; *schema evolution* deals with problems arising when a database schema evolves. Since a database schema is usually globally defined, changes to the schema may have large impact on the applications. Therefore, schema evolution has been extensively investigated in the area of databases.

Both class evolution and schema evolution[6] are concerned with preserving consistency within a class hierarchy. Schema evolution also deals with consequences on the data in the database. There are also some other differences between class evolution and schema evolution. Class evolution deals with internal changes of a

[5] In (Odberg, 1995), a relation is defined as a named association between two classes, which establishes that relationships of this relation may be created to relate objects which are instances of the respective classes.

[6] If not otherwise explicitly stated, schema evolution means in this thesis schema evolution in the object-oriented context.

class, like adding and removing fields and methods, and changes to a class hierarchy. Schema evolution also deals with compound changes of several classes, like merging two classes and splitting a class. A typical unit of versioning in class evolution is a class, while it can be a class, a subschema or a schema in schema evolution. In our opinion, there is no other reason for the last two differences between schema and class evolution but tradition.

Even if the research presented in this thesis does not deal with consequences of changes on the data in the database, we consider related work from both schema and class evolution. In our opinion, this provides a more comprehensive picture of related concepts and problems.

### 2.1.1 Database Application Life-Cycle Model

This section describes database schema evolution in the context of a database application system life-cycle model. We propose a database application system life-cycle (Figure 2.2) that represents the following phases: feasibility analysis, requirements collection and analysis, design of the database and applications, implementation, integration, testing and validation, data loading and conversion, and operation/maintenance.[7] These phases are not strictly sequential — there are feedback loops between them. For simplicity, testing and loading of data are not presented in this figure.

The database design phase produces a high-level description of the data to be stored in the database. This description, also called a conceptual schema, is *independent* of the actual database management system (DBMS), but dependent on the data model of the DBMS. During the implementation phase the conceptual schema is converted into a database schema that is *dependent* on the chosen DBMS, and a database is created.

Similarly, the application design phase produces a high-level application specification that is converted to application code during the application implementation phase. The database application system is the result of the integration of the schema and applications. The system is tested and, if available, data is loaded into the database. Thereafter, the database application system becomes operational. This phase is called maintenance.

Change requests appear during the maintenance phase and include correction of errors, performance enhancements, improvements to user interfaces and new or changed functionality. Changes in the latter category require new requirements analysis and database redesign. Other changes like error corrections and performance improvements will typically be carried out directly on the database schema.

[7] This model is similar to other software life-cycle models such as the spiral model (Boehm, 1988) and the fountain model (Henderson-Sellers and Edwards, 1990) and to the database life-cycle model (Connolly *et al.*, 1996). However, our model includes more details about a database application system.

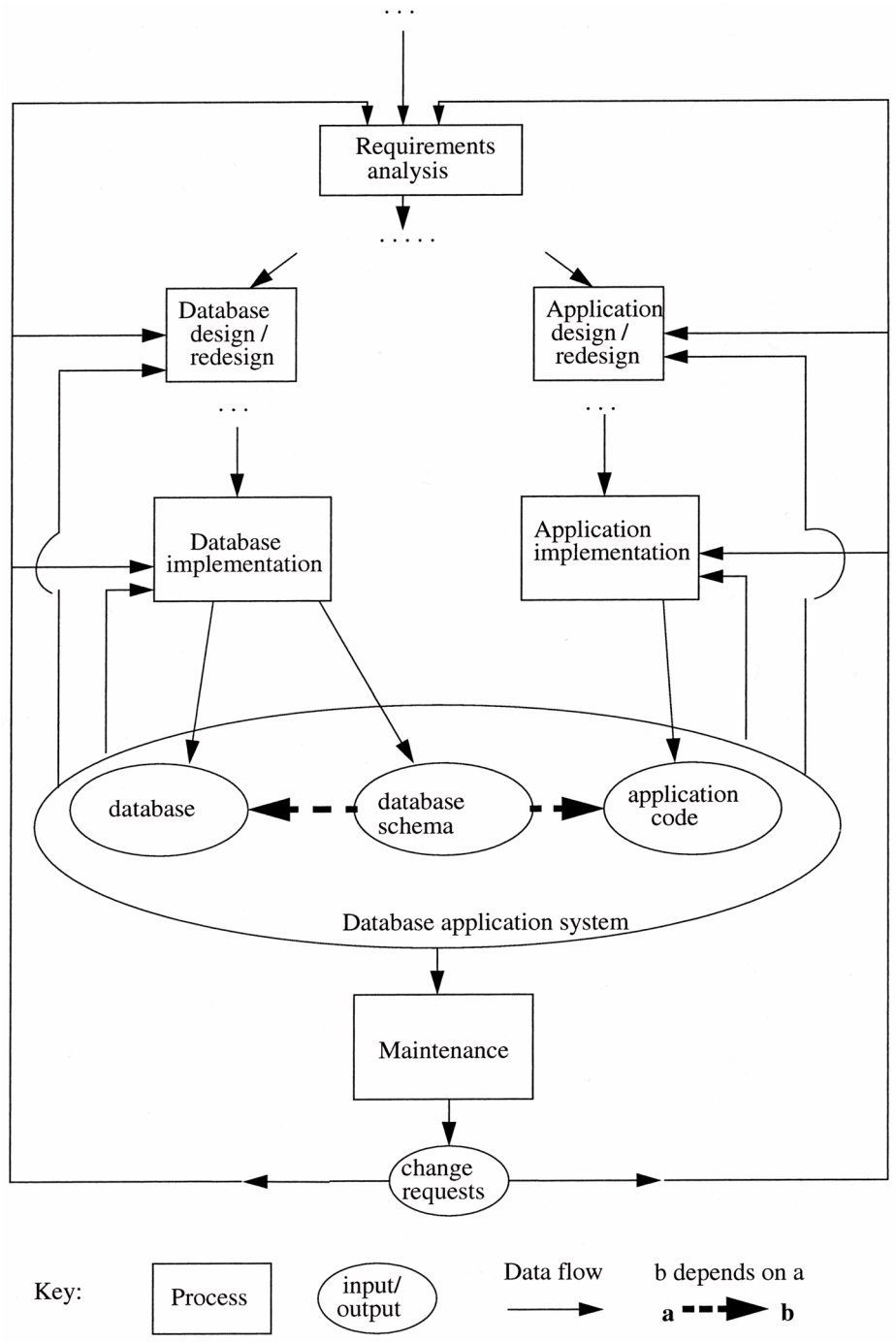



**Fig. 2.2.** Database application system life-cycle model

### 2.1.2 Model of Software Change

In the area of software change impact analysis, a *software change* (Bohner and Arnold, 1996) is considered as a process consisting of the activities given in Table 2.2.

**Table 2.2.** Activities of software change process

| | Activity |
|---|---|
| 1. | Creating specification of the change |
| 2. | Finding impacts of the change |
| 3. | Approving or refusing the change |
| 4. | Planning and conducting the change |
| 5. | (Re)Testing of affected components and the system as a whole |

We consider a schema change as a special kind of software change consisting of the *change of the schema itself*, *change of the data* in the database and *change of the application* while maintaining the consistency of database application system. The activities of Table 2.2 can be applied to each of these three kinds of change.

Figure 2.3 illustrates a typical scenario for a schema evolution process.

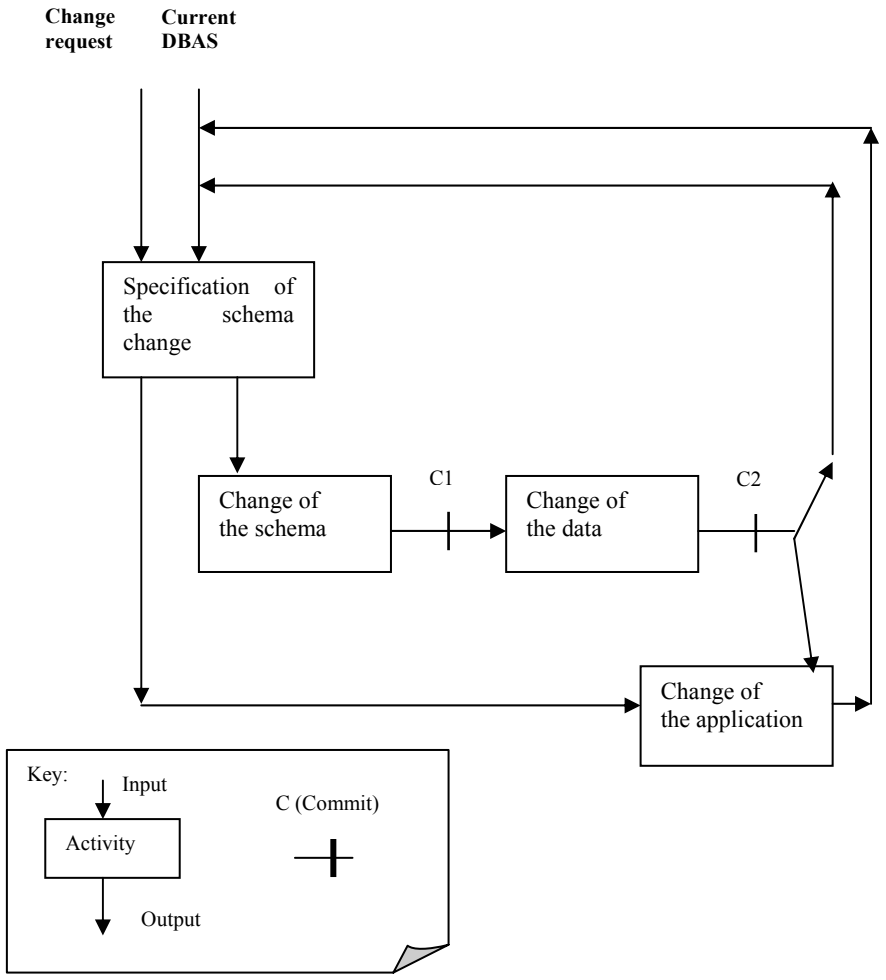



**Fig. 2.3.** Schema evolution process

Let us assume that a *change request* for the *current database application system* (DBAS) has been submitted. The first step, *specification of the change*, is common for the whole DBAS. The next steps are determined by the needs of a schema or application developer. The developer can be interested in:

- Propagating the change only to the rest of the schema. In this case the *change of the schema* consisting of the activities 2–5 (Table 2.2.) is conducted. After that,

this change is committed (commit C1) and the new schema becomes the current one.

- Propagating the change to the schema and the data in the database. This can be executed as a nested transaction consisting of the *change of the schema* (activities 2–5) and *change of the data* (activities 2–5). First, data change is committed first (commit C2), and then, if the new schema is successfully tested, the schema change is committed. This results in a new state of the database application system. If versioning is supported, it is not necessary to change the application immediately.
- Propagating the change to the schema, to the data in the database and to one or more applications. In this case the *change of the schema* and *change of the data* are followed by the *change of the application* consisting of the activities 2–5.
- Adapting a selected application that was working on the old schema version to the new schema version. If the log file of the schema changes that have been conducted to produce the new schema version is unavailable, the schema changes first have to be detected. This is followed by *change of the application.*

### 2.1.3 Related Work on Schema Evolution

Significant research has been conducted in the area of schema evolution. The bibliography database INSPEC lists 164 papers dealing with database schema evolution for the decade of 1991 to 2000 (Figure 2.4.). The database was searched with the keywords:

(schema AND evolution)

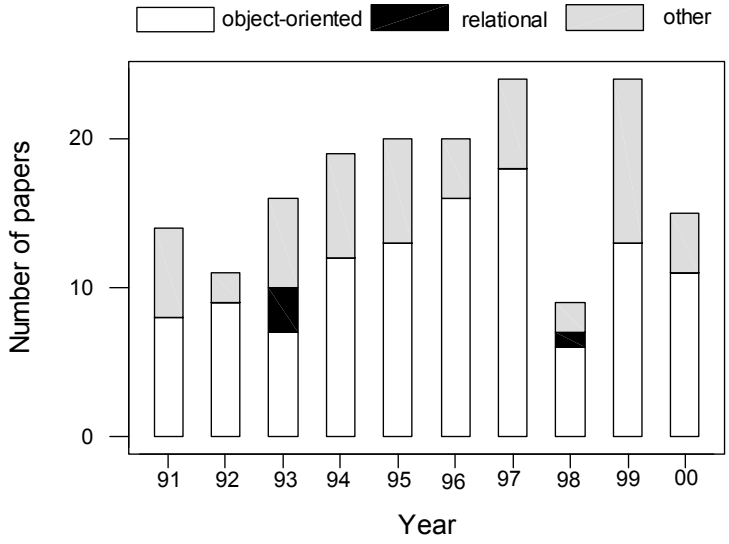



**Fig. 2.4.** Number of papers published in the area of schema evolution

This search resulted in 210 abstracts of papers published in several languages. Then, the author examined the abstracts of the resulting papers. Papers focusing on other research areas, e.g. human genome research, and only mentioning schema evolution were excluded, resulting in 172 papers left.

This survey indicates that schema evolution has become a mature research area. It also indicates that there is still ongoing research in this area. Object-oriented databases are intended to support non-traditional applications such as CAD/CAM, software engineering, office automation and multimedia information systems where schema changes are common. Not surprisingly, the majority of published papers (113) were in the area of object-oriented databases. The proportion of object-oriented database papers within the schema evolution field has slightly decreased the last years (from 75 % in 1997 to 66% in 2000). This may reflect that "the object-oriented database community is going through a frustration phase, mainly because of market reasons" (Guerrini *et al.*, 1999). The schema evolution research conducted in the area of relational databases was rarely reported (4 papers). This maybe because of the assumed stability of relation schemata or because of the shifting focus of the research community to object-oriented databases. Other papers (55) were related to conceptual schemata, temporal databases, active databases or federated databases.

Research conducted in the area of schema evolution falls into three main approaches. These are *direct schema evolution*[8], *schema versioning* and *schema evolution* by *views*.

- *Direct schema evolution* (Banerjee *et al.*, 1987; Penney and Stein, 1987; Nguyen and Rieu, 1989; Lerner and Habermann, 1990; Tresch, 1991; Barbedette, 1991; Zikari, 1991; Tresch and Scholl, 1992; Connor *et al.*, 1994; Zicari and Ferrandina, 1997; Dmitriev and Atkinson, 1999; Atkinson *et al.*, 2000) is supported when schema changes are allowed without loss of existing data. The old schema and the old state of the schema extension are inaccessible after the change took place.
- *Schema versioning* (Skarra and Zdonik, 1987; Kim and Chou, 1988; Bjørnerstedt and Hulten, 1989; Andany *et al.*, 1991; Bratsberg, 1992; Monk and Sommerville, 1992; Monk, 1993; Odberg, 1994a; Odberg, 1994b; Clamen, 1994; Brazile and Dongil, 1995; Fornari *et al.*, 1995; Lautemann, 1996; Lautemann *et al.*, 1997; Lautemann, 1997; Liu, 1997 ) is supported when schema changes produce a new state of the schema (new schema version) and a new extension associated with it. Access of different schema versions and corresponding extensions is allowed both retrospectively and prospectively. An application program works on top of a particular schema version.
- Schema evolution can also be supported by *views* (Bertino, 1992; Tresch and Scholl, 1993; Breche *et al.*, 1995; Bellahsene, 1996; Young-Gook and Rundensteiner, 1997). When a schema change request is received, a new schema view is computed with the desired semantic. All existing views are preserved and old application programs can be used with the schema for which they were designed.

[8] This approach is also denoted as *schema evolution* (Roddick, 1995; Jensen and Dyreson, 1998) and *adaptational approach* to schema changes (Ferrandina and Lautemann, 1996).

## 2.2 Evaluation of Technologies Supporting Application Consistency

The purpose of this section is:

- to propose a framework for comparing and evaluating technologies supporting application consistency in an object-oriented context, and
- to evaluate representative technologies within it.

Schema evolution technologies have been evaluated in the literature according to different criteria (Section 2.2.1). The framework we propose focuses on a set of features related to *maintaining application consistency*, and explores their usability (Section 2.2.3). Representative schema and class evolution technologies are compared according to the framework (Section 2.2.3).

### 2.2.1 Related Work on Evaluation of Schema Evolution Technologies

Schema evolution technologies proposed in the literature have been evaluated according to performance, utility and compliance with standards.

#### *2.2.1.1 Performance Evaluation*

A performance evaluation of *O2* was reported in (Ferrandina *et al.*, 1994; Zicari and Ferrandina, 1997). An adapted version of the OO1 benchmark (Cattell and Skeen, 1992) was run on both small and large databases to compare the performance of immediate and deferred database transformations. The scalability and recoverability of the system were also evaluated. An extension of the OO7 benchmark (Carey *et al.*, 1993) with schema evolution operations was proposed in (Al-Jadir and Leonard, 1999). The schema evolution mechanism of the F2 DBMS (Al-Jadir *et al.*, 1995) was evaluated by this benchmark. An evaluation of the performance for the *PJama1* platform was reported in (Atkinson *et al.*, 2000). The time that it takes for the platform to evolve a certain number of objects was measured. The following parameters varied: the number of evolving objects, the number of non-evolving objects per evolving object, complexity of the objects and of the conversion code. In addition to the synthetic tests based on the OO1 benchmark, several experiments with real-life applications were performed.

#### *2.2.1.2 Compliance with Standards*

In the evaluation of commercial object-oriented database management systems given in (Rashid and Sawyer, 1999), four systems (POET, Versant, *O2* and Jasmine) were compared according to compliance with the ODMG standard (read-only access to the schema) and according to the extended features. The extended features are addition and deletion of both leaf and non-leaf classes; addition, deletion and modification of the class attributes and methods; linear and branch class versioning; linear and branch object versioning; support for long transactions and advanced persistent locks of objects.

*2.2.1.3 Utility evaluation*

Taxonomies of supported schema changes are often used to evaluate and compare schema evolution technologies. Several taxonomies have been proposed (Roddick *et al.*, 1993; Banerjee *et al.*, 1987; Connor *et al.*, 1994; Breche *et al.*, 1995; Ferrandina and Lautemann, 1996; Ridgway and Wileden, 1998). They categorise schema changes according to:

- **Semantic** (Connor *et al.*, 1994) — *Additive schema changes* introduce new semantic knowledge, *e.g.*, a new field in a class. *Subtractive schema changes* model less semantic knowledge, while *descriptive schema changes* model the same semantic knowledge in a different manner.
- **Terms used to define a change** — Schema changes can be defined in terms of changes to the relations and fields (Roddick *et al.*, 1993); changes to the class definitions; changes to the structure of a class hierarchy (Banerjee *et al.*, 1987) and changes involving several classes at the same time (Breche *et al.*, 1995).
- **Effects on a database application system** — One can distinguish between the changes that may affect the database and the changes that may not affect it (Ridgway and Wileden, 1998). Regarding effects on applications one can distinguish between *schema extending*, *compilation safe* and *compilation unsafe* changes (Ferrandina and Lautemann, 1996). Schema extending changes have no immediate impacts on applications; compilation safe changes do neither delete nor modify any of the information used by applications; compilation unsafe changes do either delete or modify information of at least one application. In the Java Language Specification (Gosling *et al.*, 1996), *binary incompatible* changes are changes that prevent successful compilation or linking. In (Dmitriev, 2001), *source incompatible changes* are defined as changes that prevent successful compilation or correct program execution.

Proposals addressing application consistency are compared in (Young-Gook and Rundensteiner, 1997) according to the ability to share objects by different schemas; manual effort required when conducting the change; flexibility to build a new schema from class versions; subschema evolution and combination of views with schema changes.

Schema evolution models and tools are compared in (Odberg, 1995) according to the support for versioning; their ability to manage additive changes, eagerness of object conversion and ability to handle changes that affect more than one class.

A survey of schema versioning database systems (Roddick, 1995) discusses modelling, architectural and query language issues for both relational and object-oriented database systems. A survey (Li, 1999) presents recent research in schema evolution in object-oriented databases according to six aspects: instance mutation, instance versioning, schema mutation, schema versioning, data model mutation, and data model versioning.

The criteria for evaluation and relevant issues identified in the research described above constitute a foundation for evaluation of schema evolution technologies. However, they have some weaknesses. Even if the primary goal of the research in schema evolution has been to support schema and application developers in software development and maintenance, little attention has been paid to the usability evaluation

of the proposed schema evolution technologies. Furthermore, insufficient attention has been paid to the support for application consistency, which is in stark contrast to its importance.

### 2.2.2 Framework for Evaluation of Technologies Supporting Application Consistency

A list of desired features of schema evolution technologies usually reflects the experience of the authors from the development of their own system. To define our framework more formally, we based it on a model of software change (Section 2.1.3) and a model of system acceptability (Section 2.2.2.1). We then selected features related to maintaining consistency between schemata and application (Section 2.2.2.2).

#### *2.2.2.1 Acceptability of Schema Evolution Technologies*

Schema evolution technologies may be compared and evaluated from different viewpoints. Managers may be mostly interested in the costs and functionality of a schema evolution technology. Schema and application developers may be mostly interested in functionality and usability of the same technology. Technical support engineers are often interested in reliability and costs. Researchers may be interested in novelties of the technology, implementation details and research methods used to evaluate the technology.

We have adapted a model of system acceptability presented in (Nielsen, 1993) to include attributes of schema evolution technologies (Figure 2.5). The overall *acceptability* of a schema evolution technology is a combination of its *social acceptability* and its *practical acceptability*. The practical acceptability of the technology is determined by context, compliance with standards, by traditional attributes of acceptability such as performance, reliability and costs, and by usefulness (Nielsen, 1993).

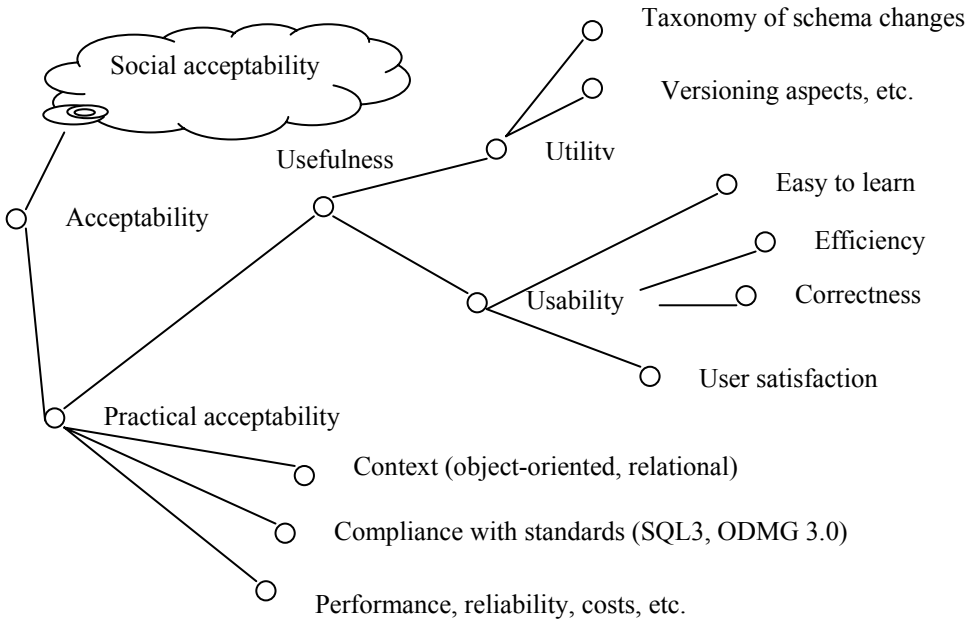



**Fig. 2.5.** Overall acceptability of schema evolution technologies[9]

*Context* is determined by the data model on which a database schema is based, and whether we consider a single schema or a several schemata. A data model e.g., the relational model, the object-oriented model and the entity-relationship model, determines the concept of database schema and rules for producing such schemata. The data model used in a database application system can be changed during the lifetime of a database application system. A schema evolution technology can deal with a single schema or several schemata. In this thesis we consider schema evolution models and tools dealing with a single schema that is based on the object-oriented data model.

ODMG 3.0 (Cattell and Barry, 2000) is a standard for object-oriented databases developed by the ODMG consortium. ODMG 3.0 has two specification languages for object database management systems (ODMS): The Object Definition Language (ODL) and the Object Interchange Format (OIF). Currently, this standard offers only read-only access to the schema. The ODMG C++ Schema Access API allows traversal of the meta-object hierarchy and access to the structure of the meta-objects. Schema change commands are not included. An extension of the ODMG ODL for generalised schema versioning support has been proposed in (Grandi and Mandreoli, 1999). This extension consists of statements for schema version selection, the schema

[9] We have defined the following attributes specific for schema evolution technologies: context, compliance with standards, taxonomy of schema changes and versioning aspects. Other attributes are taken from (Nielsen, 1993).

version creation, deletion and modification. Schema change commands are not part of this proposal.

Several benchmarks are used for measuring the *performance* of schema evolution technologies. OO1[10] is a general benchmark for measuring the performance of engineering applications. OO7 is a more complete and complex benchmark for object-oriented databases. An extension of OO7 for schema evolution operations is proposed in (Al-Jadir and Leonard, 1999).

*Usefulness* of a schema evolution technology concerns to what extent the technology supports schema and application developers to *manage* schema changes. Usefulness deals with two related categories: utility and usability (Grudin, 1992). *Utility* concerns the functionality provided to manage schema changes. *Usability* is defined in ISO 924-11 as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" (Catarci, 2000).[11] Usability is not only the appearance of user interface (Ferre *et al.*, 2001). The technology should be easy to learn and efficient in use. Users should make few errors during the use, and the technology should be pleasant to use. Finding optimal balance between utility and usability is quite complex due to mutual interaction among them. While new functionality may improve effectiveness, it may be difficult to learn, and therefore not used. Negative attitude towards the technology may cause more errors. Long-term studies are needed to increase our understanding of these effects (Kay and Thomas, 1995).

Ensuring satisfactory usability of technology is not easy or obvious (Juristo *et al.*, 2001). Usability of a technology should be evaluated as early possible. Evaluation of 'pen and paper' prototypes can be used during the design of a technology (Ferre *et al.*, 2001). Usability evaluation is, however, evaluation of an implemented technology. Significant differences between the results of 'pen and paper' and 'tools' evaluations may appear (Catarci, 2000).

#### *2.2.2.2 Framework*

We have the following requirements to our framework:

- It should be 'user oriented'.
- It should be complete.
- It should provide a basis for empirical evaluation.

Research in schema evolution has primarily been driven by the need of schema and application developers for better support for schema changes. Therefore, we focus on the attributes of schema evolution technologies directly relevant for the users of these technologies, namely utility and usability (Figure 2.5).

Further, the framework should be complete. By this we mean that it should be possible to evaluate features supported by most existing schema evolution technologies. All features supported by schema evolution technologies are used in some of the *activities* of the schema evolution process. Therefore, relating elements of the framework to the *activities* should provide relatively complete framework.

---

[10] An adapted version is given in (Zicari and Ferrandina, 1997).

[11] This definition was first given in (Bevan and Macleod, 1993).

Our third requirement is that the framework should provide a basis for empirical evaluation of schema evolution technologies. For the framework elements that represent utility we give a list of *applicable values*. For the framework elements that represent usability we propose a list of *measures*. The effects of a particular utility[12] on usability can be expressed in terms of these measures.

Table 2.3 presents the utility elements of our framework.

**Table 2.3.** Utility elements of the framework

| Element | Activity | Applicable values |
|---|---|---|
| Consistency | All activities | Schema consistency (structural, behavioural);<br>Structural consistency;<br>Consistency with application (behavioural, downward, upward; by impact analysis) |
| Taxonomy of schema changes | Specification of the change | Changes to the:<br>class definitions;<br>class hierarchy;<br>several classes |
| Way of specification | Specification of the change | Schema change language;<br>Source code changes;<br>Graphical schema editor |
| Impacts identification | Finding impacts | Impacts on the: data, schema, applications;<br>Granularity: files, classes, fields, methods |
| Impact presentation | Finding impacts | List; graph;<br>No |
| Automatic conversion | Conducting the change | Conversion of:<br>data, schema, applications<br>Conversion functions,<br>No |
| Validation | (Re)Testing | Yes;<br>No |
| User interface | All activities | Textual;<br>GUI;<br>Unknown |
| Versioning support | All activities | Yes;<br>No |
| Other functionality | All activities | Other functionality available to the user;<br>Unknown; None |

When a schema is changed, three kinds of *consistency* should be maintained: schema consistency, structural consistency and application consistency (Section 1.2). The

[12] This corresponds to the applying a value on an element.

kinds of consistency supported by various schema evolution technologies can be used for a first, rough classification of these technologies.

We consider the following *taxonomy of schema changes* in our framework:

- Changes to the class definition, i.e., add, remove or rename of fields and methods, change the domain of a field, change the code or the signature of a method.
- Changes to the structure of a class hierarchy, i.e., add, remove a superclass or change order of superclasses.
- Changes to a class, i.e., add, remove or rename a class.
- Changes to the connection of a class to a file or a package, i.e., add, remove a class from/to a file or a package.
- Complex changes — simultaneous changes of more than one class like merge, split or move fields or methods between classes (Breche *et al.*, 1995; Lerner, 2000).

This taxonomy differs from the Orion taxonomy (Banerjee *et al.*, 1987) in omitting changes that are specific to the Orion data model, such as *change to the order of inheritance priority*, *add a shared value*, etc. Furthermore, this taxonomy includes complex changes.

There are two *ways of specifying* schema changes:

- Schema change commands are issued using some defined schema change language.
- The source code of schema classes is changed directly.

The majority of the schema evolution technologies *identify impacts* of the schema change on the rest of the schema and the data in the database (Banerjee *et al.*, 1987; Nguyen and Rieu, 1989; Tresch and Scholl, 1992; Lerner and Habermann, 1990; Penney and Stein, 1987; Barbedette, 1991; Connor *et al.*, 1994; Zicari and Ferrandina, 1997; Zikari, 1991; Dmitriev and Atkinson, 1999; Atkinson *et al.*, 2000). Some technologies also support identifying impacts on applications (Deruelle *et al.*, 1999).

When impacts are identified, there are basically two possibilities: *automatic/semi-automatic conversion* of the affected data and/or schema and application parts, and *impact presentation* to the developers. Identified impacts can be at the granularity of file, class or method.

In addition to testing whether the new schema maintains conceptual and behavioural consistency, the quality of the new schema can be *validated*. The quality of the schema can be measured according to some set of criteria, for example, redundancies in the inheritance graph. This feature is currently not supported by any schema evolution technology. However, plans (needs) for implementing this feature are reported in (Barbedette, 1991).

A *user interface* of a schema evolution technology allows specification of schema changes, specification of conversion functions and presentation of impacts. Schema changes can be specified via textual or graphical user interfaces or by changing source code directly. If conversion of data is not completely automated, a user needs to specify conversion functions. This can be done via an interface or by changing the source code. Impacts of schema changes can be displayed as a list, or marked on a graph presenting dependencies between the components.

*Versioning support* refers to the ability to remember schema versions and their extensions. Navigation between the different versions should be provided. A schema derivation structure can be a sequence, a tree or a Directed Acyclic Graph (DAG). A derivation structure based on a DAG allows parallel design and integration. The ability to co-operate on schema versions is related to the existence of several schema versions. A check-in/check-out mechanism and transaction model could be provided to support co-operative work. If applications are working on different schema versions, it should be possible to co-operate on shared instances (schema versioning co-operation) (Lautemann, 1997).

Table 2.4 presents the usability elements of the framework.

**Table 2.4.** Usability elements of the framework

| Element | Measure |
|---|---|
| Easy to learn | Time needed to learn |
| Efficiency | Time needed to conduct a schema change (for different changes; for different size and complexity of application systems) |
| Correctness | Number of errors when conducting a schema change (for different changes; for different size and complexity of application systems) |
| User satisfaction | Seven-point scale based on (Lewis, 1995);<br>Other |

A schema evolution technology in its entirety, as well as its utility elements, should be *easy to learn*. The developers should be able to use tools and procedures in a way that is close to their normal working methods. This can be achieved by keeping the amount of additional concepts and steps as minimal as possible (Atkinson *et al.*, 2000). Whether the technology is easy to learn can be measured by the time needed to achieve a particular proficiency level.

*Efficiency* in use is measured by the time needed to conduct a schema change (including maintaining all consistencies).

*Correctness* is measured in number of introduced errors when conducting a schema change. These errors can be compilation or run-time errors. *User satisfaction* can be measured by usability questionnaires, like IBM Post-Study System Usability Satisfaction Questionnaire (Section 3.4.2). The size and complexity of a database application system affect efficiency, correctness and user satisfaction when using a particular schema evolution technology and should therefore be considered. The size of the system may be measured in lines of source code, number of persistent classes, and memory used for storing data in the database. The complexity of the system may be measured by some of measures proposed for object-oriented systems (Briand *et al.*, 1999) or traditional databases (Piattini and Martinez, 2000).

### 2.2.3 Evaluating Schema and Class Evolution Technologies

To illustrate the use of our framework, and to provide some validation of it, we have selected schema and class evolution technologies representing different approaches and evaluated them according to our framework.

*2.2.3.1 Schema Evolution Technologies*

Orion (Banerjee *et al.*, 1987) was a prototype object-oriented database system. It was amongst the first systems offering support for schema evolution in object-oriented environments. The taxonomy of schema changes and invariant properties of a database schema proposed in Orion are used as a basis for numerous research and commercial systems supporting schema evolution in object-oriented environments.

Another schema evolution technology is the class evolution tool for PJama (Dmitriev and Atkinson, 1999; Atkinson *et al.*, 2000; Dmitriev, 2001). PJama (Atkinson *et al.*, 1996) is a persistent programming language (Atkinson and Morrison, 1985). PJama writes persistent objects, their classes and all classes reachable from them to a persistent store. All code needed to interpret the saved data is also saved. Once a class become persistent it is in the persistent store and cannot be replaced by modifying its source code; the schema evolution tool must be used. These two technologies are based on the direct schema evolution approach, which is currently adhered to the majority of commercially available systems.

COAST, the Complex Object and Schema Transformation project (Ferrandina and Lautemann, 1996; Lautemann, 1997; Lautemann *et al.*, 1997; Coast, 1999) is one of the most sophisticated systems supporting versioning that we found. Old schema and database states are kept as versions to allow continuous operation of existing programs. Versioning is supported both at the instance and at the schema level. The COAST Schema Editor offers graphical user interface to develop new schema versions and to specify conversion functions.

The Transparent Schema-Evolution System (TSE) (Young-Gook and Rundensteiner, 1997) is based on object-oriented views. It is built on top of GemStone and includes a capacity-augmenting view mechanism. The persistent data is shared by different views of the schema, i.e., both old and newly developed applications can continue to operate.

The Source Code Software Components Structural Model ($SC^2SM$)(Deruelle *et al.*, 1999) is the only system that we are aware of which identifies and presents impacts of schema changes on applications. Local and federated schemata, source code of applications and their relationship are represented by software components graphs. Schema changes are simulated by using a knowledge base system.

A comparison of the technologies according to the utility elements is given in Tables 2.5–2.8.

**Table 2.5.** Types of consistency

| Technology | Schema | Data | Applications |
|---|---|---|---|
| Orion | conceptual | structural | No |
| PJama | conceptual, behavioural | structural | behavioural |
| COAST | conceptual | structural | downward |
| TSE | conceptual | structural | downward |
| $SC^2SM$ | impact analysis | no | impact analysis |

The selected technologies are compared according to the supported kind of *consistency* in Table 2.5. Orion provides structural and conceptual consistency (Section 1.2). Support for structural consistency has some limitations. When a class is deleted in Orion, all instances of that class are deleted. This can lead to dangling references. Whereas the systems based on versioning or views (COAST and TSE) maintain consistency between the data, schema and application by accessing an old schema version, PJama takes a different approach. PJama does not maintain an explicit database schema. Classes, included the code, are preserved along with persistent instances in the persistent store. This assumes that all classes in the store belong to the same application and are changed at the same time. The $SC^2SM$ system identifies potential impacts of schema changes. This information helps the developer to accomplish the change.

**Table 2.6.** Taxonomies of Schema Changes

| Technology | Class definition | Class hierarchy | Complex changes |
|---|---|---|---|
| Orion | Add/delete/rename an attribute;<br>Change the domain of an attribute;<br>Change the inheritance of an attribute;<br>Add/delete/rename a method;<br>Change the code of a method;<br>Change the inheritance of the method; | Add/delete/rename a class;<br>Add/remove a superclass;<br>Change the order of superclasses; | No |
| PJama | All changes supported | All changes supported | No |
| COAST | Add/delete/rename an attribute;<br>Change the domain of an attribute;<br>Add/delete/rename a method;<br>Change the code of a method; | Add/delete/rename a class;<br>Add/remove a superclass; | No |
| TSE | Add/delete/rename an attribute;<br>Change the domain of an attribute;<br>Add/delete/rename a method;<br>Change the code of a method; | Add/delete/rename a class;<br>Add/remove a superclass;<br>Change the order of superclasses; | No |
| $SC^2SM$ | Add/delete an attribute;<br>Change the domain of an attribute;<br>Add/delete a method; | Add/delete a class | No |

Table 2.6 summarises the kinds of *taxonomies of schema changes* supported by different schema evolution technologies. The taxonomy of Orion is used as basis for numerous research and commercial systems supporting schema evolution in the object-oriented environments. However, this taxonomy has some limitations. For example, a domain can be generalised, but not specialised. There is no support for changes outside the given taxonomy.

**Table 2.7.** Specification of change, identification and presentation of impacts

| Technology | Specification way | Impact identification | Impact presentation |
|---|---|---|---|
| Orion | Schema change commands; graphical-based schema editor | Impacts on the schema and data | No |
| PJama | Schema change commands (for class hierarchy changes); source code changes | Impacts on the schema, data and application; impacts on classes | Text |
| COAST | Schema change commands | No | No |
| TSE | Schema change commands | No | No |
| $SC^2SM$ | Source code changes | Impacts on the schema, and application; impacts on classes and methods | Graph |

Table 2.7 presents ways of *specifying* schema changes. Orion, COAST and TSE use schema change languages (schema change operators) for specification of schema changes. Orion also facilitates specification through a graphic-based editor. Unfortunately, we unaware of its detailed description. When schema change languages or graphic-based editors are used, only the changes defined by the language can be specified.

PJama provides the possibility to detect changes from the source code. This allows a wider spectrum of schema changes to be specified. PJama also allows specification of new or changed classes in bytecode form (classes obtained from third parties, for example). Some changes have to be explicitly specified. These are removing of a class alone or together with instances. In $SC^2SM$, schema changes are specified as a combination of changes of the source code and schema change commands.

Our primary concern is application consistency. Therefore, we explain how the selected technologies identify and present impacts on the applications, and whether they provide means for automatic conversion of applications. Information about impacts on the schema and the data in the database, and their conversion is given briefly in respectively Tables 2.7 and 2.8.

As PJama writes to a persistent store, persistent objects, their classes and all classes reachable from them, there is no difference between the schema and the applications. Impacts of a proposed change are identified at the granularity of classes and presented as a list. COAST and TSE support application consistency by allowing access to old schema versions and views, and therefore do not need to identify and presents impacts of schema changes on applications.

In (Ferrandina and Lautemann, 1996) improvements of the COAST system by integrating the direct schema evolution model and the schema versioning model are

proposed although not yet implemented. Schema changes are categorised into schema extending and schema modifying changes, whereby only the latter might require the schema versioning approach. To recognise the case where the schema change has no effect on the applications, the system maintains an internal *application table* for a particular schema. The application table contains information about usage of class attributes by applications. This table is allocated when a schema is created, and updated each time the schema or an application is changed. Class inheritance is modelled by repeating the superclass attributes in the application table. There is no information about usage of class methods by applications. These improvements are, however, not implemented.

