Empirical Assessment of Changeability in Object-Oriented Software

Erik Arisholm

Thesis submitted for the degree of Dr. Scient.

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

February 22, 2001

Abstract

The combination of evolutionary development processes and object-oriented methods may be a new "silver bullet" for the software community. Evolutionary development may result in poor structure and outdated documentation, which in turn may have a negative impact on the changeability of the software. However, there is surprisingly little scientific evidence to support or refute claims regarding evolutionary development processes in general, and consequences regarding the changeability of resulting object-oriented software in particular. A prerequisite for obtaining a better scientific knowledge regarding the consequences of evolutionary development on changeability is to assess the changeability of the developed software in an accurate way.

The research presented in this thesis proposes and validates a measurement framework for assessing the changeability of object-oriented software. First, changeability is defined in a concise manner. Then, three alternative approaches to measuring changeability are identified: *Structural Attribute Measurement (SAM), Change Profile Measurement (CPM)* and *Benchmarking.* Methods for collection and analysis of change data and for empirical validation of the measurement framework are also described.

The *SAM* approach quantifies, amongst others, the static and dynamic coupling between classes and the size of the classes. The *CPM* approach combines these structural attribute measures with measures of the actual changes on the software. The SAM approach can be used to quantify structural properties at the system level, whereas the CPM approach can be used to assess how changes actually propagate through the software structure. Based on the SAM and CPM measures, models predicting the difference in change effort are built. These models indicate the extent to which the structural properties of object-oriented designs affect changeability. As an alternative approach, *benchmarking* can be used to determine the total effort to implement a given collection of benchmark change tasks on different versions or alternative designs of a software system. In addition to the impact of deteriorating structure, other aspects (e.g., inconsistent documentation) may be reflected in the benchmarking results.

Several empirical studies have been performed to validate the changeability measurement framework. Most of the research has been undertaken in an industrial context, demonstrating the practical use of the framework. Potential *causes* of changeability decay in evolutionary development of object-oriented software are also identified. This research establishes the measurement framework as a viable foundation for future changeability assessment studies related to evolutionary development of object-oriented software.

Acknowledgements

First of all I am indebted to my supervisor Dag Sjøberg for his continuous support, contributions, guidance and encouragement. He has made this important period of my life a fun and challenging learning experience.

I am grateful for the many useful comments, interesting discussions and valuable research collaborations with Magne Jørgensen, Bente Anda, Lars Bratthall, and Amela Karahasanovic at the ISU group. I thank my M.Sc. students, Anette Cecilie Lien and Lars Hiim. Anette Lien contributed substantially to the design and analysis of the TelMont interview. Lars Hiim implemented the Java coupling parser. I also thank the students who participated in the coffee-machine experiment.

This thesis would not have been possible without the cooperation from several companies in Norway. I gratefully acknowledge the support from Erik Amundrud, Jon Skandsen, Knut Sagli and Stein Grimstad at Genera AS. Genera supported me financially and gave me access to data in the Wings and Genova development projects. Stein Grimstad implemented the Visual Basic coupling parser. I am also indebted to Lasse Bjerde and Anne-Lise Skaar at Numerica-Taskon AS (now Mogul). Anne-Lise Skaar implemented the dynamic coupling parser for SmallTalk. In addition, representatives from EDB 4Tel, Telenor Nett, Icon Medialab and the SPIQ project have contributed to the research presented in this thesis.

Furthermore, Lionel Briand, Letizia Jaccheri, Barbara Kitchenham, Harvey Siy, Ray Welland, the ESSDE'99 workshop participants and anonymous reviewers gave useful comments and influenced the work in several ways.

The research presented in this thesis was funded by a PhD fellowship from the Research Council of Norway through the industry-projects SPIQ (Software Process Improvement for better Quality) and PROFIT (PROcess improvement For the IT industry).

Last but not least, I thank my friends and family for their support and encouragement. A special thanks goes to my wife Farnaz for supporting me in every possible way to finalize this work.

Table of Contents

1 INTRODUCTION 1

1.1 THE PROBLEM OF CHANGE 1

1.2 CHANGEABILITY IN EVOLUTIONARY DEVELOPMENT 2

1.3 GOALS 3

1.4 CONTRIBUTION 3

1.5 THESIS ORGANIZATION 4

2 CHANGEABILITY 6

2.1 SOFTWARE PRODUCT QUALITY 6
2.1.1 ISO Quality Model 6

2.2 WHAT IS CHANGEABILITY? 7
2.2.1 Relationship between Productivity and Changeability 9
2.2.2 Relationship between Maintainability and Changeability 9
2.2.3 Relationship between Code Decay and Changeability Decay 10
2.2.4 Relationship to the ISO quality model and Change Impact 10

3 RESEARCH METHODS IN EMPIRICAL SOFTWARE ENGINEERING. 12

3.1 SOFTWARE ENGINEERING AS A SCIENCE 12
3.1.1 Requirements to a Scientific Approach 12
3.1.2 Scientific Validity 13
3.1.3 State of Practice 13

3.2 RESEARCH METHODS IN EMPIRICAL SOFTWARE ENGINEERING 14
3.2.1 Surveys – Research in the Large 14
3.2.1.1 Guidelines for Conducting Surveys 16
3.2.2 Experiments – Research in the Small 16
3.2.2.1 Guidelines for Conducting Experiments 17
3.2.3 Case Studies – Research in the Typical 18
3.2.3.1 Guidelines for Conducting Case Studies 19
3.2.4 Qualitative versus Quantitative Methods 20

3.3 SUMMARY 20

4 EVOLUTIONARY DEVELOPMENT 22

4.1 SOFTWARE PROCESS MODELS 22
4.1.1 Linear-Sequential Development 23
4.1.2 Incremental Development 24
4.1.3 Evolutionary Development 25
4.1.3.1 Evolutionary Prototyping 25
4.1.3.2 Evolutionary, Incremental Development 25

4.2 CONSEQUENCES FOR CHANGEABILITY..27
4.2.1 Sommerville..*27*
4.2.2 Lehman et al....*27*
4.2.3 Parnas..*27*
4.2.4 Boehm et al....*28*
4.2.5 Royce...*29*
4.2.6 Brownsword et al....*29*
4.2.7 May and Zimmer...*30*
4.2.8 Zamperoni et al...*30*
4.2.9 Ehn...*31*
4.2.10 Emam et al...*31*
4.2.11 Lichter et al....*32*
4.3 SUMMARY..32
5 EMPIRICAL STUDIES OF OBJECT-ORIENTED SOFTWARE 34
5.1 OVERVIEW OF EXISTING EMPIRICAL STUDIES..34
5.1.1 Measurement Validation Principles..*35*
5.1.1.1 Structural Attribute Measures for Object-Oriented Software..........35
5.1.2 Empirical Assessment of Object-Oriented Technologies....................*35*
5.1.3 Quality and Productivity Models for Object-Oriented Software.........*36*
5.1.3.1 Fault Proneness 36
5.1.3.2 Change Impact...37
5.1.3.3 Effort 37
5.2 SUCCESS FACTORS FOR EMPIRICAL STUDIES OF OBJECT-ORIENTATION......38
5.2.1 Nature of the Data...*38*
5.2.2 Consistent Terminology...*39*
5.2.3 Nature of the Research..*39*
5.3 RESEARCH DIRECTIONS 39
5.3.1 Objectives..*39*
5.3.2 Research Questions...*40*
5.3.2.1 Identify Important Factors...40
5.3.2.2 Evaluation of Object-Oriented Technologies.................................41
5.3.2.3 Building Quality and Productivity Models.....................................41
5.3.2.4 Meta-level issues 42
5.4 SUMMARY..42
6 MEASURING CHANGEABILITY 44
6.1 OVERVIEW OF THE MEASUREMENT APPROACHES..44
6.2 STRUCTURAL ATTRIBUTE MEASUREMENT (SAM).......................................45
6.2.1 Selection of Structural Attributes..*46*
6.2.2 Specification of Static Coupling Measures..*47*
6.2.3 Specification of Dynamic Coupling Measures....................................*48*
6.2.3.1 Direction of Coupling: Import and Export Coupling 48