$SC^2SM$ models applications manipulating persistent objects by graphs. Based on the set of propagation rules the system processes the change impacts and identifies impacts of a proposed schema change on the application.

**Table 2.8.** Automatic conversion and user interface

| Technology | Automatic conversion | User interface |
|---|---|---|
| Orion | Data; schema | GUI (not described) |
| PJama | Data; schema; applications; conversion functions | Textual |
| COAST | Data; schema; conversion functions | GUI |
| TSE | Data; schema | Unknown |
| $SC^2SM$ | No | GUI |

PJama is the only technology providing partly automatic conversion of an application. The tool recompiles those classes for which the source has changed. The build tool compares the resulting classes with their original version stored in the store. If a class has changed in a potentially incompatible way, the tool forces recompilation of all potentially affected classes. If recompilation fails, the evolution is aborted. If a conversion of instances is required, the user may choose between the default conversion and writing their own conversion functions in Java.

The GUI of COAST consists of the Browser, the Schema Editor, the Schema Version Editor and the Class Editor. The browser allows viewing of the schemata, schema versions and classes. The schema editor allows viewing and editing a single schema potentially having several versions. It also allows editing of conversion functions. A particular schema version can be viewed and updated by the schema editor. The class editor allows viewing and editing a single class. A textual user interface is also provided. An Object Definition Language (ODL) is used to specify both single schema versions and schema update primitives.

In $SC^2SM$, components of the schema (classes, fields and methods) and relationships between them (manipulate, compose, instance_of, inherit) are presented as nodes of a graph. Affected nodes are labelled *affected*.

COAST and TSE provide versioning support. Old states of the schema can be remembered and accessed. They both provide co-operation on instances shared between the different schema versions.

A variety of other potentially useful features can be supported by schema evolution technologies. TSE, for example, allows subschema evolution and flexibility of composing schema.

Usability is evaluated only for PJama. The first version of the PJama evolution tool (Dmitriev, 1998) was evaluated by own judgement during the work on the improvement of the Geographical Information System developed in PJama at the University of Glasgow. The system consisted of nearly 80 classes (25 of them were persistent). Frequent small changes and occasional larger changes were conducted. It has been reported that the tool facilitated greatly the work on the persistent applications (Dmitriev, 1998).

#### *2.2.3.2 Class Evolution Technologies*

Table 2.9 describes the Object-Oriented Testing and Maintenance Environment (OOTME) (Kung *et al.*, 1994). It automatically identifies the changes to an object-oriented program and finds the impact of these changes. This environment is not intended to be used with databases. However, it deals with similar problems as do schema evolution technologies and is therefore included in this survey.

**Table 2.9.** The OOTME system

| Utility elements \ System | | OOTME |
|---|---|---|
| Types of consistency | Schema | Impact analysis |
| | Data | No |
| | Application | Impact analysis |
| Taxonomy | Class definition | All changes supported: not intended to be used with database |
| | Class hierarchy | All changes supported; not intended to be used with database |
| | Complex changes | No |
| Specification way | | Source code changes |
| Impact identification | | Impacts on the applications; impacts on files and classes |
| Impact presentation | | Text; graph |
| Automatic conversion | | No |
| User interface | | GUI;textual |

The semantic of schema changes of OOTME is not quite identical to the others. Rename of a field in OOTME is equivalent to deletion and creation of the field with

the new name. Such an interpretation of the same change would lead to the losing of the data in a database application system.

The *change identifier* of OOTME automatically finds the code changes between two different versions of the same class library.

The graphical user interfaces of OOTME is based on X-Windows. Information about classes and their interrelations is extracted and represented in graphs, which in turn are used to show the parts of the object-oriented library affected by the changes. Textual information is also provided. The smallest component affected by a change identified by this tool is a class.

OOTME is integrated in a testing and maintenance environment. Various metrics, such as complexity, are planned to be included.

### 2.2.4 Summary

We claimed the need for empirical evaluation of usefulness of the technologies supporting application consistency. We described related work on evaluation of schema evolution technologies. We then presented a framework for evaluating technologies supporting maintenance application consistency from the perspective of schema and application developers. The framework is based on a proposed model of system acceptability and a model of software change. The framework consists of utility and usability elements. Utility elements are associated with a list of applicable values and the usability elements are associated with a list of measures. The effect of a particular utility on usability can be expressed in terms of these measures.

Five representative schema evolution technologies and one class evolution technology were evaluated within this framework. The class evolution technology we studied — OOTME provides, of course, no support for structural consistency. Regarding other aspects we found it very similar to the schema evolution technologies.

All the technologies that we have studied support changes to class definitions. Most of the technologies support arbitrary changes to class hierarchy, but none of them support complex schema changes. The effects of basic changes are usually smaller than the effects of complex changes. Applying several basic changes one after the other might have effects that cannot be easily overviewed (Breche *et al.*, 1995). It has been argued that change management suitable for software engineering applications should also provide support for complex changes (Breche *et al.*, 1995; Lerner, 2000). We have identified only minor differences in the semantics of supported schema changes between OOTME and the other technologies.

Application consistency is supported by allowing old applications to work on the top of an old schema version (COAST, TSE), by automatic conversion of code (PJama) and by presenting impacts of changes ($SC^2SM$ and OOTME).

The PJama evolution technology is currently the only one providing automatic conversion of application for changes to class definition. For changes to class hierarchies this conversion is semi-automatic. This approach assumes that all classes and data stored in the persistent store are part of the same application and will be converted simultaneously. This is, however, not always the case.

Allowing old applications to work on top of an older schema version, like in COAST and TSE, is very useful when the resources needed to change the application are insufficient, or if the source code is unavailable. The purpose of introducing a

change is usually to make use of it. The applications should therefore be adapted to the new schema version when possible. Identifying impacts of changes on the applications is an uneasy task. A case study of an industrial object-oriented project showed that the number of classes predicted to be changed was largely underestimated and that only one third of the set of changed classes were identified (Lindvall, 1997).

The usability of these technologies was not evaluated. It is unclear to which degree the presentation forms used by these technologies are convenient for schema and application developers.

## 2.3 Impact Analysis

Software change impact analysis is a research area considering consequences and implementation of a software change on software artefacts like design documents, documentation, source code and test material. It focuses on *"identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change"* (Bohner and Arnold, 1996). According to (Bohner and Arnold, 1996) there are two areas of impact analysis: *dependency analysis* and *traceability analysis*.

Dependency analysis[13] deals with low-level dependencies in code, such as dependencies among variables and modules. Dependency analysis is used in tools such as compilers, cross-reference tools, browsers and test tools. An example of commercially available cross-reference tool is Source Navigator (Navigator, 2001). It can find places where a given function is called and find files that includes a given header file.

Traceability analysis deals with dependencies among software artefacts like requirements and design documents. These dependencies are typically not very detailed.

Dependency analysis can be *static* or *dynamic*. Static impact analysis focuses on static code structure and identifies change impacts without executing the code. Dynamic impact analysis identifies the impacts by executing the code. The research presented in this thesis is concerned with static dependency analysis. Details on dynamic dependency analysis can be found in (Wilde and Huitt, 1992; Chen *et al.*, 1996). Several impact analysis techniques have been developed. Among them are:

- **Transitive closure**. The *transitive closure algorithm* (Warshall, 1962) is considered to be one of the fundamental impact-analysis techniques (Bohner and Arnold, 1996). It is general enough to be used in different contexts and numerous impact analysis tools are based on this algorithm. The technology we propose for supporting application consistency (Chapter 6) is based on the transitive closure algorithm. The algorithm is explained in more details in Section 2.2.1.
- **Program slicing**. Program slicing introduced by (Weiser, 1984) is a technique used in code debugging, integrating and maintenance (Horwitz *et al.*, 1990; Gallagher and Lyle, 1991). A *program slice* S(x,p) taken with respect to a variable *x* and a program point *p* is executable program consisting of all statements of the program that may affect the value of x at point p. S(x,p) is called a *slicing criterion*. Tools using this technique eliminate parts of the program that are not important for the execution. This can be helpful in debugging and testing complex programs.
- ***Truth-maintenance***. A *truth-maintenance system* (Doyle, 1979) is based on a model of assumptions. Dependencies within a system are stored in a rule base as facts. Rules specify the way to infer new facts from existing ones. When the assumption model is changed, the truth-maintenance system identifies

[13] Dependency analysis is also known as source-code analysis.

inconsistencies and reports them to the user. The advantage of this technique is its flexibility. As the system changes one can update the facts and the rules.

### 2.3.1 Transitive Closure

The transitive closure algorithm takes as input a directed acyclic graph with components[14] as nodes and relationships between them as edges. It then finds all components reachable from a given component. Figure 2.6 illustrates the algorithm.

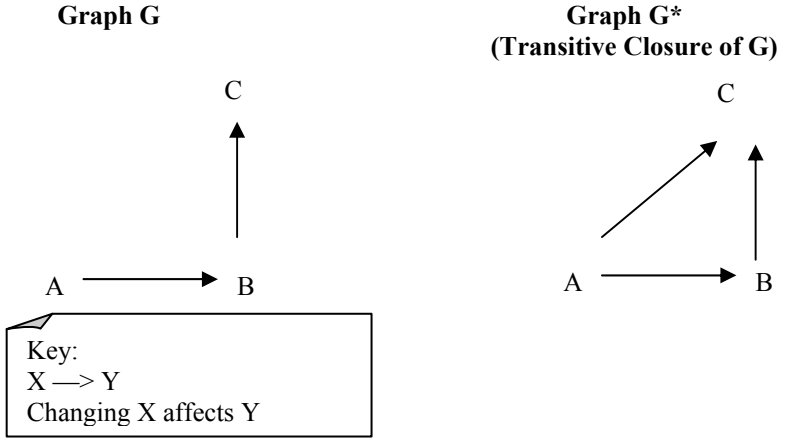



**Fig. 2.6.** Transitive closure algorithm

The graph G defines the dependencies between the components of a system. A dependency between A and B is a relationship (A,B) such that changing A may affect B. G gives direct impacts pairs {(A,B),(B,C)}. The graph G* gives the transitive closure {(A,B),(B,C),(A,C)}. G* shows that changing A may affect C.

Real-life database application systems typically consist of numerous components. The original transitive closure algorithm (Warshall, 1962) with execution time $O(n^3)$, where $n$ is the number of components, will thus often give unsatisfactory performance. Hence, several improved variant have been developed (Qadah *et al.*, 1991; Purdom, 1970). In (Qadah *et al.*, 1991), execution times of three variants of this algorithm are evaluated. These are the *wavefront* algorithm (Qadah *et al.*, 1991), the *δ-wavefront* algorithm (Qadah *et al.*, 1991) and the *super-TC* algorithm (Qadah *et al.*, 1991). The results of the study suggest superiority of one variant of the *super-TC* algorithm — the *gfsuper-TC* algorithm (Qadah *et al.*, 1991) over other algorithms and superiority of the *δ-wavefront* algorithm over the *wavefront* algorithm. Superiority of the *super-TC* algorithm is achieved by low-level memory management. Whereas other algorithms may read a given disc page more than once, the *super-TC* algorithm reads the disc page at most once.

The technology we propose for supporting application consistency (Chapter 5) is based on the *δ-wavefront* algorithm (Fig. 2.7). This algorithm is chosen because it is

[14] A *component* here means *a constituent part of something*, as defined in (Oxford, 1993).

more efficient than the original algorithm and independent of low-level memory management. We present this algorithm here in the terms of relation algebra. Details on other algorithms can be found in (Qadah *et al.*, 1991).

**procedure** δ-wavefront(input: query_constants, A(base_relation); output: closure)
**begin**
/*
wave and closure are unary relations with an X_attribute
A is binary relation with an X_attribute and a Y_attribute */
wave = {query_constants};
closure = ∅;

**while** (wave ≠ ∅) **do**

$$\text{wave} = \pi_{Y} \{\text{wave}(X) \,|x|\, A(X,Y)\}_{\text{wave.X=A.X}}$$

wave = wave – closure;
closure = wave ∪ closure;
**endwhile**
**end**

**Fig. 2.7**. The δ-wavefront algorithm

The *δ-wavefront* algorithm is a breadth-first search algorithm. It starts searching from a set of initial nodes stored in a unary relation, ***wave,*** and iterates to find all nodes reachable from the initial ones. ***A*** in Figure 2.7 is a binary relation where, for each tuple *(x,y)* of ***A,*** there exists a directed edge from node *x* to node *y* of the transitive closure graph. Figure 2.8 gives an example. ***A*** is presented as a table and as a graph.

| X | Y |
|---|---|
| a | b |
| b | c |
| b | d |
| c | d |
| d | b |
| d | e |

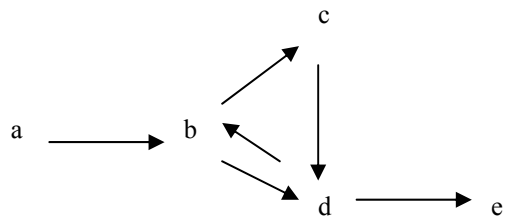



**Fig. 2.8.** The binary relation A

***Wave*** stores the nodes found during one iteration, while ***closure*** accumulates the nodes found during the different iterations. At the beginning of an iteration, ***wave***, which contains the set of nodes collected during the previous iterations, is semi-joined

with ***A***.[15] The result relation is projected on its second attribute to produce the set of nodes that can be reached from ***wave*** in one step.[16] The iteration process continues until ***closure*** does not change between two iterations. The nodes in ***closure*** are the solution of the query. Only nodes not encountered during previous iterations (stored in ***closure)*** are processed.[17] This eliminates redundant processing. Figure 2.9 presents the processing of the relation **A** using the δ-wavefront algorithm with the node *a* as a starting node (query_constants). The algorithm identifies the nodes {*b,c,d,e*} as a transitive closure of the node *a* after four iterations.

| iteration | old_wave | new_wave | closure |
|---|---|---|---|
| 1 | a | b | b |
| 2 | b | c,d | b,c,d |
| 3 | c,d | e | b,c,d,e |
| 4 | e | ∅ | b,c,d,e |

**Fig. 2.9.** The processing of the relation A

[15] The join operation on the relations R and S with join condition satisfied is written **R**|**X**|<joincondition>**S**.

[16] The projection of the relation R over the attributes from the attribute list is written **π**<attributelist>**R**.

[17] During any iteration.

## 2.4 Information Visualisation

Visualising means forming a mental picture of something (Oxford, 1993). In many application areas like medicine, architecture and geography pictures are routinely used to present information. It is pointed out in (Strothotte, 1988) that visualisation is complex due its context sensitivity; details in the image in surgery books, for example, are selected and emphasised to fit the context (Figure 2.10). Some of images (made by humans) have such a high quality that they have been used in many textbooks for decades. Information visualisation facilitates insight graphical information. Research suggests that in many application domains like reverse engineering, software restructuring and information retrieval (Larkin and Simon, 1987; Veerasamy and Belkin, 1996; Storey *et al.*, 1996; Griswold *et al.*, 1997; Morse *et al.*, 2000; Lanza, 2001), information visualisation may improve efficiency, accuracy and user satisfaction when solving tasks. We assume that visualisation will have a similar effect in the area of schema evolution. A tool that intends to visualise impacts of schema changes should employ generally accepted lore from the area of visualisation and cope with domain specific problems like scalability and change.

This section presents general guidelines for user interface design including guidelines for use of colours, and describes diagrams as a means for visualisation. It also presents some ways of coping with scalability and change.

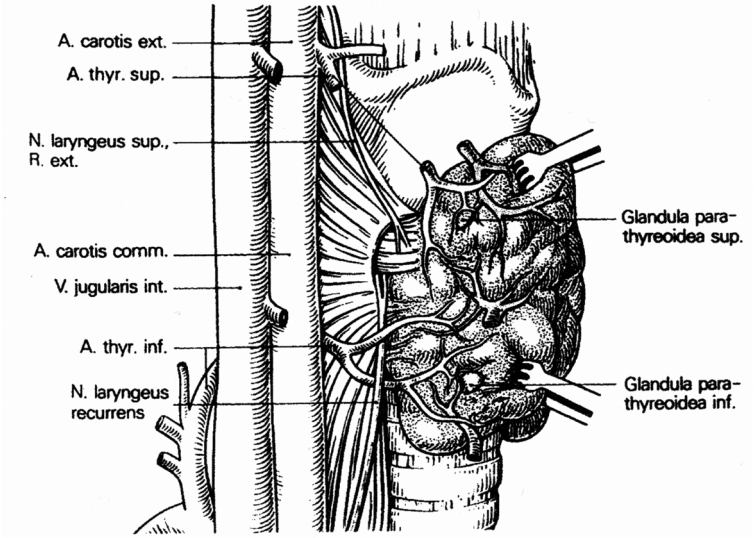



**Fig. 2.10.** A drawing from a textbook on surgery (Strothotte, 1988)

### 2.4.1 Guidelines for User Interface Design

Two general principles for user interface design, the *consistency principle* and the *clear and concise language principle,* are suggested in (Smith and Mosier, 1986). The

first principle states that similar things should look similar and different things should look different. The second principle states that names or images given to functions should be easy to understand.

In (Shneiderman, 1998) the eight golden rules of interface design are given. As our primary concern is information presentation, the first and the eighth rules are relevant for us. The first rule states that one should strive for consistency. Consistent sequence of actions should be required in similar situations; consistent colour and layout should be employed throughout. The eight rule states that short-term memory load should be reduced. As humans can remember "seven-plus or minus-two chunks" of information, the displays should be kept as simple as possible.

#### *2.4.1.1 Use of Colours*

Colours can be used to improve usability of user interfaces. Colours can be used to increase comprehensibility of a display or to draw the operator's attention to important events.

Guidelines for using colour in user interfaces are given in (Shneiderman, 1998). From our point of view, the most important guidelines are:

- Use colours conservatively and limit the number of colours. No more than four or five different colours should be used in a window. They should be used selectively and consistently.
- Be consistent in colour coding. Use the same colour-coding rules throughout the system.
- Use colour coding to support the task that the users are trying to perform. If they have to identify similarities in two programs, highlight these using a different colour.
- Use colour changes to indicate status changes. Colour can be used to draw operator's attention. In an oil refinery, for example, pressure indicators might change colour when the value is above acceptable limits.

When using colour, one should be aware of the following potential problems (Shneiderman, 1998):

- Colour pairings; people cannot focus on red and blue colour simultaneously. Too little contrast, such as yellow letters on a white background, could also be a problem.
- Colour deficiency; about 8 percent of users (North America and Europe) have some colour deficiency, e.g., red-green blindness.
- Meaning of colours; in some cultures, specific colours can have deeply anchored meanings (e.g., red light for stop).

### 2.4.2 Diagrams

Diagrams are commonly accepted as an efficient means to communicate information. They are often composed of schematic figures that serve as graphical primitives (Tversky *et al.*, 2000). The following properties of a diagram carry semantic weight:

- location of the semantic figure in the diagram,
- type of schematic figures (blobs, straight lines, curved lines, crosses, arrows, etc.),
- size and color of the schematic figures,
- local conventions established by the diagram, and
- conventions established by the application domain.

File system and web site diagrams typically use blobs to present files and directories, and lines to denote relationships between them. Size can be determined by quantity in bar graphs, by the length of the text in blobs representing file systems, etc. Orientation of arrows can indicate order of functional operations. Orientation can also be explicitly defined by a particular notation such as UML (Booch *et al.*, 1997).

### 2.4.3 Scalability

Displaying large amount of mutually related objects[18] is a problem in several application domains like geography, reverse engineering and administration of large web sites and large software systems. Extracting meaningful information from such large structures is difficult and existing approaches try to limit the number of presented objects or to structure them in some way. Some of the proposed presentation techniques are:

- multiple windows (Storey *et al.*, 1996; Shneiderman, 1998),
- hierarchical graphs (Storey *et al.*, 1996),
- graphs with differently sized nodes and edges, for example, *fish-eye* (Sarkar and Brown, 1992) and
- the CONE trees (Preim, 1998).

In the multiple window approach, individual overlapping windows are used to present large hierarchical structures. Each window presents a specific part of the hierarchy. One can open windows to display a particular level in the hierarchy, a specific neighbourhood around a selected object, a projection or flattening of the hierarchy, or the whole hierarchy.

In the hierarchical graph approach, nested graphs represent the hierarchy as a whole. One can explore the hierarchy and by opening nodes get more information about the next level of the hierarchy.

Fish-eye views are based on observations on some characteristics of human perception. The resolution at the centre of the retina is quite high which enables us to read, for example. The resolution falls outside the central region. We are able to see dominant objects and movements outside the central region, but without all details. In (Sarkar and Brown, 1992) the fish-eye technique is applied to geographic data (Figure 2.11). The user can place a focus point such that the focused region is enlarged, and other regions are reduced in size.

[18] Large amount of mutually related objects is called a complex information space in (Preim, 1998).

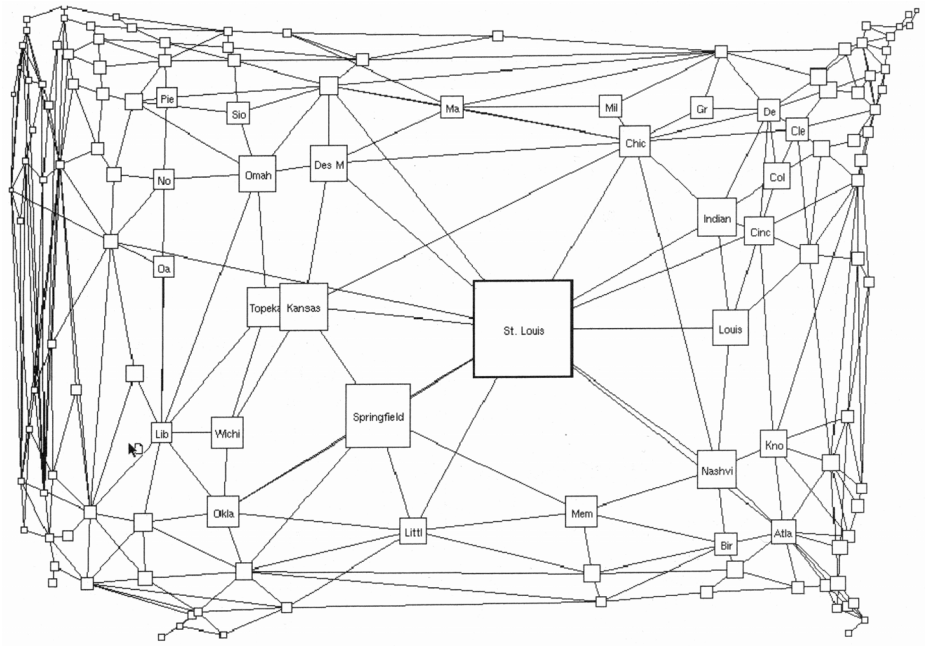



**Fig. 2.11.** Fisheye view of U.S cities, with the focus on St. Louis (Shneiderman, 1998)

CONE trees (Preim, 1998) allow presenting large hierarchies as 3D trees. Any node and its children are mapped to a 3D cone yielding a 3D model of the entire tree. Choosing a focus nodes automatically rotates cones to bring the focus to the front of the display. CAM trees (Figure 2.12) are a horizontal variant of CONE trees. They enable labelling nodes with a text.

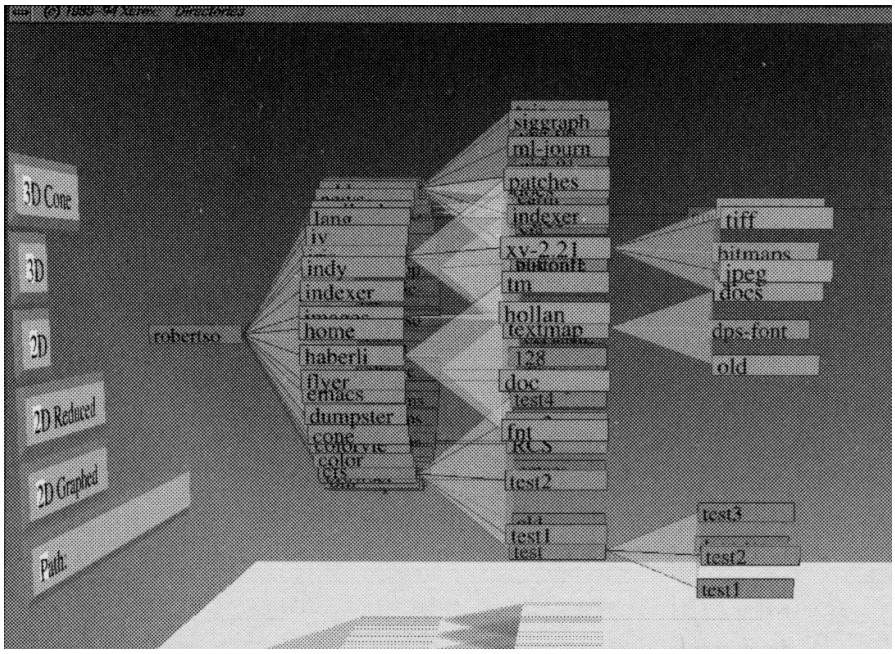



**Fig. 2.12.** CAM TREES (Preim, 1998)

### 2.4.4 Presenting Change

Visualisation on computers offers numerous opportunities for presenting change, such as dynamic highlighting or animation. Animation is used, for example, for presenting demographic change (Southall and White, 1997), support for developing software architectures (Grundy and Hosking, 2000) and for educational purposes (Jones and Scaife, 2000). Further investigation is needed to validate the benefits of the different approaches (Jones and Scaife, 2000).

## 2.5 Chapter Summary

This chapter has established a conceptual framework for our consideration of evolving object-oriented systems. We described concepts related to class and schema evolution. We considered schema evolution in the context of a database application life-cycle model and surveyed the related work. Furthermore we described the concepts from *software change impact analysis* and *information visualisation* that could be employed to support evolution.

# 3 Research Methods

The purpose of this chapter is to describe research methods used for empirical evaluation of software engineering technologies.

In general, the choice of research methods is determined by epistemological assumptions and by the type of the research question posed. Epistemological assumptions are discussed in Section 3.1. Section 3.2 describes research methods in computer science. Methods used for empirical evaluation of software engineering technologies are presented in Section 3.3. Conducting empirical studies on usability of software engineering technologies requires appropriate means for data collection. Section 3.4 discusses means for data collection during usability studies, and presents a tool we have developed. Our preliminary work on the implementation of this tool is also reported in (Karahasanović *et al.*, 2001). Section 3.5 describes the research presented in this thesis regarding the research methods used. Section 3.5 summarises.

## 3.1 Epistemological Assumptions

*Epistemology* deals with the varieties, grounds and validity of knowledge (Oxford, 1993). All research is based on some epistemological assumptions about the validity of research and adequacy of research methods.

*Modernism* is a set of epistemological assumptions summarised by the "Enlightenment's ideals of perfection and the notion, inspired by the modern science, of the infinite progress of knowledge and an infinite advance toward social and moral betterment"(Habermas, 1992). Modernism is deeply anchored in the belief that there is a rational explanation of natural phenomena or a rational solution to real world problems. It has been argued (Robinson *et al.*, 1998) that software development and software engineering are strongly influenced by modernism. The world is viewed as a composition of problems. Rational solutions to these problems are possible via application of technology.

One of the main characteristics of *postmodernism* is suspicion towards universal solutions and themes (Robinson *et al.*, 1998). Postmodernism highlights the need for local solutions. In software engineering this means for example negotiations between different stakeholders in software development.

Orlikowski and Baroudi (Orlikowski and Baroudi, 1991) suggest three categories of research based on the underlying research epistemology:

- **Positivist research**: positivist researchers assume that reality is objectively given and can be described by measurable properties independently of the observer and his instruments.
- **Interpretive research**: interpretive researchers assume that reality is accessed only through social constructions such as language and shared meanings.

- **Critical research**: critical researchers assume that social reality is historically constituted and that it is produced and reproduced by people. Focus is on the conflicts and contradictions in the modern society.

A notion of theory testing is crucial within positivism. A theory is a possible explanation of a phenomenon (Basili *et al.*, 1999). According to Popper (Popper, 1968) a scientific theory must be falsifiable, logically consistent, predictive, and its predictions must be confirmed by observations during tests for falsification. By testing theories, the positivists increase the predictive understanding of the studied phenomena.

While the positivist view is typical in natural sciences, the interpretive and critical views are typical for social sciences. Computer science involves technology, human beings as individuals, and human beings as members of organisations and societies.[19] Therefore, computer science may involve positivist, interpretative and critical research. It has been pointed out that the distinctions between positivist, interpretative and critical research are not always clear in practice (Lee, 1989). These underlying epistemologies are not necessarily contradicting; they can be combined within the same study. The research presented in this thesis is based on the positivist assumptions and influenced by the ideas of postmodernism.

### 3.2 Research Methods in Computer Science

Computer science has it sources in different disciplines, like mathematics and electrical engineering and has later included aspects from cognitive psychology and management science. This diversity gives rise to a variety of approaches within computer science. Computer science has been viewed as "not a science, but a synthetic, an engineering discipline." (Brooks, 1996) Hoare, in his influential paper (Hoare, 1984), states that "the programmer of today shares many attributes with the high priest." Software development characterised by its use of mathematical proofs "will pay the highest dividends in practical terms of reducing costs, increasing performance, and in directing the great sources of computational power on the surface of a silicon chip to the use and convenience of man." It has been claimed that one of the reasons for the success of relational databases was the mathematical rigour with which the relational model was defined (Eaglestone and Ridley, 1998). The need for empirical studies has been emphasised in software engineering (Zelkowitz and Wallace, 1998; Tichy, 1998; Kitchenham *et al.*, 1997; Basili, 1996) and in information visualisation (Chen and Yue, 2000; Catarci *et al.*, 1997; Whitley, 1997). It has been stated that empirical studies are needed to make software engineering a science rather than an art (Fenton *et al.*, 1994).

These different approaches within computer science imply use of different research methods. According to (Adrion, 1993), there are four research methods:

[19] We use here the term *computer science* in a quite broad sense; it includes also isoftware engineering and information systems research.

- ***Scientific method***. A theory is developed to explain a phenomenon; a hypothesis is proposed and alternative variations of the hypothesis are tested. Data collected during the test is used to verify or refute the hypothesis.
- ***Engineering method***. A solution to a hypothesis is developed and tested. Based upon the results of the test, the solution is improved until no further improvements are required.
- ***Empirical method***. A hypothesis is validated by statistical methods. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis.
- ***Analytical method***. A formal theory is developed and, if possible, results derived from the theory are compared with empirical observations.

Another classification (Snyder, 1994) suggests three ways of hypothesis validation within computer science. These are:

- ***Proof of Existence***. It gives evidence of the establishment of a new computational artefact. This artefact contributes to the validation of a theory. Implementation of the first object-oriented language, Simula, is an example of a proof of existence.
- ***Proof of Performance***. It gives evidence of an improvement of previous implementation. It may compare a new implementation with previous ones or it may compare competing systems. Benchmarks like the database benchmarks 001 (Cattell and Skeen, 1992) and 007 (Carey *et al.*, 1993) are used for comparison.
- ***Proof of Concept***. It is a demonstration of how a particular set of ideas achieves its objectives. It often involves a human factor analysis. A proof of concept investigates, for example, whether we can build a particular system to run in certain space or time.

Development of new methods and tools to support those methods is recognised as a dominant activity within computer science (Tichy *et al.*, 1995; Kitchenham *et al.*, 1997; Zelkowitz and Wallace, 1998). It is necessary to establish a new computational artefact when presenting new ideas (*engineering method* or *proof of existence*) or when applying knowledge from one research area to another. However, "progress comes when what is actually true can be separated from what is only believed to be true" (Basili *et al.*, 1999). This can be achieved by empirical evaluation of the proposed methods and tools (*empirical method* or *proof of performance*). Thus, engineering and empirical methods do not exclude but complement each other.

### 3.3 Empirical Evaluation of Software Engineering Technologies

Empirical evaluation of software engineering technologies[20] contributes to software engineering both theoretically and practically. Empirical testing of hypotheses related to the usability or performance of a particular technology contributes to the validation of an underlying theory.[21] Empirical results on patterns of tool usage may provide

[20] By software engineering technologies, we mean software engineering methods and tools.
[21] This is called theory-testing research in (Jarvinen, 1999).

useful guidelines to the designers of similar tools or be used as a basis for formulating theories.[22] One can evaluate technical aspects of a tool (performance). Further, one can evaluate how the tool helps the users to conduct their tasks (usability) or evaluate the tool in the wider context of an organisation or a society.

Figure 3.1 depicts evaluation of a technology. Based on a problem, a researcher generates a hypotheses related to a technology. During the empirical validation of the hypothesis the researcher exploits different evaluation methods. The results of the validation may support or refute the hypothesis. Both outcomes contribute to building a body of knowledge. By replicating evaluation studies, their validity can be increased (Basili *et al.*, 1999). New hypotheses can be formulated during the evaluation.[23]

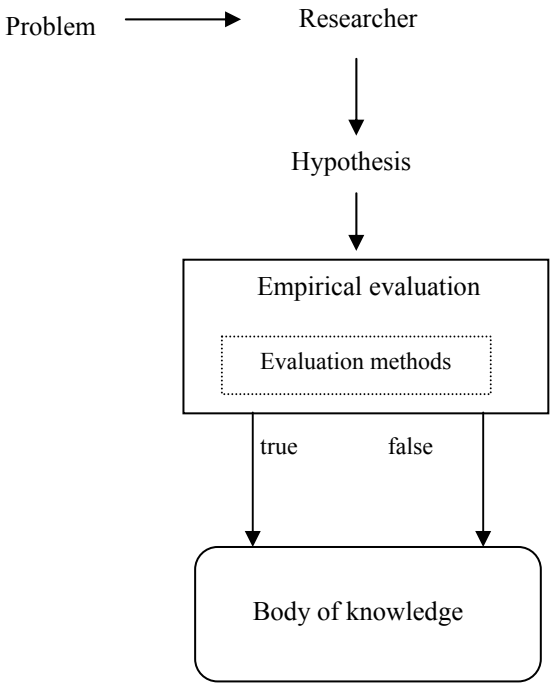



**Fig. 3.1.** Empirical evaluation of software engineering tools

### 3.3.1 Evaluation Methods

Depending on the purpose of an evaluation one can choose between several methods for evaluating software engineering technologies. Robson (Robson, 1993) identifies three major evaluation methods (research strategies): surveys, case studies and experiments. Kitchenham (Kitchenham, 1996) divides nine evaluation methods into

[22] This is called theory-building research in (Jarvinen, 1999).
[23] For simplicity, this is not presented in Figure 3.1.

three general categories: qualitative evaluations, quantitative evaluations and benchmarks. A more comprehensive classification is given in (Zelkowitz and Wallace, 1998). Twelve different evaluation methods are grouped into three categories: observational, historical and controlled. We describe below the five most commonly used evaluation methods. We also present a method often used in other research fields than software engineering — the controlled one-subject explorative study method.

#### *3.3.1.1 Literature Search*

Literature search[24] is a feature analysis based on literature describing technologies. The evaluation is performed by a single investigator (a person or a team). The investigator analyses the documentation, published articles and other literature describing a technology. It is suitable for evaluating large number of technologies or as a first stage in a more complex evaluation (Kitchenham, 1997). It can also be used to confirm a hypothesis on the basis of previously published data, using a quantitative synthesis method called meta-analysis (Zelkowitz and Wallace, 1998). The evaluator is responsible for identifying candidate technologies, identifying evaluation criteria, compiling information, scoring each feature, analysing the scores and presenting the report (Kitchenham, 1997).

The main advantages of the literature search method are its flexibility and low costs. It is therefore used very often. A major weakness of this method is subjectivity in selecting both features and their importance (selection biases). When using the literature search method for technology evaluation in an industrial context, one has to be aware that the evaluator have a different technical education and experience than the potential users, and therefore may not be representative.

The documentation describing commercial products may also be incomplete. In a study of the Oracle8i Database Management System (Section 4.2), for example, we experienced that some features were not described (both supported and unsupported features). It was left to the user to discover some features by the 'trial and error' method.

The analysis of published papers may be biased by the tendency of authors and journal editors to publish positive results (Zelkowitz and Wallace, 1998). It has been recommended by Tichy (Tichy, 2000) that negative and 'obvious' results should also be published. This would increase the validity of meta-analysis based on published papers.

#### *3.3.1.2 Survey*

A survey is often an investigation performed in retrospect, when a technology has been used for a while (Pfleeger, 1994). Qualitative and quantitative data are collected by interviews and questionnaires. According to Kitchenham (Kitchenham, 1997) the evaluator is responsible for:

- identifying the candidate technologies,
- identifying the evaluation criteria ,

[24] It is called qualitative screening in (Kitchenham *et al.*, 1997).

- designing and running the survey including: deciding on the type of the survey (postal, e-mail, personal), creating the survey documentation (questionnaire, instructions), selecting participants, and running the survey, and
- analysing the answers.

The main advantages of a survey are that it can be conducted on real-life projects (like a case study) and can be replicated (like an experiment). Further, it can provide results in shorter time than a case study and is less human intensive than an experiment (Kitchenham, 1997). The main disadvantage is that the evaluator cannot control the experience, background and capability of the participants (Kitchenham, 1997).

Surveys based on questionnaires may cover a large sample of the population. However, one must be aware of the potential biases of questionnaires. The results of an investigation on quality of questionnaire based software maintenance studies show that several problems with questionnaires can appear. These are inconsistency in the use of words, inconsistency in opinion, incompleteness and incorrectness in answering questions (Jørgensen, 1995b).

#### *3.3.1.3 Assertion*

An assertion is an experiment where a developer of a technology evaluates the technology by using it. This method is often used for a preliminary evaluation of a technology. The survey of papers published in 1985, 1990, and 1995 in *IEEE Transactions on Software Engineering*, *IEEE Software* and the proceedings of the *International Conference on Software Engineerin*g (Zelkowitz and Wallace, 1998) examined 562 papers. Nearly a third of the papers reported use of the assertion method. However, the percentage decreased from 37 percent in 1990 to 29 percent in 1995.

Use of the assertion method can be 'use the tool to test a simple 100-line program to show that it can find all errors' (Zelkowitz and Wallace, 1998) or it can be conducted in an industrial context. As the developer is both experimenter and subject of study, this method is considered potentially biased. However, if the experiment is done in the context of a large industrial project, it can be classified as a case study, since the experimenter has no influence on experimental conditions.

#### *3.3.1.4 Case Study*

The case study method is an empirical enquiry that investigates a contemporary phenomenon within its real-life context when the boundaries between phenomenon and context are not clearly evident (Yin, 1994). In a case study an evaluator monitors a real project and collects data over time. Certain attributes like product quality and developer productivity are monitored during the project and data measuring them is collected. When case studies are used in feature evaluation of technology, the evaluator (evaluation team) is responsible for (Kitchenham, 1997):

- selecting the technology to be evaluated
- identifying evaluation criteria

- selecting one or more pilot projects that will try the technology. The pilot project should be typical for the organisation. Software developers use the technology in the pilot projects and score each feature of the technology.
- analysing the scores and report the results

The main strength of the case study method is that data is collected in a real life context. With a relatively minimal addition to project costs we can collect valuable information about the project and use of the technology (Zelkowitz and Wallace, 1998). However, the prerequisite for conducting a case study is that an organisation is attuned to the need for conducting the study. The importance of ensuring usefulness of the results for the organisation is emphasised in (Arisholm *et al.*, 1999). The weakness of the case study method is that it is very context biased (each development is unique). Furthermore, the presence of the evaluators and the process of evaluation may have the side effects of making the staff aware of some issues and influences the results.

Case studies can be used for evaluating commercial tools and research prototypes. Evaluating research prototypes in industrial context has, in our opinion, some additional weaknesses. The developers need to learn on the prototype. The prototype is typically not so well documented and tested as commercial tools. The use of the prototype may cause delays of the project and may be a risk that management is not willing to undertake. A solution may be that the researcher uses the prototype in the industrial context.

#### *3.3.1.5 Experiment*

During an experiment several subjects perform a task in multiple ways. The researcher set factors that can influence the results[25], e.g., duration and staff experience. Therefore, it is possible to provide a greater level of statistical validity than in case studies. In the Computer-Aided Empirical Software Engineering framework (Tari and Li, 1994) an experiment consists of five subprocesses:

- ***Needs Analysis —*** the phase where the problem is identified, and goals and purposes of the study are specified and hypotheses are formulated.
- ***Experiment design —*** the phase where detailed specification of the experiment is given. This specification includes experimental tools, number and type of subjects and analysis methods and tools.
- ***Experimentation —*** the phase where the experiment is conducted. Experiments are monitored and data is collected.
- ***Data Analysis —*** the phase where the collected data is analysed.
- ***Packaging —*** the phase where "knowledge" obtained from a series of experiments is compiled into a "package".

The costs of conducting experiments with developers from industry with professional development tools are very high. Therefore, the experiments are often conducted with

[25] Called control variables (Zelkowitz and Wallace, 1998) or state variables (Kitchenham *et al.*, 1997).

students instead of professionals and with pen and paper instead of professional development tools. Effects of these substitutions on validity of the experiments are rarely reported. A study has been conducted to compare presentations of students and professionals in lead-time impact assessment (Høst *et al.*, 2000). No significant difference was found. On the contrary, significant differences between 'pen and paper' and 'tools' tests has been reported in a study comparing Query by Example with SQL (Catarci, 2000). The generality of these results needs further exploration. As recommended by Tichy (Tichy, 2000) studies with students can be used to eliminate alternative hypotheses and to test experimental design.

#### *3.3.1.6 Controlled One-Subject Explorative Study (N=1 Experiment)*

Explorative studies of one subject over a longer period of time in a laboratory environment are often used in the fields of psychology, psychiatry and education. It has been argued that this method, also called N=1 experiment, could be useful in software engineering (Harrison, 2000). The classical A-B design (Barlow and Hersen, 1984) of a controlled one-subject explorative study defines two phases of a study (*A* and *B*). During the first phase the subject is observed without any particular treatment. During the second phase a treatment is applied on the subject. In software engineering the treatment is use of a technology. In such studies it is assumed that the performance of the subject "stabilises" after a period of time in each of the phases. After some stabilisation point, the experimenter attempts to measure effects of the use of a particular technology on the performance, and not effects of learning the technology, learning the application, or stress caused by the experimental situation. When used for generating hypotheses, one-subject controlled explorative study (Harrison, 2000) is similar to a case study but imposes a rigid control on dependent and independent variables.

## 3.4 Data Collection during Usability Evaluation of Software Engineering Tools

Conducting empirical studies in general, and usability evaluation of software engineering tools in particular, require efficient and reliable data collection. To conduct a series of studies on usability evaluation of software engineering tools, we envisage a tool that automatically collects data about subjects, their interaction with the tool under study and their evaluation of the tool under study. This section discusses techniques and tools for data collection during usability studies, and presents a tool that we have developed.

### 3.4.1 Means for Data Collection

Data collection during the usability evaluation can be done manually, automatically or as a combination of these two. The means for manual data collection are fill-in questionnaires, interviews, observations, and programming tasks with pen and paper. Manual data collection is often used in human-intensive experiments (Wohlin *et al.*, 1999).

Manual data collection has some disadvantages. The validity of the results can be decreased by missing data as reported, for example, in an experiment on maintainability (Briand *et al.*, 2001). Transforming data from paper to a computer readable form during experiments with manual data collection requires a vast amount of resources (Arisholm *et al.*, 2001; Jørgensen and Sjøberg, 2000).

Automatic data collection includes the use of tools for logging the user's actions *e.g.,* audio/video recording and special purpose equipment. A monitoring system was developed to study the use of the *sem* text editor (Kay and Thomas, 1995). This system was used in a three-year study of over 2000 users. The source code of the editor was extended to log invoked commands and operators. Similarly, the Napier88 Workshop was extended to log the use of tools and operations (Welland *et al.*, 1997) and was used in a usability evaluation of the Workshop. The Ginger-2 Computer-Aided Empirical Software Engineering framework includes logging of *emacs* commands, windows commands, mouse movements and keystrokes (Torii *et al.*, 1999). These logging tools do not disturb the user and provide a detailed picture of the user's actions. However, it may be difficult to relate low-level command logs with users' intentions (Welland *et al.*, 1997).

The think-aloud protocol (von Mayrhauser and Lang, 1999) is a standard way of collecting behavioural data. The users are asked to tell what they are thinking and doing during the experiment and the experiment is audio recorded. This protocol enables useful qualitative information collection about how the technology under study is used. It should be noted that the think-aloud collects subjective information. The experiment participants describe their own thoughts and actions themselves. Audio recording disallows conducting experiments with a group of students working in the same laboratory. Some participants of a study were disturbed by the think-aloud protocol and spoke little (Bratthall *et al.*, 2001).

Special purpose equipment for motion measurement, skin resistance level measurement, eye tracking, and video recording can provide quite complete data collection during usability evaluation studies. Collection of behavioural data by special purpose equipment can be very advanced, such as Ginger-2 (Torii *et al.*, 1999). This system includes audio and video recording, eye tracking systems, 3D-motion measurement and skin resistance level measurement. The audio and video recorder can be used to record behaviour, speech protocol or screen images. The eye tracking system tracks where the eyes of the experiment participants are currently focused. The 3D-motion measurement system collects data such as movement of the participant's head, hands and shoulders. The skin resistance level measurement system captures data on how much resistance the participant's skin has when a low level of electricity is given. Continuous data types like skin resistance level are transformed into discrete data and then coded into a special designed language. Data collected in such a manner is rather complete and objective. However, such equipment is quite expensive and may be unpleasant for experiment participants. The participants wear special eyeglasses and have attached sensor pads and movement balls. The equipment also needs individual calibration. Problems with processing large quantity of collected data may also appear (Torii *et al.*, 1999).

### 3.4.2 Logging Tool

We posed the following requirements on our logging tool. It had to automatically collect data about:

- the subjects (including personal information and the user's evaluation of the technology under study) and
- the interaction between the subjects and the software technology under study.

We were restricted to low costs of building of the tool. Furthermore, we wanted the tool to be obtrusive in the less possible degree.

The collection of personal information is done in the Personal Information Questionnaire screen. The questionnaire is given in Appendix A.1. Information regarding the user's evaluation of the technology under study and the evaluation of the logging tool are collected in a similar way. The User Evaluation Questionnaire screen (Figure 3.2) appears at the end of the experiment. The questionnaire is given in Appendix A.2. This questionnaire is based on the IBM Post-Study System Usability Questionnaire (Lewis, 1995). It consists of statements addressing the usability of the technology under study and adequacy of the logging tool. The users express their agreement or disagreement with statements by selecting a number on a seven-point scale (see Figure 3.2). Score 1 is used when the subjects of the experiment fully agree with the positive statements on the usability of the technology; score 7 when they fully disagree.

The logging tool was used for evaluation of a tool, SEMT, which identifies and visualises impacts of schema changes on applications (Chapter 5 and 6). The statements presented in Figure 3.2 address the usability of SEMT.

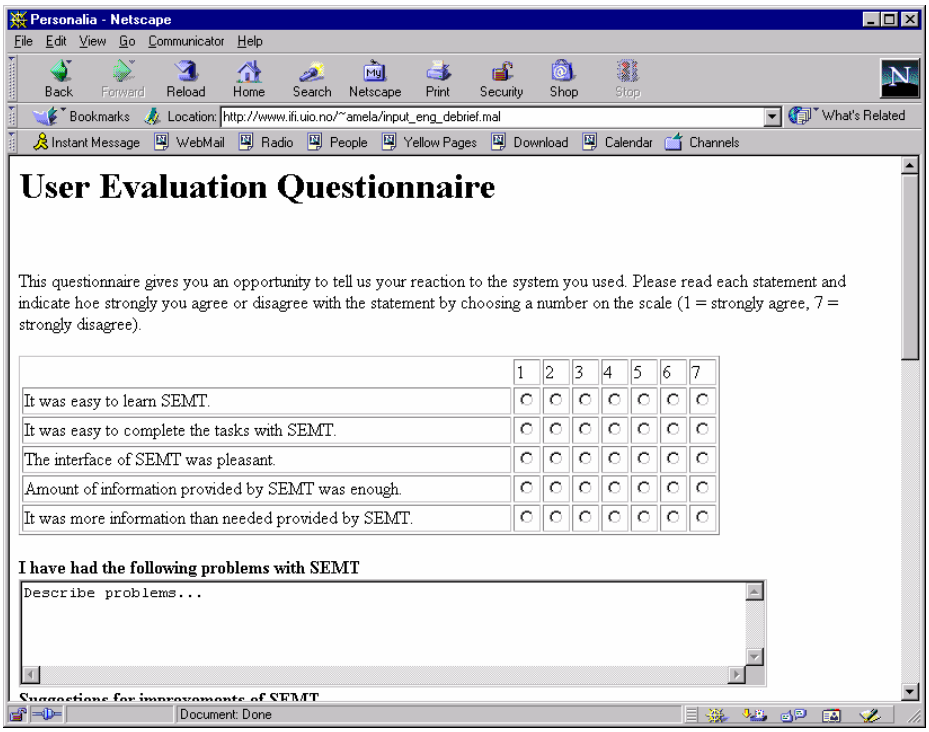



**Fig. 3.2.** User evaluation questionnaire screen.

We have developed a *'think-aloud' screen* to collect behavioural data. The experiment participants are asked to briefly write what they are thinking during the experiment in the screen. To remind them, the screen appears every 10 minutes and the text *'What are you doing now? What are you thinking and feeling?'* appears. The 'think-aloud' screen is given in Appendix A.3. These comments written by the subject are saved in a text file, and the time used for writing them is recorded. Figure 3.3 gives an excerpt from a think-aloud file. The comments written by the subject of the experiment are given in bold type.

***** Input given 20:28:25, 24.7.2001 ***** username isueea
**I have used SN to find all the classes and their methods that access the LibraryBorrowerInformation class and now I am editing this class.**
***** Input given 20:39:12, 24.7.2001 ***** username isueea
**I am editing all the different files that refer to the class LibraryBorrowerInformation**.

**Fig. 3.3.** Comments in the think-aloud log file for the user *isueea*

Data about interaction between the subjects and the operating system is similarly collected. SEMT is implemented in the context of *unix*. Usage of *unix* commands is logged for each command in the following format:

*start time* : *command argument1 argument2 ...*
*end time*

Figure 3.4 gives an example of the *unix* log file. End time for each command is given in bold type.

19:42:18 20/7 2001: cd bo
**19:42:18 20/7 2001**
19:42:19 20/7 2001: ls
**19:42:20 20/7 2001**
19:42:40 20/7 2001 : grep State *.java

**Fig. 3.4.** Unix log file

All keystrokes and operations on the mouse buttons within an adapted *emacs* window are recorded. The logging tool records usage of commands of SEMT in a file together with execution timestamps and user name. Figure 3.5 gives an example of the SEMT log file. The user isueea displayed the components of the theatre application (Display Search Space – Theatre), clicked on the node representing a class (Click Node), and displayed the fields of the class (Expand Fields).

Tue Oct 02 18:18:00 GMT+02:00 2001| isueea |Display Search Space – Theatre
Tue Oct 02 18:18:30 GMT+02:00 2001| isueea |Click Node
Tue Oct 02 18:18:42 GMT+02:00 2001| isueea |Expand Fields

**Fig. 3.5.** SEMT log file for the user *isueea*

The logging mechanism is implemented in the context of *unix*. The think-aloud and questionnaire screens are implemented in HTML. Logging of *emacs* usage is

achieved by modifying the source code of *emacs* in LISP. Logging of *unix* commands is implemented in *perl*. The *Unix* Window Manager is modified to record window operations. Logging of SEMT commands is achieved by modifying the source code of SEMT.

## 3.5 Research Methods and Questions

The research questions of this thesis raised in Section 1.3 can be described with respect to the research methods used. The relation between the research questions and research methods is summarised in Table 3.1.

**Table 3.1. Summary of research questions and methods**

| Research questions | | Research methods | Presented in |
|---|---|---|---|
| **RQ1** | How do the existing schema evolution tools support application consistency? | Assertion;<br>interviews;<br>qualitative screening; | Chapters 2 and 4 |
| **RQ2** | To what extent can former work within the field of *software change impact analysis in programming languages* contribute to identify impacts of schema/class changes on the applications? | Implementation (SEMT);<br>case study; | Chapters 5 and 6<br>Papers (Karahasanović and Sjøberg, 1999; Karahasanović, 2000, Karahasanović, 2001) |
| **RQ3** | What is the trade-off between a textual and visual presentation of the impacts of schema/class changes? | Implementation (SEMT)<br>case study;<br>experiment;<br>N=1 experiment | Chapters 5 and 6<br>Paper (Karahasanović and Sjøberg, 2001) |
| **RQ4** | How should schema and class evolution technologies (including conducting empirical studies) be evaluated regarding their support for application consistency? | Implementation (logging tool)<br>Experiment;<br>N=1 experiment;<br>assertion | Chapters 3 and 6<br>Paper (Karahasanović *et al.*, 2001) |

## 3.6 Chapter Summary

This chapter has discussed research methods used in computer science. Computer science has arisen out of different disciplines, like mathematics and electrical engineering and has later included aspects from cognitive psychology and management science. This diversity gives rise to a variety of approaches within

computer science and implies use of different research methods. According to Salomon,

> *"Knowledge is commonly socially constructed, through collaborative effort toward shared objectives or by dialogues and challenges brought about by differences in persons' perspectives."*
>
> *(Salomon, 1993)*

Likewise, we believe that the different approaches within computer science and different research methods all contribute to the body of knowledge. Empirical validation of the usability of a tool, for example, may contribute to the validation of an underlying theory, generate new hypotheses or provide guidelines to the designers of similar tools. Numerous evaluation methods are available. We have discussed their strengths, weaknesses and validity. We have surveyed the means for automatic collection during usability evaluation of software engineering tools. We have also presented a tool we have developed to conduct the studies on usability evaluation of software engineering tools. This tool has been used in the studies reported in Chapter 6. Section 6.5 reports our experience with this logging tool.

# 4 Supporting Application Consistency – Examples from Current Practice

This chapter describes two studies that we have conducted to get insight into current practice regarding how application consistency is maintained during schema evolution. Relational and object/relational databases are widely used in the industry. Therefore, we collected anecdotal evidences on problems in current practice related to application consistency in the context of relational and object/relational databases, and object-oriented languages.[26] We reckoned that the problems related to application consistency in this context are similar to the problems that are present in the context of object-oriented databases.

Many commercially available CASE tools, database administration tools and data dictionaries are used to support schema changes. The functionality of these tools and the way they are used in an organisation may affect the maintenance of application consistency. To explore this we have interviewed database administrators in three large organisations. Section 4.1 describes the results of the interviews.

The Oracle corporation currently controls around 40% of the database systems market and is the biggest supplier of database management systems (Serwer, 2000). The way the Oracle products support application consistency has therefore consequences for numerous companies and governmental organisations. Section 4.2 describes the results of a study we conducted to evaluate the Oracle8i database management system (Leverenz and Rehfield, 1999) regarding its support for application consistency.

Based on the results of these two studies, Section 4.3 identifies lines of research that need further investigations.

## 4.1 Supporting Application Consistency in an Industrial Context

To get insight into how application consistency is supported by database management systems and tools widely used in industry, we interviewed three experienced database administrators. The data collection and analysis were qualitative.

Section 4.1.1 describes the data collection and analysis. Section 4.1.2 presents the results of the interviews. Section 4.1.3 highlights the limitations of the data collection method. Section 4.1.4 summarises the results.

### 4.1.1 Data Collection and Analysis

Open-ended interviews (Yin, 1994) were used for data collection. The author interviewed the three subjects. The interviews were guided by a simple questionnaire developed by the author. The questionnaire is presented in Appendix B. Each

[26] One application developed in COBOL was also described.

interview lasted between one and two hours. The subjects were selected due to their experience. The notes from the interviews were processed using the *constant comparison method* (Seaman, 1999). Pieces of text relevant to the subject of the study were coded by attaching labels to them and grouped according to the codes. To assure quality of the results and confidentiality, the results were presented to the interviewees. They were asked if there was something that they wanted to change or remove from the report. The results presented here are based on such modified report.

### 4.1.2 Results

Table 4.1 gives an overview of the related experience of the subjects. They are all responsible for database administration in large organisations, have many years of experience and are well educated. They have experience with different database management systems, CASE tools and database administration tools.

**Table 4.1.** Overview of related experience of the subjects

| Question | Subject 1 | Subject 2 | Subject 3 |
|---|---|---|---|
| **Current work title** | Database administrator | Database chief | Database administrator |
| **Organisation** | Banking and insurance company | Software development company | Governmental organisation |
| **Work experience** | 10 years | 10 years | 25 years (18 years with databases) |
| **Education** | MSc | MSc | BSc |
| **Experience with DMBS** | DB2 | Oracle, Sybase, Informix | Oracle, Sybase |
| **Experience with CASE tools and DBA tools** | BMC, ENDEVOR, COOL | Rational Rose, Power Builder | Oracle Designer, Enterprise Manager, TOAD |

Subject 1 had experience as a database administrator in a large Norwegian banking and insurance company (System A) where he has been working for 10 years.

Subject 2 had experience with two different database application systems. The first one (System B) was developed by a software development company located in Norway and delivered as a standard product to about 2100 customers. The second application system (System C) was developed by a company located in the USA. The branch of the company located in Norway was responsible for adaptations of the product to the local market.

Subject 3 worked many years as a principal database administrator at the two largest Norwegian oil platforms. She now works as a DBA in a governmental organisation. The databases and applications in the oil industry were significantly larger in size and complexity. However, the subject said that the problems related to application consistency are basically the same. Table 4.2 gives an overview of the systems described by the subjects.

**Table 4.2.** Overview of the database application systems

| Characteristic | System A | System B | System C | System D |
|---|---|---|---|---|
| **Application domain** | Banking and insurance | Customer relationship management | Account management, logistics | Account management, marketing |
| **DBMS and supporting tools** | DBMS DB2; Administration tool BMC[27]; Versioning tool ENDEVOR[28]; Design tool COOL[29]; Applications in Cobol, PLM | DBMSs Oracle, Sybase, Informix, SQL Server; CASE tool Rational Rose; In-house developed data dictionary | DBMS Sybase, In-house developed CASE tool | DBMS Oracle; Administration tools TOAD[30], Oracle Enterprise Manager; Applications developed by Oracle Forms, Reports, Power Builder |
| **Size and complexity** | Several hundred databases; average number of relations in a database is approximately 30 | About 200 relations and 1000000 C++ LOC | About 80 databases, about 3500 relations; about 11000 stored procedures | About 100 relations and 130 objects (screens and reports); applications developed in-house or by others |

The database administrators were asked to describe how they change the schemata (Table 4.3).

**Table 4.3.** Overview of the ways to specify schema changes

| System | Specification of schema changes |
|---|---|
| **System A** | Directly by SQL statements or by the design tool |
| **System B** | By an in-house developed program |
| **System C** | By an in-house developed CASE tool |
| **System D** | By Oracle Enterprise Manager |

The commercial and in-house developed programs and tools that are described by the subjects provide semi-automatic support to database administrators in specifying schema changes, identifying impacts of the changes on the rest of the schema, and conducting the change. The program/tool has an interface to system catalogues that contain the database schema and information about users, access rights and indices. A database administrator uses a tool/program to specify schema changes, to generate data definition files, and to verify and report correctness of a new schema. An in-

[27] BMC is an administration tool for DB2 (BMC, 2002).
[28] ENDEVOR is a versioning and configuration management system developed by Rainier Associates.
[29] COOL Enterprise is a design tool (COOL, 2002).
[30] TOAD is developed by Quest Software.

house developed program that is used to maintain System B generates files for data conversion. Oracle Enterprise Manager that is used to maintain System D also generates files for data conversion. Its Dependency Analyser helps identifying impacts on triggers and stored procedures of the database schema.

The subjects were asked how they identify impacts of schema changes on applications (Table 4.4).

**Table 4.4.** Overview of the ways used to identify impacts of schema changes

| System | Administration tool | Documentation | Knowledge of the system |
|---|---|---|---|
| **System A** | Yes | Yes | Yes |
| **System B** | | Yes | Yes (more than 50%) |
| **System C** | | Yes | Yes |
| **System D** | | Yes | Yes |

The administration tool BMC is used to identify impacts of schema changes on applications in System A. The administration tool produces a list with programs affected by the change. Correctness of the report with affected modules produced by the administration tool depends on the developers. They must have good knowledge of the system, the versioning tool, and the administration tool to maintain the data model and the DDL files in a consistent manner. Different groups of developers also manually maintain overviews of the changes in their 'private' documents.

In System B, an API was developed that connects applications with a database. This API is a set of wrapper classes developed on the top of ODBC classes. The applications access the database only via this API. Thus, impacts of schema changes are limited to the API classes. To identify impacts of schema changes on this API developers use Rational Rose class diagrams and overviews of the system written in Word files. These files are manually maintained and not always up-to-date. The good knowledge of the system is essential for identifying impacts and the subject said that 'the developers rely on their own knowledge of the system in more than 50% of cases'.

In Systems C and D, the developers used documentation and their knowledge of the system to identify impacts of schema changes on the applications.

Typical size of the application modules that are affected by a schema change and that can identified by the tool (for System A) or by the developers (for other systems) is between 100 and 500 LOC.

The database administrators were also asked to describe the strategies that they used to limit the problems related to application consistency. Table 4.5 gives an overview of these strategies.

**Table 4.5.** Overview of the strategies used to limit problems related to application consistency

| Strategy | System A | System B | System C | System D |
|---|---|---|---|---|
| Avoiding the change | | Yes | | Yes |
| Minimising impacts by 'good' design | | Yes | | |
| Delegating responsibility | Yes | Yes | | Yes |
| No special strategy | | | Yes | |

In System B, one tried to avoid the change as long as possible. There are about 2100 customers and the change of the schema, conversion of the data and applications are very expensive. Application domain of this system is described by the subjects as quite stable.

In System D, one also tried to avoid the change whenever possible. As there will be no technical support for Oracle 7, the system has to be moved from Oracle 7 to Oracle8 or Oracle 9. However, due to the high cost of schema changes[31] the decision was made to make no changes to the schema. This means that new, better features offered by the DMBS, such as partitioning and clustering, will not be exploited.

In System B, one tried to minimise impacts of schema changes during the design phase of the system. All database related code is isolated in an API and impacts of the schema changes are limited to the API classes.

In Systems A, B and D, only one person is responsible for conducting changes of the schema, updating the data model, generating DDL files and generating test databases. This is believed to be important for limiting the problems related to the maintaining schema and application consistency.

System C was generated by a CASE tool developed in the USA branch of the company. Due to limited functionality of this tool, parts of the application that was developed in Norway had to be implemented from scratch, and no special strategy was used.

The subjects also pointed out the following:

- None of CASE tools and third part tools for database administration is 'complete'. Several tools have to be used simultaneously; this is described as 'a nightmare' by one subject.
- The database administrator should be included early in the design of a database application system. This would provide better quality of the schema ('less complex design') and it would consequently be easier to change the applications.

### 4.1.3 Threats to Validity

The main threat to the validity of the results in this section is that we interviewed only three subjects. However, considering the level of their experience and variety of the database management systems, CASE tools and database administration tools that

[31] The costs related to schema changes due to its integration with an economy system were about one million NOK.

they used, we believe that the current practice in supporting application consistency is similar in other organisations within similar areas of business. Another threat to validity is that both the choice of the questions and the interpretation of the results may be biased by the interests of the researcher.

### 4.1.4 Summary of Results

The results of the interviews indicate that trying to avoid the change as long as possible is a quite common practice. Furthermore, 'human factors' are quite important for maintaining application consistency. These are good knowledge of the supporting tools, discipline in their use, discipline in maintaining documentation, and good knowledge of the schema and the applications. 'Good design' and isolating database related parts of an application also seem helpful for maintaining application consistency. When adapting applications to schema changes, developers rely in a large degree on their own knowledge of the applications or 'private' documentation. This indicates that there is a place for improvement of the existing tools that are used to maintain application consistency.

## 4.2 Application Consistency Using the Oracle8i Database Management System

This section describes a study we conducted to evaluate the Oracle8i database management system (Leverenz and Rehfield, 1999) regarding support for application consistency. Views are commonly used to provide authorisation in databases and to simplify complex operations on databases. It has been claimed that views are helpful for maintaining application consistency (Bertino, 1992; Tresch and Scholl, 1993; Breche *et al.*, 1995; Bellahsene, 1996; Young-Gook and Rundensteiner, 1997). The purpose of the study presented here was to investigate whether the use of views may decrease the amount of work needed to preserve application consistency in Oracle8i.

Section 4.1.1 describes the method used for the evaluation. Section 4.1.2 presents and discusses the results. Section 4.1.3 highlights the limitations of the evaluation method.

### 4.2.1 Method

We have developed two versions of an application for customer relationship management. Two versions of the application had the same functionality. The first one accessed the database directly and the second one by views. Changes of the underlying database schema were conducted and these two versions of the application were adapted to the changes. For each change and for each application the number of lines, classes and files were measured before and after the change. This gave an indication of efforts needed to preserve consistency between the schema and the applications.

#### *4.2.1.1 Application under Study*

The application provides functionality for browsing, deleting and updating information about employees, departments, customers and their orders. It is an extension of the *company database* that is commonly used as an example in the database literature (Elmasri and Navathe, 1994). The database consists of six relations. The ER diagram of the company database and an example of the applications user interface are given in Appendix C.

The JDeveloper[32] wizards (Oracle, 2001) and the Business Components[33] were used to generate the application source code in Java and XML. The application model consisted of three tiers. At the lowest tier was the database server, in the middle tier were the Business Components and at the highest tier were the client applications or the user interface applications. Each Business Component is represented by an XML file and one or more Java files. The XML file stores metadata, while the Java files stores the code that implements application specific behaviour. An *Entity Object* is a Business Component that encapsulates business logic for a database relation or view. An *Association* is a Business Component that specifies a relationship between two

[32] Oracle JDeveloper is an integrated development environment for building, debugging and deploying component-based applications. It provides visual tools and wizards for application development.

[33] Oracle Business Components are Java classes customised for database access.

Entity Objects. The JDeveloper wizards generate default values for associations, but also allow user to define values. Figure 4.1 depicts the two versions of the application.

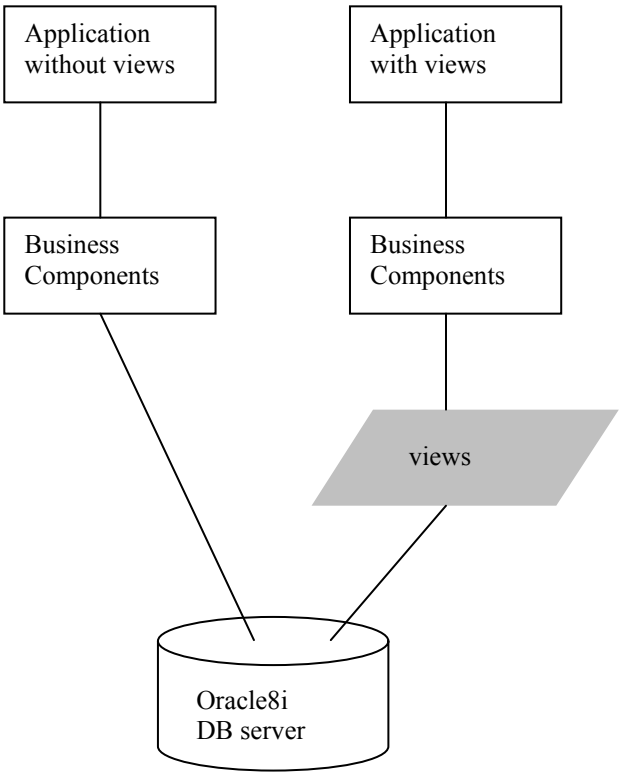



**Fig. 4.1.** Two different ways of accessing the database: without views and with views

The data about these two versions of the application is given in Table 4.6. Although the two versions have the same functionality, there was a slight difference in number of lines of code, classes and files. The version with direct access to the database was completely automatically generated. The version with views was also automatically generated but there was one exception: the associations describing the foreign integrity constraints were manually defined in the XML files. The reason is that JDeveloper generates no default association when views were used. Thus, the larger number of XML files was generated automatically (version without views) than manually (version with views).

**Table 4.6** The total number of lines of source code (LOC), functions and files in the two versions of the application

| Application | LOC | Classes | Files | XML Files | Java Files |
|---|---|---|---|---|---|
| Application without views | 5406 | 172 | 41 | 20 | 21 |
| Application with views | 5121 | 164 | 38 | 17 | 21 |

#### *4.2.1.2 Data Collection*

We were interested in measuring the effort needed to adapt the applications to a schema change. Empirical results shows that the effort needed to develop a program is highly correlated with number of lines of source code (LOC) (Henderson-Sellers, 1996). LOC is widely used as a size/complexity metric due to its simplicity, ease of application and its intuitive appeal (Henderson-Sellers, 1996). Number of lines of code added/deleted may be a good indication of the effort needed to adapt the applications to a schema change. The number of classes and files added/deleted may also indicate this effort.

The number of lines of code was measured for the two versions of the application using *MS Word 97*. This was done before and after each change. Each line of code, whether it was a comment line or a line of code was taken into account. Blocks comprising of more than two blank lines were ignored. The number of classes was measured using an *awk* script. The *awk* is a *unix* script language for pattern recognition and it used here to search and count classes. The number of files was measured by the *unix ls* command.

### 4.2.2 Results and Discussion

Table 4.7 presents the results. The letter *S* indicates that the same number of places was affected for these two versions of the application. The letter *N* indicates that the application was not affected by the change. For example, adding a relation had no effect on the application in the approach we selected.

We used a taxonomy of schema changes for relational databases (Roddick *et al.*, 1993; Kelley *et al.*, 1995) and conducted one change of each type. The changes were conducted on the relation *Employee*. This relation is used by most of the applications functions. We assumed therefore that the changes of *Employee* affected more places in the application than changes of other relations.

We defined views for different users of the application. A user, for example, could access data about a customer without the *Credit_limit* attribute. The wizards in JDeveloper use predefined templates for generating applications and these templates pose restrictions on the use of views. The views can access attributes from only one relation. As the JDeveloper wizards were used for code generation, the views could access attributes from only one database relation.

**Table 4.7.** Impacts of schema changes on two versions of the application in the number of lines of source code added/deleted, number of classes and number of files.

| Schema changes | Application without views | | | Application with views | | |
|---|---|---|---|---|---|---|
| | LOC | Classes | Files | LOC | Classes | Files |
| Add an attribute | S(110) | S(6) | S(4) | S(110) | S(6) | S(4) |
| Delete an attribute | S(95) | S(6) | S(4) | S(95) | S(6) | S(4) |
| Rename an attribute | 30 | 2 | 5 | 0 | 0 | 0 |
| Modify a domain of an attribute | S(64) | S(6) | S(4) | S(64) | S(6) | S(4) |
| Add a relation | N | N | N | N | N | N |
| Delete a relation | S(562) | S(22) | S(10) | S(562) | S(22) | S(10) |
| Rename a relation | 13 | 0 | 2 | 0 | 0 | 0 |
| Split a relation | 1148 | 59 | 14 | 0 | 0 | 0 |
| Merge two relations[34] | 1148 | 59 | 14 | 0 | 0 | 0 |
| Partition a relation | N | N | N | N | N | N |
| Merge partitions of a relation | N | N | N | N | N | N |

Among 11 schema changes, 8 affected the applications. For 4 out of these 8 schema changes, the number of affected places was reduced by use of simple views as implemented in the Oracle8i database management system.

Several schema changes affected exactly the same number of places in the two versions of the application. Deleting the attribute *Salary* of the *Employee* relation, for example, affected screens and reports about employees and departments in both versions.

Views were helpful in reducing the effects of the schema modifications on applications for the following schema changes: renaming an attribute/relation, and splitting/merging relations. These changes had no effects on the application that used views. When we, for example, merged the relation *Employee* with the relation *Job*, the application used the views instead of the new relation. There was no effect on the application. However, a new trigger (less than 50 lines) had to be defined. As the triggers were not part of the application code this is not presented in the table. Number of added/deleted lines was about 40 times greater for splitting/merging a relation than for renaming of an attribute/relation.

In the study reported in (Sjøberg, 1993), renaming of an attribute occurred quite frequently, while renaming, splitting and merging a relations were rare. In the example we used the benefits of the use of views were larger for splitting/merging a relation (that may rarely occur) than for renaming an attribute (that may frequently occur).

It has to be noted that we did not used views to simulate schema changes as, for example, in (Bertino, 1992). When views are used to simulate schema changes, new applications access the view, and the old applications access the original relation. For example, a view can be created that is a projection of the *Employee* relation that does

[34] Splitting a relation and merging two relations affected the same number of places in the version of the application without views. This is because we split and merged the same relations.

not include the attribute *Salary*, but includes all other attributes. In that case, the version of the application with views would use the view instead of the relation and there would be no effects on the application code.

The applications were changed to keep the same functionality as before the schema changes were carried out, but they did not use the new semantic of the database. If a new attribute was added, for example, the applications were not changed to use it. This decreased the number of places affected by schema changes. From the results of this study no conclusions can be drawn about the usefulness of views when applications are adapted to the new semantic of the database.

### 4.2.3 Threats to Validity

Due to limitations of the methods used in this study, one has to be cautious in drawing general conclusions from the presented results. The following possible threats to validity have been identified:

- We studied only one application with limited functionality and limited number of relations and views. The database and applications may not be representative regarding size and complexity.
- The views that were created in the database were restricted to access attributes from only one database relation.

For practical reasons the size and complexity of the database and applications were relatively small (six relations and about 5000 lines of code). It is possible that conclusions would be different for larger systems and other application domains. However, we believe that the functionality of the application is somewhat representative for business applications.

As the JDeveloper wizards were used for code generation, the views could access attributes from only one database relation. Oracle applications are typically developed using JDeveloper and similar environments. Therefore, we believe that this restriction on the use of views did not influence the results.

## 4.3 Chapter Summary

This chapter presented two studies focusing on maintaining application consistency in an industrial context. Relational and object/relational databases are widely used in the industry. Therefore, we collected anecdotal evidence on problems in current practice related to application consistency in the context of relational and object/relational databases, and object-oriented languages. We assumed that the problems that we identified in this context are similar to the problems in the context of object-oriented databases.

The results from the first study suggest that ‘human factors’ as good knowledge of the supporting tools, discipline in their use, discipline in maintaining documentation, and good knowledge of the schema and the applications are essential to maintaining application consistency. Furthermore, ‘good design’ and isolating database related parts of an application also seem to be helpful for maintaining application consistency. When adapting applications to schema changes, developers rely in a

large degree on their own knowledge of the applications or the 'private' documentation that they manually maintain. This indicates that there is a place for improvement of the existing tools that are used to maintain application consistency.

The results from the second study suggest that views can be used to support application consistency by absorbing the effects of schema changes on the applications. Views were helpful in reducing the effects of the schema modifications on applications for the following schema changes: renaming an attribute/relation, and splitting/merging relations. The benefits of views were greater for splitting/merging a relation than for renaming of an attribute/relation. Views were not helpful for other schema changes.

The results of the studies cannot be generalised to the object-oriented databases. Still, we believe that the results can be used to identify lines of research that need further investigations in the area of object/relational and object-oriented databases. In our opinion, the following lines of research are worth investigating:

- considering application consistency during the design of conceptual schemata,
- identifying impacts of schema changes on the application at the source code level, and
- using views to support application consistency during a longer period of time. Are benefits of using views larger than cost related to the additional complexity?

The research presented in this thesis focuses on the second line of research.

# 5 SEMT: Schema Evolution Management Tool

This chapter describes the Schema Evolution Management Tool (SEMT), a technology that supports application consistency in evolving object-oriented systems by identifying and visualising the potential impacts of changes. Section 5.1 identifies the requirements we posed on our technology. Section 5.2 gives the rationale behind the development of different versions of SEMT. Section 5.3 describes a component-based model of object-oriented systems that is used for identifying impacts. Section 5.4 describes the functionality and implementation of SEMT. Section 5.5 describes limitations of the proposed technology. Section 5.6 summarises.

## 5.1 Requirements for the SEMT Technology

In the previous chapters we have demonstrated the need for technologies that support application consistency in evolving object-oriented systems. Particularly, we have demonstrated the need for a technology that identifies impact of changes. We also claimed the need for focusing on the usability of such technology. The main objective of the technology we propose is to improve the process of maintaining application consistency in evolving object-oriented systems from the perspective of schema or application developers. More precisely, the technology should:

- **Identify the potential impacts of changes** to an object-oriented system. Changes to a set of classes describing a particular application domain (a class hierarchy/database schema) affect the rest of the hierarchy/schema and the applications (Section 2.1). The technology should identify these impacts for changes to the class definitions, changes to a class, changes to the structure of a class hierarchy and complex changes.
- **Present this information** in a way that improves efficiency, accuracy and user satisfaction of developers when conducting change tasks. The technology should present the object-oriented system and impacts of its change in a way that is convenient for the developer. It should follow the consistency principle and the clear and concise language principle (Section 2.4). Furthermore, it should be able to present changes to a large amount of objects.

Also other requirements could be posed on technologies that should support application consistency in evolving object-oriented systems. For example, application consistency could be supported by versioning or views. Advantages and disadvantages of this approach were discussed in Section 2.1.

Based on the results from schema evolution related application domains such as reverse engineering and software restructuring (Larkin and Simon, 1987; Morse *et al.*, 2000; Storey *et al.*, 1996; Griswold *et al.*, 1997), we assumed that visualising change impacts may improve efficiency, accuracy and user satisfaction also in schema evolution tasks.

Efficiency, accuracy and user satisfaction are related to, and are dependent on other attributes of acceptability of an evolution technology (Section 2.2.2). For example, the performance of the technology and precision of impact analysis necessarily affect its usability. However, we decided not to specify good performance and high precision of impact analysis among our requirements. It was unrealistic to both fulfil these two requirements and implement the technology at the level appropriate for usability evaluation within the resources available for this research project.

## 5.2 History of SEMT

SEMT evolved through several intermediate versions and their evaluations (Table 5.1). The first version of the technology (Karahasanović and Sjøberg, 1999) was called SEMT — Schema Evolution Management Technology. Although SEMT supports only application consistency and provides no support for consistency between the data in the database and the schema,[35] we decided to keep this quite general and maybe somewhat misleading name.

Java (Gosling *et al.*, 1996) shares many features common to most object-oriented programming languages in use nowadays. It is used for development of Web applications as well as a general-purpose programming language. A large number of application systems have been developed in Java and its popularity is still increasing. Therefore, we put our research in the context of Java.

A Norwegian company developing a CASE tool in the context of C++ and the object-oriented DBMS Pse/Pro (ObjectStore, 2000) provided us with the source code of several versions of this tool. We decided to take this opportunity for evaluation of SEMT in an industrial context, and therefore also developed a C++ version of SEMT.

[35] The main difference between schema evolution technologies and class evolution technologies is that only the former support structural consistency (consistency between the database schema and the data in the database), see Chapter 2.