6.2.3.2 Mapping: Object-level and Class-level Coupling 48
6.2.3.3 Strength of Coupling 49
6.2.3.4 Resulting Coupling Measures 50

6.3 CHANGE PROFILE MEASUREMENT (CPM) 51

6.4 BENCHMARKING 53
6.4.1.1 Design of a Benchmarking Procedure 53
6.4.1.2 Composition of Benchmarks 54

6.5 RELATIONSHIPS BETWEEN CHANGEABILITY MEASURES 56
6.5.1 Accuracy of the Measurement Approaches 56
6.5.2 Cost of the Measurement Approaches 56
6.5.3 Practical Use 56

6.6 VALIDATION ISSUES 57
6.6.1 Evaluation of Practical Issues 57
6.6.2 Building Prediction Models Using SAM and CPM 57
6.6.2.1 Stepwise Multiple Linear Regression 59
6.6.2.2 Principal Component Analysis 59
6.6.2.3 Cross-Validation 60
6.6.2.4 Checking the Model Assumptions 60
6.6.3 Validation of the Accuracy of Benchmarking 60

6.7 SUMMARY AND RELATED WORK 61
6.7.1 Object-Oriented Metrics 61
6.7.2 SAAM 61
6.7.3 COMPARE 61
6.7.4 TAC++ 62

7 EMPIRICAL STUDIES OF CHANGEABILITY 63

7.1 CHANGEABILITY IN EVOLUTIONARY DEVELOPMENT PROJECTS 64
7.1.1 The Braathens Case Study 64
7.1.1.1 System under Study 64
7.1.1.2 Data Collection 65
7.1.1.3 Evaluation of Evolutionary Development 65
7.1.1.4 Validation of SAM and CPM 67
7.1.2 The Genera Case Study 71
7.1.2.1 System under Study 71
7.1.2.2 Data Collection 71
7.1.2.3 Evaluation of Evolutionary Development 74
7.1.2.4 Validation of SAM and CPM 76
7.1.3 Summary 83
7.1.3.1 Using the Change Log 83
7.1.3.2 Evolutionary Development 83
7.1.3.3 Validation of SAM and CPM 84

7.2 ASSESSING THE CHANGEABILITY OF TWO OBJECT-ORIENTED DESIGN ALTERNATIVES – A CONTROLLED EXPERIMENT 86

7.2.1 Design of the Study *86*
7.2.1.1 Treatments: The Coffee-Machine Design Problem 87
7.2.1.2 Description of the Design Alternatives 87
7.2.1.3 The Mocca Programming Language 90
7.2.1.4 The Programming Tasks 91
7.2.1.5 The Change Task Questionnaire 92
7.2.1.6 Experimental Design 92
7.2.1.7 Design of the Pilot Experiment 93
7.2.1.8 Design of the Main Experiment 93
7.2.1.9 Dependent Variables 93
7.2.2 Results of the Pilot Experiment *96*
7.2.2.1 Evaluation of Blocking Strategies 96
7.2.2.2 Preliminary Assessment of Change Effort for MF and RD 97
7.2.3 Results of the Main Experiment *98*
7.2.3.1 Formal Hypotheses 98
7.2.3.2 Change Effort (H1) 99
7.2.3.3 Learning Curve (H2) 100
7.2.3.4 Correctness (H3) 101
7.2.3.5 Subjective Change Complexity (H4) 101
7.2.3.6 Structural Stability (H5) 101
7.2.3.7 Attempting to Explain the Results 102
7.2.4 Summary of Results *104*
7.2.4.1 Comparing the Results with Related Research 105
7.2.5 Threats to Validity *106*
7.2.5.1 Experimental Materials 107
7.2.5.2 Size and Choice of Design Alternatives and Change Tasks 107
7.2.5.3 Pen and Paper 108
7.2.5.4 Subject Selection 108
7.2.5.5 Group Assignment 108
7.2.6 Future Work *110*

7.3 DEFINITION AND EVALUATION OF DYNAMIC COUPLING 111
7.3.1 The Case Study *111*
7.3.1.1 Collection of the Change Data 111
7.3.1.2 Collection of the Coupling Measures 112
7.3.1.3 Descriptive Statistics of the Coupling Measures 113
7.3.1.4 Principal Component Analysis 114
7.3.2 Assessing Changeability with the Change Profile Measures *115*
7.3.3 Change Proneness *117*
7.3.3.1 Hypotheses and Statistical Analysis 117
7.3.3.2 Results 118
7.3.4 Using Dynamic Coupling for Impact Analysis *120*
7.3.4.1 Collection of the Measures 120
7.3.4.2 Identification of Prediction Models for Common Changes 121
7.3.4.3 Model Evaluation – Prediction of Common Changes 123
7.3.5 Summary and Future Work *124*
7.3.5.1 Comparing Dynamic Coupling with Static Coupling 125
7.3.5.2 Within-Object Coupling 125

7.3.5.3 Using Dynamic Coupling Measures to Support Impact Analysis .125
7.3.5.4 Using Dynamic Coupling Measures to Assess Understandability 126
7.3.5.5 Data Collection Algorithms...127
7.3.5.6 Implementation Issues 128

7.4 CAUSES OF INCREASED CHANGE EFFORT AND PROJECT DELAYS 129
7.4.1 Design of the Study...129
7.4.2 Results 129
7.4.3 Summary of Results 133
7.4.3.1 Threats to Validity..133

7.5 EVALUATION OF AN EVOLUTIONARY DEVELOPMENT PROJECT 134
7.5.1 Design of the Study...134
7.5.1.1 Subject Selection 134
7.5.1.2 Interview Technique..134
7.5.1.3 Transcription, Data Analysis and Unification 135
7.5.1.4 Quality Assurance 135
7.5.2 The TelMont Project..135
7.5.2.1 Project Activities and Milestones...136
7.5.2.2 Customer/Software Vendor Relationship.....................................137
7.5.2.3 Estimation 137
7.5.2.4 Process 137
7.5.2.5 User Participation..138
7.5.2.6 Prototyping..138
7.5.2.7 Requirements Specification...139
7.5.2.8 UML and Rational Rose..139
7.5.2.9 Lessons Learned 139
7.5.3 Summary..140

8 CONCLUSIONS AND FUTURE WORK 142

8.1 SUMMARY OF RESULTS...142
8.1.1 Definition of Changeability...142
8.1.2 Empirical Validation of the Measurement Framework.....................143
8.1.3 Changeability in Evolutionary Development 144

8.2 FUTURE WORK..145
8.2.1 Improvements of the Measurement Framework 145
8.2.1.1 Industrial Evaluation 146
8.2.1.2 Building Generic Benchmarks 146
8.2.1.3 Combining CPM with Scenario Elicitation 146
8.2.1.4 Dynamic Coupling 147
8.2.2 Evolutionary Development Processes 147

8.3 CONCLUDING REMARKS...148

APPENDIX A: RAW DATA FOR THE GENERA CASE STUDY 149

APPENDIX B: THE COFFEE-MACHINE EXPERIMENT.............................151

B.1 EXPERIENCE LEVEL QUESTIONNAIRE 151
B.2 CHANGE TASKS 152
B.3 CHANGE TASK QUESTIONNAIRE 155
B.4 MESSAGE SEQUENCE CHARTS FOR THE DESIGNS 156
B.5 CODE FRAGMENTS FROM THE DESIGNS 158
B.6 RAW DATA FROM THE MAIN EXPERIMENT 160
APPENDIX C: RAW DATA FROM THE OORAM CASE STUDY 165
BIBLIOGRAPHY 166

1 Introduction

1.1 The problem of Change

Handling *change* is a fundamental research topic within software engineering. The following problem statement from an influential early research paper illustrates the topic:

> *This paper is written because the following complaints about software systems are so common.*
>
> *1) "We were behind schedule and wanted to deliver an early release with only a proper subset of the intended capabilities, but found that the subset would not work until everything worked."*
> *2) "We wanted to add a simple capability, but to do so would have meant rewriting all or most of the current code."*
> *3) "We wanted to simplify and speed up the system by removing the unneeded capability, but to take advantage of this simplification we would have had to rewrite major sections of the code."*
> *4) "Our SYSGEN was intended to allow us to tailor a system to our customers' needs but it was not flexible enough to suit us."*
>
> *After studying a number of such systems, I have identified some simple concepts that can help programmers to design software so that subsets and extensions are more easily obtained…*
>
> *(Parnas, 1979)*

Software systems must be changed as a result of, for example, misunderstood, new or changing user requirements, changing laws or regulations, errors in design and code, adaptations to new hardware and operating systems, and adaptations to other software systems (Lientz *et al.*, 1978). Consequently, the computer science and software engineering communities are continuously attempting to provide better solutions for handling change.