**Table 5.1.** Versions of SEMT

| Version | Description | Presented in |
|---|---|---|
| 1 | Java; textual interface | (Karahasanović and Sjøberg, 1999) |
| 2 | C++;<br>textual interface;<br>graphical interface (the first version) | (Karahasanović, 2000)<br>Section 5.3.4 |
| 3 | Java;<br>graphical interface (the second version);hierarchical graphs;<br>fine granularity: identifies affected parts of applications at the granularity of packages, classes, interfaces | (Karahasanović and Sjøberg, 2001,<br>Karahasanović, 2001) |
| 4 | Java;<br>graphical interface (the third version);<br>hierarchical graphs;<br>coarse granularity: identifies affected parts of applications at the granularity of fields and methods | (Karahasanović and Sjøberg, 2001,<br>Karahasanović, 2001) |
| 5 | Java;<br>graphical interface (the fourth version);<br>hierarchical graphs;<br>fine and coarse granularity;<br>complex schema changes supported | Chapter 5;<br>Section 7.3 |

## 5.3 Component-Based Model for Identifying Impacts

We have defined an *application system* as a class hierarchy describing an application domain (a class library or a database schema), applications built around them, and, in the case of persistent capable classes, a database (Section 1.2).

This section proposes a component-based model of an object-oriented application system as a basis for identifying change impacts. In this model, classes describing a particular application system (a class hierarchy or a database schema), applications and changes are expressed in terms of *components* and *bindings*. This makes possible to exploit existing impact analysis algorithms. The model is developed in context of Java.

### 5.3.1 Application System

We introduce the following definitions of components and bindings of an application system:

**Definition 1**

A component, *c*, is an element in one of the following sets of an application system:

- *the set of packages*
- *the set of classes*
- *the set of interfaces*
- *the set of abstract classes*
- *the set of method names*
- *the set of signatures of methods*
- *the set of parameters*
- *the set of methods implementations*
- *the set of class fields*
- *the set of fields domains*
- *the set of initial values for fields.*

**Definition 2**

For the set of components *C*, a binding *b*, is a tuple ($c_i$, relationship, $c_j$) where $c_i, c_j \in C$ and $i \neq j$.

Relationships between components are:

- **Part_of**, which expresses encapsulation and aggregation.
- **Of**, which describes a usage of a domain and an initial value.
- **Extended_by**, which expresses inheritance.
- **Used_by**, which expresses usage of one component by another component, by method calls or a direct access to a field.
- **Implemented_by**, which expresses that a class implements an interface.

Table 5.2 gives an overview of the bindings.

**Table 5.2.** Bindings

| Component | Relationship | Component |
|---|---|---|
| class, interface | Part_of | package |
| method implementation, signature, field | Part_of | class |
| signature, field | Part_of | abstract class, interface |
| parameter | Part_of | signature |
| domain | Of | field, parameter |
| initial value | Of | field |
| interface, class | Extended_by | interface, class |
| interface | Implemented_by | class |
| package, class, interface, method, field | Used_by | package, class, interface, method |

The choice of bindings is influenced by the Java language specification (Gosling *et al.*, 1996). The relationships Part_of, Of, Extended_by, Implemented_by and Used_by are expressed in the source code of a schema and applications, and can be found by static source code analysis.[36]

A class hierarchy describing a particular application domain and its associated applications can now be expressed in terms of components and bindings.

**Definition 3**

A class hierarchy of an application system consists of a name of the hierarchy and all the components and bindings specified in the source code of the packages, classes and interfaces describing a particular application domain.

Classes with the property that their instances (objects) can be made persistent are defined as persistence capable classes by ODMG 3.0 (Cattell and Barry, 2000). We extend this definition by the following two definitions:

**Definition 4**

*A persistent capable interface* is an interface implemented by a persistent capable class.

**Definition 5**

*A persistent capable package* is a package with at least one persistent class or interface.

**Definition 6**

A database schema of an application system consists of a schema name and all components and bindings specified in the source code of the persistent capable packages, classes and interfaces defined within that application system.

[36] Dynamic aspects are not considered in this model.

**Definition 7**

An application in an application system consists of an application name and the components and bindings specified in the source code of the packages, classes and interfaces defined within that application.

**Definition 8**

A search space of an application system is the space in which the consequences of a given change of the code are considered. It consists of a schema or a class hierarchy, zero or more applications, and bindings between their components.

**Example 1** Let a schema *S* consist of a class *C1* with a method *M1*, a class *C2* with a method *M2* and a class *C3*, which is subclass of *C2*. M1 calls *M2*. This schema will be described by the following set of components and bindings: {C1, C2, C3, M1, M2, (M1, Part_of, C1), (M2, Part_of, C2), (C2, Extended_by, C3), (M2, Used_by, M1)}.

### 5.3.2 Taxonomy of Changes

The taxonomy of changes in our model is presented in Table 5.3. This taxonomy is derived from the general taxonomy of changes for a standard object model (Section 2.2) but it includes changes that are specific for Java.

It should be noted that classes and interfaces are not completely interchangeable. Firstly, a Java class can extend only one class, while an interface can extend several interfaces. Furthermore, methods of interfaces and abstract classes have no implementation.

Changes to interfaces and methods can be propagated both to the rest of the class hierarchy/schema and to the applications. Therefore, such changes are included in the taxonomy of changes.

Recognising common functionality of several classes would initiate creating a superclass and moving a method to it. Therefore**,** a change for moving a method from one class/interface to another is included in the taxonomy.

**Table 5.3.** Taxonomy of changes

| **Changes to components** |
|---|
| Create/Remove/Rename a package |
| Add/Remove/Rename a class |
| Add/Remove/Rename an interface |
| **Changes to bindings** |
| Add/Remove a class/interface to/from a package |
| Add/Remove an interface to/from the implements clause of a class |
| Move a class/interface from one package to another |
| Add/Remove a class C1 to/from an extends clause of the class C2 |
| Add/Remove an interface I1 to/from the extends clause of the interface I2 |
| **Changes to the contents of a component: (a class/an interface)** |
| Change modifiers/fields/signatures of a class/interface |
| Change methods/constructors/ initializers of a class |
| Merge two classes |
| Split a class |
| **Changes concerning fields** |
| Add/rename/remove of a field |
| Change a domain/initial value of a field |
| **Changes concerning methods** |
| Add/remove a method to/from a class/interface |
| Move a method from one class/interface to another |
| Changes to method modifiers |
| Changes to parameters |
| Changes to implementation code of a method |

**Example 2**

Let a schema include a class *C*, which implements an interface *I*. Assume C is to be changed to not implement I. The binding (I, Implemented_by, C) will then be removed from the schema.

### 5.3.3 Impact Analysis

We apply the transitive closure algorithm (see Section 2.3.1) on the model described above to find the effects of schema/class hierarchy changes on applications in the following way. Components in the search space of an application system are represented as nodes of a transitive closure graph, while bindings are represented as edges. When a change request appears, there are two possibilities:

- this change deletes/modifies a component/binding in the search space
- this change inserts a new component/binding in the search space

In the first case, the transitive closure algorithm starts with a given component; in the second case, it starts with a component semantically enclosing a given component. For example, if a field is to be added to a class, then this class encloses the new field. As bindings present relationships between components, this algorithm finds

components in the search space potentially affected by a given change. We use the δ-wavefront algorithm, which was explained in Section 2.3.1.

**Example 3**

Let *AP* be an application in an application system with the schema S described in Example 1. Let AP consist of a class *C4* with a method *M4*. M4 calls M1 defined in S. In addition to the components and the bindings in S, the search space of the application system will include the following components and bindings: {C4, M4, (M4, Part_of, C4), (M1, Used_by, M4)}.

The components and the bindings in the search space are stored in the relation ***A*** of the δ-wavefront algorithm. Figure 5.1 presents the relation ***A*** as a table and as a graph.

| X | Y | Relationship |
|---|---|---|
| M1 | C1 | Part_of |
| M2 | C2 | Part_of |
| C2 | C3 | Extended_by |
| M4 | C4 | Part_of |
| M2 | M1 | Used_by |
| M1 | M4 | Used_by |

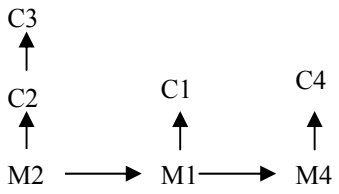



**Fig. 5.1.** Relation A

If M2 is to be deleted from the schema, we are interested in finding all the components affected by that change. Figure 5.2 presents the processing of the relation ***A*** using the δ-wavefront algorithm with M2 as a starting node. The algorithm identifies the components {C1, C2, C3, C4, M1, M4} as the transitive closure of the component M2. The dependencies between the components that are identified by the algorithm are presented by dashed lines in the graph.

| iteration | old_wave | new_wave | closure |
|---|---|---|---|
| 1 | M2 | M1,C2 | M1,C2 |
| 2 | M1,C2 | C1,C3,M4 | M1,C1,C2,C3,M4 |
| 3 | C1,C3,M4 | C4 | M1,C1,C2,C3,M4,C4 |
| 4 | C4 | ∅ | M1,C1,C2,C3,M4,C4 |

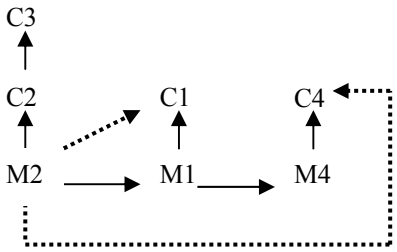



**Fig. 5.2.** The processing of relation *A*

### 5.3.4 Possible Improvements of the Algorithm

The algorithm described above takes a conservative 'worst-case' approach. It calculates all classes and methods that *might be* affected by the proposed change. Some types of change that will not necessarily affect other parts of system will be calculated in the transitive closure relation. The main problem with this approach is its potentially low precision.

Our choice was to focus on appropriate presentation forms rather than to provide higher precision of impact analysis. Our observations, although based on a limited number of applications and changes,[37] show that the precision was acceptable. However, there was one case that required immediate improvements of the algorithm. It is illustrated by the following example (Figure 5.3).

```
public class Class_C1 {
        public void Method_M1() {};
        public void Method_M2() {};
}
public class  Class_C2{
   public  C1  tmpC1;
   …tmpC1.M2() … }
```

**Fig. 5.3.** An example of Java source code

[37] A detailed description of the applications and changes used for evaluation of the technology is given in Chapter 6.

The following bindings are identified by the algorithm:

(Method_M1, *Part_of,* Class_C1)
(Method_M2, *Part_of*, Class_C1)
(Class_C1, *Used_by*, Class_C2)
(Method_M2, *Used_by*, Class_C2)

If Method_M1 is in the set of affected components (the ***closure*** relation), Class_C2 will also be added to the set of affected components, even if Class_C2 actually uses Method_M2 of Class_C1 because of the transitive dependency between Method_M1 and Class_C2. All components that use or extend Class_C2 will also be added to the set of affected components.

The number of components that were added to the set of affected components without actually being affected was relatively high in such cases[38] for the applications and changes we tested. Only 40% of the components that were identified were actually affected by the proposed change.

To improve the precision, we extended the δ-wavefront algorithm described in Section 2.3.1 as follows:

1. Initialise the ***wave*** and ***closure*** relations
2. Calculate the ***closure*** relationship by the *δ-wavefront* algorithm (Figure 2.7)
3 **For each** class in ***closure***
**if** (child of this class used by some component in closure)
**then** remove the class from closure

The algorithm removes the classes from the *closure* relation if its child is already in *closure*. This simple improvement increased the precision of the algorithm. The overall precision was about 80-90% for the applications and changes we used (see Chapter 6).

## 5.4 Supporting Application Consistency by Impact Analysis and Visualisation

Based on the model and the algorithms explained above, we have implemented a tool that supports application consistency by impact analysis and visualisation of the impacts.

### 5.4.1 Functionality

SEMT offers the following functionality:

- identify a search space. This is the space where consequences of a given change should be considered.

[38] Where some other method of that class was used by a particular class.

- display the search space as a directed acyclic graph. Files, classes, fields and methods are displayed as nodes of the graph, while relationships between them are displayed as edges.
- find and display the potential effects of a proposed change. Affected components are identified and marked on the graph.
- cancel a change
- commit a change.

SEMT offers two types of user commands: commands for identifying a search space and one command for finding impacts of changes. The commands for identifying a search space are:

- delete everything from the search space
- display the content of the search space
- put the content of a directory given by the user to the search space
- add the content (classes, methods and fields) of a file given by the user to the search space

The command for finding impacts of a change takes the change given by the user as a parameter. Table 5.4 shows the schema changes currently supported by SEMT.

**Table 5.4.** Schema changes supported by SEMT

| Overall Change | Specific Change |
|---|---|
| Changes to the properties of a class | add/delete/rename a field |
| | add/delete/rename a method |
| Changes to the inheritance graph | add a class to the superclass list of a class |
| | delete a class from the superclass list of a class |
| Changes to classes | add/delete a class |
| Complex schema changes | merge two classes |
| | specialise a class |
| | generalise two classes |

### 5.4.2 SEMT Architecture

The main components of the SEMT architecture are described in Figure 5.4. The parts of the system are isolated from each other and communicate via well-defined interfaces. This isolation allows replacement of each part of the system without changing the other parts. For example, we changed the visualizer without changing the parser.

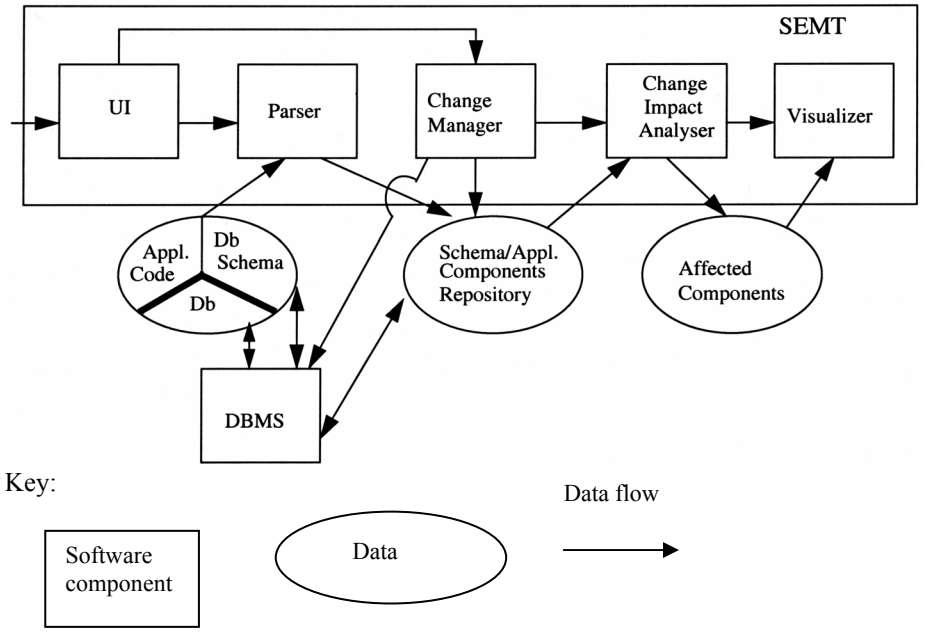



**Fig. 5.4.** SEMT architecture

- The ***User Interface.*** Both a textual and a graphical interface are provided. The textual interface allows specification of SEMT commands from a command line. The graphical interface is windows based and allows specification of SEMT commands through windows and menus.
- The ***Parser*** converts the schema and applications (Java or C++ source files, schema classes defined through ODMG schema bindings) defined to be in the search space, to the components and relationships of the schema and the application components/relationships and store them in the repository.
- The ***Components and Relationships Repository*** is a database where the schema and application components and relationships between them are stored.
- The ***Change Manager***
  - If a change request is received by SEMT, the Change Manager identifies the components being subject of change and initiates the change impact analysis.
  - If a change request is committed (in the next cycle[39]), the Change Manager updates the Schema/Application Components Repository.

- The ***Change Impact Analyser*** analyses the impact of the requested change in the search space. The δ-viewfront algorithm is used here. Other transitive closure algorithms (Qadah *et al.*, 1991) can easily be used within the SEMT architecture.

[39] Schema evolution process was described in Section 2.1.2.

- The ***Visualiser*** displays the content of the search space as a graph and marks nodes of the graph affected by the proposed change. The content of the search space and affected components can also be displayed as lists.
- The ***Affected Components*** is a relation where the impacts of a change are stored.

### 5.4.3 Implementation

The SEMT implementation is based on the architecture explained above.

#### *5.4.3.1 User Interface*

The textual interface of SEMT is implemented in Java. The graphical user interface is integrated with the Visualizer. It is implemented by the Java AWT API and consists of windows and menus for communication.

#### *5.4.3.2 Parser and Preprocessor*

The parser is generated by the Java Compiler Compiler (JavaCC) (Metamata, 2000). JavaCC is a parser generator, which reads a grammar specification and transforms it into a parser written in Java. The parser recognises whether input conforms to the given grammar. Java semantic actions can be associated with the grammar productions. A repository with grammar specifications for different programming languages is also provided (Lee, 1998). We have generated a C++ parser and a Java parser by JavaCC.

This C++ parser generated from the C++ grammar specification (Viswanadha, 1997) does not include preprocessing. When applied JavaCC produces a Java program called CPPParser.java. Therefore, the author implemented a simple preprocessor for C++ files in Java. The preprocessor takes as input C++ files. It uses the Java StringTokenizer class for recognising the #include operator. User defined header files are then textually included. Other C++ macro directives are not processed. For implementation of the Java parser we used the grammar specification for Java1.02.jj (Sankar, 1996), which is based on the Java Language Specification (Gosling *et al.*, 1996) and is the most thoroughly tested Java grammar available (Sankar, 1996).

We need symbol table information to identify dependencies between components of an application system. As the given grammar specifications have no support for symbol tables, we extended their productions with actions identifying and storing declarations of components (files, classes, functions and fields for the C++ parser; packages, classes, interfaces, fields and methods for the Java parser) and relationships between them (Part_of, Extended_by and Used_by).

#### *5.4.3.3 Components and Relationships Repository*

The *Components and Relationships Repository* is a Sybase database consisting of relations needed for the transitive closure algorithm (Section 2.3.1). The base relation of the algorithm (*A* in Figure 5.1) includes information about existence of the edge between two nodes. We also include information about component and relationship type. The Java Database Connectivity (JDBC) API is used for accessing the Sybase database.

*5.4.3.4 Change Impact Analyser*

The δ-wavefront algorithm is implemented in Java. Relational operations are implemented using SQL via JDBC. As the δ-wavefront algorithm is given in terms of relational algebra, it was easiest to implement it in SQL. This implementation gives good performances on relatively large application systems. This is reported in Section 6.1.

*5.4.3.5 Graph Visualisation System daVinci*

We used the interactive graph visualisation system *daVinci* to implement the graphical user interface and the visualizer. *daVinci* is developed at the Computer Science Department at University of Bremen (Werner, 1998). The basic philosophy of *daVinci* is separation of two levels of responsibility:

- The Visualisation layer (*daVinci*) is responsible for the graphical representation.
- The Application layer (some application program connected to the *daVinci* API) decides what to do if an operation should be applied that modifies the structure of a graph.

Figure 5.5 illustrates the data flow between an application program and *daVinci.*

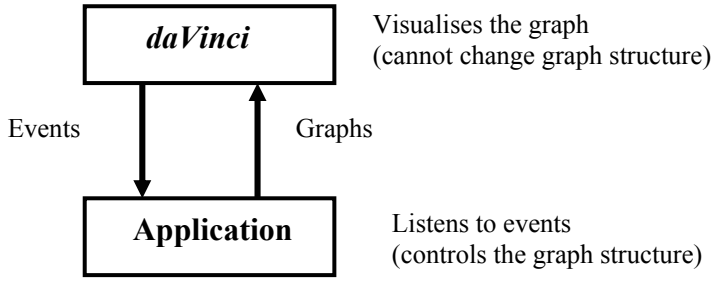



**Fig. 5.5.** Connecting *daVinci* with a application

Communication between the visualisation and the application layer is realised with two UNIX pipes: one to send *daVinci* API commands (e.g. graphs) and one to receive the answers (e.g. events). With API commands, the application is able to send graphs for visualisation and attach its own dialog elements (i.e., menus and icons) to the *daVinci* user interface. The application, then, waits answers from *daVinci*. These are events triggered by user interactions in *daVinci's* graph visualisation windows, such as a menu selection or a node selection. Based on this event, the application conducts the selected operation on its local graph structure. It then sends the new graph (or its increment) to *daVinci* to update the representation of the graph.

Several commands are provided for manipulation of a loaded graph. One can minimise edge crossing, minimise edge bending, change the orientation of the graph, and zoom into different parts of the graph.

Graphs are loaded in *daVinci* using a format called *term representation*. The term representation is a plain text ASCII format. A term is a structure of type `parent[child1, child2, …]`, where a parent and a child are nodes of a graph. One can send a complete graph in term representation format or update a graph by adding and deleting nodes and edges. Further, it is possible to change the attributes of nodes and edges, like colour and font. One can also change the type of nodes and edges. Figure 5.6 shows *daVinci* attributes for a graph node that should be drawn as a box with light blue background and the text "SEMT" written using the default font (*a* denotes an attribute in the term representation):

```
[a("OBJECT,"SEMT"),a("COLOR","lightblue")]
```

**Fig. 5.6.** An example of *daVinci* attributes

*5.4.3.6 Visualizer*

The Visualizer has three layers:

- The **Visualisation** layer (*daVinci*) is responsible for graphical representation of the search space, marking nodes of the graph affected by the proposed schema change, and receiving user interaction in *daVinci's* graph visualisation windows, such as a menu selection and a node selection.
- The **Listener** layer reads in *daVinci* answers and parses them. It creates the events from these answers and fires them to the repository layer. It also forwards graphs and dialog elements from the repository layer to the visualisation layer. The listener layer is implemented in Java.
- The **Repository** layer is based on the events received from the listener layer. The repository layer decides which operations need to be applied on the local graph structure that is stored in the repository. After modifying the graph, the repository layer generates appropriate *daVinci* terms and sends them to the listener layer. For example, when a command for displaying a search space is selected from the menu, the repository layer reads the components and bindings of the search space stored in the repository, generates appropriate *daVinci* terms and sends them to the listener layer. The repository layer is implemented in Java. JDBC is used for accessing the repository.

*daVinci* and the application can be connected in two ways. Either *daVinci* connects the application (*daVinci* is parent process) or the application connects *daVinci*. While the latter way gives more flexibility, we have chosen the former one because it was easier to implement. *daVinci* starts the listener layer, which has to wait for an OK answer before sending API commands to the standard output. Answers from *daVinci*

are available for the listener layer by reading from the standard input. API commands for sending and updating graphs are used.

### 5.4.4 User Interface

The user of SEMT has three tasks to support:

- enable commands,
- display components/relationships in the search space, and
- mark components affected by the proposed change.

#### *5.4.4.1 Textual Interface*

The textual interface has been evaluated with the C++ version of SEMT (Karahasanović, 2000) and examples presented in this section are, therefore, given in C++. Table 5.5 shows the commands of the textual interface of SEMT.

**Table 5.5.** SEMT textual interface

| Command | Function |
|---|---|
| DeleteSearchSpace | Deletes the content of the search space. |
| DisplaySearchSpace | Displays the content of the search space. |
| PutInSearchSpace <file_name/directory_name> | Puts in the search space a file or a directory |
| FindImpacts –d/a class/function/field_name | Finds impacts of deleting (d), adding (a) a class, a function or a field. |
| FindImpacts –di/ai class_name1,class_name2 | Finds impacts of deleting (di), adding (ai) a class from/to a subclass list of another class. |

When using the textual interface, the user can delete the content of the search space by running the script ***DeleteSearchSpace*** from the command line. The content of the search space is displayed as a component/relationship list by running the script ***DisplaySearchSpace***. The name and type of components (file, class, function and field) are displayed together with type of the relationship between them (part_of, extended_by and used_by).

The following example is used in this section. A C++ file called sptestfile.cpp consists of the classes Spurious and eSpurious. They include several functions and class variables. The file is given in Figure 5.7.

```
#include <stream.h>
class Spurious {
       private: int i; float f;
       public: Spurious();
       Spurious(int, float);
       int ival();
       float fval();};
       Spurious::Spurious(): i(0), f(0.0) {}
       Spurious::Spurious(int iI, float fI) : i(iI), f(fI) {}
       Int Spurious::ival() {return i;}
       float Spurious::fval() {return f;}
class eSpurious : public  Spurious { public: eSpurious(); int
zval(); };
eSpurious::eSpurious() {}
int eSpurious::zval() {return 1;}
void main()
{
       Spurious d1;
       Spurious d2 (2, 6.926);
       cout << "d1 = (" << d1.ival() << ",";
       cout << d1.fval() << ")\n";
       cout << "d2 = (" << d2.ival() << ",";
       cout << d2.fval() << ")\n";
}
```

**Fig. 5.7.** The source code of sptest.cpp

Figure 5.8 shows the components and the relationships of the sptestfile.cpp file. The type of the first components is given in the first column. The names of the components are given in the second and the firth column. The relationship between them is given in the third column.

>***DisplaySearchSpace***

| | | | |
|---|---|---|---|
| Class | Spurious | Part_of | sptest.cpp |
| Field | I | Part_of | Spurious |
| Field | f | Part_of | Spurious |
| Constructor | Spurious | Part_of | Spurious |
| Function | ival | Part_of | Spurious |
| Function | fval | Part_of | Spurious |
| Class | eSpurious | Part_of | sptest.cpp |
| Class | Spurious | Extended_by | eSpurious |
| Constructor | eSpurious | Part_of | eSpurious |
| Function | zval | Part_of | eSpurious |
| Function | main | Part_of | sptest.cpp |
| Function | ival | Used_by | main |
| Function | fval | Used_by | main |

**Fig. 5.8.** Displaying search space

When the users want to define the search space, they run the script ***PutInSearchSpace*** from the command line with the file or directory name as the

parameter (Figure 5.9). Commands and parameters given by the users are typed in bold. SEMT reports whether a program parsed successfully.

```
> PutInSearchSpace sptest.cpp
C++ Parser Version 0.1: Reading from file sptest.cpp
C++ Parser Version 0.1: Program parsed successfully.
```

**Fig. 5.9.** Defining the search space

Impacts of a particular schema change are found by running the script ***FindImpacts***. It takes the kind of schema change (-d for delete, -a for add), the type (class, function or field) and the name of the component to be changed as parameters from the command line. If the user wants to change the inheritance graph, the kind of change (-di for delete inheritance, -ai for add inheritance) is followed by the name of the class and the name of the superclass. The ***FindImpacts*** script displays a list of the components (their type and name) affected by the proposed change. Figure 5.10 presents the components potentially affected by deleting the function *fval*.

```
> FindImpacts -d function fval
Affected components:
Type     Name
----------------------------
file     sptest.cpp
class    Spurious
class    eSpurious
function main
```

**Fig. 5.10.** Components affected by the change delete function *fval*

#### *5.4.4.2 Graphical Interface*

When using the graphical interface, SEMT commands are given by selecting menu items, and clicking buttons and nodes on the graph. *daVinci* allows adding menu items in the ***Edit*** menu; we added the following:

- Redraw
- Update Search Space
- Display Search Space
- Expand Node
- Find Impacts

When the ***Redraw*** menu item is selected, the presented graph is redrawn. The effects of all other commands than ***Display Search Space*** become invisible. These are, for example, change of the shapes of the affected nodes of the graph. When the ***Update Search Space*** menu item is selected, the ***Update Search Space*** screen is created. This screen has three buttons: ***Put Directory in Search Space***, ***Put File in Search Space*** and ***Delete Search Space***. By clicking on the selected button, the corresponding

function is performed. Browsing of the directories and files is currently not supported. Therefore, the directory or file name has to be entered from the text field. When the ***Display Search Space*** menu item is selected, the content of the search space is displayed. By selecting the ***Expand Node*** menu, one can display the methods and fields of a class. When the ***Find Impacts*** menu item is selected, the ***Find Impacts*** menu list appears. It consists of the menu items corresponding to the schema changes (***Delete***, ***Add***, ***Rename***, ***Merge, Generalise, Specialise***). How SEMT presents the content of the search space and the impacts of a proposed schema change was described in Section 5.2.5.

The layout of the graph can be improved by selecting the ***Improve All*****,** the ***Improve Nodes*** and the ***Improve Edges*** menu item from the ***Layout*** menu. By selecting the ***Layout Dimensions*** menu item from the ***Options*** menu one can change font size.

### 5.4.5 Visual Syntax of SEMT

Due to the potentially large number of the components affected by a proposed schema change, visualising components of a schema and applications, relationships among them and impacts of the change is a challenging task. This section describes in an informal manner the visual language that we propose.

The visual syntax of SEMT is based on the component-based model (Section 5.2) and on the guidelines and principles for user interface design (Section 2.4). We anticipate that SEMT users are schema and application developers with good knowledge of the application domain and the tasks they want to perform, but unfamiliar with the visual syntax of SEMT, that is, *knowledgeable intermittent users* (Shneiderman, 1998). Therefore, we try to make the visual syntax of SEMT as simple and intuitive as possible. To achieve *efficient information assimilation* (Smith and Mosier, 1986), we propose usage of graphs with limited number of schematic figure types (rectangles, ellipses, lines, and arrows) and colours. To achieve *consistency of data display* (Smith and Mosier, 1986), we propose use of the same colours and shapes for the same type of components and relationships.

Using Java determines the set of components, relationships between them, taxonomy of changes and inheritance model (single-inheritance between classes). However, the visual language and the tool can, with some adaptations, be used with other object-oriented languages. For example, an earlier version of SEMT (Karahasanović, 2000) was implemented in the context of C++ and the object-oriented DBMS Pse/Pro. The visual syntax for the C++ version of SEMT is given in Section 5.4.5.4.

#### *5.4.5.1 Presenting a Search Space*

SEMT displays a search space as a graph. Nodes of this graph are packages, classes, interfaces, methods and fields, whereas the relationships between them are represented as edges. All component types are displayed as rectangles with the name of the component written inside. The size of the rectangles carries no additional information. Colours are used for denoting different component types. These are given in Table 5.6.

**Table 5.6.** Components and colours

| Component | Colour |
|---|---|
| Package | light pink |
| Class | light blue |
| Interface | light green |
| Field | light yellow |
| Method | white |

Coloured directed arrows denote different relationships between the components. Arrows indicate direction in space, time, and causality (Tversky *et al.*, 2000). As pointed out in (Tversky *et al.*, 2000), Horn counted 250 meanings of arrows in his survey of diagrams (Horn, 1998). In the standard Java book (Arnold and Gosling, 1996), for example, the concept of specialisation is depicted by an arrow pointing to the new extended class (subclass). The Unified Modeling Language (Booch *et al.*, 1997), on the contrary, depicts the same concept by an arrow pointing to the class being extended (superclass). It is, therefore, difficult to know in advance what is a 'standard' or 'intuitive' orientation of arrows for the potential users of SEMT. Thus, we simply chose one orientation that we considered intuitive. Table 5.7 gives an overview of the relationships.

**Table 5.7.** Relationships and colours

| Abstraction | Relationship | Presented by | Pointing to |
|---|---|---|---|
| Aggregation | Part_of | blue arrow | part |
| | Implemented_by | green arrow | interface to be implemented |
| Inheritance | Extended_by | green arrow | superclass |
| Usage | Used_by | red arrow | component that invokes another component |

In Figure 5.11, four excerpts of Java source code are given together with the corresponding graphs. In the first example, Package_P consists of the classes Class_C1 and Class_C2. Class_C2 consists of the field Field_F1 and the methods Method_M1 and Method_M2. In the second example, SubClass extends SuperClass. In the third example, Class_C implements Interface_I. In the fourth example, Method_M1 of the class Class_C1 invokes Method_M2 of the class Class_C2. This means that Class_C2 is used by Class_C1.

**Aggregation**

Class_C1.java

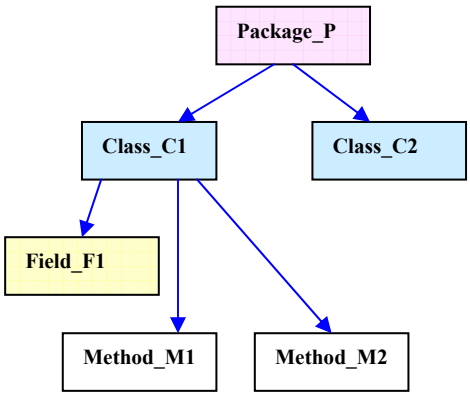
---

```
package Package_P;
public class Class_C1 {
  public int Field_F1;
  public void Method_M1(int p1) {};
  public void Method_M2(int p2) {};
}
```

Class_C2.java

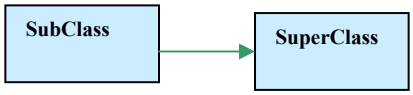
---

```
package Package_P;
public class Class_C2 {}
```



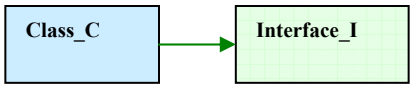

**Specialisation**

```
public class SubClass
            extends SuperClass {}
```




**Interfaces**

```
public class Class_C implements
                    Interface_I {}
```




**Usage**

Class_C1.java

---

```
public class Class_C1 {
public void Method_M1(int p1) {
…Method_M2(a)…;
}
}
```

Class_C2.java

---

```
public class Class_C2 {
public void Method_M2(int p2) {}
}
```


**Fig. 5.11.** Examples of visual syntax of SEMT

The techniques explained above will produce very large and incomprehensible graphs when applied to real-life database application systems. Therefore, we decided to first present only components and dependencies at the coarse granularity (packages, classes and interfaces). If the user wants to explore the graph more closely, he may expand classes and interfaces. The user may choose between presentation of fields or/and methods. Corresponding components and dependencies at the granularity of fields and methods are then presented. Our solution is a variation of hierarchical graphs. The choice of shapes and support for hierarchical graphs are limited by the implementation technology we had access to (Werner, 1998).

Figure 5.12 presents a theatre database application system. It consists of the classes *Theatre*, *Customer* and *Occasionally*. This application is a support system for making reservations for theatre performances. It can store, delete, update and report information about the theatre performances and the customers in the system. An occasional customer only *occasionally* visits the theatre. The frequency of his visits is recorded. The class *Theatre* is used in the classes *Customer* and *Occasionally*. The user decided to expand only the fields of the class *Customer*. It consists of the fields *name* and *telephone.*

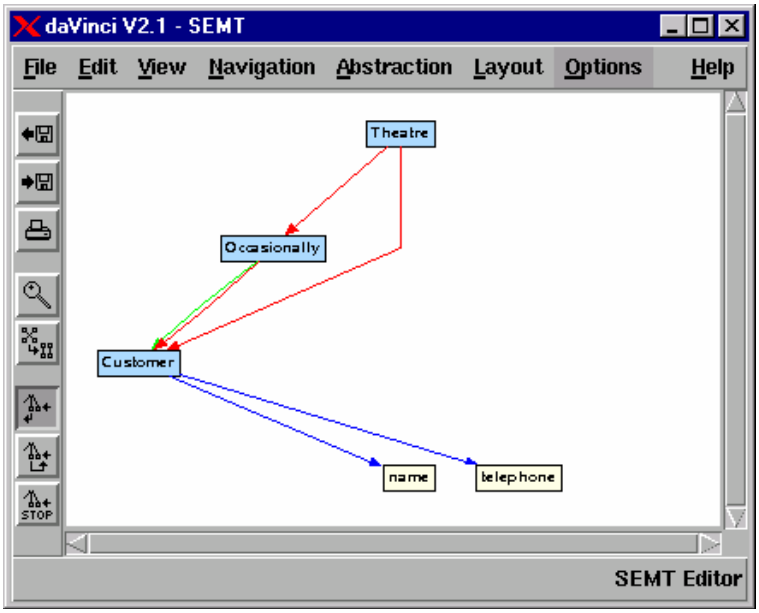



**Fig. 5.12.** Theatre database application system

*5.4.5.2 Presenting Impacts of Change*

To present the effects of a proposed schema change, we use a combination of changing the shapes and lines. The component we intend to change is surrounded by double lines. Components potentially affected by the proposed schema change are denoted by changing their shape to ellipses. Fields and methods are presented in the

graph if the corresponding class or interface is expanded or if they are affected by the change. This solution appears to be both visible and semantic preserving. An alternative is to mark the nodes with the *affected* label, as in (Deruelle *et al.*, 1999), but this would increase the size of the graph and is thus inappropriate for real-life database application systems. Another alternative is to denote the affected components by changing the colour to red, for example. It would be easy to recognise these components on the graph, but information about type of the component would be missing. It would not be clear if the node is a class or a method, for example.

The example in Figure 5.13 shows the effect of deleting the field *name*. The field *name* is now surrounded by double lines. The classes *Customer*, *Occasionally* and *Theatre* change their shape from rectangles to ellipses. The methods *getName*, *orderTicket* and the constructor *Occasionally* appear in the graph as ellipses. This means that the method *getName* of the class *Costumer,* the constructor *Occasionally* and the method *orderTicket* of the class *Theatre* can potentially be affected by deleting the field *name*.

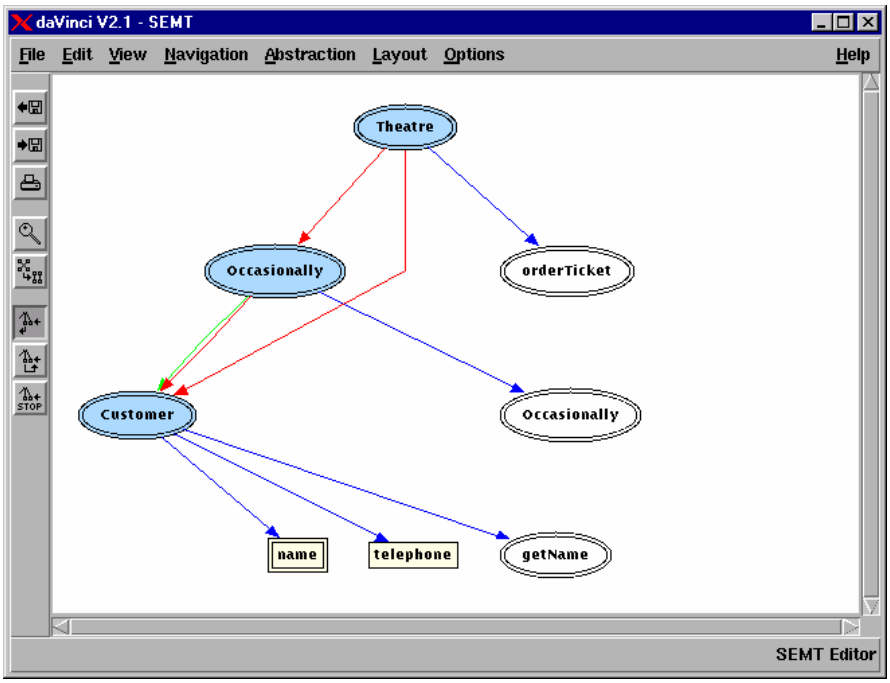



**Fig. 5.13.** Components affected by deleting the field *name*

### *5.4.5.3 Fine and Coarse Granularity*

Two versions of SEMT are available. One version identifies affected parts of applications at the granularity of at the granularity of fields and methods (fine granularity), whereas the other version identifies affected parts at the level of packages, classes, and interfaces (coarse granularity). In the example given above

(Figure 5.13) we described the version of SEMT that identifies affected parts at the granularity of fields and methods. When the coarse granularity version of SEMT is used, the affected packages, classes and interfaces change their shape to ellipses, but fields and methods are not displayed. Figure 5.14 shows the same example that is now presented using the coarse granularity version of SEMT.

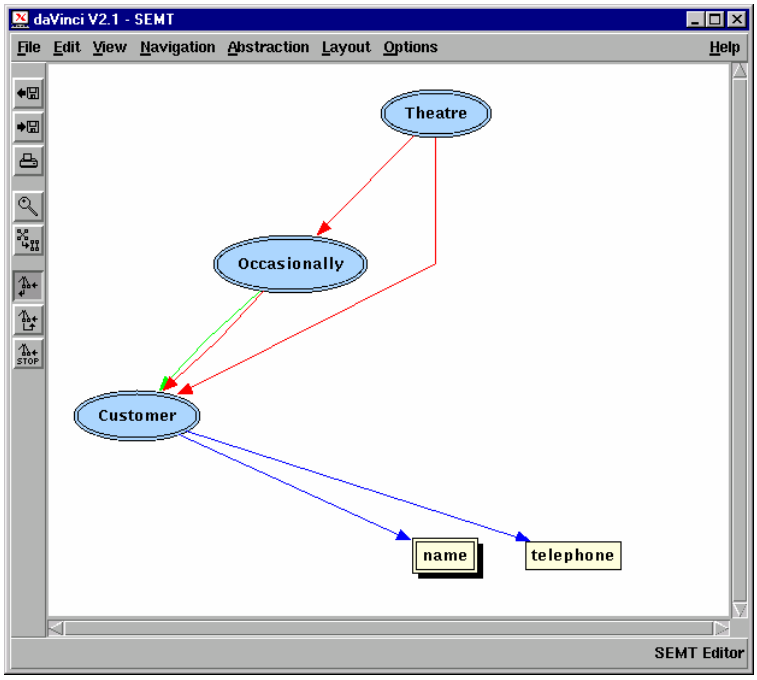



**Fig. 5.14.** Classes affected by deleting the field *name*

*5.4.5.4 Visual Syntax for the C++ Version of SEMT*

Nodes of the graph are components in the search space while the relationships between them are edges. All types of the components are displayed as rectangles with the name of the component written inside. Different colours are used for denoting different component types. These are presented in Table 5.8.

**Table 5.8.** Components and colours

| Component | Presented by |
|---|---|
| File | light blue rectangle |
| Class | light blue rectangle |
| Method | white rectangle |
| Field | white rectangle |

Colours are also used for denoting different types of relationships (presented as edges) between the components. These are presented in Table 5.9.

**Table 5.9.** Relationships and colours

| Abstraction | Relationship | Presented by | Pointing to |
|---|---|---|---|
| Aggregation | Part_of | blue arrow | aggregated component |
| Inheritance | Extended_by | green arrow | subclass |
| Usage | Used_by | red arrow | component that invoke another component |

The components affected by the specified change are denoted by changing the colour to a darker nuance.

The example in Figure 5.15 shows the effects of deleting the function *fval* from the class Spurious (Figure 5.7). When function *fval* is proposed to be deleted the classes and files potentially affected by this change are coloured darker on the graph presenting content of the search space.

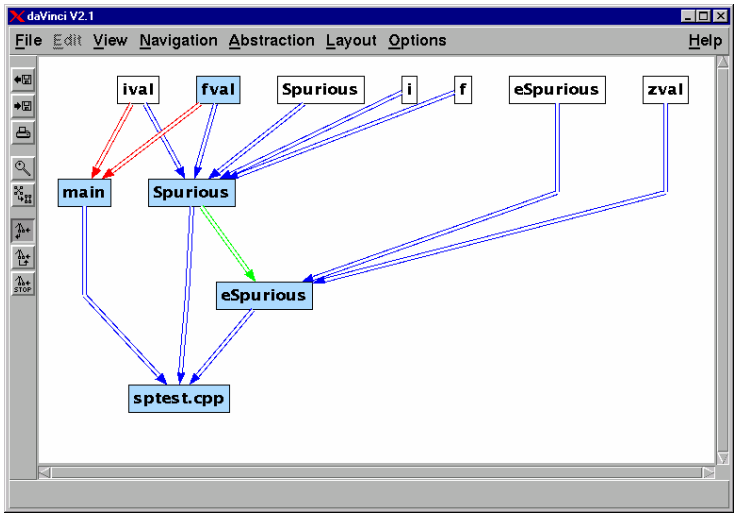



**Fig. 5.15.** Components affected by deleting function *fval*

## 5.5 Present Limitations of the Technology

In this section we describe present limitations of the proposed technology and suggest some ways of dealing with them in further research.

### 5.5.1 Precision of the Impact Analysis

The algorithm that we implemented distinguishes between the changes that delete/modify a component/binding in the search space and the changes that insert a new component/binding in the search space. The algorithm removes the class name from the set of the affected components if the child of the class is used by a component that is already in the set of the affected components. The problem with this approach is its potentially low precision. Some types of change that will not necessarily affect other parts of system are calculated in the transitive closure relation.

The algorithm can be refined on the basis of the detailed information about the impacts of each particular change. Chapter 13 of the book on the Java Language Specification (Gosling *et al.*, 1996) describes impacts of *binary incompatible* changes, which are changes that prevent successful compilation and linking.

In (Dmitriev, 2001) a list of *source incompatible changes* that are detected by the PJama evolution technology is given together with the description of the effects of these changes. Source incompatible changes are changes that prevent successful compilation or correct program execution. These descriptions can be extended to include additive changes, *e.g.*, add a class, and complex changes like merging two classes. Tables 5.10 – 5.12 describe changes and their consequences. Table 5.10 describes the changes to a component, Table 5.11 describes the changes to bindings and Table 5.12 describes the complex changes. This list is given to illustrate effects of changes and activities that have to be carried out. The list is, however, not exhaustive. The changes of modifiers, changes of fields and changes of methods should be added.

**Table 5.10.** Impacts of changes to a component

| **Changes to a component** | **Potentially affected parts of the application** |
|---|---|
| Add a class/an interface | At least one place in the application should use the class; the packages that use the package that contains the class may be affected |
| Delete a class[40] | All places where this class is used are affected; they should be deleted |
| Rename a class | All places where this class is used are affected; the old name should be replaced by the new one |

[40] An interface is intended to be used by a class that extends it. Therefore, the changes delete and rename an interface are described in Table 5.10.

**Table 5.11.** Impacts of changes to bindings

| **Changes to bindings** | **Potentially affected parts of the application** |
|---|---|
| Add an interface to the implement clause of a class | At least one place in the class should implement the methods of the interface; at least one place (where this class or its subclass is used ) should use this method |
| Remove an interface form the implement clause of a class | All places in the class where the methods of the interface are implemented; all the places in the classes that uses or extent this class where the methods of the interface are referenced |
| Rename an interface that is in the implement clause of a class | All places where the interface is used; the old name should be replaced by the new one |
| Add a class/an interface C1 to the extends clause of the class/interface C2 | All places where C2 is used; at least one of these places should use one or more public methods and fields of C1 |
| Remove a class/an interface C1 from the extends clause of the class/interface C2 | All places where C1 and C2 are used |
| Rename a class/an interface C1 that is in the extends clause of the class/interface C2 | All places where C1 is used; the old name should be replaced by the new one |

**Table 5.12.** Impacts of complex changes

| **Complex changes** | **Potentially affected parts of the application** |
|---|---|
| Merging two classes, C1 and C2 | One of the classes (C1) is deleted, the other one (C2) is renamed; methods and fields from C1 are moved to C2.<br>All places where the public methods of these classes are used are affected |
| Generalisation of two classes, C1 and C2 | A new class (G) is created;<br>Methods/fields with the common functionality are removed from both classes and moved to the general class;<br>G is in the *extends* clause of both classes.<br>If C1 were subclass of another class, for example C0, G will also be subclass of C0.<br>All places using the methods that are moved to the general class are affected.<br>All places connected by these classes by inheritance are affected |

A mechanism suggested in (Li and Offutt, 1996) can, for example, be used to connect this description with the algorithm. Based on the description, each change is assigned an *attribute* that is a combination of *attribute bytes* like *impact_on_children*, *impact_on_current*, and *impact_all*. The algorithm then uses this information to decide how to propagate impacts. Implementation of such an improved impact analysis algorithm, and measuring the effects of the improvements on a large real-life application system, might be an interesting research topic.

### 5.5.2 Impacts on the Data in the Database

We did not consider effects of schema changes on the data in the database in the model that we have proposed. One possible way of doing this could be to introduce the association *Has_an_instance*, which expresses that there exists a persistent instance of a class in the database. Bindings with *Has_an_instance* relationships could be detected by a DBMS and stored in the repository. These bindings would also be included in the impact analysis. The Change Manager could be extended to forward a change request to the DMBS that would then conduct the change.

An issue that is worth considering is extending the ODMG standard (Cattell and Barry, 2000) with schema change commands. A standardisation structure offered by such an extension would be a step towards a portable schema evolution technology. The Change Manager of SEMT could generate such standardised schema change commands and be used with different DBMSs.

### 5.5.3 Integration with a Programming Environment

Integration of different tools that are used in maintenance of application systems is an advantage from the point of view of developers (Section 4.1). SEMT is currently not integrated with any development environment.

An alternative to the visual syntax of SEMT was to extend UML (Booch *et al.*, 1997). This would have had the advantage of easier integration with a programming environment. The graph notation would also have been familiar to schema and application developers. However, we had no access to the source code of a UML tool, which would have been necessary to tailor the tool to our studies.

Based on the results and experiences with the visual syntax of SEMT an extension of UML that should include presenting change impacts, could be proposed and evaluated. We believe that it is a fruitful research topic to be explored in the future.

## 5.6 Chapter Summary

In this chapter, we presented a technology that supports application consistency in evolving object-oriented systems. We identified requirements that we wanted this technology to satisfy. These are identifying the potential impacts of changes and presenting this information with focus on the usability of the proposed solutions. We presented briefly how we ended up with the present technology through several versions and evaluations of SEMT. We described then the component-based model of object-oriented systems that is used as basis for identifying impacts. In this model,

classes describing a particular application system (a class hierarchy or a database schema), applications and changes are expressed in terms of components and bindings. The components of this model are packages, classes, interfaces, fields and methods; the relationships between them are encapsulation, inheritance, aggregation and usage. An improved version of the transitive closure algorithm, δ-wavefront algorithm, is then applied on the model to identify change impacts. We described the functionality and implementation of a tool based on this model. The tool displays components of an application system as a graph. Components potentially affected by a change are indicated by changing the shape of the boxes representing those components.

Research on visualisation in the area of databases focuses mainly on visual query languages (Catarci, 2000; Catarci *et al.*, 1997; Benzi *et al.*, 1999; Chen *et al.*, 2000; Olston *et al.*, 1998). The visual language that we propose for visualising impacts of changes on applications in object-oriented systems, is based on simple graphical primitives and generally accepted lore from the area of visualisation. It is similar to *See Data* (Antis *et al.*, 1996), which is a system for visualising database schema and relationships to application code in the context of the relation databases. However, the visual language of SEMT presents dependencies in an object-oriented context, namely dependencies between classes, methods and fields caused by inheritance and aggregation.

We also described some limitation of the technology and proposed how to overcome them.

# 6 Evaluation of SEMT

This chapter describes the studies that we have conducted to evaluate the proposed technology. The primary goal of the studies was to validate SEMT, the technology that supports application consistency in evolving object-oriented systems by impact analysis and visualisation. The main focus of validation was on usability of the technology. SEMT was described in Chapter 5.

The studies also aimed to gain experience regarding methods and tools used in the evaluation of SEMT. More precisely, the goals were:

- to validate the logging tool that is described in Section 3.4.2,
- to gain experience regarding controlled one-subject explorative studies, and
- to validate the framework for evaluation of technologies supporting application consistency that we proposed in Section 2.2.2.

The empirical studies consisted of two experiments, a case study and evaluation of SEMT within the proposed framework. The data collection and analysis were both qualitative and quantitative. The remainder of this chapter is organised as follows. Section 6.1 describes the results of a preliminary evaluation of SEMT. Section 6.2 describes the results of a controlled student experiment evaluating the visual language of SEMT with respect to productivity and user satisfaction. Section 6.3 describes the results of a controlled one-subject study evaluating further the visual language of SEMT and reports our experience with this kind of studies. In Section 6.4 SEMT was evaluated within the framework for evaluation of technologies supporting application consistency. Section 6.5 presents our experience with the logging tool. Section 6.6 discusses some ethical issues raised by our studies. Section 6.7 summarises.

## 6.1 Our Evaluation of SEMT

This section presents our experience with the first version of SEMT. These results are partly published in (Karahasanović, 2000).

Our main concern regarding the technology we proposed was that performance could be bad due to the large number of components in the repository. Furthermore, presenting impacts of changes at the level of fields and methods could lead to large and incomprehensible graphs.

Thus, the goals of the preliminary judgement of SEMT were to evaluate performance and usability of presentation form of SEMT. The method we used was assertion conducted in an industrial context; the researcher itself evaluated the technology by using it on the source code from the industry.[41]

[41] It has been argued that the assertion method can be classified as a case study if it is conducted in the context of a large industrial project (Zelkowitz and Wallace, 1998).

A Norwegian company developing a 4GL CASE tool provided us with the source code of the tool. The tool is integrated with Rational Rose. Based on a UML model made in Rational Rose, one can generate application prototypes. The tool contains both an application generator and a database generator. The tool is implemented by using C++ and the object-oriented DBMS Pse/Pro (ObjectStore, 2000).

The company provided us with two versions of three directories containing the source code of the tool. The directories with the source code before the change have postfix *before*. The directories with the source code after the change have postfix *after*. There was about 600 files and about 135 000 lines of code stored on three directories for each version (Table 6.1).

**Table 6.1.** Files and lines of code (LOC) of the CASE tool

| **Directory** | **Header Files** | **LOC** | **CPP Files** | **LOC** |
|---|---|---|---|---|
| ai.before | 141 | 12878 | 132 | 41673 |
| dmddialog.before | 74 | 5620 | 135 | 38571 |
| ds.before | 30 | 2905 | 113 | 33232 |
| **Total** | 245 | 21403 | 380 | 113476 |
| ai.after | 140 | 12783 | 164 | 41676 |
| dmddialog.after | 74 | 5620 | 111 | 38635 |
| ds.after | 30 | 2905 | 114 | 33265 |
| **Total** | 244 | 21308 | 389 | 113576 |

Firstly, the author measured the time needed to generate the *Schema/Application Components Repository* and the time needed to identify impacts of changes. It took about 20 minutes to build the repository for the source code we had. There were about 5000 records in the repository. We analysed *diff* files to identify changes of the two versions of header files. We then measured the time SEMT spent to identify potential impacts of some of these changes, when the textual interface was used. It took between 5 and 30 seconds.

Performance depends on the computer, number of components, network traffic, optimisation of the database, etc. As building of the repository is not normally done so often, we considered performance to be acceptable. The performance could be improved by use of indices, for example.

The graphical interface was also evaluated by the author. All classes, fields and functions were presented in the same graph. We found the graph completely incomprehensible, and decided to implement hierarchical graphs in the next version of SEMT.

Figure 6.1 presents classes, functions and fields of file *ExpressionAI.cpp*, and effects of deleting function *storeValue*.

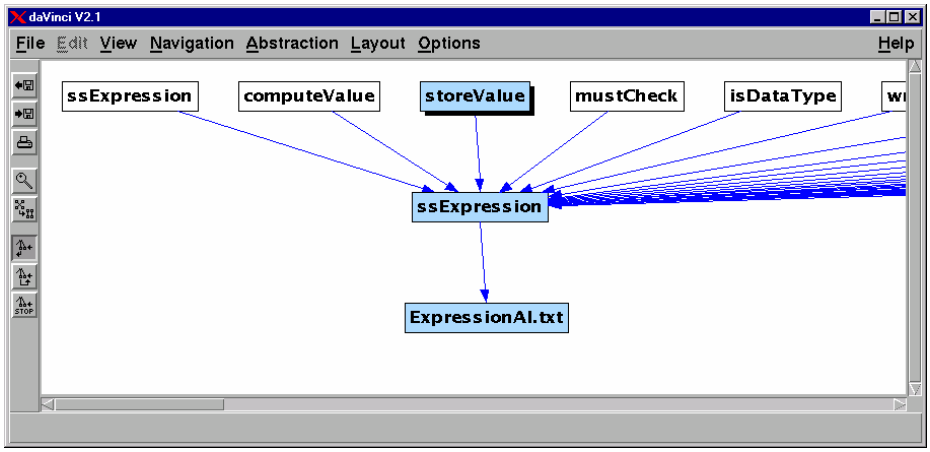



**Fig. 6.1.** Components affected by deleting function *storeValue*

The method we used for the evaluation is potentially biased since the developer was both experimenter and subject of the study. Nevertheless, as the evaluation was done in the context of a relatively large industrial project, we consider the method as appropriate for a preliminary evaluation of SEMT.

## 6.2 Visualisation Experiment

In Chapter 2 we claimed the need for a usability evaluation of technologies supporting application consistency. This section describes the design and results of an experiment on the usability of SEMT. The goal was to evaluate the granularity levels of SEMT regarding productivity and user satisfaction of schema and application developers. The results of this study are also published in (Karahasanović and Sjøberg, 2001).

The remainder of this section is organised as follows. Section 6.2.1 describes the design of the study. Section 6.2.2 describes the results. Section 6.2.3 discusses threats to validity. Section 6.2.4 summarises.

### 6.2.1 Design of the Study

One of the goals of our research is to evaluate how visualising the impacts can help to maintain application consistency. Several aspects of visualising can be explored, e.g., effects of animation, colours, arrows orientation. In Section 5.3.5, two versions of SEMT are described. One version identifies affected parts of applications at the granularity of packages, classes, and interfaces, whereas the other version identifies affected parts at the finer granularity of fields and methods. This study focuses on these two granularity levels. The Goal Question Metric template (Basili *et al.*, 1999) is used to define the purpose of the study.

***Null hypothesis:*** *There is no significant difference in effectiveness and user satisfaction between the group using the version of SEMT that identifies change impacts at a fine granularity (fields and methods) and the group using the version of SEMT that identifies change impacts at a coarse granularity (packages, classes, and interfaces).*

The independent variables of the experiment were the granularity level, complexity of the application system, complexity of the change task, and level of user expertise. The dependent variables were time needed to conduct impact analysis, correctness of the answers, and subjective user satisfaction.

#### *6.2.1.1 Subjects*

The subjects of this experiment were 13 students from the Computer Science Department, University of Oslo, who conducted schema changes on a library application. All of them were unfamiliar with the functionality of SEMT. They were provided short training (15 minutes). They had access to the SEMT documentation during the experiment. During two advanced courses in databases and software engineering the students were motivated to participate in the experiment. The students were paid for the participation.

#### *6.2.1.2 Experiment Organisation*

A pilot experiment with one student was undertaken to ensure that change tasks, training and documentation were appropriate. In the main experiment, the 13 students were divided into two groups with respectively 7 students and 6 students. The assignments to the respective groups were done at random. Both groups were asked to conduct two database schema changes (add a field and delete a field) on the library application system. The change tasks are given in Appendix D.1.

The first group used the SEMT version operating at the fine granularity, while the second group used the SEMT version that identifies affected parts at the coarse granularity.

#### *6.2.1.3 Application System*

The application used in this study is a support system for a library borrowed from (Eriksson and Penker, 1998). A library lends books and magazines. The books and the magazines are registered in the system. A library handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books and magazines are removed when they are out of date or in a poor condition. The librarians can easily create update, delete and browse information about the titles in the system. The borrowers can browse information about the titles. They can reserve a title if it is not available.

The application consists of five packages. The data about the application is given in Table 6.2. The overall structure of the application is given in Figure 6.2. A code fragment is given in Appendix D.2.

**Table 6.2.** The number of files, lines of source code (LOC) and functionality of the library application

| Package | Files | LOC | Functionality |
|---|---|---|---|
| library | 1 | 24 | The main program. |
| library.util | 1 | 67 | Services that are used in other packages of the system. |
| library.db | 1 | 293 | Services that are used to store data persistently. |
| library.bo | 5 | 567 | The domain classes such as Title, Item. Loan. |
| library.ui | 10 | 1393 | Classes for the entire user interface |
| **Total** | 53 | 2344 | |

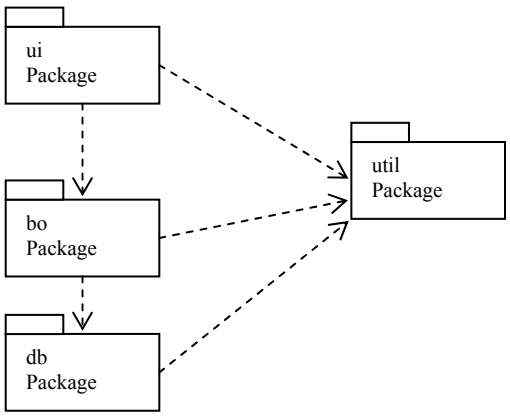



**Fig. 6.2.** A class diagram showing an architectural overview with the application packages and their dependencies.

*6.2.1.4 Data Collection*

Data about the subjects of the experiment and their usage of the technology under study were collected by the tailored logging tool (Karahasanović *et al.*, 2001) that is described in Section 3.4.2. When the experiment started, the students filled in the Personal Information Questionnaire screen (Karahasanović *et al.*, 2001) which requests information about their experience, cognitive style (Kirton, 1994) and attitude to learning new tools. At the end of the experiment, the students were asked to evaluate the tool by filling in the User Evaluation Questionnaire screen (Karahasanović *et al.*, 2001), which is based on the IBM Post-Study System Usability Satisfaction Questionnaire (Lewis, 1995).

### 6.2.2 Results and Discussion

The main focus of our analysis was to identify trends and significant results on time, correctness and user satisfaction. Raw data is given in Appendix D.3. Minitab (Minitab, 1998) was used for data analysis. A boxplot is used to present time spent to solve the tasks. How to interpret a boxplot is depicted in Figure 6.3. Interquartile Range Box (IQ) shows the interquartile range, with the box bottom at the 25th percentile and box top at the 75th percentile. Whiskers shows the values 1.5*IR range.

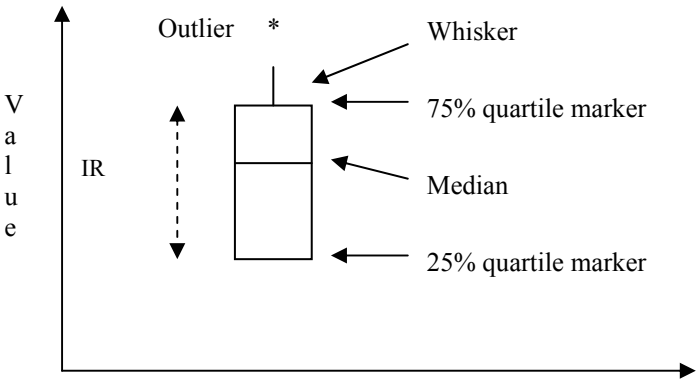



**Fig. 6.3.** Interpretation of boxplots

*6.2.2.1 Time*

The students were asked to conduct two change tasks in 90 minutes. Since they did not finish the second task on time, we compare time only for the first task, but correctness for both tasks. Table 6.3 and Figure 6.4 show the results. The median time used for solving the task was 6 minutes shorter (13% of the total mean time) for the fine granularity version of SEMT than for the coarse granularity version ($p = 0.11$). The data comparisons were conducted using the non-parametric 'distribution free' Mann-Whitney rank test (our data was not normally distributed).

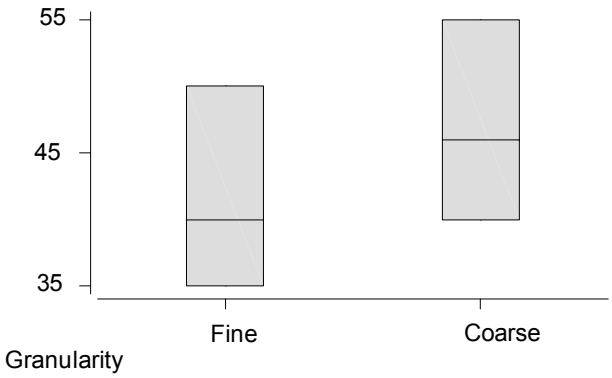



**Fig. 6.4.** Box plots of minutes spent to solve the first task by granularity

**Table 6.3.** Minutes spent to solve the first task

| Granularity | N | Mean | Med | StDev | Min | Max |
|---|---|---|---|---|---|---|
| Fine | 7 | 42.1 | 40 | 6.3 | 35 | 50 |
| Coarse | 6 | 47.0 | 46 | 7.2 | 40 | 55 |

Following guidelines for conducting empirical studies of information visualisation given in (Chen and Yue, 2000), we present the value of *p* rather than determine an *alpha* value in advance.

#### *6.2.2.2 Correctness*

The first change task was to delete a field and identify all impacts of this change. This change had impacts on 27 places in 4 different files. We counted all errors that could lead to compilation or execution problems. The data comparisons were conducted using the Mann-Whitney rank test (Table 6.4). The students using the fine granularity version of SEMT made fewer errors than those using the coarse granularity version did ($p = 0.04$).

**Table 6.4.** Number of errors for the first task by granularity

| granularity | N | Mean | Med | StDev | Min | Max |
|---|---|---|---|---|---|---|
| fine | 7 | 2.4 | 1.0 | 3.0 | 0 | 8 |
| coarse | 6 | 7.7 | 6.5 | 6.6 | 1 | 17 |

When the fine granularity version of SEMT was used only one error type appeared: if several places in the same method were affected by the change, only the first place was discovered. Among the 7 students, 3 made this error. While this may be avoided by providing more time for learning the functionality of SEMT, it seems that indicating the exact number of affected places in the affected methods could be

useful. When the coarse granularity version of SEMT was used, different error types appeared (from not discovering affected classes to not discovering several affected places in the same method).

The second change task was to add a functionality to the library system. This change had impacts on 20 places in 4 different files. We counted all errors that could lead to compilation or execution problems. The data comparisons were conducted using the Mann-Whitney rank test (Table 6.5). The students using the fine granularity version of SEMT made fewer errors than those using the coarse granularity version did ($p = 0.02$). However, these results should be treated cautiously since most students did not finish this task on time.

**Table 6.5.** Number of errors for the second task by granularity

| granularity | N | Mean | Med | StDev | Min | Max |
|---|---|---|---|---|---|---|
| fine | 7 | 6.7 | 5.0 | 3.6 | 5 | 15 |
| coarse | 6 | 11.8 | 14.5 | 4.5 | 6 | 15 |

#### *6.2.2.3 User Satisfaction*

The students were asked to express their agreement with statements addressing adequacy of the SEMT technology by selecting a number on a seven-point scale. Score 1 was used when the students fully agreed with the positive statements on the usability of SEMT; score 7 when they fully disagreed. The general user satisfaction was relatively high (median 2 for the fine granularity; 3 for the coarse granularity). This is similar with the results of the study (Morse *et al.*, 2000) where visual representation was preferred for solving information retrieval tasks. Table 6.6 presents the results.

**Table 6.6.** User satisfaction on a seven point scale by granularity

| granularity | N | Mean | Med | StDev | Min | Max |
|---|---|---|---|---|---|---|
| fine | 7 | 3.1 | 2 | 2.0 | 1 | 6 |
| coarse | 6 | 2.7 | 3 | 1.0 | 1 | 4 |

The data comparisons for the two groups were conducted using the Mann-Whitney rank test. There was no statistically significant difference in user satisfaction for the two versions of SEMT ($p = 0.5$).

The students were also asked to express their agreement with statements addressing other usability aspects of the technology by selecting a number on the same seven-point scale. Table 6.7 presents the results.

**Table 6.7.** Other aspect of usability on a seven point scale by granularity

| Variable | granularity | N | Mean | Med | StDev | Min | Max |
|---|---|---|---|---|---|---|---|
| Easy to learn | fine | 7 | 2.7 | 2 | 1.7 | 1 | 6 |
| | coarse | 6 | 3.5 | 4 | 1.6 | 1 | 5 |
| Easy to complete the tasks | fine | 7 | 3.5 | 4 | 0.9 | 2 | 5 |
| | coarse | 6 | 4 | 4 | 0.6 | 3 | 5 |
| Enough information | fine | 7 | 4.7 | 5 | 0.9 | 3 | 6 |
| | coarse | 6 | 3.1 | 3 | 0.9 | 2 | 4 |
| More information | fine | 7 | 4.8 | 5 | 1.9 | 2 | 7 |
| | coarse | 6 | 5.3 | 6 | 1.9 | 2 | 7 |

The students answered that SEMT was relatively easy to learn (median 2 for the fine granularity; 4 for the coarse granularity) in spite of the short training time (15 minutes). However, some students reported a need for better training in use of SEMT in the think-aloud screen.

The students answered that it was not so easy to solve the tasks (median 4). The answers related to adequacy of the provided information were difficult to interpret.

For the fine granularity version of SEMT, the students disagreed with the statement that SEMT provided enough information (median 4.7) and disagreed with the statement that SEMT provided more information than needed (median 4.8). For the coarse granularity version of SEMT, the answers were consistent. The students agreed with the statement that SEMT provided enough information (median 3.1) and disagreed with the statement that SEMT provided more information than needed (median 5.3). The inconsistency in the answers related to the fine granularity indicates that the questions should be better formulated.

What was behind the high and low scores? The students that were satisfied with SEMT reported that *"SEMT gives good overview of the code"* and *"it was easy to find impacts with SEMT."* The students that were dissatisfied with SEMT reported bad performances (while drawing the graphs), missing help functions *("I had to use documentation several times"*). Several students (from both groups) suggested that SEMT should be connected with an editor and that changes on the code should be immediately displayed by SEMT. No misunderstanding of the visual language of SEMT was reported.

#### *6.2.2.4 Experiment Evaluation*

The students were asked to report the problems they experienced while solving the tasks. Among 13 students, 11 answered that it was not enough time for solving the tasks; 7 answered that it was difficult to understand the tasks; 5 answered that it was difficult to understand the application; 2 reported problems with the programming environment. The students explicitly complained about the short time for performing the tasks ("*I felt time pressure"* and *"It was like an exam"*).

In our research group we often use students in our experiments. We wish that participation in the experiments is useful and interesting for the students. The students were asked to express their agreement with statements addressing the experiment by selecting a number on the same seven-point scale as explained in 6.2.2.3. Table 6.8 presents the results.

**Table 6.8.** Evaluation of the experiment

| Variable | Mean | Median | StDev | Min | Max |
|---|---|---|---|---|---|
| It was useful /interesting for me to participate | 2.1 | 2 | 1.2 | 1 | 5 |
| Organisation of the experiment was good | 2.6 | 3 | 1.2 | 1 | 5 |
| I was well motivated to participate | 2.1 | 2 | 1.3 | 1 | 6 |

The general satisfaction of the subjects with their participation in the experiment was relatively high (median 2 and 3). The students were well motivated to participate in the experiment (median 2).

### 6.2.3 Threats to validity

The following possible threats to validity have been identified:

- The time for learning SEMT and the time for conducting the change tasks were relatively short because of the practical reasons.
- A learning effect is caused by the students learning SEMT during the experiment. The threat to this study was that the students were more familiar with the functionality of SEMT during the second change task than during the first change task.
- The database application system and schema changes may not be representative in their size and complexity.
- The students who participated in this experiment were not professionals.

To help reduce the first two threats we provided the students with the SEMT documentation and the assistance with use of SEMT. The last two threats are more difficult to handle. Although there may be difference between computer science students and professionals, it has been argued that they are close enough to provide useful results in software engineering experiments (Tichy, 2000). Nevertheless, the hypotheses supported in this experiment should be further tested in more realistic industrial settings.

### 6.2.4 Summary

This study evaluated two versions of SEMT regarding productivity and user satisfaction. One version of SEMT identifies affected parts of applications after a schema change at the granularity of classes, whereas another version identifies affected parts at the granularity of fields and methods. We conducted a controlled student experiment with 13 students to validate our visual language, particularly the two levels of granularity. The general user satisfaction was relatively high for both versions of SEMT. The students made fewer errors and spent somewhat shorter time to solve the task with the fine granularity version compared with the coarse granularity version.

## 6.3 Controlled One-Subject Explorative Study

We conducted a controlled one-subject explorative study (Section 3.3.1.6) with the following goals:

- To explore whether a tool that presents impacts of schema changes as a graph (SEMT) improves the productivity of maintaining application consistency compared with tools that presents impacts as a text (Source Navigator and *unix* command-line tools). SEMT, Source Navigator (Navigator, 2001) and *unix* tools *(find*, *grep*) were evaluated regarding productivity and user satisfaction. This qualitative study extends the experiment reported above (Section 6.2) in several important ways. Firstly, SEMT is compared with other tools. Secondly, we defined an extended list of change requests. Thirdly, we studied a larger system. And finally, the subject was studied over a longer period of time. The results of the study were used as an empirical basis for generating hypotheses.
- To gain experience regarding controlled one-subject explorative studies and the logging tool. Conducting explorative studies of one subject over a longer period of time in a laboratory environment (N=1 experiment) raised issues of adequate experimental design and appropriate data collection tools.

Section 6.3.1 describes the design of the study. Section 6.3.2 presents and discusses the results. Section 6.3.3 discusses threats to the validity of the results. Section 6.3.4 reports lessons learned. Section 6.3.5 summarises the results.

### 6.3.1 Design of the Study

The subject of this study was a professional with 11 months of experience in systems development who was systematically studied while conducting schema changes on a medium sized (6600 lines of source code) library application. The study took place within 35 hours divided in 9 sessions through an 11-week period.

We used the A-B-C design for this study. This is our extension of the classical A-B design that was described in Section 3.3.1. The study we conducted consisted of three phases as shown in Table 6.9. Each phase consisted of three sessions. The length of a session varied from 2,5 to 7,5 hours. The subject was asked to identify a stabilisation point for each of the phases. The subject had relatively good knowledge of the tools and about 40 hours experience with the library application. No additional training was provided.

**Table 6.9.** Phases of the study

| Phase | Definition |
|---|---|
| A | The subject used *unix* tools (*grep*, *find*) to conduct schema changes |
| B | The subject used Source Navigator to conduct schema changes |
| C | The subject used SEMT to conduct schema changes |

*6.3.1.1 Subject of the Experiment*

The subject of the experiment was a professional currently employed as a system developer at a Norwegian software development company. Table 6.10 gives an overview of his relevant experience. The subject worked for three companies with several languages and tools. He preferred software development tools with graphical presentation of programs. The subject thought that he easily learned new tools and environments. He was paid for participation in the experiment, but he said that his main motivation was 'interest in new tools and empirical studies'.

**Table 6.10.** Overview of experience for the subject

| Question | Subject's experience |
|---|---|
| Current work title | System developer |
| Work experience | 11 months |
| Education | MSc of Comp.Sc. |
| Oracle | 15 months |
| JDeveloper (CASE tool) | 1 year |
| Other CASE tools | 2 months |
| Assembler | 7 months |
| C++ | 3 months |
| Java | 16 months |
| Simula | related with university courses |
| *unix* and *emacs* | more than 18 months |

*6.3.1.2 Application*

The application system used in this study was an the library application described in Section 6.2.1.3 extended with the following functionality:

- The library can also lend books and magazines to/from other libraries. The borrowers and the librarians can browse information about the titles in the other libraries.
- The borrowers are informed by e-mail when a reserved title is available.

The data about the application is given in Table 6.11.

**Table 6.11.** The number of files, lines of source code (LOC) and functionality of the library application

| Package | Files | LOC | Functionality |
|---|---|---|---|
| library | 1 | 24 | The main program. |
| library.util | 1 | 67 | Services that are used in other packages of the system. |
| library.db | 1 | 505 | Services that are used to store data persistently. |
| library.bo | 13 | 1252 | The domain classes such as Title, Item. Loan. |
| library.ui | 27 | 4781 | Classes for the entire user interface |
| **Total** | 53 | 6629 | |

*6.3.1.3 Change Tasks*

The subject was asked to conduct a set of 10 schema change tasks during the each phase. The tasks were chosen to include changes to the properties of a class, changes to the inheritance graph, changes to the classes, and complex changes. To avoid a learning effect, the sets of tasks were chosen to focus on different classes. These tasks were selected to be as similar possible regarding number of places to be changed.[42] For each general change task, three concrete tasks were given (Table 6.12).

[42] Task 9 is the same for *unix* and SEMT as it was difficult to find the appropriate tasks. As it was 8 weeks between conducting this task with *unix* and SEMT (the subject was in full-time job all the time), we believe that there was no learning effect.

**Table 6.12.** Schema change tasks

| General description | Phase A | Phase B | Phase C |
|---|---|---|---|
| Change task 1<br>add a field;<br>add a method | Add the *country* field (*Address* class) | Add the *type* field (*Borrower* class) | Add the *year* field (*Title* class) |
| Change task 2<br>rename a field | Rename the *address* field (*Borrower* class) | Rename the *title* field (*Item* class) | Rename the *name* field (*Title* class) |
| Change task 3<br>rename a method | Rename the *write* method (writeToFile) | Rename the *read* method (*Persistent* class) | Rename the method *printToFile* (*PrintList* class) |
| Change task 4<br>delete a field;<br>delete a method | Delete the *state* field (*Persistent* class) | Delete the *reservations* field (*Borrower* class) | Delete the *titleName* field (*Title* class) |
| Change task 5<br>change a domain of a field | Change the domain of the *zip* field (Address class) from String to integer | Change the domain of the *isbn* field from String to integer | Change the domain of the *type* field from integer to String |
| Change task 6<br>introduce a new functionality;<br>Required: add a field, add a method, change parameters of a method, change modifiers of a method, change implementation code of a method | Introduce a functionality to inform a borrower when a loan is due | Introduce a functionality to remove a non-active borrower | Introduce a functionality to remove a reservation after three months |
| Change task 7<br>delete a class from the class hierarchy | Delete the *Borrower* class | Delete the *Title* class | Delete the *GeneralPrintClass* class |
| Change task 8<br>generalisation of two classes | Generalise the *printLoanLetter* and *PrintLetter* classes | Generalise the *BorowerFrame* and *LibraryBorrower-Frame* classes | Generalise the *LendItemFrame* and *LendRemoteItem-Frame* classes |
| Change task 9<br>specialisation of a class | Specialise the *Title* class to the *RemoteTitle* class | Specialise the *Address* class to the *CompanyAddress* class | Specialise the *Title* class to the *RemoteTitle* class |
| Change task 10<br>merge two classes | Merge the *Loan* and *Reservation* classes | Merge the *Borrower* and *BorrowerInformation* classes | Merge the *PrintLoanLetter* and *PrintLetter* classes |

#### *6.3.1.4 Organisation of the Study*

The experiment was conducted in a terminal room with other students; the author was not present but available by phone. The first session was conducted on Saturday. All other sessions were conducted in the afternoons, after the subject finished his job. The break between the Phases B and C, as well as the break between Sessions 8 and 9

(Phase C) were due to illness in the subjects' family. During the experiment, the tasks were ordered according to their complexity, and the nature. For example, the task that required some change on a class was conducted before the task that required deleting of the same class. Table 6.13 gives details about the organisation of the study.

**Table 6.13.** Organisation of the study

| Phase | Tool | Session | Tasks | Week | Length (hours) |
|---|---|---|---|---|---|
| A | unix | 1 | 2,1,5,6 | I | 7,5 |
| | | 2 | 10,7,9,4 | II | 3,5 |
| | | 3 | 8,6,3 | II | 4 |
| B | Source Navigator | 4 | 2,1,5 | II | 3,5 |
| | | 5 | 6,8,9 | III | 3,5 |
| | | 6 | 10,4,3,7 | III | 3 ,5 |
| C | SEMT | 7 | 2,1,5 | VIII | 3,5 |
| | | 8 | 6,8,7,10 | VIII | 3 |
| | | 9 | 3,4, 9 | XI | 3 |

*6.3.1.5 Data Collection*

Data about the subject and his usage of the technologies was collected by a tailored logging tool (Section 3.2.4). The tool collected personal information and the user's evaluation of the technology under study by questionnaire screens. It logged the use of the technology in log files and collected qualitative information about how the technology under study was used by a *'think-aloud' screen*. At the end of each phase the author interviewed the subject to increase the validity of the collected data.