The last decade has seen an explosive growth in the number of software engineering methods and tools, each one offering to improve some characteristics of software, its development, or its maintenance (Kitchenham *et al.*, 1995a). The proposed solutions may be divided into the following categories:

- *Improved programming languages*, such as object-oriented programming languages (e.g., SmallTalk, C++, Java).
- *Improved CASE tools*, such as configuration management systems, design modeling tools, test tools, automatic code generators and tools supporting the automatic restructuring of code.
- *Improved design techniques and principles,* such as those proposed in (Parnas, 1979; Coad and Yourdon, 1991a; Coad and Yourdon, 1991b; Booch, 1994).
- *Improved development processes*, such as the incorporation of prototyping (Floyd, 1984) and evolutionary, incremental development processes (Boehm,

1988; Gilb, 1988; Cotton, 1996; Kruchten and Royce, 1996; May and Zimmer, 1996).

For example, object-oriented programming languages may improve productivity through code reuse (Basili *et al.*, 1996a) and make it simpler to understand and change software. CASE tools may help locate, design, code, test, deploy and track changes. Proper design techniques or principles may reduce the cost of both anticipated and unanticipated changes by providing an extensible design that reduces the (negative) impact of changes. Evolutionary development may help discover the "real" requirements of a software system early, reducing the size of the system and the number of unanticipated and often costly changes later in the software lifecycle.

1.2 Changeability in Evolutionary Development

Evolutionary development has been proposed as an efficient way to deal with risks regarding new technology and imprecise or changing requirements (Boehm, 1988). The main idea is to resolve risks early by incrementally evolving the system towards completion instead of relying on the traditional "big-bang" waterfall (Royce, 1970) approach.

While some experience reports show a great deal of success in the application of evolutionary development (Gilb, 1988; Zamperoni *et al.*, 1995), the continuous incremental changes supported by evolutionary development are believed to result in poor structure (Boehm *et al.*, 1984; Boehm, 1988; Sommerville, 1996). For this reason, the design and maintenance of an "open-ended architecture" may be critical for the success of evolutionary software engineering processes such as Gilb's EVO (Gilb, 1988), HP Evolutionary Fusion (Cotton, 1996), Dynamic Systems Development Method (DSDM) and Rational Unified Process (Jacobson *et al.*, 1999). This is also exemplified by a rather rhetoric exercise given in Sommerville's software engineering textbook:

> Explain why programs which are developed using evolutionary development are likely to be difficult to maintain.
>
> (Sommerville, 2001)

In our experience, the frequent changes may also lead to inconsistent and outdated requirement specifications and design documentation. This and similar statements or claims are given by many researchers and practitioners in the software industry. However, in most cases, the claims have a very limited scientific foundation. Software engineering needs science to observe, gain general experience, establish knowledge, and create and validate theories and models. Such scientific investigation is essential to make software engineering a science rather than an art (Fenton *et al.*, 1994). Until recently, the most commonly used form of validation of new software engineering technology has been *advocacy research* (Glass, 1994), i.e., the validation has been based on intuition, subjective experience and opinions of "gurus" instead of carefully designed scientific validation studies.

1.3 Goals

The discussion above motivates this thesis. Does evolutionary development cause structural degradations to code, or does it actually improve the design? To get in a position where this question can be answered, several issues need to be resolved:

- What do we mean by "structural degradation" and "design improvements"?
- How can such "degradations" or "improvements" be measured?
- How can we determine whether one design is "better" or more "open-ended" than another?

In an attempt to answer these questions, this thesis focuses on the assessment of one quality aspect of software, namely *changeability*, reflecting the effort required to incorporate new or changed functionality into a software system. Thus, the overall goal of this thesis is

- to define changeability in a concise manner,
- to develop a measurement framework for assessing changeability, and
- to identify factors affecting changeability in evolutionary development.

1.4 Contribution

The main contribution of this thesis is a proposed measurement framework for assessing changeability in object-oriented software. Three alternative approaches to measuring changeability are identified: (1) *Structural Attribute Measurement (SAM),* (2) *Change Profile Measurement (CPM)* and (3) *benchmarking.* Methods for the collection and analysis of change data and for the empirical validation of the measurement framework are also proposed.

The motivation for *structural attribute measurement* is that the (external) changeability attribute of a software system is very difficult to quantify directly in real-life development projects. This main reason for this difficulty is that a given change is implemented only once, and hence there is no real baseline allowing analysis of trends in change effort. Thus, it would be advantageous to identify *indicators* of changeability based on measures of the structural attributes of the software system. The changes in the measurement values of these attributes over time may then be used as indicators of changeability decay.

The proposed *change profile measurement* combines structural attribute measures with measures of the actual changes on the software. It quantifies structural properties of the parts of the system being changed instead of the overall system structure. We believe change profile measurement is a more accurate indicator of changeability than structure measurement, because, unlike structure measurement, it accounts for how changes actually propagate through the software structure.

As an alternative approach, we propose using *benchmarks* to measure change effort more directly. Benchmarking can be used to determine the total effort to implement a given collection of "benchmark changes" on different versions or alternative designs of a software system. Implementing the same changes on different versions of the software provides the necessary baseline to ensure that change efforts can be

compared. In addition to the impact of deteriorating structure, other aspects (e.g., inconsistent documentation and the incorporation of new technology) may be reflected in the benchmarking results. However, the utilization of benchmarks introduces new methodological challenges. A research methodology for the development and use of benchmarks and benchmarking procedures is therefore proposed.

The proposed measurement framework, the associated data collection methods, and the empirical validation techniques have been evaluated in several industrial development projects and controlled experiments. This research demonstrates that the proposed measurement framework has considerable potential for changeability assessment in evolutionary development of object-oriented software.

The second contribution of this thesis is the identification of factors potentially causing changeability decay in evolutionary development of object-oriented software. In several of the case studies described in this thesis, a considerable amount of unnecessary, costly and untimely rework is observed. This rework is caused by a lack of formal and incremental testing, too little up-front analysis and design based on initial requirements, and too little focus on early technology prototyping. The resulting rework may be a major contributor to changeability decay. Another problem identified in this thesis is that frequent changes may result in outdated documentation. Up-to-date documentation is important for understanding how to change the code (Tryggeseth, 1997). The use of modern CASE tools, such as Rational Rose, seems to be inadequate for keeping documentation consistent with code. This thesis illustrates how a reverse-engineering tool producing models of dynamic aspects of a system can be developed.

At present, only a very limited number of empirical studies of evolutionary development projects exist. We believe the results of the case studies described in this thesis extend the current state of knowledge regarding evolutionary development.

1.5 Thesis Organization

This thesis is organized in eight chapters. In addition, there are three appendixes providing details of data and statistical analysis from the presented empirical studies. Table 1.1 gives an overview of the chapters of this thesis.