### 6.3.2 Results of the Study

The main focus of the analysis was to identify trends and results on time, correctness, user satisfaction and frequency of commands' usage. Usability of a technology can be measured in terms of the time needed to conduct tasks and by the correctness of the solutions. It is context determined if error-free software or shorter development time has higher priority. Therefore, we give no correlation between time and correctness. The subject claimed that the correctness had the highest priority.

*6.3.2.1 Time*

Some problems with *emacs* and SEMT occurred during the study that influenced the time the subject spent to solve the tasks. These are given in Table 6.14.

**Table 6.14.** Problems during the study

| Phase | Task | Comment |
|---|---|---|
| A — *unix* | Task 1 | Problems with *emacs*. |
| | Task 6 | The subject reported in the think-aloud screen that he was tired. |
| C — SEMT | Task 2 | Problems with SEMT; error when opening a class; SEMT was restarted by the experimenter. |
| | Task 6 | Problems with *emacs*; |
| | Task 8 | The subject was unfamiliar with the command *FindImpactsOfTwoClasses*. The help was provided by the experimenter. |

The subject spent much longer time to solve Task 1 with *unix* than with SEMT and Source Navigator. By the analysis of the think-aloud file and log files we discovered that there were problems with *emacs* and that the subject used time trying to restart it.

The subject spent much longer time to solve Task 6 with *unix* than with SEMT and Source Navigator. Analysing the think-aloud file, we discovered that the subject wrote that he was very tired while he was using *unix* (phase A). The subject spent somewhat longer time to solve Task 6 with SEMT than with Source Navigator. There were problems with *emacs* while he used SEMT (phase C).

The time log shows that the subject spent longer time to solve Task 8 with SEMT than with *unix* and Source Navigator. In the think-aloud screen the subject wrote that he did not understand the command for finding impacts on two classes. The additional explanation was provided by the experimenter. Analysing the SEMT log file, we identified when the subject actually started to work on this task. The time he spent to solve this task was actually shorter than with *unix* and Source Navigator. Based on the information from the log files and the think-aloud screen we corrected times for Task 1 (*unix*), Tasks 6 and 8 (SEMT). It was difficult to accurately determine the effect of the fact that the subject was tired on the time he spent to solve Task 6. Therefore, the time for Task 6 (*unix)* was not corrected.

Figure 6.5 shows the time spent to solve the tasks as recorded by the logging tool and Figure 6.6 presents the results after the correction.

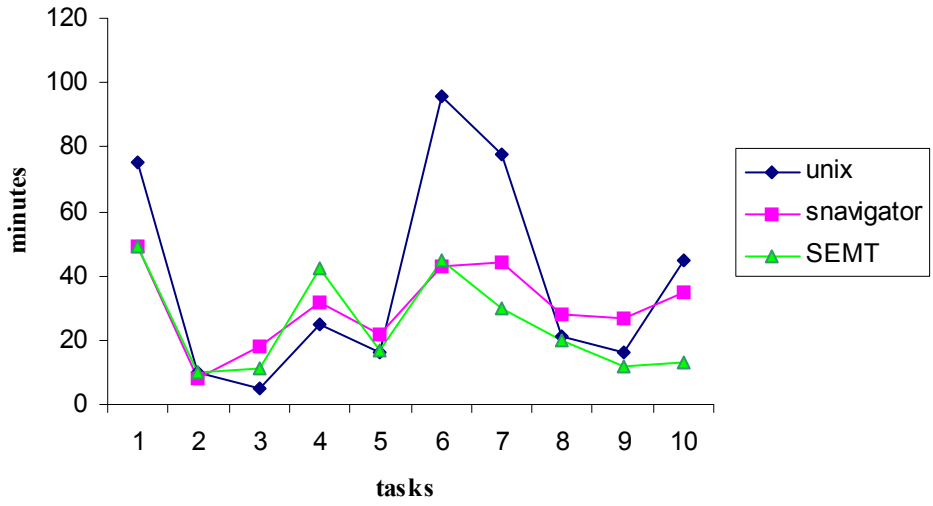



**Fig. 6.5.** Time spent to solve the tasks

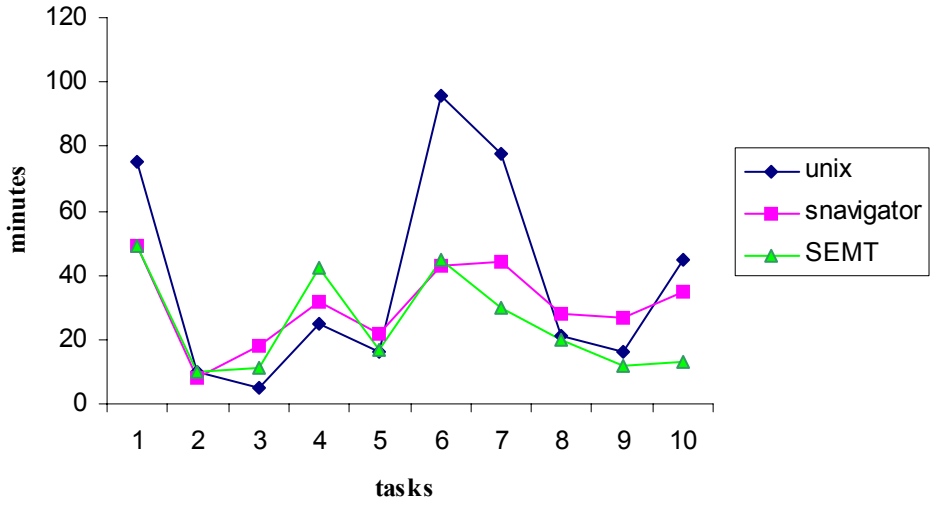



**Fig. 6.6.** Time spent to solve the tasks after the correction

The results show that the subject spent totally shorter time to solve the tasks by SEMT (249 minutes) than by Source Navigator (306 minutes) and *unix* (387 minutes). The subject spent shorter time to solve Tasks 1, 6, 7, 8, 9 and 10 by SEMT than by *unix*. The subject spent shorter time to solve Tasks 3, 5, 7, 8, 9 and 10 by SEMT than by Source Navigator. SEMT scored better than both Source Navigator and *unix* for the

following tasks: delete a class from the hierarchy (Task 7), generalisation (Task 8), specialisation (Task 9) and merging (Task 10). SEMT scored worse than both Source Navigator and *unix* for Task 4 (delete a field and a method).

*6.3.2.2 Correctness*

The solutions for the given change tasks were prepared in advance by the author. Each change affected several places in the source code task (*number of affected places*). The solutions were reviewed by the author and the places that were correctly identified by the subject were counted. [43] Table 6.15 presents the results.

**Table 6.15.** Number of places to be changed that were correctly identified

| **Task** | **unix** | | **Source Navigator** | | **SEMT** | |
|---|---|---|---|---|---|---|
| | correct | affected places | correct | affected places | correct | affected places |
| 1 | 13 | 14 | 13 | 13 | 12 | 14 |
| 2 | 10 | 12 | 6 | 6 | 6 | 6 |
| 3 | 6 | 8 | 7 | 8 | 2 | 2 |
| 4 | 16 | 16 | 20 | 20 | 18 | 18 |
| 5 | 5 | 12 | 11 | 12 | 11 | 12 |
| 6 | 4 | 4 | 4 | 4 | 4 | 4 |
| 7 | 22 | 24 | 26 | 26 | 24 | 24 |
| 8 | 3 | 5 | 3 | 4 | 7 | 8 |
| 9 | 7 | 7 | 7 | 8 | 8 | 9 |
| 10 | 18 | 21 | 15 | 20 | 17 | 20 |

We tried to select the change tasks to be as similar as possible regarding number of places to be changed. However, the changes were not identical. Therefore we introduced a measure for *correctness of the impact analysis — imp_corr*, which is the number of places to be changed that are correctly identified. It was normalised on a scale of 0 to 1 and calculated *as imp_corr = number of correct places/number of affected places.* Figure 6.7 presents the results.

[43] The *unix* tool *diff* was used to identify changes in the source code.

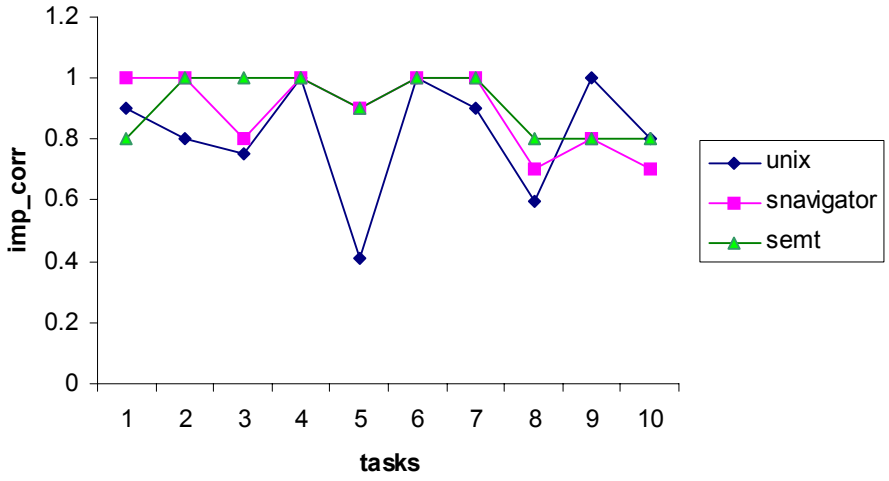



**Fig. 6.7.** Correctness of the impact analysis

Overall correctness was relatively high for Source Navigator and SEMT and somewhat lower for *unix*. For 6 of 10 changes, correctness was the same for SEMT and Source Navigator. For Task 1 (add a field; add a method), the subject achieved better correctness by Source Navigator and for Tasks 3 (rename a method), 8 (generalisation) and 10 (merging) by SEMT.

*6.3.2.3 User Satisfaction*

In an interview, the subject was asked to give more details on the usability of the tools. His main objection regarding the *unix* tools was that they were not precise (too much information) and tedious to use. Compared with Source Navigator and SEMT, they were also more difficult to learn. The main problem with Source Navigator was that it did not report dependencies for classes related by inheritance. The subject suggested that the nodes of the SEMT graph should be closer to each other (or grouped somehow), and that better performances when drawing the graph are necessary for industrial use of the tool. He also suggested connecting SEMT with an editor. Even if he prefered graphical presentation of programs, the subject said that both Source Navigator and SEMT interfaces were equally good.

The subject was also asked to express his agreement with the statements addressing adequacy of the technologies by selecting a number on a seven-point scale. The statements and the results are presented in Table 6.16.

**Table 6.16.** User satisfaction (1 means fully agree; 7 means fully disagree)

| Statement | unix | Source Navigator | SEMT |
|---|---|---|---|
| It was easy to learn the tool | 3 | 2 | 2 |
| It was easy to complete the task with this tool | 5 | 2 | 2 |
| The interface was pleasant | 4 | 1 | 1 |
| Amount of information provided by the tool was enough | 1 | 1 | 1 |
| It was more information than needed provide by the tool | 1 | 7 | 7 |

The results show that Source Navigator and SEMT scored higher than *unix* tools.

*6.3.2.4 Use of the Commands*

During this 35 hours long study our logging tool collected more than 1,5 MB data in 65 log files. We decided to start by analysing the *unix* and SEMT log files. We implemented some simple Java programs that count the commands used by the subject (Tables 6.17 and 6.18). When we discovered something (subjectively) interesting or unexpected we further manually analysed the log files. We also asked the subject to try to explain his actions in several situations.

**Table 6.17.** Use of unix commands

| Command | Phase A unix | Phase B snavigator | Phase C SEMT |
|---|---|---|---|
| cd | 18 | 7 | 5 |
| ls | 5 | 8 | 3 |
| rm | 12 | 6 | 0 |
| grep | 27 | 0 | 0 |
| emacs | 6 | 6 | 8 |
| semt | 0 | 0 | 6 |
| snavigator | 0 | 5 | 0 |

The subject mostly used the tools as we expected. The *unix* command *grep* was not used in the Phases B and C. This indicates that the information provided by Source Navigator and SEMT was good enough. The analysis of the unix log file shows that the *ls* command is more often used with Source Navigator (8 times) than with SEMT (3 times). It may be due the presentation form, but it also may be due to learning effect. The *rm* command was not used with SEMT because the code that should be deleted was written as a comment.

**Table 6.18.** Frequency of use of the SEMT commands

| Command | Frequency | Percent |
|---|---|---|
| Display Search Space (Library) | 3 | 4.7 |
| Display Search Space (Theater) | 1 | 1.6 |
| Redraw | 13 | 20.3 |
| Refresh | 2 | 3.1 |
| Expand Fields | 14 | 21.8 |
| Expand Methods | 11 | 17.2 |
| Impacts | 13 | 20.3 |
| Class Impacts | 4 | 6.3 |
| Impacts of Two Classes | 3 | 4.7 |
| **Total** | 64 | 100 |

The analysis of the SEMT log file shows that the redraw command was used quite often (20.3 %). In the interview after the study the subject explained that he often needed to hide the methods and the fields he expanded in the previous step. As SEMT has no such command, he had to redraw the graph.

In this controlled study, the usage of commands is determined by the given tasks. It can only give an indication whether the tool is used as intended and can identify potential problems. A long-term study of the tool, like the study of the *sem* text editor (Kay and Thomas, 1995), is needed to provide evidence on adequacy of its commands.

### 6.3.3 Threats to Validity

A person's knowledge of an application and tools increases with time. It is possible that the subject performed better in the latter phases of the experiment due to the learning effect. The standard way of dealing with the learning effect in N=1 experiments is the identification of a stabilisation point. The subject was asked to identify a point when he felt that he had good knowledge of the application and the tool for each phase. The subject identified the stabilisation point to be during the first task in all phases. We cannot be sure in which degree we can rely on this subjective information. However, it seems that the order of conducting the task had no effect on the performance of the subject.

The largest threat to the validity of this study is in the nature of the N=1 experiment; the results based on the study of one subject cannot be generalised. We plan to carry out a similar experiment with larger number of subjects.

### 6.3.4 Lessons Learned

This section reports our experience with the N=1 experiment. We have identified the following advantages of N=1 experiment.

- There was no time pressure. In the previous experiment (Section 6.2) most of the subjects (11 of 13) reported that they felt time pressure. The subject of this study, on the contrary, reported that he felt no time pressure and that the experiment was like a development project in industry.
- Higher degree of realism. Due the longer duration of the experiment a wider spectrum of change tasks were conducted. The application was also larger in size and complexity than in the previous experiment.
- Possibility for improvements. If there are some problems with the original design of an N=1 study, improvements during the study are possible. In our case, however, we changed nothing during the study.
- Lower costs than an 'ideal' experiment. Conducting the same experiment with larger number of experts would require very high costs. Therefore, we find N=1 experiment useful not only for generating hypotheses, but also for testing an experimental design before conducting the experiment with larger number of experts.

Conducting this type of study had several disadvantages:

- Relatively high initial costs. The design of the experiment and preparation of the experimental material required much more time for this 35 hours long experiment than for a standard three hours long experiment.[44]
- Lower level of control and high risk related to dropout. Conducting a study with one subject over a longer period can be related to practical problems. We experienced longer unexpected breaks between some sessions due to the illness in the subjects' family. If the subject is unable to participate in the last 5 of 35 hours, for example, the study is completely or partly unsuccessful.
- Importance of the subject description. It is important in all studies to select the subjects that are representative for the population under the study. One subject cannot be representative for the population of software engineers. It is, therefore, important that the subject is described as accurately as possible.
- Limitations of the collected data. As there was only one participant of the study, the results could not be used to statistically validate hypotheses, but we find this type of study useful for indicating trends and generating hypotheses.

### 6.3.5 Summary of Results

This study was explorative in its nature. The main objective was to get insight into the usage of the tools that present impact of schema changes as a text (*unix* and Source Navigator) and as a graph (SEMT). Based on the results of the study we generate the

[44] The majority of the experiments reported in the software engineering literature lasted between one and three hours.

following hypotheses:

- A tool that presents change impacts as a graph (SEMT) improves effectiveness compared with tools that presents change impacts as a text (*unix*, Source Navigator).
- A tool that identifies impacts of complex schema changes (merge, generalize, specialize) by special tailored commands (SEMT) improves effectiveness compared with the tools that have no such commands (unix, Source Navigator).

The subject suggested the following improvements of SEMT:

- Improving performances when drawing the graph.
- Connecting SEMT with an editor.
- Providing a command to hide the methods and the fields that were expanded in the previous step.
- Better layout. Related nodes of the SEMT graph should be closer to each other or better grouped.

## 6.4 Evaluation of SEMT in the Framework

This section evaluates SEMT within the evaluation framework that was described in Section 2.2.2. The results of the studies reported in Sections 6.2 and 6.3 are used for this evaluation. This section also evaluates the framework.

### 6.4.1 Evaluation of SEMT

Table 6.19 presents the utility elements of the framework for SEMT, i.e., the functionality provided by SEMT.

**Table 6.19.** Utility elements of the framework for SEMT

| Utility elements \ System | | SEMT |
|---|---|---|
| **Types of consistency** | **Schema** | Impact analysis |
| | **Data** | No |
| | **Application** | Impact analysis |
| **Taxonomy** | **Class definition** | Add/delete/rename a field;<br>Add/delete/rename a method;<br>Add/delete a class |
| | **Class hierarchy** | Add/delete a class to/from the superclass list of a class |
| | **Complex changes** | Merge, specialise, generalise |
| **Specification way** | | Change commands |
| **Impact identification** | | Impacts on the applications;<br>Impacts on a schema/class hierarchy;<br>Fine and coarse granularity impacts |
| **Impact presentation** | | Text;<br>Variation of a hierarchical graph;<br>Changing the shapes |
| **Automatic conversion** | | No |
| **User interface** | | GUI;textual |

Identifying impacts of changes of an object-oriented system and presenting them as a graph is not a new idea. Both the OOTME system (Kung *et al.*, 1994) and the SC$^2$SM system (Deruelle *et al.*, 1999) use information provided by compilers to identify and present impacts of a proposed change. The main difference between SEMT and these tools is that SEMT identify impacts of complex changes (i.e., merge, specialise).

Furthermore, the visual language of SEMT is different from the visual language of OOTME and SC$^2$SM in that those two systems do not address the problem of displaying larger amount[45] of objects. SC$^2$SM presents both the components and relationships of a system as graph nodes. This produces large graphs even for small programs. OOTME presents relationships by different type of lines. Whereas SEMT presents impacts both at the granularity of packages, classes and interfaces (coarse granularity) and at the granularity of fields and methods (fine granularity), OOTME identifies and presents impacts at the granularity of classes only. Its presentation form is similar to the coarse granularity version of SEMT.

To display a large number of objects, SEMT uses a variation of hierarchical graphs. This choice was determined by the implementation technology we used (Werner, 1998). Large number of objects can also be supported by multiple windows, hierarchical graphs (Storey *et al.*, 1996), node elision and path trimming (Griswold *et*

[45] By the larger number of objects, we mean classes, fields and methods of medium applications (10000 LOC, for example).

*al.*, 1997), ownership trees (Hill *et al.*, 2000) or as a graph with differently sized nodes and edges, for example, *fish-eye* (Sarkar and Brown, 1992). Further research is needed to determine which of the presentation techniques are the most suitable ones for presenting components of object-oriented application systems.

To present the effects of a proposed schema change, we decided to use a combination of changing the shapes and lines. In $SC^2SM$, affected nodes are marked with the *affected* label. This increases the size of the graph and may be awkward for real-life application systems.

Our evaluation of SEMT focused on its usability. The question we attempted to answer was whether the functionality provided by SEMT really improves the process of maintaining application consistency in evolving object-oriented systems from the perspective of schema or application developers. Table 6.20 presents the results of the usability evaluation of SEMT within the framework.

**Table 6.20.** Usability elements of the framework for SEMT

| Element | SEMT |
|---|---|
| **Easy to learn** | Not evaluated |
| **Efficiency** | The subjects spent somewhat shorter time to solve the tasks (delete a field, add a field) with the fine granularity version compared with the coarse granularity version (Section 6.2).<br><br>SEMT scored better than both Source Navigator and *unix* for the following tasks: delete a class from the hierarchy, generalisation, specialisation and merging (Section 6.3). |
| **Correctness** | The subjects made fewer errors when solving the tasks (delete a field, add a field) with the fine granularity version compared with the coarse granularity version (Section 6.2).<br><br>Overall correctness was relatively high for Source Navigator and SEMT and somewhat lower for *unix* (Section 6.3). |
| **User satisfaction** | The general user satisfaction was high for both (coarse and fine granularity) versions (Section 6.2).<br><br>In an interview, the subject said that the *unix* tools were not precise (too much information) and were tedious to use (Section 6.3). Compared with Source Navigator and SEMT, they were also more difficult to learn. The main problem with Source Navigator was that it did not report dependencies for classes related by inheritance. The subject suggested that nodes of the SEMT graph should be closer to each other (or grouped somehow), and that better performances when drawing the graph are necessary for industrial use of the tool. He also suggested connecting SEMT with an editor (Section 6.2). |

Conducting studies focusing on usability requires a lot of resources and is therefore rarely reported in the literature. No studies of the usability of OOTME and $SC^2SM$ were reported, and we had no access to OOTME and $SC^2SM$ so we could not compare efficiency, correctness and user satisfaction achieved by SEMT to those achieved by other similar tools such as OOTME and $SC^2SM$.

### 6.4.2 Evaluation of the Framework

We proposed a framework for evaluation of technologies supporting application consistency (Section 2.2.2). Within this framework we evaluated six representative schema and class evolution technologies (Section 2.2.3) and SEMT (Section 6.4.1).

This framework differs from the evaluations criteria used in (Odberg, 1995; Roddick, 1995; Young-Gook and Rundensteiner, 1997; Rashid and Sawyer, 1999)[46] by focusing on application compatibility and usability of the schema and class evolution technologies. We wanted the framework to allow evaluation of features supported by most existing schema evolution technologies, user oriented and that it can be used as a basis for empirical evaluations.

The framework allowed us to evaluate features of existing technologies in a distinguishable manner. However, completeness[47] of an evaluation framework is difficult to achieve. The framework, for example, does not highlight how a technology is implemented. $SC^2SM$ uses a knowledge-based system to identify impacts; OOTME and SEMT use modified algorithms developed by the authors. These two approaches can differ in development costs and efficiency.

Further evaluation of the framework by applying it on other technologies is needed to make it more complete.

By introducing usability attributes the framework become user oriented. Some proposed measures provided a basis for empirical evaluation of the technologies. The results of the empirical studies on usability of SEMT were summarised within the framework. The framework could be further improved by adding attributes describing type of study, and size and type of applications.

Our evaluation of the technologies shows that validation of the quality of the new schema is not supported by any technology.

We believe that the proposed framework can assist schema and application developers in understanding, comparing and evaluating technologies supporting application consistency. Researchers may also use this framework to assess their own work and to identify research directions.

## 6.5 Evaluation of the Logging Tool

In Section 3.4.2 we described a logging tool that we developed to conduct usability studies of software engineering tools. The logging tool was used in the studies reported in Sections 6.2 and 6.3. A study designed to evaluate the logging tool is described in Section 6.5.1. Section 6.5.2 presents the results of the study. Validity threats of this study are discussed in Section 6.5.3. Section 6.5.4 summarises.

### 6.5.1 Evaluation of the Logging Tool — Study Design

Means for automatic data collection can be compared and evaluated according to different attributes of the model of system acceptability (Section 2.2.2.1). Performance, reliability, system building costs and operational costs are some of aspects of practical acceptability. The usefulness of a system is determined by its utility and usability. The quality of the data that can be collected and satisfactorily analysed is an aspect of utility. The degree in which logging tools disturb the subjects

[46] A more detailed description of the related work was given in Section 2.2.1.

[47] By completeness we mean that features of existing technologies can be presented within the framework (see Section 2.2.2.2).

of the experiments and the degree in which the subject can control the logged data are aspects of usability.[48]

The usefulness of a system for automatic data collection is closely related to the validity of an experiment. The system should not only alleviate data collection and analysis but should also deal with the threats to the validity of the experiment. Using a single data source includes a risk that if there is a measurement bias, the results of the experiment can be misleading (mono-method bias). This threat to construct validity can be handled by involving different data sources that are cross-checked against each other (Seaman, 1999; Wohlin *et al.*, 1999, Bratthall and Jørgensen, 2002). When the think-aloud protocol (von Mayrhauser and Lang, 1999) is used for collecting behavioural data, the validity of the conclusion depends in high degree on the subjects of the experiment. If they do not explain what they are doing, no valid conclusions can be drawn. The logging tool itself can affect the internal validity of the experiment. If the subjects of an experiment are disturbed by the logging tool, it is possible that the effects we measure are not due to the treatment but due to this disturbance (Wohlin *et al.*, 1999).

The novelty of our logging tool compared with the logging tools (Kay and Thomas, 1995; Welland *et al.*, 1997) is that it combines low-level objective data sources (user command logs) with a high-level subjective data sources (think-aloud screen and evaluation screen).[49] Therefore, our evaluation of the logging tool focuses on the effects that these novelties may have on the validity of the experiments. The study presented in this section aims to answer:

- Whether the combination of data sources collected by the tool provided new or more reliable information than if only a single data source had been used?
- Whether the participants reported what they were doing in the think-aloud screen?
- Whether the participants of the studies were disturbed by the think-aloud screen?

The study analyses data collected by the logging tool during a controlled student experiment (Section 6.2) and a controlled one-subject explorative study (Section 6.3) evaluating usability of SEMT. Figure 6.8 depicts the study.

[48] The list of the evaluation attributes is based on the work reported in (Kay and Thomas, 1995; Welland *et al.*, 1997; Torii *et al.*, 1999).

[49] By an objective data source we mean a data source where there is no human judgement involved. A subjective data source, on the contrary, involves some sort of human judgement. Usability evaluation questionnaires, for example, involve human judgement and different results can be achieved if the subject is asked to answer the same questions again.

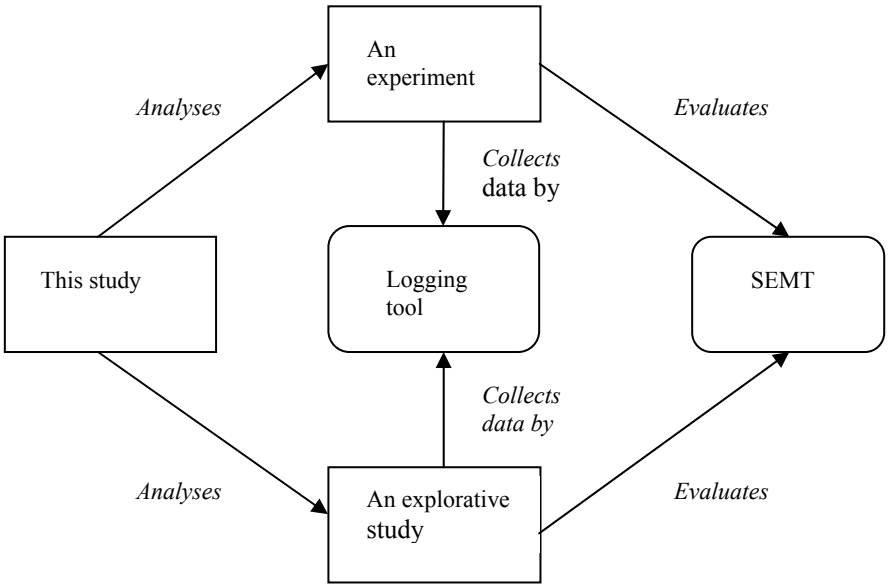



**Fig. 6.8.** The study that evaluates the logging tool

To explore whether the combination of data sources collected by the tool provides new or more reliable information, we applied cross-case analysis (Seaman, 1999) on our data. It is an analysis method where the data is grouped according to some attribute (type of the case study or data source). A list of propositions[50] is generated for each of the groups. These lists are than compared to find if the propositions from one list support or refute the propositions from the other list.

To investigate what kind of data was collected by the think-aloud screen, we used the *coding* process (Seaman, 1999), that is the comments from the think-aloud files were coded by attaching labels to them and grouped according to the codes. We used the coding schema proposed in (von Mayrhauser and Lang, 1999).

To evaluate whether the participants where disturbed by the think-aloud screen, we used data from the evaluation screen and the think-aloud screen.

### 6.5.2 Results and Discussion

The main focus of our analysis was on the data collected by the logging tool and possible disturbance by the think-aloud screen.

#### *6.5.2.1 Providing New Information*

The results from the controlled student experiment (Section 6.2) are summarised in Table 6.21 A "G" indicates that a proposition has been generated from a data source.

[50] Statements about usability of SEMT.

An "R" indicates that a preposition has been refuted by the data source. An "S" indicates that a preposition has been supported the data source. A "D" indicates that the data source provided more details on reason for refuting ("RD") or supporting the proposition ("SD"). A proposition that has been generated by a subjective source code can be refuted by another objective source. A proposition that has been generated by an objective source code can be refuted by a subjective data source if it provides a reason for refuting the preposition.

**Table 6.21.** Propositions and data sources from the controlled student experiment

| **Propositions** | | **Evaluation screen** | **Logged time**[51] | **Source code** | **Think-aloud** |
|---|---|---|---|---|---|
| **R1** | Task 1 and 2 — no difference in time and correctness between the fine and coarse granularity versions of SEMT | | R | R | RD |
| **R2** | The subject has insufficient knowledge of Java | | R | R | G |
| **R3** | The subject feels time pressure | G | | S | SD |

The primary goal of the controlled student experiment was to test the hypothesis related to the two levels of granularity (proposition R1). The null hypothesis was rejected by the logged time and analysis of the changed source code. The comments in the think-aloud file gave a more detailed explanation why the hypothesis was refuted. The students, for example, wrote that the identification of the methods affected by the change provided by SEMT helped them solve the tasks.

Some students (4 of 13) reported in the think-aloud screen that they had insufficient knowledge of Java (proposition R2). However, their results (time and number of errors) were as the average. As the logged time and the source code are more objective data sources, the preposition is refuted.[52]

Most of the students (11 of 13) reported in the evaluation screen that they had not enough time for solving the tasks. This was supported by the analysis of the source code (they did not complete the second task) and by the comments written in the think-aloud screen (R3).

[51] Logged time is *End_time – Start_time* for each task. *End_time* and *Start_time* are extracted from the *unix* log files.

[52] Whereas source code is an objective data source, the number of errors detected in that source code involves human judgement and cannot be considered as completely objective information. To reduce researcher bias, correct solutions were proposed by a person that was not involved in this research and correct answers were counted by the *diff* tool. Nevertheless, it is possible that we introduce some errors when we counted errors.

The controlled one-subject study (Section 6.3) was explorative. The results were used as empirical basis for generating hypotheses. Table 6.22 summarises the results.[53] The same notation is used as in Table 6.21.

[53] The results on time and correctness of solved tasks were given in Section 6.3.

**Table 6.22.** Propositions and data sources from the controlled one-subject explorative study

| Propositions | | Logged time | Source code | Think-aloud | Unix log | SEMT Log | Interview |
|---|---|---|---|---|---|---|---|
| P1 | Task 1 — much longer time with unix | G | | RD | RD | | |
| P2 | Task 6 — longer time with unix | G | | RD | | | |
| P3 | Task 6 — longer time with SEMT than snavigator | G | | | | RD | |
| P4 | Task 7 — longer time with unix and snavigator | G | | SD | S | S | |
| P5 | Task 8 — longer time with SEMT | G | | RD | | R | |
| P6 | Task 9 — shorter time with SEMT | G | | SD | S | S | |
| P7 | Task 10 — shorter time with SEMT | G | | SD | S | S | |
| P8 | The *ls* command is often used with snavigator | | | | G | | |
| P9 | The redraw command of SEMT is used quite often | | | | | G | D |

The subject spent much longer time solving Task 1 with *unix* than with SEMT and Source Navigator. Analysing the think-aloud file and log files, we discovered that there were problems with *emacs* and that the subject spent a lot of time trying to restart *emacs*. We therefore refuted the proposition P1. More precisely we reformulated P1 to be "longer time" instead of "much longer".

The subject spent much longer time to solve Task 6 with *unix* than with SEMT and Source Navigator. In the think-aloud file the subject wrote that he was very tired while he was using *unix*. We therefore refuted the proposition P2.

The subject spent somewhat longer time to solve Task 6 with SEMT than with Source Navigator. The subject wrote in the think-aloud screen that he had problems with *emacs* while he used SEMT. We therefore refuted the proposition P3.

The prepositions P4, P6 and P7 were generated on the basis of the logged time and supported by the analysis of the think-aloud files, and the *unix* and SEMT log files.

The time log shows that the subject spent longer time to solve Task 8 with SEMT than with *unix* and Source Navigator. In the think-aloud screen, the subject wrote that he did not understand the command for finding impacts on two classes using SEMT. Analysing the SEMT log file, we identified when the subject actually started to work on this task. The time he spent to solve this task was actually shorter than with *unix* and Source Navigator. The proposition P5 was therefore refuted.

The analysis of the unix log file shows that *ls* command is more often used with Source Navigator (8 times) than with SEMT (3 times). Other data sources including the interview provided no explanation for this preposition (P8).

The analysis of the SEMT log file shows that the redraw command was used quite often (20.3 %). Other automatically collected data sources provided no explanation for this preposition (P9). In the interview after the study the subject explained that he often needed to hide the methods and the fields he expanded in the previous step. As SEMT has no such command, he had to redraw the graph.

For a proposition generated from a data source, the following cases occurred:

- It was refuted by other data sources
- It was supported by other data sources
- Other data sources provided no information that could support or refute it

For the preposition R2 generated from the think-aloud file, the log files and the source code files (objective data sources) provided information that refuted the preposition.

For the propositions P1, P2, P3 and P5, the think-aloud file provided new information that contributes to refuting the prepositions. The comments written in the think-aloud file together with the log files helped us discover some special events that occurred during the experiment, which might indicate a causal relationship.

For the propositions R1 and R3 and the propositions P4, P6 and P7, the think-aloud file provided more evidence that supported the prepositions.

The data automatically collected by our logging tool provided no information that could support, reject or explain the prepositions P8 and P9.

These results are similar to the results of others. It has been shown that analyses based on multiple data sources, called *data triangulation* (Patton, 1987), increase the validity of case studies (Yin, 1994; Jørgensen, 1995b; Bratthall and Jørgensen, 2002).

#### *6.5.2.2 Data Collected by the Think-Aloud Screen*

The think-aloud screen was frequently used during both studies. Table 6.23 presents the comments written in the think-aloud screen and their frequency. There were totally 125 comments in the first study and 152 comments in the second study. To explore what kind of comments the subjects wrote in the think-aloud screen, we

classified them according to the coding schema described in (von Mayrhauser and Lang, 1999). The dominant activities described by the subjects of the student experiment were writing code (20 of 57) and trying to understand the relationships among the program components (12 of 57). The explorative study was similar. Writing code was described by 49 of 68 comments and trying to understand the relationships was described by 15 of 68 comments. As the explorative study was conducted with one subject, the comments were obviously less diverse.

**Table 6.23.** Type and frequency of the comments. The codes are from (von Mayrhauser and Lang, 1999).

| **Comment** | **Code** | **Count** | |
|---|---|---|---|
| | | **Study I** | **Study II** |
| Reading a manual | man.rea | | 4 |
| Searching file hierarchy | fil.sea | 2 | |
| Reading instructions | ins.rea | 3 | |
| Searching instructions | ins.sea | 3 | |
| Reviewing instructions | ins.rev | 2 | |
| Searching mental knowledge | knw.sea | 1 | |
| Searching a connection diagram | con.sea | 12 | 15 |
| Writing code | cod.wri | 20 | 49 |
| Reviewing code | cod.rev | 2 | |
| Reading code | cod.rea | 3 | |
| Generating code | cod.gen | 2 | |
| Searching variables | var.sea | 5 | |
| Re-examining previous actions | var.act | 1 | |
| Reading comments | com.rea | 1 | |
| Other | unclassified | 68 | 84 |
| **Total** | | 125 | 152 |

Most of the comments (68 of 125 from the student experiment and 84 of 152 from the explorative study) cannot be classified according to this schema. Table 6.24 shows a sample of such comments. The first group of comments describes what the subjects were doing; the second group of comments expresses the feelings of the participants; the third group of comments describes what the subjects were thinking during the experiment; the fourth group includes questions. Note that even though a human observer (the author) was present during the study, some questions were directed to the think-aloud screen.

**Table 6.24.** Comments written in the think-aloud screen

| **Comments describing activities** |
|---|
| I started to work on Task 3 |
| I am looking at the source code … |
| I am trying to find the relationship between classes… |
| I am trying to find the isbn field…have found it |
| I am working… |
| I can't see everything in one screen, have to go back and forward |
| Problems with emacs. I have to start it again. |
| I have finished…playing and testing the tool… |
| **Comments describing feelings** |
| I am frustrated; I don't understand the task… |
| (the same student 10 minutes latter)…finished Task 2 |
| I feel time pressure… |
| I am little bit confused…. It's OK… |
| **Comments describing thoughts** |
| I am reflecting on the class Title |
| Oops! I should be more familiar with emacs… |
| It was long time ago I used Java, I should master it better… |
| It would be easier if I had learned the tool better…. |
| It is difficult to start… |
| **Questions** |
| What should I do to present the changes? |
| Should I delete the code? |

These results suggest that the subjects used the think-aloud screen as intended. They described what they were doing and the causes of their actions and frustrations. However, some pieces of information might be missing because they might find them unimportant, or they might deliberately wish to give positive feedback. Furthermore, the comments collected during the explorative study were less diverse, probably because the study was conducted with only one subject, but it also may be due to the long duration of the study.

*6.5.2.3 Disturbance by the Think-aloud Screen*

The subjects of the first study were asked to express their agreement with statements addressing adequacy of the logging technology by selecting a number on a seven-point scale. The results are presented in Table 6.25. We assumed that the think-aloud screen, which appears every 10 minutes, would disturb the participants. Even if the students were not quite used to write comments (median 3 on the seven-point scale),

they claimed that they were not disturbed by the think-aloud screen (median 6) and that the screen did not appear too often (median 6). The students also claimed that their work was not influenced by the logging tool (median 7).

**Table 6.25.** User evaluation of the think-aloud screen on a seven-point scale (1 means fully agree; 7 means fully disagree).

| Variable | Mean | Med | StDev | Min | Max |
|---|---|---|---|---|---|
| The think-aloud screen disturbed me in my work | 5.4 | 6 | 2.1 | 1 | 7 |
| The think-aloud screen appeared too often | 5.1 | 6 | 2.0 | 2 | 7 |
| I am used to write the comments while I am working | 2.9 | 3 | 1.4 | 1 | 5 |
| I worked differently because I knew that everything was logged | 6.1 | 7 | 1.7 | 2 | 7 |

There were three positive statements from the students. One said: "The think-aloud screen is a very good idea. It helps me to keep the focus". Two others said: "I like that screen with comments." There were no negative statements. It should be noted that in the student experiment, the logging technology was used for a relatively short period of time (90 minutes). This may influence the evaluation of the students. In the second study the logging technology was used 35 hours. The subject of the explorative study also claimed that he was not disturbed by the logging technology and that his work was not influenced by the technology.

This subjective evaluation of the influence of the verbalisation of the problem solving process is similar to the results of others. A study on complex search tasks is reported where the users who were thinking aloud performed 9% faster than the users who were not thinking aloud (Berry and Broadbent, 1990). Another study is reported on the use of disk utility package (e.g., traverse a directory structure, load a program, change a program) where the users who think aloud made less errors than the users who did not think aloud (Wright and Converse, 1992).

### 6.5.3 Threats to Validity

The main threat to the external validity of the evaluation of the logging tool may be that it is based on the results from two similar studies. The logging tool was used in two studies to evaluate three tools by totally 15 participants[54] during totally 56 hours[55]. These two studies were similar — they both focused on visualising of change impacts and usability of the tool we have developed (SEMT). This may limit the

[54] A pilot experiment with one subject was conducted before the experiment reported in Section 6.2.

[55] 14 participants used the logging tool 90 minutes each and one participant used it 35 hours.

ability to generalise the results. However, we believe that our results can be used in software engineering experiments where focus is on usability and on visualisation.

Furthermore, the participants of these two studies were paid and well motivated to participate. The researcher was not involved in teaching of the participants of the first study, but the subject of the second study was a former MSc student of the researcher. If the participants had had negative attitude to the logging tool, the results might be different. A way to deal with this is to delegate control of the logged data to the participants and give them the possibility to turn off the logging tool.

Another threat to validity of this study is researcher bias. While processing the large amount of data from the log files, we had to select some propositions as starting points for the cross-case analysis. Our choice of propositions may be influenced by our interest in complex change tasks. To decrease the researcher bias, we have randomly selected some time points and conducted cross-case analysis. We discovered no new propositions in this manner. However, future development of the logging tool should include automatic help for integration of data collected from different data sources based on the timestamps. This would not only reduce the researcher bias, but would also allow us to discover causal relationships that we were unaware and thus initiate new research.

### 6.5.4 Summary of Results

The results show that for 5 of 12 propositions the think-aloud file provided evidence that supported propositions. For 4 of 12 propositions the think-aloud file together with the log files helped us discover some special events that occurred during the experiment. This information contributed to refuting the prepositions. For one proposition generated from the think-aloud file (subjective data source), the log files and the source code files (objective data sources) provided information that refuted the preposition. For two propositions generated from command log files other data sources provided no information that could support, reject or explain the prepositions. These results suggest that the subjects used the think-aloud as intended. They frequently described what they were doing and described causes of their actions and frustrations. The students claimed that they were not disturbed by the think-aloud screen; they were quite positive about it.

By collecting data from several data sources, our logging tool allowed cross-check analysis and thus increased the construct the validity of the studies. The results suggest that the subjects described frequently their actions in the think-aloud screen. This increased validity of the conclusions. The think-aloud screen itself did not disturb the subjects and thus did not affect the internal validity of the studies.

The novelties of our logging tool compared with the logging tools described in (Kay and Thomas, 1995; Welland *et al.*, 1997) are that it introduces the think-aloud screen and that it combines low-level objective data sources (user command logs) with a high-level subjective data sources (think-aloud screen and evaluation screen) to increase the validity of studies. When the user interaction with a technology is automatically logged, it may be difficult to discover some events that occur during an experiment and the intention behind the user actions (Kay and Thomas, 1995; Welland *et al.*, 1997). It cannot be deduced from the command log files whether the subject pauses for thought or for cup of coffee (Welland *et al.*, 1997); whether the

subject has a problem with the software or whether a book is accidentally laying on the mouse.

Similar information could be collected by audio-video recording or by a human observer. However, the think-aloud screen has several practical advantages. It is cheaper to collect and analyse data in this manner. The students wrote causes of their actions directly in the think-aloud screen, which alleviates data interpretation. It is possible to conduct experiments with groups of students working in the same room without disturbance from the think-aloud. It may also be used for experiments that are conducted via web (Morse *et al.*, 2000; Sjøberg *et al.*, 2001). Furthermore, we believe the think-aloud screen may help reduce the disturbance by the think-aloud. It has been reported that the subjects have been disturbed by the classical audio-recorded think-aloud and that some of them did not want to answer in the study (Bratthall *et al.*, 2001). Writing comments instead of talking while solving the tasks (as in the classical think-aloud) is more similar to everyday practice of software engineers. Therefore, the subjects of experiments would be less disturbed[56] by the think-aloud screen than by the classical think-aloud. However, more studies are needed to explore further in which degree the think-aloud screen disturbs the participants.

The think-aloud screen has some disadvantages. It records *subjective* information. The experiment participants write their own feelings and describe their activities in their own way. Some pieces of information may be missing because the subjects do not write them for various reasons, for example, they may find them unimportant, or they may deliberately wish to give positive feedback. It is therefore important to combine think-aloud screens with objective data sources such as log files.

The logging tool is used in our experiments that involved limited number of tools and subjects. It is therefore difficult to generalise from the results regarding the logging tool. However, we believe that our results are applicable to software engineering experiments where focus is on usability and on visualisation.

We plan to further develop the logging tool. The next version should allow the participants to read their own data, and delete them if they want.

### 6.6 Ethical Issues

Conducting empirical studies on usability necessarily involves human subjects and deals with ethical dilemmas. This section discusses the ethical issues raised by our studies. It also discusses some conflicts between ethical standards and assuring validity of the studies that we have experienced.

Several risks for the participants involved in empirical research studies have been identified in (Sieber, 2001). We find the following two relevant to our studies: inconvenience and psychological risk. Inconvenience may be that the participants of the study use by a new and unknown technology. They also may feel that they are wasting their time. Psychological risk is, for example to worry about breach of confidentiality and evaluation of their work. The evaluation may affect their grades and employment opportunities. Logging all users commands, as we did, may make it even worse. All their errors are visible. They may also feel pressure to be positive

[56] But still disturbed.

about the technology under evaluation because they are students of the researcher or because they are paid for the participation.

The participants of both studies were volunteers and they were paid for the participation. They were also motivated by the possibility to learn a new tool and gaining experience from participating in a usability study. We assumed that this could be a compensation for the inconvenience. The participants of the studies were asked to comment on the study and their motivation in the evaluation screen. Most of the participants were motivated and found interesting and useful to participate in the study (see Section 6.2). They also said that they were motivated to participate in the study.

The researcher was not involved in teaching of the participants of the first study, but the subject of the second study was a former MSc student of the researcher. At the beginning of the both studies we explained to the participants the goals and procedure of the study, the study procedure, and the logging tool. We assured the participants that the data would be treated confidentially. During the studies we tried to emphasise that we were evaluating the tools, not the participants. But, have we really succeeded in that? All of the participants claimed in the evaluation screen that their work was not affected at all by the logging technology. However, some of them wrote in the think-aloud screen some 'excuse' comments, such as 'I should know Java better' or 'I should be more familiar with *emacs*'. This indicates that they wanted their results to be as good as possible.

Our concern regarding validity of our usability studies was that the subjects might be too positive as they were paid volunteers or due to the connection to the researcher. In our opinion, conflicts between validity and ethics are difficult to avoid. The subjects cannot participate in such studies without some form of compensation. When analysing and reporting the results, the researchers should report potential biases. We believe that motivation of the participants in our two studies might affect overall usability assessment of the tool under study (SEMT). However, the subjects were straightforward in their criticism and suggestion for improvements of the tool. We believe that motivation might affect the overall usability assessment but had probably no effect on the conclusions regarding granularity and complex change tasks. The subjects simply could not guess the hypotheses. Motivation might also affect their positive evaluation of the logging tool, but had no effect on our conclusion related to multiple data sources.

At the end of the study we presented the results to the participants. However, the participants should have better control of the logged data (autonomy of the subjects (Sieber, 2001)). The next version of the logging tool should deal with these issues.

## 6.7 Chapter Summary

This chapter presented the results of three empirical studies evaluating the usability of the visual language of SEMT with respect to productivity and user satisfaction. The main results were:

- The performance was acceptable for a large industrial application system (114000 LOC).
- The overall user satisfaction was relatively high.

- Effectiveness and correctness were higher with the fine granularity than with the coarse granularity version of SEMT for a small application system (2400 LOC).
- Effectiveness and user satisfaction were higher with SEMT than with unix and Source Navigator for a medium application system (6700 LOC). Identifying complex changes improved effectiveness.

SEMT was also evaluated within the framework for evaluation of technologies supporting application consistency. This chapter then presented an evaluation of the methods and tools we used to evaluate SEMT:

- a study that evaluated the usefulness of the logging tool
- experience with a controlled one-subject explorative study
- an evaluation of the framework we proposed

Some ethical issues raised by our studies were also discussed in this chapter.

# 7 Conclusions and Future Work

Improving the process of maintaining application consistency in an object-oriented context from the perspective of schema or application developers requires a systematic understanding of schema and class evolution process, related technologies and needs of the developers. Impact analysis is one approach to supporting application consistency. The current knowledge of the usability of such solutions is limited.

The main contribution of this thesis was the development a technology that identifies impacts of changes to an object-oriented system and presents this information in an appropriate way. The proposed technology was evaluated by two experiments and one case study.

The second contribution of this thesis was a framework for evaluating technologies supporting application consistency and the application of the framework to evaluate the existing schema evolution technologies. A tool that automatically collects data during the experiments in software engineering has been developed.

## 7.1 Summary of Results

Maintaining application consistency during schema and class evolution is a non-trivial task. This thesis has demonstrated that identifying and visualising impacts of changes in evolving object-oriented systems is a step towards improving the process of maintaining application consistency in an object-oriented context from the perspective of schema or application developers.

### 7.1.1 Schema Evolution Management Tool: SEMT

SEMT (Schema Evolution Management Tool) is an impact analysis tool. It takes as input the database schema files and the Java source files of an application. The packages, classes, interfaces, fields and methods are extracted from these files. Relationships among them that are identified by SEMT are encapsulation, inheritance, aggregation and usage. Two versions of the tool are available. One version identifies affected parts of applications at the granularity level of packages, classes and interfaces, whereas the other version identifies affected parts at the finer granularity of fields and methods. SEMT displays components of a schema and applications as a graph. Nodes of this graph are packages, classes, interfaces, methods and fields, whereas the relationships among them are represented as edges. They are displayed as rectangles with the name written inside. Colours are used for denoting different types (*e.g.,* classes, methods and fields). Coloured directed arrows denote relationships. Effects of a proposed schema change are presented by changing the shapes and lines. Although similar tools have been built in different environments, this tool is original in several aspects: SEMT has more sophisticated visual syntax; it supports complex changes; it presents fine-grained impacts.

### 7.1.2 Evaluation of SEMT

Empirical studies evaluating the usability of software engineering tools are needed to assess whether a proposed technology meets the requirements. Conducting such studies requires a lot of resources and is therefore rarely done.

This thesis has reported the results of three empirical studies focusing on the usability of SEMT. Particularly, the visual language of SEMT was evaluated. Several aspects of visualising can be explored, e.g., the effects of animation, colours and arrows orientation. Our studies focused on the overall usability of SEMT, usability of two granularity levels and the usability of presenting impacts of change as a graph versus presenting them as a text. An evaluation of performance was also carried out.

The preliminary evaluation of SEMT was conducted by own judgement on a large real-life application. The performance was acceptable but the presenting all classes, fields and functions in the same graph was inappropriate for large applications. These results initiated the implementation of a new version of SEMT that supports a variant of hierarchical graphs.

A controlled student experiment was conducted to validate the two levels of granularity provided by SEMT. The main findings were that identifying impacts at the finer level of granularity (fields and methods) reduced the time needed to conduct changes and the number of errors. The general user satisfaction was relatively high for both the fine and coarse granularity versions of SEMT.

A controlled one-subject study was conducted to explore whether a tool that presents impacts of schema changes as a graph (SEMT) improves the productivity of maintaining application consistency compared with tools that present impacts as a text (Source Navigator and *unix* command-line tools). SEMT, Source Navigator (Navigator, 2001) and *unix* tools *(find*, *grep*) were evaluated regarding productivity and user satisfaction.

Based on the results of the study the following hypotheses were generated:

- A tool that presents change impacts as a graph (SEMT) improves effectiveness compared with tools that presents change impacts as a text (unix, Source Navigator).
- A tool that identifies impacts of complex schema changes (merge, generalize, specialize) by special tailored commands (SEMT) improves effectiveness compared with the tools that have no such commands (unix, Source Navigator).

The studies demonstrated that a technology that identifies and visualises impacts of changes to an object-oriented system improves efficiency, accuracy and user satisfaction of developers who conduct change tasks. Several improvements of the technology related to performance, presentation form and integration of SEMT with a programming environment were suggested.

It is difficult to generalise the results of the usability evaluation studies. Typically, one particular aspect of visualisation is evaluated in a particular domain by using a particular tool or a set of tools with a limited number of subjects, tasks and applications. This, however, cannot be a reason for not conducting such studies. On the contrary, increasing the number of empirical studies focusing on usability enables synthesising results across different studies. Following guidelines for empirical research in software engineering (Kitchenham *et al.*, 2001) and visualisation (Chen

and Yue, 2000), would alleviate meta-analysis of reported studies and contribute to building of explanatory theories. One of the central hypotheses of the information visualisation theory is that "*users tend to perform better with interfaces with visualisation components than interfaces without such features*"(Chen and Yue, 2000). The studies presented in this thesis validated different aspects of this hypothesis and thus contributed to the building of information visualisation theory. They also increased the understanding of use of schema and class evolution technologies and provided feedback to their further development. Achieving generality and triangulation across research disciplines in usability studies is difficult (Karat *et al.*, 1998). However, we believe that the results of the studies can also be of interest for other communities dealing with visualisation in, for example, software restructuring and reverse engineering.

### 7.1.3 Examples from Current Practice

Relational and object/relational databases are widely used in the industry. This thesis presents anecdotal evidence on problems in current practice related to application consistency in the context of relational and object/relational databases, and object-oriented languages. Based on interviews with experienced database administrators, the following techniques for maintaining application consistency were identified: avoiding change, minimising dependencies between schemata and applications by *meta-programming*, and manually maintaining 'private' documentation. This indicates that there is a room for improvement of the existing tools that are used to maintain application consistency.

Based on measurements of an Oracle8i application, views are identified as a useful means for supporting application consistency. By absorbing the effects of schema changes on the applications, the effects of the schema modifications on applications for the following schema changes were reduced: renaming an attribute/relation and splitting/merging relations. The benefits of views were greater for splitting/merging a relation than for renaming an attribute/relation. Views were not helpful for other schema changes.

### 7.1.4 Framework for Evaluation

This thesis has introduced a framework for evaluation technologies supporting application consistency in an object-oriented context from the perspective of schema and application developers. The framework consists of utility and usability elements. Utility elements are associated with a list of applicable values. The framework provides a basis for empirical evaluation of schema and class evolution technologies. A list of *measures* is included in the framework. The effects of a particular functionality on usability can be expressed in terms of these measures (*e.g.,* the time needed to conduct a schema change).

Five representative schema evolution technologies and one class evolution technology were evaluated within this framework. The need for support for complex changes was identified.

### 7.1.5 Logging Tool

Conducting empirical studies in general, and usability evaluation of software engineering tools in particular, requires efficient and reliable data collection. To conduct usability evaluation of SEMT, a tool was developed to automatically collect data about subjects, their interaction with the technology under study and their evaluation of the technology under study.

The novelties of our logging tool compared with similar logging tools (Kay and Thomas, 1995; Welland *et al.*, 1997) are that it introduces a think-aloud screen and that it combines low-level objective data sources (user command logs) with a high-level subjective data sources (think-aloud screen and evaluation screen). It has been demonstrated that this combination increased the validity of the conducted studies in two ways. By collecting data from several data sources, the logging tool allowed cross-check analysis and thus increased the construct validity of the studies. The subjects described frequently their actions in the think-aloud screen. This increased the validity of the conclusions based on the comments written in the think-aloud screen.

While the similar information could be collected by audio-video recording or by a human observer, the think-aloud screen has several practical advantages. It is cheaper to collect and analyse data collected in this manner. It is possible to conduct experiments with groups of students working in the same room without disturbance from the think-aloud or use it for experiments that are conducted via web.

The logging tool was used in experiments with only a limited number of tools and subjects. It is therefore difficult to generalise the results regarding usefulness of the logging tool for other kinds of tools and subjects. However, we believe that our results can be used in software engineering experiments where focus is on usability and visualisation.

## 7.2 Future Work

This section outlines areas for future work related to the main contributions of this thesis.

### 7.2.1 SEMT

The algorithm used by SEMT to identify impacts of changes takes a conservative 'worst-case' approach in that that it calculates all classes and methods that *might be* affected by a proposed change. Some types of change that will not necessarily affect other parts of a system will be calculated in the transitive closure relation. The main problem with this approach is its potentially low precision.

A way to deal with this problem is refinement of the algorithm based on the detailed information about the impacts of each particular change. One can distinguish among changes that have impacts on the class itself, impacts on its children and impacts on all the classes related to that class by the transitive closure (Section 5.5.1). Future research should address issues of efficient implementation of such an improved impact analysis algorithm, and measure the effects of the improvements on large real-life application systems.

At present SEMT does not identify effects off schema changes on the data in the database. However, the model could be extended to manage this. Bindings indicating existence of a persistent instance could be detected by a DBMS and stored in the repository. These bindings could then also be included in the impact analysis.

However, both the impact analysis algorithm and the presentation form need to be adapted to manage new information. Furthermore, SEMT has to be connected with an object-oriented DBMS. A solution could be extending the ODMG standard (Cattell and Barry, 2000) with schema change commands. SEMT could generate such standardised schema change commands and be used with different DBMSs.

A problem with SEMT that was identified by the participants of our studies is lack of its integration with a development environment. Based the results and experiences with the visual syntax of SEMT, an extension of UML (Booch *et al.*, 1997) that includes presenting change impacts, could be proposed. This would have the advantage of easier integration with a development environment. The graph notation would also have been familiar to the schema and application developers.

SEMT could also be extended to validate the quality of the new schema/class hierarchy according to some set of criteria. Redundancies in the inheritance graph could be detected. The complexity of the system before and after the proposed change may be measured by some of measures proposed for object-oriented systems (Briand *et al.*, 1999). We anticipate that this could be useful for restructuring the system.

### 7.2.2 Logging Tool

"Experimentation in software engineering is necessary but difficult" (Basili *et al.*, 1999). This thesis demonstrated the usefulness of a tool that automatically collects data about subjects, their interaction with the technology under study and their evaluation of the technology during experiments on usability of software engineering tools. The challenge is to provide support for analysis of data collected in this manner. One approach could be to integrate data collected from different data sources based on the timestamps. Several types of data can be visualised in the same window as in Ginger2 (Torii *et al.*, 1999). For example, a Cartesian graph can combine logs of keystrokes typed in *emacs* windows with *unix* and SEMT commands logs. The time can be presented on the X-axis. The number of keystrokes, and usage of SEMT and *unix* commands can be presented on the Y-axis. Number 1 can denote that a command is used and 0 that it is not used. By analysing such a graph one can identify patterns of command usage and thus identify potential causal relationships. Combining high-level subjective data sources (think-aloud screen and evaluation screen) with low-level objective data sources (command log files) creates some new opportunities. A think-aloud file can be searched for a certain word, for instance "problem", and graphs presenting command logs for the corresponding period of time can be displayed. Thus, causes of repeating the same command several times can be identified, for example.

It has been pointed out that participants of the experiments should have control of the logged data (autonomy of the subjects) (Sieber, 2001). The next version of the logging tool should allow the participants to select a logging level, read their own data, and delete it if they really want. Extensions of the logging tool in that direction might benefit from the work reported in (Kay and Thomas, 1995; Atkinson *et al.*, 2001).

Future work on this logging tool should also include its integration with the Simula Experiment Support Environment (SESE) (Sjøberg *et al.*, 2001). SESE is a web-based technology that provides support for defining a new experiment, filling in questionnaires, downloading task descriptions and other documents, and storing the results in a database.

### 7.2.3 Other Directions of Future Work

Other directions of future work may include empirical studies of benefits of views for maintaining application consistency. The study reported in this thesis that evaluated support for application consistency by views in the Oracle8i system was explorative in its nature. The benefits of views need to be evaluated on large industrial applications during a longer period of time (several years). Statistics could be collected on both effects of changes on the applications and effort needed to restructure the database.

## 7.3 Final Remarks

In recent years, the demand for evaluating usability of software technologies has increased. This, in turn, has increased the demand for empirical studies evaluating the proposed technologies and consequently the need for efficient and reliable data collection during such studies. Supporting application consistency in evolving object-oriented systems is generally considered important, but difficult.

The work presented in this thesis is, we believe, a step towards improving usability of a technology that support application consistency by impact analysis. The logging technology that is proposed in this thesis is a step towards simplifying data collection and analysis during usability studies.

# Appendix A: The Logging Tool

## A.1 Personal Information Questionnaire

Name:
Total number of credits:
Total number of credits in programming:
Total number of credits in databases:

Experience
Please write the three DBMSs, CASE tools and programming languages that you have most experience with. Please specify if you experience was related to university courses or industry work. For work in industry, please give you experience.

| Experience with DBMS (Oracle, Sybase, Informix, O2, ObjectStore, other)? | | |
|---|---|---|
| DBMS | Related to courses | Experience (months) |
| | | |
| | | |
| | | |

| Experience with DBMS (StP, Designer-Oracle, S-Designer, other)? | | |
|---|---|---|
| CASE tool | Related to courses | Experience (months) |
| | | |
| | | |
| | | |

| Experience with programming languages (Java, C++, Simula, Smalltalk, Pascal, C, Fortran, other)? | | |
|---|---|---|
| Prog. language | Related to courses | Experience (months/LOC) |
| | | |
| | | |
| | | |

Do you think that is easier to change an object-oriented program if you have its graphical presentation? Yes [ ] No [ ]

Do you consider that you learn new programs and tools easily? Yes[ ] No [ ]

### A.2 Usability Evaluation Questionnaire

*There are three versions of this questionnaire: for unix, for Source Navigator and for SEMT. We present here the version with questions related to SEMT.*

This questionnaire gives you an opportunity to reaction to the system you used. Please read each statement and indicate that you strongly agree or disagree with the statement by choosing a number on the scale (1=strongly agree, 7= strongly disagree).

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| It was easy to learn SEMT | | | | | | | |
| It was easy to complete the tasks with SEMT | | | | | | | |
| The interface of SEMT was pleasant. | | | | | | | |
| SEMT provided enough information. | | | | | | | |
| SEMT provided more information than needed. | | | | | | | |

I had the following problems with SEMT.

| Please, describe the problem….. |
|---|
| |

I have the following suggestions for improvements of SEMT.

| Please, write your suggestions….. |
|---|
| |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| The change tasks had appropriate level of difficulty. | | | | | | | |
| The change tasks were too difficult. | | | | | | | |
| It was enough time to solve the change tasks. | | | | | | | |

I have not finished change tasks because:

| | |
|---|---|
| it was not enough time. | |
| it was difficult to understand the tasks. | |
| it was difficult to understand the application. | |
| there were problems with the programming environment. | |

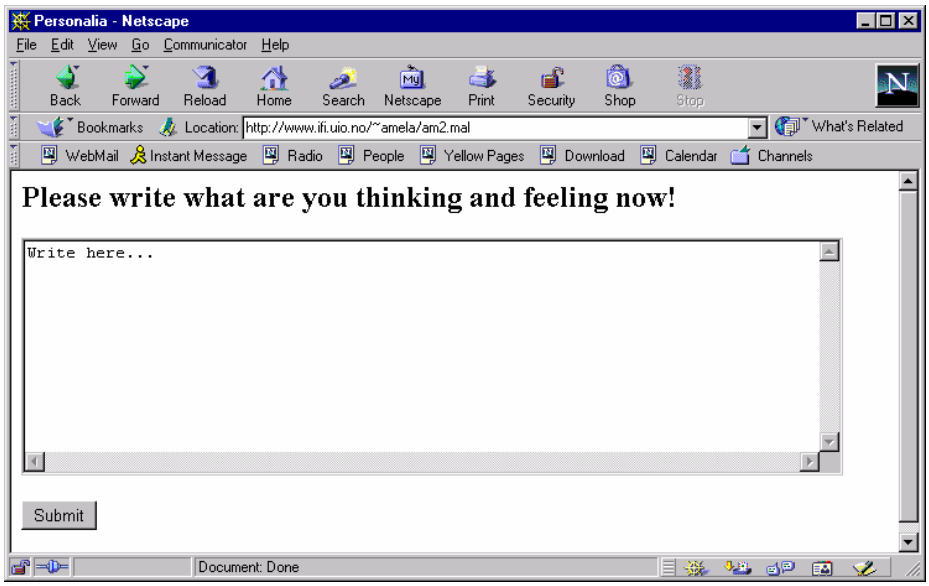
| |
|---|
| If you had other problems, please describe here…. |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| It was useful/interesting for me to participate in the experiment. | | | | | | | |
| I was well motivated to participate. | | | | | | | |
| Organisation of the experiment was good. | | | | | | | |
| 'What do you think' screen appeared too often. | | | | | | | |
| 'What do you think' screen disturbed me in my work. | | | | | | | |
| I am used to write comments while I am programming. | | | | | | | |
| I worked differently because I knew that everything was logged. | | | | | | | |

| |
|---|
| If you have suggestions for the improvement of the experiment, please write here…. |

| |
|---|
| If you have any other comments, please write here…. |

## A.3 Think Aloud Screen



# Appendix B: Supporting Application Consistency in an Industrial Context

## B.1 Experience Level Questionnaire

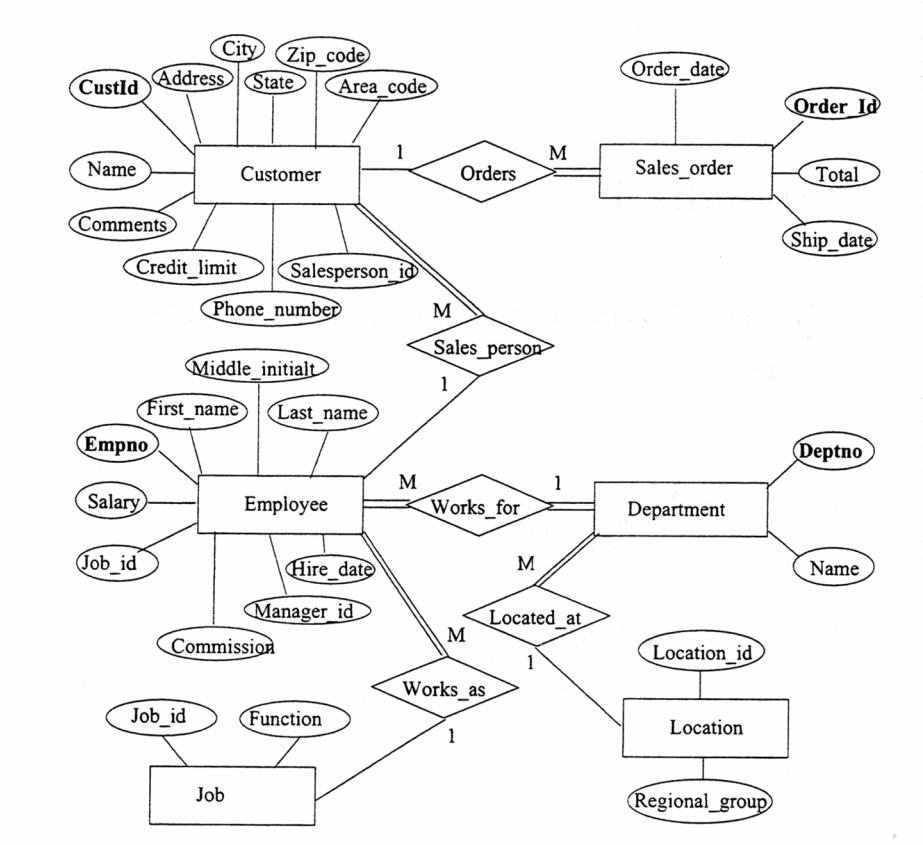
| Question |
|---|
| Current work title |
| Organisation |
| Work experience |
| Education |
| Experience with DMBS |
| Experience with CASE tools and DBA tools |

## B.2 Current Practice Questionnaire

| Question |
|---|
| Could you describe the system (size and complexity)? |
| How do you specify the schema changes? |
| How do you find impacts of database schema changes on applications? |
| What is typical size of the modules that are reported as affected by the schema change? |
| Who is responsible for updating the data dictionary? |
| What are the routines needed to assure up-to-date state of the data dictionary? |
| Do the schema and application developers follow these routines? If not, why? |
| Could you give your insight into problems regarding maintaining application consistency? |

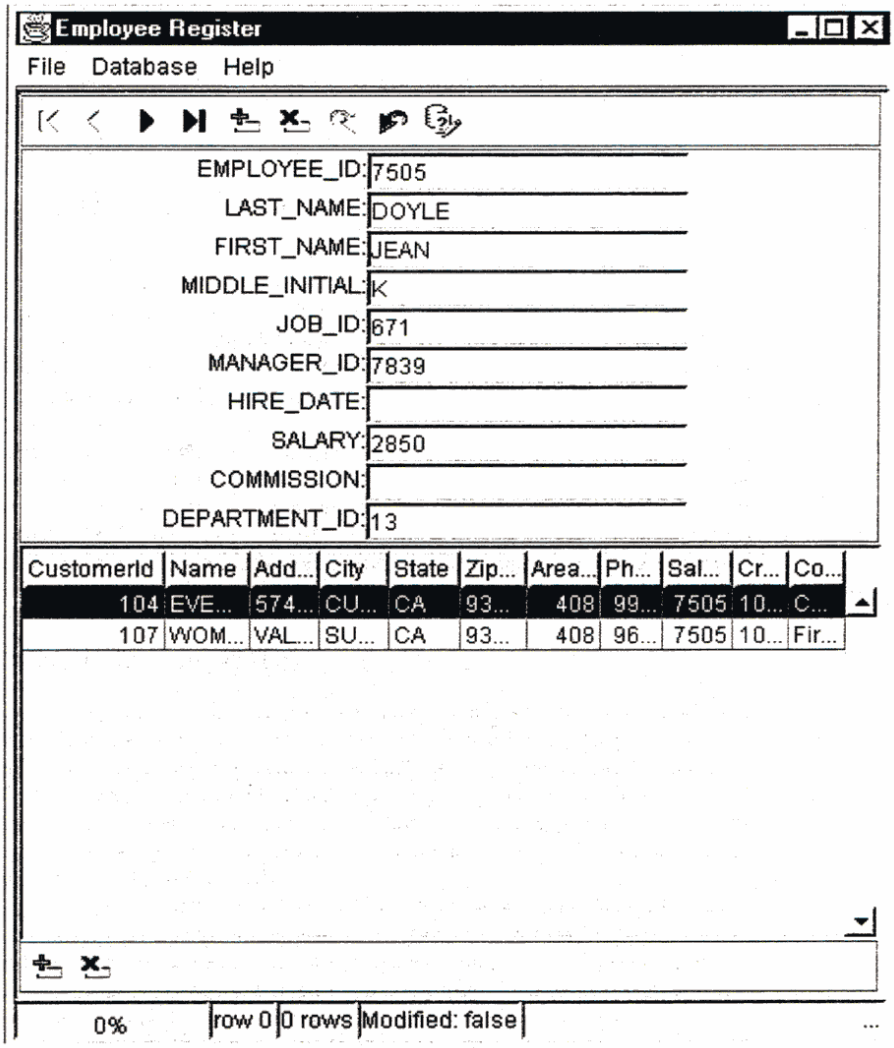
# Appendix C: Application Consistency Using Oracle8i

## C.1 Entity Relationship Diagram of the Company Database



## C.2 User Interface of the Application

The screen presents information about an employee and all customers he is responsible for.



# Appendix D: Visualisation Experiment

## D.1 Experiment Documentation — Change Tasks

### 1. Starting the Experiment Session and Filling in the Background Questionnaire

During this experiment all your keystrokes will be recorded. Please don't worry! All data will be processed anonymously and cannot in any way be related with your name. But, of course, you have to do your best.

And again WE DON'T EVALUATE YOU – YOU EVALUATE THE TOOL!

To the automatic data collection program, type start in X-window.

isuee1>**start**

A Netscape window with questions about your programming experience will appear on the screen. Please answer these questions as precisely as you can. When you finish, the recorded data will be displayed. You can change or confirm the data.

Then an *emacs* window will appear. You must use this and only this *emacs* window to change needed files! Please do not kill this *emacs* window or use any other editor!

### 2. Change Tasks 1, 2 and 3

Read carefully the tasks and implement them. The source code you are going to change is on the directory called Library. To find out places that you have to change use SEMT. Once again, for editing files you must use the *emacs* window that appeared when you started the experiment! You must not kill this *emacs* window or use any other editor! You can start the original application to see how it works, or compile and test the files that you have changed if this will help you. However, it is not required that your solution is without compile or run-time errors. It is more important that you discover all the places to be changed. During this session a small window will appear every 10 minutes asking you what are you thinking. Please describe briefly what you are doing in that moment and what problems you have. Two or three short notes will be enough. Don't worry if you cannot finish the change tasks before the time has run out. If you finish earlier, you shall try to improve your solutions or solve change task 3.

**Change Task 1.** By now the ISBN numbers of the titles are recorded by the system. As a new international categorisation system is accepted, we do not need this information. You should remove it from the all places where it is used.

**Change Task 2.** Extend the handling of titles so that they can be placed in different categories, and add user-defined information to each title (e.g., a review of a book).

Hint. Add a field called Category to the class Title.

**Change task 3.** Add the Book Title and Magazine Title classes to the design, and add some new appropriate attributes to each of these classes. Make the existing Title class abstract and make sure the new classes can be stored persistently.

You have done your best. It is time to evaluate the tool and answer some other questions. Type exit in the
X-window:

isuee1>**exit**

A Netscape window with questions about the tool will appear on the screen. Your evaluation of the tool is very important to us, so please take your time and think before you answer. This is your opportunity to say what you think about the tool. When you finish, the recorded data will be displayed. You can change or confirm the data.

Thank you for your time!

## D.2 Code Fragment from the Library Application

```
package bo;
import util.ObjId;
import db.*;
import java.io.*;
public class Item extends Persistent
{
   private int itemid;
   private ObjId title;
   private ObjId loan = new ObjId();
   public Item()
   {
   }
   public Item(ObjId titleid, int idno)
   {
      title = titleid;
      itemid = idno;
   }
   public int getId() { return itemid; }
   public String getTitleName()
   {
      Title t = (Title) Persistent.getObject(title);
      return t.getTitle();
   }
   public void setLoan(ObjId loanid)
   {
      loan = loanid;
   }
   public Loan getLoan()
   {
      Loan ret = (Loan) Persistent.getObject(loan);
      return ret;
   }
   public boolean isBorrowed()
   {
      if (loan.getName().equals("NotSet"))
         return false;
      else
         return true;
   }
   public void write(RandomAccessFile out)
      throws IOException
   {
      out.writeInt(itemid);
      title.write(out);
      loan.write(out);
```

```
    }
    public void read(RandomAccessFile in)
        throws IOException
    {
        itemid = in.readInt();
        title = new ObjId();
        title.read(in);
        loan.read(in);
    }
}
```

### D.3 Raw Data from the Experiment

The usability evaluation questionnaire was given in Appendix A.2.

**Table D.1.** Data for change tasks and user satisfaction

| Subject | Granularity Coarse(1) Fine(0) | C1-time minutes | C1 errors | C2 errors | User satisfaction | Learning SEMT |
|---|---|---|---|---|---|---|
| 1 | 1 | 55 | 1 | 6 | 4 | 2 |
| 2 | 1 | 40 | 1 | 14 | 1 | 4 |
| 3 | 1 | 40 | 6 | 15 | 2 | 5 |
| 4 | 1 | 50 | 17 | 15 | 3 | 5 |
| 5 | 1 | 55 | 14 | 15 | 3 | 4 |
| 6 | 1 | 42 | 7 | 6 | 3 | 1 |
| 7 | 0 | 40 | 0 | 5 | 1 | 2 |
| 8 | 0 | 35 | 8 | 15 | 2 | 3 |
| 9 | 0 | 40 | 5 | 6 | 6 | 6 |
| 10 | 0 | 50 | 1 | 5 | 6 | 2 |
| 11 | 0 | 35 | 0 | 5 | 3 | 1 |
| 12 | 0 | 50 | 2 | 6 | 2 | 4 |
| 13 | 0 | 45 | 1 | 5 | 2 | 1 |

**Table D.2.** Data for user satisfaction

| Subject | Helped to solve the tasks | Adequate Info. | Too much Info. | Too often think-aloud | Think-aloud disturbed | Write comments | Work different |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 4 | 4 | 7 | 7 | 2 | 7 |
| 2 | 4 | 2 | 7 | 2 | 7 | 4 | 7 |
| 3 | 4 | 4 | 7 | 3 | 3 | 3 | 7 |
| 4 | 4 | 3 | 2 | 6 | 6 | 2 | 5 |
| 5 | 5 | 4 | 6 | 5 | 6 | 2 | 3 |
| 6 | 4 | 2 | 6 | 4 | 6 | 5 | 7 |
| 7 | 3 | 3 | 3 | 3 | 7 | 4 | 7 |
| 8 | 4 | 4 | 5 | 7 | 7 | 5 | 7 |
| 9 | 5 | 5 | 2 | 2 | 1 | 4 | 2 |
| 10 | 2 | 5 | 6 | 7 | 2 | 2 | 7 |
| 11 | 4 | 5 | 4 | 6 | 6 | 3 | 7 |
| 12 | 4 | 6 | 7 | 7 | 5 | 1 | 7 |
| 13 | 3 | 5 | 7 | 7 | 7 | 1 | 7 |

**Table D.3.** Data about the experiment

| Subject | Adequate tasks | Difficult tasks | Enough time | Not enough time | Underst. tasks | Underst. application | Software problems |
|---|---|---|---|---|---|---|---|
| 1 | 4 | 6 | 6 | 1 | 0 | 0 | 0 |
| 2 | 6 | 2 | 5 | 0 | 1 | 0 | 1 |
| 3 | 4 | 4 | 7 | 1 | 0 | 1 | 1 |
| 4 | 5 | 2 | 5 | 1 | 1 | 1 | 1 |
| 5 | 2 | 3 | 6 | 1 | 1 | 1 | 0 |
| 6 | 4 | 4 | 4 | 1 | 1 | 1 | 1 |
| 7 | 4 | 4 | 5 | 1 | 1 | 0 | 0 |
| 8 | 7 | 1 | 1 | 1 | 1 | 1 | 0 |
| 9 | 5 | 2 | 3 | 1 | 0 | 0 | 0 |
| 10 | 2 | 6 | 3 | 1 | 0 | 0 | 0 |
| 11 | 3 | 3 | 4 | 0 | 1 | 0 | 0 |
| 12 | 2 | 7 | 6 | 1 | 1 | 1 | 0 |
| 13 | 5 | 7 | 7 | 1 | 0 | 0 | 0 |

**Table D.4.** Data about experiment

| Subject | Interesting experiment | Motivated | Organisation |
|---|---|---|---|
| 1 | 3 | 2 | 4 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 2 | 2 |
| 4 | 2 | 1 | 4 |
| 5 | 2 | 2 | 3 |
| 6 | 3 | 3 | 3 |
| 7 | 2 | 2 | 2 |
| 8 | 1 | 1 | 3 |
| 9 | 5 | 6 | 5 |
| 10 | 2 | 2 | 3 |
| 11 | 4 | 3 | 3 |
| 12 | 1 | 2 | 1 |
| 13 | 1 | 1 | 1 |

## Bibliography


Adrion, W.R. (1993), Research Methodology in Software Engineering: Summary of the Dagstuhl Workshop on Future Directions in Software Engineering, *SigSoft Software Eng. Notes, ACM Press, New York*, Vol. 18, No. 1, pp. 36–37, 1993.

Al-Jadir, L., Estier, T. and Falquet, G. (1995), Evolution Features of the F2 OODBMS, In *5th. Int. Conf. on Database Systems for Advanced Applications, DASFAA,* Singapore, pp. 284–291.

Al-Jadir, L. and Leonard, M. (1999), Transposed Storage of an Object Database to Reduce the Cost of Schema Changes, In *Advances in Conceptual Modeling/ER'99, Workshop on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling. Proceedings (LNCS 1727),* Springer-Verlag, Paris, France, pp. 48–61.