Table 1.1. Overview of the thesis organization and content

Chapter	Content
Chapter 1	**Introduction** provides the motivation for the selected research and gives an overview of the chosen approach and the thesis.
Chapter 2	**Changeability** presents an overview of the concepts underlying the remainder of this thesis. What is changeability, and how does it relate to other software qualities?
Chapter 3	**Research Methods in Empirical Software Engineering** provides background information on the research methods used in empirical software engineering. It also discusses validity issues related to the different research methods.
Chapter 4	**Evolutionary Development** gives an overview of evolutionary development processes and terminology, and describes important existing empirical studies. The results from these empirical studies are used to describe how the changeability of software may be affected by evolutionary development.
Chapter 5	**Empirical Studies of Object-Oriented Software** gives an overview of the state-of-the-art of the empirical studies of object-oriented software that are relevant for measuring changeability.
Chapter 6	**Measuring Changeability** proposes a comprehensive measurement framework for quantitative assessment of changeability in object-oriented software. A methodology for data collection and empirical validation of the measures is described.
Chapter 7	**Empirical Studies of Changeability** describes the results of four case studies and one experiment. They are used to validate the measurement framework proposed in Chapter 6, and to evaluate issues related to the changeability of software in evolutionary development projects.
Chapter 8	**Conclusions and Future Work** provides a detailed summary of the results and research contributions. Several areas of future research are outlined.
Appendix A	This appendix describes raw data from the Genera case study (Section 7.1).
Appendix B	This appendix describes experimental materials and raw data for the coffee-machine experiment (Section 7.2).
Appendix C	This appendix describes raw data from the Ooram case study (Section 7.3).

2 Changeability

The main topic of this thesis is to study how object-oriented design principles and evolutionary development processes affect the changeability of software. Software engineering is concerned with the evaluation of the proposed solutions. A prerequisite for such a scientific evaluation is to define the concepts under study in a precise manner – to avoid confusions and enable accumulation of knowledge. Unfortunately, the concept of changeability is not fully understood to the extent that one can claim to know exactly how it should be defined and measured. The ambition of this chapter is to provide a description of the concept changeability that defines the scope of this thesis. The description provides the foundation for the measures and empirical studies outlined in later chapters.

2.1 Software Product Quality

There are many different views of what 'software quality' is. By referring to the work by Garvin (Garvin, 1984), Kitchenham points out that there are three important views of software quality (Kitchenham, 1996a):

- The *user view*, in which software quality is determined by the degree of meeting the needs of the user.
- The *manufacturing view*, focusing on producing the "right" product and minimizing of costs associated with rework during development and after delivery.
- The *product view*, considering the inherent characteristics of the software, such as structural properties of the software.

A similar distinction is done by Jørgensen, who points out that the most common types of software quality definitions are determined by (1) user satisfaction, (2) by the errors or unexpected behavior of the software, or (3) by a set of quality factors (Jørgensen, 1999).

2.1.1 ISO Quality Model

Many so-called quality models have been proposed (Kitchenham, 1996a). They attempt to provide a consistent and comprehensive characterization of what software quality *is*. One of the more recent quality models is the ISO 9126 model (ISO9126, 1992), which decomposes software quality into six quality characteristics:

- *Functionality*, consisting of the sub-*characteristics suitability, accuracy, interoperability* and *security*.
- *Reliability*, consisting of the sub-characteristics *maturity, fault tolerance* and *recoverability*.
- *Usability*, consisting of the sub-characteristics *understandability*, *learnability* and *operability*

- *Efficiency*, consisting of the sub-characteristics *time behavior and resource behavior*.
- *Maintainability*, consisting of the sub-characteristics *analyzability, changeability, stability* and *testability.*
- *Portability*, consisting of the sub-characteristics *adaptability, installability, conformance* and *replaceability*.

However, according to Kitchenham,

> *…the selection of quality characteristics and sub-characteristics still seems to be rather arbitrary; for example, it is not clear why portability is a top-level characteristic but interoperability is a sub-characteristic of functionality.*
>
> (Kitchenham, 1996a)

Likewise, it seems rather arbitrary that changeability is a sub-characteristic of maintainability whereas adaptability is a sub-characteristic of portability. Why is adaptability not a sub-characteristic of maintainability when changeability is? Furthermore, most of the characteristics have not been defined at an operational level.

2.2 What is Changeability?

From the previous section, it is clear that changeability is only one of many important quality characteristics of software. Furthermore, changeability is primarily concerned with the manufacturing and product view of software quality. The user view of software quality represented by characteristics such as usability, reliability and efficiency are not important aspects of changeability.[1] The changeability of a software system characterizes the ease of implementing changes to the system. Consequently, changeability overlaps with quality characteristics such as modifiability, adaptability, maintainability, extensibility, testability, program comprehensibility, and fault proneness. The ambition of this section is to provide a relatively precise description of the concept changeability as it is used in the remainder of this thesis, but not to provide a 'universally valid' definition.

According to Webster's Revised Unabridged Dictionary, *changeability is the quality of being capable of change*. From this definition, one may deduce two important consequences:

- Being capable of change implies that the task of changing the software requires little *effort*.
- Being capable of change implies a capability of avoiding a gradual decline from a sound or satisfactory state to an unsatisfactory state.

Thus, changeability may be viewed as a two-dimensional quality characteristic, related both to the *effort* to implement changes and to the resulting *quality* of the software after the changes. This view presents many issues that need clarification. The term *quality* designates all kinds of quality characteristics of software, such as

[1] However, the quality characteristics are not independent. For example, a potential tradeoff between usability and changeability is illustrated in the discussion of evolutionary development (Chapter 4).

reusability, consistency, reliability, usability, efficiency, portability, and changeability. Thus, *the decline from a satisfactory to an unsatisfactory state* implies a recursive definition of changeability, since changeability is also a quality characteristic that is prone to decline or decay! There is, apparently, no simple way to provide an operational definition of changeability. From a more pragmatic perspective, however, there is clearly an interesting tradeoff between the two identified dimensions of changeability. Which system has better changeability; a system that initially requires less effort to implement changes or a system that initially requires more change effort but that is less prone to decay? Based on empirical results, this is discussed further in Chapter 7.

One may also take the view that the changeability of software is ultimately reflected by the lifetime development effort to implement changes such as new functionality, changed functionality, adaptive changes, corrective changes and preventive restructuring. Thus, to determine whether one design alternative actually has better changeability than another design alternative, one would need to measure the lifetime effort to implement real changes to the software by the actual development team, based on both design alternatives. Such an evaluation is clearly not practical. In addition, this "lifetime effort" view of changeability may not be very useful when comparing design alternatives. During the lifetime of a software product, the design may *change* several times (e.g., as a result of restructuring). Thus, the "lifetime development effort" is not a measure of the changeability of *a design*, but may at best be a measure that somehow combines the changeability of *several, consecutive designs* of an evolving software system.

The above discussion highlights some of the difficulties in defining quality characteristics or concepts such as changeability. In summary, the changeability is reflected by the change effort to implement changes and the resulting quality of software reflected by the increase in change effort for future changes. The following working definitions are proposed:

> **Definition 1 (Changeability)** Apply a given change *c* to two alternative implementations of a software system s*1* and *s2*. Let *e1* and *e2* be the total effort to implement *c,* and the consequential change propagation to preserve the consistency of the total system, on *s1* and *s2,* respectively. The changeability of *s1* is better than the changeability of *s2*, with respect to *c*, if *e2 > e1*.
>
> **Definition 2 (Changeability Decay)** Apply a given change *c* to versions *v1* and *v2* of a software system, where *v2* is a later version of the software than *v1*. Let *e1* and *e2* be the total effort to implement *c,* and the consequential change propagation to preserve consistency of the total system, on *v1* and *v2,* respectively. The changeability is decayed with respect to *c* if *e2 > e1*.
>
> (Arisholm and Sjøberg, 2000)

The "given change *c*" is not viewed in isolation, but also includes the additional work associated with change propagation to ensure that the consistency of the system remains at the same level as before the change (Sjøberg *et al.*, 1997a; Sjøberg *et al.*, 1997b). Included in the consequences are thus new errors (the ripple effect). One study found that more than 50% of all errors were due to previous changes (Collofello and Buck, 1987).

As pointed out in (Eick *et al.*, 1999), the actual change effort depends on the ability of the developer implementing the change. Thus, changeability may be viewed not only as an attribute of the software system but also as an attribute of "people". In principle, when referring to "the total effort" in the definition of changeability and changeability decay, we could have added *"with respect to a given developer"*. However, experimental designs and statistical techniques may be used to control for differences in the individual skill level of developers. This is one of the main topics of Chapters 6 and 7.

2.2.1 Relationship between Productivity and Changeability

In (Arisholm and Sjøberg, 1999; Arisholm and Sjøberg, 2000), a productivity measure (the change size in SLOC divided by the change effort) was used as an indicator of changeability. Based on our current understanding of the underlying concepts, such an approach seems inappropriate. Whether 10 or 100 lines of code are implemented per hour does not represent a good indicator of changeability. For example, if design *d1* requires 100 lines of code and 2 hours to implement a given change, the productivity is 50 LOC/hour. If design *d2* requires 10 lines of code and 1 hour to implement the same, given change, the productivity is 10 LOC/hour. Still, *d2* is certainly easier to change than *d1* with respect to the given change.

2.2.2 Relationship between Maintainability and Changeability

Software changeability, as it has been defined above, is closely related to software maintainability. The ISO quality model described earlier defines changeability as a sub-characteristic of maintainability. However, we will argue that there are important reasons to distinguish between changeability and maintainability, and that these quality characteristics may (should) coexist on an equal level in a hierarchical quality model. According to Peercy,

> *...software maintainability is a quality of software which reflects the effort required to perform the following actions: (1) Removal/correction of latent errors (2) Addition of new features/capabilities (3) Deletion of unused/undesirable features (4) Modification of software to be compatible with hardware changes. Implicit in the above definition are the concepts that the software should be understandable, modifiable and testable in order to have effective maintainability.*
>
> (Peercy, 1981)

This definition corresponds well with the focus of this thesis. However, the empirical studies presented in this thesis primarily study change and change effort during initial development of software. Although there are many definitions of software maintenance, most of them are related to modification of operational software after system delivery (Jørgensen, 1994). The distribution of the types of changes may be quite different in initial development compared with maintenance. Furthermore, programmers maintaining software may be less experienced than the implementors (Munson, 1981), and may perform different tasks and use different tools (e.g., to convert data, to "port" code to a different platform or to improve performance). Consequently, the quality factors affecting change effort during maintenance may be different from the quality factors affecting the change effort during initial development.

2.2.3 Relationship between Code Decay and Changeability Decay

Our research is related to the early work of Lehman and Belady on program evolution (Lehman and Belady, 1985) and the recent Code Decay project (Eick *et al.*, 1999) at Bell Labs:

- Lehman & Belady – *Law of increasing complexity: As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.*
- Code Decay Project (Bell Labs) – *Code is decayed if it is more difficult to change than it should be, as reflected by three key responses: (1) COST of the change, which is effectively only the personnel cost for the developers who implement it; (2) INTERVAL to complete the change – the calendar/clock time required; and (3) QUALITY of the changed software.*

Like Lehman & Belady's "Law of Increasing Complexity", the research presented in this thesis is concerned with *"deteriorating structure"*, but only to the extent to which such deterioration actually affects the effort to implement changes.

The inherent difficulty of implementing different changes may of course vary substantially. For example, the implementation of a simple bug-fix requires significantly less effort than to implement a new accounting module in an existing software system. Thus, unlike the definition of "Code Decay", the definition of changeability decay refers explicitly to the increase in total effort required to implement the same, *given* change (including the necessary change propagation) in successive versions of the software system.

Like in the Code Decay definition, *QUALITY* is also (partially) reflected in the definition of changeability and changeability decay since a "change" includes the work required to ensure consistency. However, *INTERVAL* may not be a good indicator of changeability as the time schedule may depend on other external factors not related to the quality of the software. Thus, *INTERVAL* is not the focus of the research presented in this thesis.

2.2.4 Relationship to the ISO quality model and Change Impact

According to the ISO 9126 model, the changeability characterizes the software's *ability to sustain an on-going flow of changes*. This is very similar to our proposed definition of changeability decay. However, the ISO definition is not operational, i.e., it does not prescribe how changeability should be *measured*. In (Chaumun *et al.*, 2000), the authors base their definition of changeability on the ISO standard. Furthermore, they choose to quantify 'changeability' by assessing the *impact of changes*, defined as the set of classes that require correction as a result of a change (Chaumun *et al.*, 2000). This is related to the modularity of software. This view of changeability is also similar to the work of Lehman and Belady (Lehman and Belady, 1985), where trends in a count of the number of modules changed within a fixed time span is used as an indicator of 'system decay'.

Clearly, *change impact* is related to our definition of *changeability*. We believe that change impact may be a useful *indicator* of changeability. However, based on the definitions proposed in this chapter, change impact is only a valid indicator of changeability to the extent that it actually affects the effort to implement changes.

Such a relationship between change impact and change effort needs to be validated empirically (Kitchenham *et al.*, 1995b). That is, we take the view that changeability is an external quality characteristic that cannot be measured directly from the internal characteristic of the software. According to measurement theory, the internal characteristics of the software can only be used to *predict* external software quality attributes (Jørgensen, 1999). These measurement theoretical topics are discussed further in Chapter 6.

3 Research Methods in Empirical Software Engineering

Software engineering is a field of practice using methods and tools to solve problems where the solution is a software product. *Empirical* software engineering is the study of software engineering based on experiences and observations. In empirical software engineering one attempts to identify and establish a scientific approach for software engineering, which comprises of a set of research methods, theories, terminology, and a collection of experiences and observations (Sørumgård, 1997). The purpose of this chapter is to give an overview of the current state of practice in empirical software engineering, with emphasis on the research methods commonly used. Furthermore, an attempt is made to assess the feasibility, validity, strengths and weaknesses of various research methods.

3.1 Software Engineering as a Science

The last decade has seen explosive growth in the number of software engineering methods and tools, each one offering to improve some characteristics of software, its development, or its maintenance (Kitchenham *et al.*, 1995a). The number of silver bullets that have been proposed indicate that software engineering should be established as a scientific discipline as well as an engineering discipline (Sørumgård, 1997). Software engineering needs science to observe, gain general experience, establish knowledge, and create and validate theories and models.

3.1.1 Requirements to a Scientific Approach

Various research models to be applied within software engineering have been proposed. For example, Adrion suggests four methods (Adrion, 1993):

- **The scientific method** is based on developing a theory to explain a phenomenon observed in the real world. A hypothesis is formulated, and alternative variations of the hypothesis are tested by measurement and analysis of collected data.
- **The engineering method** evolves existing solutions. Based on results of testing existing solutions, better solutions are suggested, which in turn are developed, measured and analyzed. The cycle is repeated until no further improvements are possible.
- **The empirical method** is based on model proposals followed by empirical validation. Unlike the scientific method, a formal model or theory may not the basis for the proposal.
- **The analytical method** proposes a set of axioms upon which a theory is developed. Results from the theory are deduced, and if possible compared with empirical observations.

Jarvinen (Jarvinen, 1999) distinguishes between two different approaches for empirical studies:

- **Theory-testing studies** attempt to answer whether a part of reality correspond to a certain theory, model or framework, e.g. do our observations from, for example, experiments, case studies or surveys, confirm or falsify our theory, model or framework?
- **Theory-creating studies** attempt to obtain a theory, model or framework that best describes or explains a part of reality, e.g. which kind or theories, models or frameworks can describe the observations from, for example, experiments, case studies or surveys? Theory-creating studies are very suitable for exploratory investigations, i.e. when there is no prior knowledge of a part of reality or a phenomenon.

3.1.2 Scientific Validity

There are many views of what constitutes scientific validity of theories. By referring to Popper (Popper, 1968), Lee emphasizes that

> *…empirical validity (I) is just one requirement that a theory must satisfy in order to be scientific. There are three additional requirements, which are all associated with the concept of deductive testing of theories. One of these requirements is (II) logical consistency: as long as the different predictions that may be deduced from the theory are not mutually contradictory, the theory can be said to logically consistent. Another requirement (III) is that the theory must be at least as explanatory, or predictive, as any competing theory. The last requirement (IV) is that the theory, while falsifiable, must survive the actual attempts made at its falsification.*
>
> (Lee, 1989)

Lee describes a view of science sometimes referred to as *positivism*. In contrast, the *interpretive* view of science considers the methods of natural science to be inappropriate where human beings are concerned, mainly because different people (including researchers) will interpret situations in different ways (Braa and Vidgen, 1999). When observing and describing phenomena related to software development, the phenomena cannot be assumed to be governed by general laws and rules since there is a critical element of human behavior involved (Seaman, 1999).