Andany, J., Leonard, M. and Palisser, C. (1991), Management of Schema Evolution in Databases, In *17th Int. Conf. on Very Large Databases,* Morgan-Kaufmann, Barcelona, Spain, pp. 161–170.

Antis, J.M., Eick, S.G. and Pyrce, J.D. (1996), Visualizing the Structure of Relational Databases, *IEEE Software*, Vol. 13, No. 1, pp. 72–79, 1996.

Arisholm, E., Anda, B., Jørgensen, M. and Sjøberg, D. (1999), Guidelines on Conducting Software Process Improvement Studies in Industry, In *22nd IRIS Conference (Information Systems Research Seminar),* Keuruu, Finland, pp. 88–102.

Arisholm, E., Sjøberg, D.I.K. and Jørgensen, M. (2001), Assessing the Changeability of two Object-Oriented Design Alternatives — a Controlled Experiment, *Empirical Software Engineering*, Vol. 6, No. 3, pp. 231–277, 2001.

Arnold, K. and Gosling, J. (1996) *The Java Programming Language*, Addison-Wesley, Reading, Massachusetts, 1996.

Atkinson, M., Brown, M., Cargill, J., Crease, M., Draper, S., Evans, H., Gray, P., Mitchell, C., Ritchie, M. and Thomas, R. (2001), GRUMPS Sommer Anthology, 2001, http://grumps.dcs.gla.ac.uk.

Atkinson, M.P., Daynes, L., Jordan, M.J., Printezis, T. and Spence, S. (1996), An Orthogonally Persistent Java, *SIGMOD RECORD*, Vol. 25, pp. 68–75, 1996.

Atkinson, M.P., Dmitriev, M., Hamilton, C. and Printezis, T. (2000), Scalable and Recoverable Implementation of Object Evolution for the PJama Platform, In *9th Int. Workshop on Persistent Object Systems POS9 (LNCS 2135),* Springer, Lillehammer, Norway, pp. 292–314.

Atkinson, M.P. and Morrison, R. (1985), Procedures as Persistent Data Objects, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 539–559, 1985.

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, pp. 311–322.

Barbedette, G. (1991), Schema Modifications in the LISPO2 Persistent Object-Oriented Language, In *ECOOP '91 European Conference on Object Oriented Programming (LNCS 512),* Springer Verlag, Berlin, Germany, pp. 77–96.

Barlow, D. and Hersen, M. (1984) *Single Case Experimental Design*, Pergammon Press, 1984.

Basili, V.R. (1996), The Role of Experimentation in Software Engineering: Past, Present, Future, In *18th Int. Conf. on Software Eng.,* IEEE CS Press, Los Alamitos, CA, USA, pp. 442–449.

Basili, V.R., Shull, F. and Lanubile, F. (1999), Building Knowledge through Families of Experiments, *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 456–73, 1999.

Batini, C., Lenzerini, M. and Navathe, S.B. (1986), A Comparative Analysis of Methodologies for Database Schema Integration, *ACM Computing Surveys*, Vol. 18, No. 4, (December 1986), pp. 321–364, 1986.

Bellahsene, Z. (1996), View Mechanism for Schema Evolution in Object-Oriented DBMS, In *14th British National Conference on Databases, BNCOD 14 (LNCS 1094),* Springer, pp. 18–35.

Benzi, F., Maio, D. and Rizzi, S. (1999), VISIONARY: A Viewpoint-Based Visual Language for Querying Relation Databases, *Journal of Visual Languages & Computing*, Vol. 10, No. 2, pp. 117–145, 1999.

Berry, D.C. and Broadbent, D.E. (1990), The role of instruction and verbalization in improving performance on complex search tasks, *Behaviour & Information Technology 9*, Vol. 3 (May–June), pp. 175–190, 1990.

Bertino, E. (1992), A View Mechanism for Object-Oriented Databases, In *3rd Int'l Conf. on Extending Database Technology (LNCS 580),* Springer-Verlag, Germany.

Bevan, N. and Macleod, M. (1993), Usability assessment and measurement, In *The management of Software Quality*, Ashgate Technical/Gower Press.

Bjørnerstedt, A. and Hulten, C. (1989), Version Control in an Object-Oriented Architecture, In *Object-Oriented Concepts, Databases, and Applications*, Chapter. 18, Addison Wesley, pp. 451–485.

BMC (2002), http://www.bmc.com/supportu/prodlist.cfm, BMC Software.

Boehm, B.W. (1988), A Spiral Model of Software Development and Enhancement, *IEEE Computer*, Vol. 21 (May 1988), pp. 61–72, 1988.

Boehm, B.W. and Papaccio, P.N. (1988), Understanding and Controlling Software Costs, *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pp. 1462–1477, 1988.

Bohner, A.S. and Arnold, R.S. (1996), An Introduction to Software Change Impact Analysis, In *Software Change Impact Analysis*, IEEE Computer Society Press, Los Alamitos, CA, pp. 1–26.

Booch, G. (1991) *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.

Booch, G., Rumbaugh, J. and Jacobson, I. (1997) *Unified Modeling Language Semantics and Notation Guide 1.0.*, Rational Software Corporation, San Jose, California, 1997.

Bratsberg, S.E. (1992), Unified Class Evolution by Object-Oriented Views, In *11th Int. Conf. on the Entity-Relationship Approach,* Karlsruhe,German, pp. 423–439.

Bratthall, L., Arisholm, E. and Jørgensen, M. (2001), Program Understanding Behavior During Estimation of Enhancement Effort on Small Java Programs, In *3rd International Conference on Product Focused Software Process Improvement,* Kaiserslautern, Germany.

Bratthall, L. and Jørgensen, M. (2002), Can You Trust a Single Data-Source in Software Engineering Case-Study, In *Empirical Software Engineering*, Vol. 7, No. 1*,* March 2002, pp. 9–26.

Brazile, R.P. and Dongil, S. (1995), A Unifying Version Model for Object-Oriented Engineering Database, In *15th Annual Int. Computers in Engineering Conf., the 9th Annual ASME Engineering Database Symposium.*

Breche, P., Ferrandina, F. and Kuklok, M. (1995), Simulation of Schema Change using Views, In *Database and Expert Systems Applications (LNCS Vol. 978),* Springer, London, UK.

Briand, L.C., Bunse, C. and Daly, J.W. (2001), A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs, *IEEE Transactions on Software Engineering*, Vol. 27, No 6, June 2001, pp. 513–530, 2001.

Briand, L.C., Wust, J. and Lounis, H. (1999), Using Coupling Measurement for Impact Analysis in Object-Oriented Systems, In *Int. Conf. on Software Maintenance (ICSM'99),* IEEE Comp. Soc., pp. 475–482.

Brooks, F.P. (1996), Toolsmith, *Comm. ACM, Mar. 1996*, pp. 61–68, 1996.

Carey, M.J., DeWitt, D.J. and Naughton, J.F. (1993), The OO7 Benchmark, In *Int. Conf. on Management of Data,* ACM SIGMOD.

Casais, E. (1991) *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*, PhD thesis, Faculte des sciences economiques et sociales, University of Geneva1991.

Catarci, T. (2000), What Happened When Database Researches Met Usability, *Information Systems*, Vol. 25, No. 3, pp. 177–212, 2000.

Catarci, T., Costabile, M.F., Levialdi, S. and Batini, C. (1997), Visual Query Systems for Databases: A Survey, *Journal of Visual Languages & Computing*, Vol. 8, No. 2, pp. 215–260, 1997.

Cattell, R., G.,G. and Barry, D., K. (Eds.) (2000) *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers, San Francisco.

Cattell, R.G.G. and Skeen, J. (1992), Object Operations Benchmark, *ACM Transactions on Database Systems*, Vol. 17, No. 1, pp. 1–31, 1992.

Chen, C. and Yue, Y. (2000), Empirical Studies of Information Visualisation: A Meta-Analysis, *Int. Journal of Human-Computer Studies*, Vol. 53, No. 5, pp. 851–866, 2000.

Chen, P.-K., Chen, G.-D. and Liu, B.-J. (2000), HVQS: The Hierarchical Visual Query System for Databases, *Journal of Visual Languages & Computing*, Vol. 11, No. 1, pp. 1–26, 2000.

Chen, X., Tsai, W.T., Huang, H., Poonawala, M., Rayadurgam, S. and Wang, Y. (1996), Omega — An Integrated Environment for C++ Program Maintenance, In *Int. Conf. on Software Maintenance,* IEEE Computer Society Press, Los Alamitos, California, Monterey, California, pp. 114–123.

Clamen, S.M. (1994), Schema Evolution and Integration, *Distributed and Parallel Databases*, Vol. 2, pp. 101–126, 1994.

Coast (1999), http://www.dbis.informatik.uni-frankfurt.de/~coast/, University of Frankfurt/Main, Germany.

Coleman, D., Ash, D., Lowther, B. and Oman, P. (1994), Using Metrics to Evaluate Software System Maintainability, *IEEE Computer, August 1994*, pp. 44–49, 1994.

Connolly, T.M., Begg, C.E. and Strachan, A.D. (1996) *Database Systems, A Practical Approach to Design, Implementation and Management*, Addison-Wesley, Harlow, England, 1996.

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. and Morrison, R. (1994), Using Persistence Technology to Control Schema Evolution, In *9th ACM Symposium on Applied Computing,* ACM Press, Phoenix, Arizona, pp. 441–446.

COOL (2002), http://support.ca.com/public/cool/enterprise/enterprisesupp.html.

Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. (1972), Structured Programming, *A.P.I.C. Studies in Data Processing No.8, Academic Press, New York*, 1972.

Deruelle, L., Bouneffa, M., Goncalves, G. and Nicolas, J.C. (1999), Local and Federated Database Schemas Evolution — An Impact Propagation Model, In *10th International Conference DEXA'99 (Lecture Notes in Computer Science Vol. 1677),* Springer-Verlag, Berlin, Germany, pp. 902–911.

Dittrich, K., Tombros, D. and Geppert, A. (2000), Databases in Software Engineering: A Roadmap, In *The Future of Software Engineering, (in conjunction with ICSE 2000),* ACM Press, New York, NY, Limerick, Ireland, pp. 291–302.

Dmitriev, M. (1998), The First Experience of Class Evolution Support in PJama, In *Advances in Persistent Object Systems — Proc. of the 8th Int. W'shop on Persistent Object Systems (POS8) and the 3rd Int. W'shop on Persistence and Java (PJW3),* Morgan Kaufmann, pp. 279–296.

Dmitriev, M. (2001) *Safe Class and Data Evolution in Large and Long-Lived Java Applications, PhD thesis, Dept. of Comp.Sc., University of Glasgow*, 2001.
Dmitriev, M. and Atkinson, M. (1999), Evolutionary Data Conversion in the PJama Persistent Language, In *1st ECOOP Workshop on Object-Oriented Databases (LNCS 1743),* Lisbon, Portugal.
Doyle, J. (1979), A Truth Maintenance System, *Artifical Intelligence*, Vol. 12, No.3, pp. 231–272, 1979.
Eaglestone, B. and Ridley, M. (1998) *Object Databases: An Introduction*, McGraw-Hill Book Company, London, 1998.
Elmasri, R. and Navathe, S.B. (1994) *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.
Eriksson, H.E. and Penker, M. (1998), Case Study, In *UML Toolkit*, John Wiley & Sons, Inc., New York.
Fenton, N., Pfleeger, S.L. and Glass, R.L. (1994), Science and Substance: A Challange to Software Engineers, *IEEE Software, 1994 (July)*, pp. 86–95, 1994.
Ferrandina, F. and Lautemann, S. (1996), An Integrated Approach to Schema Evolution for Object Databases, In *1996 Int. Conf. on Object Oriented Information Systems, OOIS96,* Springer, pp. 280–294.
Ferrandina, F., Meyer, T. and Zicari, R. (1994), Schema Evolution in Object Databases: Measuring the Performance of Immediate and Deferred Updates, In *20th Int. Conf. on Very Large Data Bases,* Santiago, Chile.
Ferre, X., Juristo, N., Windl, H. and Constantine, L. (2001), Usability Basics for Software Developers, *IEEE Software, January/February 2001*, pp. 22–29, 2001.
Finkelstein, A. (2000), A Foolish Consistency: Technical Challenges in Consistency Management, In *11th International Conference, DEXA 2000 (LNCS 1873),* Springer, London, UK, pp. 1–5.
Fornari, M.R., Golendziner, L.G. and Wagner, F.R. (1995), Schema Evolution in the STAR Framework, *Electronic Design Automation Frameworks*, Vol. 4, No. 10, 1995.
Gallagher, K.B. and Lyle, J.R. (1991), Using Program Slicing in Software Maintenance, *IEEE Trans. Software Eng.*, Vol. 17, No 8, pp. 751–761, 1991.
Garlan, D., Krueger, C.W. and Lerner, B.S. (1994), TransformGen: Automating the Maintenance of Structure-Oriented Environments, *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pp. 727–774, 1994.
Gosling, J., Joym B. and Steele, G. (1996) *The Java Language Specification*, Addison-Wesley, Reading, Massachusetts, 1996.
Grandi, F. and Mandreoli, F. (1999), ODMG Language Extensions for Generalized Schema Versioning Support, In *Advances in Conceptual Modeling ER'99; Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling (LNCS 1727),* Springer-Verlag, Paris, France.
Griswold, W.G., Chen, M.I., Bowdidge, R.W., Cabaniss, J.L., Nguyen, V.B. and Morgenthaler, J.D. (1997), Tool Support for Planning the Restructuring of Data Abstractions in Large Systems, *IEEE Transaction on Software Engineering*, Vol. 24, No. 7, pp. 534–538, 1997.
Grudin, J. (1992), Utility and usability: Research issues and development contexts, *Interacting with Computers*, Vol. 4, No. 2 (August), pp. 209–217, 1992.
Grundy, J. and Hosking, J. (2000), High-Level Static and Dynamic Visualisation of Software Architectures, In *2000 IEEE International Symposium on Visual Languages (VL'00),* IEEE Comp. Soc., Seattle, WA, USA, pp. 5–12.
Guerrini, G., Merlo, I., Ferrari, E., Kappel, G. and de Miguel, A. (1999), Object-Oriented Databases, In *ECOOP'99 Workshops (LNCS 1743),* Springer-Verlag, pp. 207–221.
Habermas, J. (1992), Modernity: An Unfinished Project, In *The Post-Modern Reader*, Academy Editions, London.

Harrison, W. (2000), N=1: An Alternative for Software Engineering Research, In *Beg, Borrow, or Steal Workshop, International Conference on Software Engineering,* Limerick, Ireland.
Henderson-Sellers, B. (1996) *Object-Oriented Metrics, Measures of Complexity*, Prentice Hall, New Jersey, 1996.
Henderson-Sellers, B. and Edwards, J.M. (1990), The Object-Oriented Systems Life Cycle, *Comm. of the ACM*, Vol. 33, pp. 142–159, 1990.
Hill, T., Noble, J. and Potter, J. (2000), Visualizing the Structure of Object-Oriented Systems, In *2000 IEEE International Symposium on Visual Languages (VL'00),* IEEE Comp. Soc., Seattle, WA, USA, pp. 191–198.
Hoare, C.A.R. (1984), Programming: Sorcery or Science?, *IEEE Software*, Vol. 1, April, pp. 5–16, 1984.
Holgeid, K.K., Krogstie, J. and Sjøberg, D.I.K. (2000), A Study of Development and Maintenance in Norway: Assessing the Efficiency of Information Systems Support Using Functional Maintenance, *Information and Software Technology*, Vol. 42, No. 10, pp. 687–700, 2000.
Horn, R.E. (1998) *Visual Language*, Macro Vu, Inc., Bainbridge Island, WA, 1998.
Horwitz, S., Reps, T. and Binkley, D. (1990), Interprocedural Slicing Using Dependence Graphs, *ACM Trans. on Programming Languages and Systems*, Vol. 12, No. 1, pp. 35–46, 1990.
Høst, M., Regnell, B. and Wohlin, C. (2000), Using Students as Subjects — A Comparative Study of Students and Professionals in Lead-Time Impact Assessment, *Empirical Software Engineering*, Vol. 5, No. 3, pp. 210–214, 2000.
IEEE (1990) *Standard Glossary of Software Engineering Technology*, IEEE, New York, 1990.
Jarvinen, P. (1999), On Research Methods, ISBN 951-97113-6-8, Opinpaja Oy, Tampere, Finland.
Jensen, C.S. and Dyreson, C.E. (1998), The Consensus Glossary of Temporal Database Concept—February 1998 Version, In *Temporal Databases: Research and Practice (LNCS 1399)*, Springer-Verlag, Berlin.
Johnson, R.E. and Opdyke, W.F. (1993), Refactoring and Aggregation, In *International Symposium on Object Technologies for Advanced Software (ISOTAS '93),* Springer-Verlag, New York, pp. 264–278.
Jones, S. and Scaife, M. (2000), Animated Diagrams: An Investigation into the Cognitive Effects of Using Animation to Illustrate Dynamic Processes, In *Diagram 2000 ( LNAI Vol. 1889),* Springer-Verlag, Berlin, Edinburg, pp. 231–244.
Jørgensen, M. (1995a), An Empirical Study of Software Maintenance Tasks, *Software Maintenance: Research and Practice*, Vol. 7, pp. 27–48, 1995.
Jørgensen, M. (1995b), The Quality of Questionnaire Based Software Maintenance Studies, *ACM SIGSOFT — Software Engineering Notes*, Vol. 20, No. 1, pp. 71–73, 1995.
Jørgensen, M. and Sjøberg, D.I.K. (2000), Empirical Studies on Effort Estimation in Software Development Projects, In *IRMA 2000,* Alaska, pp. 778–779.
Juristo, N., Windl, H. and Constantine, L. (2001), Introducing Usability, *IEEE Software, Janaury/February 2001*, pp. 20–21, 2001.
Karahasanović, A. (2000), SEMT — A Tool for Finding Impacts of Schema Changes, In *NWPER'2000 Nordic Workshop on Programming Environment Research,* Lillehammer, Norway, pp. 60–75.
Karahasanović, A. and Sjøberg, D. (1999), Supporting Database Schema Evolution by Impact Analysis, In *Norwegian Conference in Informatics,* TAPIR, Trondheim, Norway, pp. 303–314.
Karahasanović, A. and Sjøberg, D. (2001), Visualising Impacts of Database Schema Changes — A Controlled Experiment, In *2001 IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering,* IEEE Computer Society, Stresa, Italy, pp. 358–365.

Karahasanović, A., Sjøberg, D. and Jørgensen, M. (2001), Data Collection in Software Engineering Experiments, In *Information Resources Management Association International Conference, Soft. Eng. Track,* Idea Group Publishing, Toronto, Ontario, Canada, pp. 1027–1028.

Karahasanović, A. (2001), Identifying Impacts of Database Schema Changes on Applications, In *CAiSE'01, 8th Doctoral Consortium on Advanced Information Systems Engineering*, Interlaken, Switzerland, pp. 93–104.

Karat, J., Jeffries, R., Miller, J., Lund, A.M., McClelland, I., John, B.E., Monk, A.F., Oviatt, S.L., Carroll, J.M., Mackay, W.E. and Newman, W.N. (1998), Commentaries on "Damaged merchandise?", *Human-Computer Interaction*, Vol. 13, No. 3, pp. 263–323, 1998.

Kay, J. and Thomas, R.C. (1995), Studying Long-Term System Use, *Communications of the ACM*, Vol. 38, No.7, pp. 61–69, 1995.

Kelley, W., Gala, S., Kim, W., Reyes, T. and Graham, B. (1995), Schema Architecture of the UniSQL/M Multidatabase System, In *Modern Database Systems, The Object Model, Interoperability and Beyond*, ACM Press, New York, pp. 621–648.

Kim, W. and Chou, H.-T. (1988), Versions of Schema for Object-Oriented Databases, In *VLDB'88,* Los Angeles, California, pp. 148–159.

Kirton, M. (1994) *Adapters and Innovators*, NY. Routledge, New York, 1994.

Kitchenham, B., Linkman, S. and Law, D. (1997), DESMET: A Methodology for Evaluating Software Engineering Methods and Tools, *Computing & Control Engineering Journal*, Vol. 8, No. 3, pp. 120–126, 1997.

Kitchenham, B.A. (1996), Evaluating Software Engineering Methods and Tools, In *SIGSoft Software Eng. Notes,* pp. 11–15.

Kitchenham, B.A. (1997), Evaluating Software Engineering Methods and Tools, Part 7 Planning Feature Analysis Evaluation, *ACM SIGSOFT Software Engineering Notes*, Vol. 22, No 4, pp. 21–24, 1997.

Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El-Emam, K. and Rosenberg, J. (2001), Preliminary Guidelines for Empirical Research in Software Engineering, accepted for publication in IEEE Trans. on Soft. Eng. 2001.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y. and Chen, C. (1994), Change Impact Identification in Object Oriented Software Maintenance, In *Int. Conf. on Software Maintenance,* IEEE Comp. Soc. Press, Los Alamitos, CA, USA, pp. 202–221.

Lanza, M. (2001), The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques, In *Int. Workshop on Principles of Software Evolution — IWPSE 2001,* Vienna, Austria, pp. 28–33.

Larkin, J. and Simon, H. (1987), Why a diagram is (sometimes) worth ten thousand words, *Cognitive Science*, Vol. 11, pp. 69–100, 1987.

Lautemann, S.E. (1996), An Introduction to Schema Versioning in OODBMS, In *7th Int. Workshop on Database and Expert Systems Applications,* IEEE Comp. Soc. Press, Los Alamitos, CA, USA.

Lautemann, S.E. (1997), A Propagation Mechanism for Populated Schema Versions, In *13th International Conference on Data Engineering,* IEEE Comp. Soc. Press, Los Alamitos, CA.

Lautemann, S.E., Eigner, P. and Wohrle, C. (1997), The COAST Project: Design and Implementation, In *5th Int. Conf. on Deductive and Object Oriented Databases DOOD '97,* Springer Verlag, Berlin, Germany, pp. 229–245.

Lee, A.S. (1989), A Scientific Methodology for MIS Case Studies, *MIS quarterly*, Vol. 13, No. 1, pp. 33–50, 1989.

Lee, D. (1998), JavaCC Grammar Repository, http://www.cobase.cs.ucla.edu/pub/javacc/.

Lehman, M. and Belady, L.A. (1985) *Program Evolution — Processes of Software Change*, Academic Press, London, 1985.

Lehman, M.M. (1974), Programs, Cities, Students — Limits to Growth, *Imp. Col. Inaug. Lect. Series*, Vol. 9, 1970–1974, pp. 211–229, 1974.
Lehman, M.M. and Ramil, J.F. (2001), An Approach to a Theory of Software Evolution, In *International Workshop on Principles of Software Evolution — IWPSE 2001,* Vienna, Austria, pp. 62–65.
Lehman, M.M.e.a. (2000), Evolution as a Noun and Evolution as a Verb, In *SOCE 2000 Workshop on Software and Organisation Co-Evolution,* Imperial College, London.
Lerner, B.S. (2000), A Model for Compound Type Changes Encountered in Schema Evolution, *ACM Transaction on Database Systems*, Vol. 25, No. 1, March 2000, pp. 83–127, 2000.
Lerner, B.S. and Habermann, A.N. (1990), Beyond Schema Evolution to Database Reorganisation, In *Joint ACM European Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA/ECOOP 90),* ACM Press, Ottawa, Canada, pp. 67–76.
Leverenz, L. and Rehfield, D. (1999) *Oracle 8i Concepts, Release 8.1.5 A67781-01*, Oracle Corporation, 1999.
Lewis, J.R. (1995), IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instructions for Use, *International Journal of Human-Computer Interaction*, Vol. 7, No. 1, pp. 57–78, 1995.
Li, L. and Offutt, A.J. (1996), Algorithmic analysis of the impact of changes to object-oriented software, In *International Conference on Software Maintenance,* IEEE Comp. Soc. Press, Los Alamitos, CA, USA.
Li, X. (1999), A Survey of Schema Evolution in Object-Oriented Databases, In *Technology of Object-Oriented Languages and Systems,* IEEE Comp. Soc. Press, Los Alamitos, CA, USA.
Lientz, B.P. (1983), Issues in Software Maintenance, *Computing Surveys*, Vol. 15, pp. 271–278, 1983.
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, 1978.
Lindvall, M. (1997), Evaluating Impact Analysis — A Case Study, *Empirical Software Engineering*, Vol. 2, No. 2, pp. 152–158, 1997.
Liu, L. (1997), Maintaining Database Consistency in the Presence of Schema Evolution, In *IFIP WG 2.6 Working Conference on Database Applications Semantics (DS-6),* Chapman & Hall, London, UK, pp. 549–571.
Metamata (2000), JavaCC — The Java Parser Generator, http://www.webgain.com/products/java_cc/, Sun Microsystems Metamata.
Minitab (1998), Monitab for Windows Version 12, Minitab Inc.
Mittermeier, R. (2001), Software Evolution, Let's sharpen the terminology before sharpening (out-of-scope) tools, In *International Workshop on Principles of Software Evolution — IWPSE 2001,* Vienna, Austria, pp. 108–138.
Monk, S. (1993), A Model for Schema Evolution in Object-Oriented Database Systems, PhD thesis, University of Lancaster.
Monk, S. and Sommerville, I. (1993), Schema Evolution in OODBS Using Class Versioning, *SIGMOD Record*, Vol. 22, No. 3, pp. 16–22, 1993.
Monk, S. and Sommerville, R. (1992), A Model for Versioning of Classes in Object-Oriented Database, In *10th British National Conference on Databases (LNCS 618),* Aberdeen, Scotland, pp. 42–58.
Morse, E., Lewis, M. and Olsen, K.A. (2000), Evaluating visualisations: using a taxonomy guide, *Int. Journal of Human-Computer Studies*, Vol. 53, No. 5, pp. 637–662, 2000.
Nakatani, T. (2001), Quantitative Observations on Object Evolution, In *International Workshop on Principles of Software Evolution — IWPSE 2001,* Vienna, Austria, pp. 147–150.
Navigator, S. (2001), http://sources.redhat.com/sourcenav/.

Nguyen, G.T. and Rieu, D. (1989), Schema Evolution in Object-Oriented Database Systems, *Data & Knowledge Engineering*, Vol. 4, pp. 43–67, 1989.
Nielsen, J. (1993) *Usability Engineering*, AP Professional, 1993.
Nosek, J.T. and Prashant, P. (1990), Software Maintenance Management: Change in the Last Decade, *Journal of Software Maintenance Research and Practice*, Vol. 2, No. 3, pp. 157–174, 1990.
Nuseibeh, B., Easterbrook, S. and Russo, A. (2000), Leveraging Inconsistency in Software Development, *IEEE Computer*, pp. 24–28, 2000.
Nuseibeh, B., Kramer, J. and Finkelstein, A.C.W. (1994), A Framework for Expressing the Relationships between Multiple Views in Requirements Specification, *IEEE Trans. Software Eng.*, Vol. 20, No.10, pp. 760–773, 1994.
Nuseibeh, B.A. (1997), Ariane 5: Who Dunnit?, *IEEE Software*, Vol. 14, No. 2, pp. 15–16, 1997.
ObjectStore (2000), Writing Applications with ObjectStore PSE for C++, Object Design.
Odberg, E. (1994a) A Global Perspective of Schema Modification Management for Object-Oriented Databases, In *6th Conference on Advanced Information Systems Engineers CAISE '94 (LNCS 811)*, Springer-Verlag, pp. 406–420, 1994.
Odberg, E. (1994b) Category Classes: Flexible Classification and Evolution in Object-Oriented Databases, In *6th Int. Workshop on Persistent Object Systems (*POS*)*, Tarascon, Provence, France, September 1994.
Odberg, E. (1995), MultiPerspectives: Object Evolution and Schema Modification Management for Object-Oriented Databases, PhD thesis, The Norwegian Institute of Technology, University of Trondheim, Norway.
Olston, C., Stonebraker, M., Aiken, A. and Hellerstein, J.M. (1998), VIQING: Visual Interactive QueryING, In *1998 IEEE Symposium on Visual Languages,* IEEE Comp. Soc., Helifax, Nova Scotia, Canada, pp. 162–169.
Oracle (2001), JDeveloper General Documentation, release 3.2, http://otn.oracle.com/docs/products/jdev/doc_index.htm, Oracle Corporation.
Orlikowski, W.J. and Baroudi, J.J. (1991), Studying Information Technology in Organizations: Research Approaches and Assumptions, *Information Systems Research*, Vol. 2, pp. 1–28, 1991.
Oxford (1993) *The New Shorter Oxford English Dictionary on Historical Principles*, Oxford University Press Inc., New York, 1993.
Parnas, D.L. (1972), On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol. 15, No. 12, pp. 1053–1058, 1972.
Patton, M.Q. (1987) *How to Use Qualitative Methods in Evaluation*, Sage, Newbury Park, CA, 1987.
Penney, D.J. and Stein, J. (1987), Class Modification in the GemStone Object-Oriented DBMS, In *Conf. On Object-Oriented Systems, Languages and Application (OOPSLA) 1987,* pp. 111–117.
Pfleeger, S. (1994), Experimental Design and Analysis in Software Engineering Part 1–5, *ACM SIGSOFT, Software Engineering Notes*, Vol. 19, No 4, pp 16–20; Vol. 20, No. 1, pp. 22–26; Vol. 20, No. 2, pp 14–16; Vol. 20, No. 3, pp 13–15; and Vol. 20, No. 4, pp 14–17, 1994–1995, 1994.
Pfleeger, S.L. (1987) *Software Engineering — The Production of Quality Software*, Macmillan, 1987.
Piattini, M. and Martinez, A. (2000), Measuring for Database Programs Maintainability, In *Database and Expert Systems and Applications, DEXA 2000 (LNCS 1873),* Springer-Verlag, pp. 65–78.
Pomberger, G. and Blaschek, G. (1996) *Object-Orientation and Prototyping in Software Engineering*, Prentice Hall, 1996.

Pomberger, G.e.a. (1991), Prototyping-Oriented Software Development — Concepts and Tools, *Structured Programming*, Vol. 12, No. 1, 1991.
Popper, K. (1968) *The Logic of Scientific Discovery*, Harper Torchbooks, New York, 1968.
Preim, B. (1998), Exploration of Complex Information Spaces, In *Computational Visualisation, Graphics, Abstraction, and Interactivity*, Springer.
Purdom, P.J. (1970), A Transitive Closure Algorithm, *BIT*, Vol. 10, pp. 76–94, 1970.
Qadah, G.Z., Henschen, L.J. and Kim, J.J. (1991), Efficient algorithms for the instantiated transitive closure queries, *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, pp. 296–309, 1991.
Rashid, A. and Sawyer, P. (1999), Evaluation for Evolution: How Well Commercial Systems Do, In *1st ECOOP Workshop on Object-Oriented Databases,* LNCS 1743, Lisbon, Portugal.
Riccardi, G. (2001) *Principles of Database Systems with Internet and Java Applications*, Addison Wesley, Boston, 2001.
Ridgway, J.V.E. and Wileden, J.C. (1998), Toward Class Evolution in Persistent Java, In *Advances in Persistent Object Systems, Proc. of The Eighth International Workshop on Persistent Object Systems (POS-8) and The Third International Workshop on Persistence and Java (PJAVA-3),* Morgan Kaufmann Publishers, Tiburon, California, pp. 353–362.
Robinson, H., Hall, P., Hovenden, F. and Rachel, J. (1998), Postmodern Software Development, *The Computer Journal*, Vol. 41, No. 6, 1998.
Robson, C. (1993) *Real World Research: A Resource for Social Scientists and Practitioners-Researchers*, Blackwell, 1993.
Roddick, J.F. (1995), A Survey of Schema Versioning Issues for Database Systems, *Information and Software Technology*, Vol. 37, No. 7, pp. 383–393, 1995.
Roddick, J.F., Al-Jadir, L., Bertossi, L., Dumas, M., Estrella, F., Gregersen, H., Hornsby, K., Lufter, J., Mandreoli, F., Mannisto, T., Mayol, E. and Wedemeijer, L. (2000), Evolution and Change in Data Management — Issues and Directions, *SIGMOD Record*, Vol. 29, No. 1, March 2000, pp. 21-25, 2000.
Roddick, J.F., Craske, N.G. and Richards, T.J. (1993), A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models, In *12th International Conference on Entity-Relationship Approach,* Springer-Verlag, Dallas, Texas, pp. 143–154.
Salomon, G. (1993) *Distributed Cognition: Psychological and Educational Consideration*, Cambridge University Press, 1993.
Sankar, S. (1996), Java1.02 Grammar, http://www.cobase.cs.ucla.edu/pub/javacc/.
Sarkar, M. and Brown, M.H. (1992), Graphical Fisheye Views of Graphs, In *CHI'92 ACM Conference on Human Factors in Computing Systems,* ACM Press, New York, NY, USA, Monterey, pp. 83–91.
Schach, S. (1997) *Software Engineering with Java*, Times Mirror Higher Education Group, 1997.
Seaman, C. (1999), Qualitative Methods in Empirical Studies of Software Engineering, *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 557–572, 1999.
Serwer, A. (2000), The Next Richest Man in the World, *Fortune Magazine 142(11), November*, 2000.
Shneiderman, B. (1998) *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading, Mass., 1998.
Sieber, J.E. (2001), Protecting Research Subjects, Employees and Researchers: Implications for Software Engineering, *Empirical Software Engineering*, Vol. 6, pp. 329–341, 2001.
Sjøberg, D., Arisholm, E. and Jørgensen, M. (2001), Conducting Experiments on Software Evolution, In *International Workshop on Principles of Software Evolution — IWPSE 2001,* Vienna, Austria, pp. 135–138.
Sjøberg, D.I.K. (1993), Quantifying Schema Evolution, *Information and Software Technology*, Vol. 35, No. 1, pp. 35–44, 1993.

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

Smith, S.L. and Mosier, J.L. (1986), Guidelines for Designing User Interface Software, Electronic Systems Division, MITRE Corporation, Bedford, MA.

Snyder, L. (1994) *Academic Careers for Experimental Computer Scientists and Engineers*, National Academy Press, USA, 1994.

Southall, H. and White, B. (1997), Visualising Past Geographies: The use of animated cartograms to represent long-run demographic change in Britain, In *Graphics, Visualisation and the Social Science: Workshop Report: AGOCG Technical Report Series No. 33.*

Storey, M.A.D., Wong, K., Fong, P., Hooper, D., Hopkins, K. and Muller, H.A. (1996), On Designing an Experiment to Evaluate a Reverse Engineering Tool, In *3rd Working Conference on Reverse Engineering (WCRE'96),* IEEE Comp. Soc. Press, pp. 31–40.

Strothotte, T. (1988), New Challenges for Computer Visualization, In *Computational Visualization, Graphics, Abstraction and Interactivity*, Springer.

Tari, Z. and Li, X. (1994), Method restructuring and consistency checking for object-oriented schemas, *Entity Relationship Approach ER '94. Business Modelling and Re Engineering. 13th International Conference on the Entity Relationship Approach. Proceedings. Springer Verlag, Berlin, Germany*, 1994.

Tichy, W.F. (1998), Should Computer Scientists Experiment More, *IEEE Computer*, Vol. 31, No. 5, pp. 32–40, 1998.

Tichy, W.F. (2000), Hints for Reviewing Empirical Work in Software Engineering, *Empirical Software Engineering*, Vol. 5, No. 4, pp. 309–312, 2000.

Tichy, W.F., Lukowicz, P., Prechelt, L. and Heinz, E.A. (1995), Experimental Evaluation in Computer Science: A Quantitative Study, *The Journal of Systems and Software*, Vol. 28, pp. 9–18, 1995.

Torii, K., Matsumoto, K., Nakakoji, K., Takada, Y., Takada, S. and Shima, K. (1999), Ginger2: An Environment for Computer-Aided Empirical Software Engineering, *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 474–492, 1999.

Tresch, M. (1991), A Framework for Schema Evolution by Meta Object Manipulation, In *3rd Int. Workshop on Foundations of Models and Languages for Data and Objects,* Aigen, Austria.

Tresch, M. and Scholl, M.H. (1992), Meta Object Management and its Application to Database Evolution, In *11th Int. Conf. on the Entity-Relationship Approach (LNCS 1000),* Springer, Karlsruhe, Germany, pp. 299–321.

Tresch, M. and Scholl, M.H. (1993), Schema Transformation without Database Reorganisation, *SIGMOD Record*, Vol. 22, No. 1, pp. 21–27, 1993.

Tversky, B., Zacks, J., Lee, P. and Heiser, J. (2000), Lines, Blobs, Crosses and Arrows: Diagrammatic Communications with Schematic Figures, In *Diagrams 2000 (LNAI Vol. 1889),* Springer-Verlag, Berlin, Edinburg, pp. 221–230.

Veerasamy, A. and Belkin, N.J. (1996), Evaluation of a tool for visualisation of information retrieval results, In *Annual ACM Conference on Research and Development in Information Retrieval,* Zurich, Switzerland, pp. 85–92.

Viswanadha, S. (1997), C++ grammar specification, http://www.cobase.cs.ucla.edu/pub/javacc/CPLUSPLUS.jj, Sun Microsystems Inc.

von Mayrhauser, A. and Lang, S. (1999), A Coding Scheme to Support Systematic Analysis of Software Comprehension, *IEEE Transactions on Software Engineering*, Vol. 25, pp. 526–540, 1999.

Warshall (1962), A Theorem on Boolean Matrices, *Journal of ACM*, Vol. 9, No. 1, pp. 11–12, 1962.

Weiser, M. (1984), Program Slicing, *IEEE Trans. Software Eng.*, Vol. 10, pp. 352–357, 1984.

Welland, R., Sjøberg, D. and Atkinson, M. (1997), Empirical Analysis based on Automatic Tool Logging, In *Empirical Assessment & Evaluation in Software Engineering (EASE97),* Keele, UK.

Werner, M. (1998), daVinci V2.1.x Online Documentation, Universitat Bremen.

Whitley, K.N. (1997), Visual Programming Languages and the Empirical Evidence For and Against, *Journal of Visual Languages and Computing*, Vol. 8, No. 1, pp. 109–142, 1997.

Wilde, N. and Huitt, R. (1992), Maintenance Support for Object-Oriented Programs, *IEEE Trans. Software Eng.*, Vol. 18, No.12, pp. 1038–1044, 1992.

Wirth, N. (1971), Program Development by Step-Wise Refinement, *Comm. of ACM*, Vol. 14, No. 4, pp. 221–227, 1971.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., C., Regnell, B. and Wesslén, A. (1999) *Experimentation in Software Engineering — An Introduction*, Kluwer Academic Publishers, Boston, 1999.

Wright, R.B. and Converse, S.A. (1992), Method bias and concurrent verbal protocol in software usability testing, In *Proc. Human Factors Society 36th Annual Meeting,* Atlanta, Georgia, pp. 1220–1224.

Yin, R.K. (1994) *Case Study Research*, SAGE Publications, Thousand Oaks, California, 1994.

Young-Gook, R. and Rundensteiner, E.A. (1997), A Transparent Schema Evolution System Based on Object-Oriented View Technology, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 4, pp. 600–624, 1997.

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

Zelkowitz, M.V. and Wallace, D.R. (1998), Experimental Models for Validating Technology, *IEEE Computer*, Vol. 31, No. 5, pp. 23–31, 1998.

Zicari, R. and Ferrandina, F. (1997), Schema and Database Evolution in Object Database Systems, In *Advanced Database Systems*, Morgan Kaufmann Publishers, inc., San Francisco, California.

Zikari, R. (1991), A Framework for Schema Updates in an Object-Oriented Database System, In *7th IEEE Int. Conf. on Data Engineering,* Kobe, Japan.