In this authors opinion, both "positivistic" and "interpretive" research methods may provide useful results. The distinction between these approaches or schools of research within our field seems to be rather superficial. Regardless of the adopted philosophy of science, obtaining scientific validity in software engineering will always been complex and difficult. The complexity arises from technical issues, from the awkward intersection of machine and human capabilities, and from the central role of human behavior in software development (Seaman, 1999).

3.1.3 State of Practice

Until recently, the most commonly used form for validation of new software engineering technology has been *advocacy research* (Glass, 1994), i.e., it has been based on intuition, subjective experience and opinions of "gurus" instead of carefully designed scientific validation studies. In (Zelkowitz and Wallace, 1998), a literature survey was performed to assess the current state of practice of empirical studies of software engineering. The survey was based on 562 published papers in *IEEE*

Software, *IEEE Transactions of Software Engineering*, and *International Conference on Software Engineering (ICSE)* for the years 1985, 1990, and 1995.

The quantitative data suggests that the most prevalent empirical validation model appears to be lessons learned (informal case studies) and case studies, at about 10 percent combined. About 30 percent of the papers had no empirical validation. However, the percentages dropped from 36 percent in 1985 to 29 percent in 1990 to only 19 percent in 1995, indicating a shift towards more empirical validation studies. The qualitative observations made were (Zelkowitz and Wallace, 1998):

- Authors often fail to state their goals clearly or to point out the value that their method or tool adds to the experimentation process.
- Authors often fail to state how they validate their hypotheses.
- Authors often use terms very loosely. Authors would use the term "case study" informally and would even use terms like "controlled experiment" or "lessons learned" indiscriminately.

The authors conclude that "While the papers with no experimental validation seem to be dropping, clearly more work needs to be done".

Recently, many authors addressing research within software engineering emphasize the importance of measurements and observations (Lee, 1989; Fenton, 1992; Courtney and Gustafson, 1993; Kraemer, 1993; Fenton, 1994; Fenton *et al.*, 1994; Yin, 1994; Briand *et al.*, 1995; Kitchenham *et al.*, 1995a; Kitchenham *et al.*, 1995b; Pfleeger, 1995; Walsham, 1995; Briand *et al.*, 1996a; Kitchenham, 1996a; Kitchenham, 1996b; Zelkowitz and Wallace, 1998; Jarvinen, 1999; Jørgensen, 1999; Seaman, 1999). Such empirical studies are essential to making software engineering a science rather than an art (Fenton *et al.*, 1994). Thus, a further investigation should focus on how empirical studies are carried out, and what appears to be the critical problems in conducting such studies.

3.2 Research Methods in Empirical Software Engineering

Empirical studies play an important role within both theory-creating and theory-testing research. There are various ways of carrying out empirical studies, which are suitable for different purposes. Thus, prior to selecting an empirical approach, one needs to consider what is the purpose of the study, what observations will be made, and how the observations will be analyzed. In this section, the most commonly used methods for empirical studies of software engineering are described.

3.2.1 Surveys – Research in the Large

Surveys often try to capture what is happening broadly over large groups of projects: "research in the large" (Kitchenham *et al.*, 1995a). Thus, a survey may help you to evaluate software engineering technology on a larger scale. The potential benefit of surveys is that they can confirm and falsify theories by generalizing to many projects and organizations, using standard statistical analysis techniques. A survey is often formulated in a questionnaire, and is a typical theory-testing research approach. However, if the questionnaire contains open questions, the survey may also be used as

a theory-creating study (Jarvinen, 1999). Surveys combine the advantages of case studies (applicability to real-world projects) with those of experiments (replication).

However, according to Kitchenham, surveys are only able to demonstrate association, not causality:

> *An example of the difference between association and causality is that it has been observed that there is a strong correlation between the birth rate and the number of storks in Scandinavia; however, few people believe that shooting storks would decrease the birth rate.*
>
> (Kitchenham, 1996b)

A survey is *not* conducting a literature survey, nor administering a questionnaire, nor conducting a field interview (Kraemer, 1993). These are data collection techniques within survey research. Survey research involves gathering information for scientific purposes from a sample of a population using standardized instruments or protocols (Jarvinen, 1999). According to (Kraemer, 1993), survey research has three distinct characteristics:

1. A survey is designed to produce quantitative descriptions of some aspects of a study population.
2. The principal means of collecting information is asking people structured, pre-defined questions.
3. Information is collected from only a fraction of the study population – a sample – and is collected in such a way as to be able to generalize the findings to the population.

However, Cunningham (Cunningham, 1997) states that randomization and the assumptions of a normal distribution are ideal but difficult to achieve in survey research. A large percentage of organizational researchers use *convenience samples* instead of *random samples,* and reviews of many samples indicate that distributions are not normal (Cunningham, 1997). A typical problem is that people or organizations that are most willing to participate are those not representative of the population, hence resulting in biased data. Furthermore, surveys require that questions are carefully formulated. Otherwise the validity of the results may be threatened. The practice of using standardized questions in survey research is based on the following assumptions (Bradburn, 1982):

1. The meaning of the questions is shared by a majority of respondents.
2. The respondents understand the stimulus or phenomenon under investigation in a roughly equivalent way.
3. The responses will be given in a manner allowing the researcher to interpret and compare them.

Surveys that require users to evaluate or make judgments about information systems and their effect on specific work activities can produce misleading results if the respondents do not interpret or answer the questions in the ways intended by the researcher (Hufnagel and Conca, 1994).

3.2.1.1 Guidelines for Conducting Surveys

In (Kerlinger, 1988), several criteria for question-writing are given:

1. *Is the question related to the research problem and the research objectives?* All the items of the schedule should have some research problem function.
2. *Is the type of question appropriate?* Some information can best be obtained with open-ended questions, e.g. reasons for behavior, intentions or attitudes. Certain other information, on the other hand, can be more expeditiously obtained with closed questions.
3. *Is the item clear and unambiguous?* An ambiguous statement or item is one that permits or invites alternative interpretations and differing responses resulting from the alternative interpretations.
4. *Is the question a leading question?* Leading questions suggest answers.
5. *Does the question demand knowledge and information that the respondent does not have?* To counter the invalidity of response due to lack of information, it is wise to use information filter questions, i.e., to check whether a respondent knows what *X* is and means.
6. *Does the question demand personal or delicate material that the respondents may resist?* Special techniques are needed to obtain information of a personal, delicate, or controversial nature.
7. *Is the question loaded with social desirability?* People tend to give responses that are socially desirable, responses that indicate or imply approval of actions or things that are generally considered to be good.

3.2.2 Experiments – Research in the Small

Since formal experiments must be carefully controlled, they are often small in scale: "research in the small" (Kitchenham *et al.*, 1995a). Conducting a formal experiment means to control as many factors belonging to the phenomenon under study as possible. For example, if you want to determine whether programming language *A* produces "higher quality" code than programming language *B*, you must, among other things, ensure that programmers (the experimental subjects) are at an equal skill level for the two groups.

The application of formal experiments is particularly useful as a theory-testing research approach, attempting to falsify an existing theory. According to the scientific model, unless a strong conceptual-analytical theory exists, such experiments should be preceded by an exploratory phase consisting of case studies, surveys, or even pilot-experiments to create the initial theories to be tested subsequently.

Formal experiments are by far the most preferred scientific approach to empirical studies of software engineering; they are the principle means for testing hypothesis and theories according to the hypothetico-deductive method. However, even with formal experiments, one must be careful; formal experiments do not generalize outside the controlled experimental conditions (Kitchenham *et al.*, 1995a). Furthermore, many of the validity problems of surveys are also applicable to formal experiments:

- It is easier to use students rather than professionals.
- The curious and the exhibitionist are likely to populate any sample of volunteers.
- Attendees tend to be brighter, better workers, more motivated, more highly educated and more educated about a topic.

In order to impose full control, formal experiments are often small, which is a problem when you try to increase the scale from the laboratory to a real project.

3.2.2.1 Guidelines for Conducting Experiments

According to (Pfleeger, 1995), there are several steps to carrying out a formal experiment:

- **conception** – define the goal of the experiment and ensure that a formal experiment is the most appropriate research technique.
- **design** – formulate formal hypotheses (null hypotheses and the alternative hypotheses), decide on an suitable experimental design for applying differing experimental conditions to the experimental subjects so that you can determine how the condition affect the observed behavior, and determine how the results should be analyzed.
- **preparation** – prepare experimental materials and subjects.
- **execution** – apply the treatments to the experimental subjects in accordance with the experimental design.
- **analysis** – quality assurance and analysis of collected data using statistical techniques.
- **dissemination and decision-making** – conclude the results and document all the key aspects of the research.

In (Harrison *et al.*, 2000), details of an experiment involving forty-eight undergraduate students were described; the experiment illustrates the need for care when undertaking such a task. Features of the experiment were:

- The experiment on the modifiability and understandability of inheritance in C++ systems was based on a previous experiment carried out by (Daly *et al.*, 1996).
- A pilot study with twelve students was carried out initially to identify problems that may have hampered the larger experiment.
- The students were randomly allocated to one of twelve different groups. Each group member was allocated to one of four systems. No *learning effects* took place since each student was allocated to just the one system.
- Random allocation of students to the groups ensured that the results were unlikely to suffer from experience bias.
- Strict limits were placed on the time available for the experiment (carried out in a forty-five minute weekly slot).
- The materials were made available on-line via the web.
- The results contradicted the earlier experiment. This was considered interesting and added to the body of knowledge about this particular aspect of object-orientated software.

Although just one example of an experiment, a number of overriding features are highlighted:

- Proper materials, for example, code listings, need to be readily available and tested to ensure that minor problems are ironed out before embarking on the full experiment. The pilot study mentioned revealed several features of the tasks required that could have undermined results of the later experiment (e.g., the wording of the questions).
- Subjects should know the tasks they have to carry out to obtain representative results, e.g., careful training may be a pre-requisite if the subjects are unfamiliar with the experimental tasks.
- The groups should be comparable in terms of experience. This is important as we know individual capability can be an overriding factor in software development. However, this is far from trivial since we do not know very well how to characterize experience in software engineering.
- The experiment should be repeatable, and the materials should be made available to a wider community. A *replication package* should be made available containing details of the experiment.

The last point is important for building up a knowledge base of past experiments. One possible offshoot of the work described would be to undertake the same experiment using experienced practitioners. This could reveal differences between student and experienced programmers; a worthwhile area of research. In the OO community, so little is known about many facets. This situation can only improve via sharing of results and resources.

3.2.3 Case Studies – Research in the Typical

A case study allows in-depth understanding of one particular case or development project: "research in the typical" (Kitchenham *et al.*, 1995a). Case studies are particularly important for industrial evaluation of software engineering technology (methods and tools) because they can avoid scale-up problems. The case study research is also considered particularly suited when studying information systems in organizations (Braa and Vidgen, 1999). Braa and Vidgen distinguish between 'hard' case studies, e.g., (Yin, 1994), 'soft' (or interpretative) case studies, e.g., (Walsham, 1995), 'action case' (Braa and Vidgen, 1999) and 'action research' (Whyte, 1991). Regardless of the detailed approach taken, case study research enables reality to be captured in detail and many variables can be analyzed.

Case studies are often incorporated into the normal software development activities. The potential for in-depth knowledge obtained from a case study means that the case study may be a good explorative means for establishing new theories, i.e., theory-creating research. Both qualitative and quantitative data collection techniques may be used to strengthen the validity of the results. Furthermore, case studies allow us to determine whether predicted effects from an existing theory apply in a given organization – hence it may be used for the purpose of "falsification" of existing, more general, theories, i.e., theory-testing research.

However, case studies cannot achieve the scientific rigor of formal experiments or surveys (Kitchenham *et al.*, 1995a). There are problems with generalization due to

lack of control and possible effects caused by the intervention of the researcher. A case study can show you the effects of a technology in a typical situation, but it cannot be generalized to every possible situation (Kitchenham *et al.*, 1995a). However, this is not necessarily a problem. For example, the purpose of the case study might be to explore ways of building better effort prediction models for a given type of organization. The actual prediction model based on the local effort and product data may not be valid outside the project or organization, but the results are still useful from the software organization's point of view. Furthermore, the case study research may have resulted in the development and preliminary evaluation of an estimation method that probably can be reused on other projects. Several case studies will together form a broader picture from which both researchers and industry can draw knowledge.

3.2.3.1 Guidelines for Conducting Case Studies

While case studies may be useful to get in-depth understanding of a phenomenon, hence potentially resulting in the creation of well-founded theories, conducting industrial case studies are inherently difficult. Care must be taken to ensure validity of data and usefulness of results for the studied organization. This section describes guidelines that are a result of the lessons learnt in six industrial case studies, some successful and others unsuccessful, presented in (Arisholm *et al.*, 1999a).

High-risk items

- Avoid large geographic distance between researcher and organization. Frequent, direct contact may alleviate communication problems and subsequent conflicts.
- Keep the number of organizational "layers" between the researcher and the studied organization to a minimum.
- Ensure necessary involvement and backing from the organization(s) before initiating the research project.
- A negative attitude may be created if a developer feels that the study can be used for individual monitoring purposes. Ensuring that this is not the case may improve the cooperation from the developers.

How to get inside?

- Know the organization (its goals, focus, earlier research on that organization, etc.).
- First presentation/discussion is essential and should focus on a realistic assessment of the usefulness of the research for the organization (ask yourself, what would make you as a manager to say "yes" to this project).
- Use your personal network.
- Organizations may consider regular contact with students as a good opportunity for recruitment.
- Results from the case study may have a marketing effect for the organization if presented in the right media. Capitalize on this factor during initiation and planning, and follow up after the results are available.
- Agree on plans (but be open to redirect goals and scope).

- Discuss expectations with each other.
- Discuss major risk factors.

How to get high quality data?

- An initial "pilot-study" might be useful to assess and reduce risks, improve the data collection process and focus the research goal.
- When introducing a tool that automatically checks the quality of software, one should ask: Who should use the tool? How should the working process be organized to benefit as much as possible from the tool? How should the project management motivate and encourage active use of the tool?
- Whenever possible, collect "real time" data. Historical data may have lower quality and validity.

How to present the results?

- Sensitive data should be considered left out of the study.
- Sign a confidentiality agreement with the organization. Let the organization read and accept publications before submission.
- Intermediate results should be presented frequently to ensure that the organization understands the motivation, goals and potential organizational benefits from the research. Write and present reports in addition to scientific papers.

3.2.4 Qualitative versus Quantitative Methods

In software engineering, the blend of technical and human behavioral aspects lend itself to combining qualitative and quantitative methods in order to take advantage of both (Seaman, 1999). The distinction between qualitative and quantitative methods are in many respects orthogonal to research method classifications. A common misconception is that qualitative data is more *subjective* than quantitative data. However, clearly the degree of subjectivity of observations is orthogonal to the (more or less mechanical) coding of the data. Qualitative data is just data represented as words and pictures, not numbers (Gilgun, 1992). Qualitative methods collect data mainly from participant observation and interviews, for example, in the context of a "think-aloud" experiment or a case study. Data is not coded into numbers; instead, the researcher delves into the complexity of the problem rather than abstracting it away. In (Arisholm *et al.*, 1999b), interviews and participant observations were the principle means to explain the irregularities indicated by related quantitative data. Thus, the analysis of qualitative data may be particularly well suited to create theories related to software development, and is often useful to explain and clarify quantified phenomena.

3.3 Summary

The design and performance of empirical studies is a difficult craft. Many pitfalls may compromise the validity of scientific results. Although general principles and techniques are available to perform empirical studies (e.g., experiments and case

studies), each discipline needs to develop its own body of experience and strategies to answer its most pressing research questions. For example, although this is an oversimplification, sociology has focused on the design of surveys, medicine on longitudinal studies, and psychology on controlled experiments. Each strategy reflects the working constraints of the respective field. In other disciplines than software engineering, many of the more revealing empirical studies have been based on prior studies that either support or refute prior claims. Progress in empirical research comes from questioning past research and learning from past mistakes or insights.

If you are trying to choose among several competing methods or tools, you may organize your study as a survey, a case study or a formal experiment, or as a variation or combination of these methods. You may also use a combination of qualitative and quantitative data collection and data analysis techniques. The choice of investigative method depends in the goal of the research. For example, a case study combining both qualitative and quantitative data may be useful for the creation of theories subsequently tested empirically through a formal experiment or a survey. All of the research methods discussed in this chapter have potentially serious validity problems, and, in our opinion, any empirical study of software engineering should contain a careful assessment of the validity issues outlined. In most cases, utilizing a combination of the research methods may increase the empirical validity of any study.

Is the current state of practice of empirical software engineering a science? There is a large amount of research to be done in terms of studying software engineering empirically. In general, there is still a need for more empirical studies to establish software engineering as a scientific discipline – moving away from the "advocacy research" approach. However, even among the studies that *have* been conducted using empirical validation methods, improvements can be made. Since the empirical study of software engineering is a relatively new field of research, we often do not have an a-priori formal theory to be "falsified" through empirical tests. According to Adrion's classification, one may conclude that software engineering increasingly uses the "empirical method" for theory creation, but may not yet have reached the maturity level of the "scientific method".

We are still exploring ways to design empirical studies and overall research programs to answer efficiently the most pressing research questions. How can we, in our field, investigate new technologies in a way that limit the cost of investigation, the risks in pilot projects, and ensure maximal benefits for software development organizations? This is a question that will take time to answer and that will require more experience. Field experiments are prohibitively expensive. Thus, controlled experiments will likely involve students during the early stages of research. If the results of such experiments show promise, this will likely be followed by low risk, small-scale pilot projects, before moving to representative development projects. Such a scheme implies the close collaboration of academic institutions and industry but also the acceptance, like in medicine, that empirical studies are worthwhile investments.

4 Evolutionary Development

The aim of this chapter is to describe how evolutionary development may affect the changeability of software. A prerequisite for meaningful discussions is use of terminology that is agreed upon and understood. The semantics of terms such as prototyping and evolutionary development is unclear, cf. (Patton, 1983). What is the difference between incremental development and evolutionary development? Is prototyping a technique or a process, or both? Thus, Section 4.1 proposes a categorization of common software process models and describes each model in some detail. Based on existing empirical research, Section 4.2 identifies important issues that may affect the resulting changeability of software. Section 4.3 summarizes.

4.1 Software Process Models

During the past decades, several system development models (also known as software life-cycle models, software process models, etc.) have been proposed. The code-and-fix paradigm of early software projects was deemed no longer adequate to deal with the increasing size and complexity of the development of software systems. One of the first proposed models was the waterfall model (Royce, 1970). It prescribes a sequence of activities, milestones and deliverables that are supposed to aid in the development of more complex software systems. The milestones and deliverables provided a formalized means of communication between customer or user and software development teams. However, one may argue that the waterfall model represents a mechanized view of software development, focusing on project management aspects and neglecting problem solving aspects. For example, what is required of users and development teams to produce a software system according to the "real" needs of the customer or end user? In an attempt to address these and other critical aspects of the waterfall model, other models of software development, such as incremental development and evolutionary development, were proposed.

There are many ways to view the process models described in the literature. In this section, common process models are categorized to clarify the terminology and the underlying concepts for the subsequent discussions.

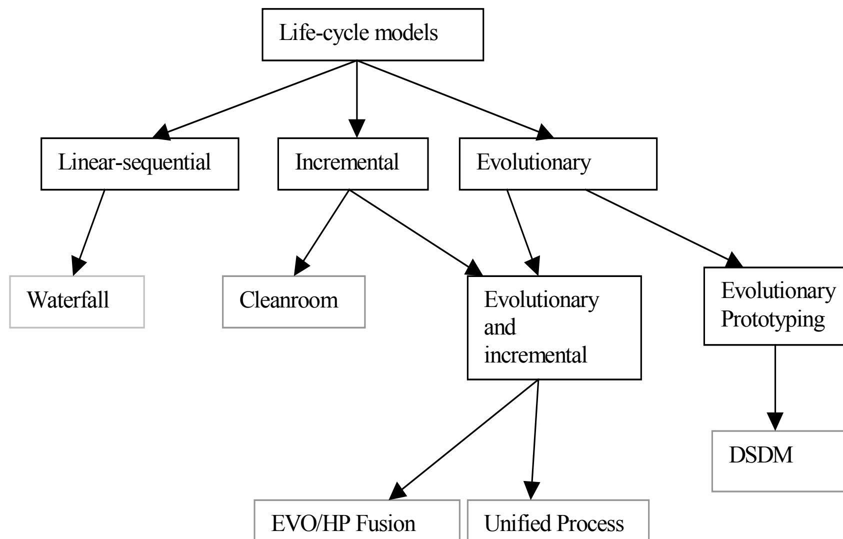

Fig. 4.1. Categorization of common process models. The semantics of the arrows is "inherits properties from the higher level model". The gray leaf-nodes are actual manifestations of process models based on higher level conceptual models.

Figure 4.1 depicts one possible view of software life-cycle models. The life-cycle models are divided into three main categories:

- Linear-Sequential Models,
- Incremental Models, and
- Evolutionary Models.

These categories will be described in the following subsections.

4.1.1 Linear-Sequential Development

The waterfall model effectively helps developers determine what they need to do. The simplicity of the model makes it easy to explain to customers who are not familiar with software development. Each phase of the process produces deliverables that constitutes concrete milestones for easy progress monitoring.

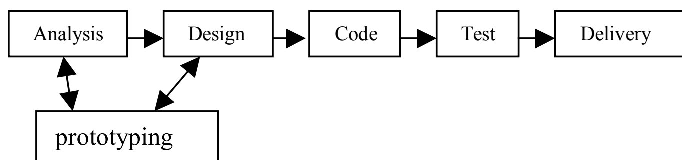

Fig. 4.2. A simplified view of the waterfall model

The biggest problem of the waterfall model is that it may not reflect the way software systems are really developed (Pfleeger, 1998). Except for very well understood problems, software development may be viewed as a problem solving process, in which software delivers a solution to a problem that has never been solved before. Such problem solving often requires a great deal of iteration, in which end users may play an integral part. The waterfall model provides no guidance to managers and developers on how to handle changes to products and activities that are likely to occur during development, typically as a result of natural but unprescribed interaction with end-users and customers. The waterfall process tells us nothing about the back-and-forth activities that are needed to create a final product.

To address some of the problems of waterfall development, one may incorporate prototyping in the early phases, as depicted in Figure 4.2. The underlying assumption is that early prototypes supply the necessary knowledge to implement a complete system following a sequential life-cycle. However, empirical results presented in (Lichter *et al.*, 1994) show that this assumption is problematic:

- Control techniques developed for waterfall development tend to hinder the incorporation of prototyping.
- Contractual enforcement of milestones and deliverables, and the results of consecutive prototyping cycles rarely fit together.
- The effort necessary for explicit evaluation of prototypes by end users is often ignored or underestimated.

4.1.2 Incremental Development

With incremental development, the product is partitioned in increments that are delivered either sequentially or in parallel. The main difference between waterfall development and incremental development is that the implementation phase contains staggered or parallel development of code modules.

Incremental development is depicted in Figure 4.3. One well-known incremental development process is the Cleanroom process (Linger, 1993). Each increment is frozen after delivery, and subsequent increments must use the interfaces provided by modules implemented in other increments. This means that process activities are repeated for new increments, but there is, in principle, no evolution of the actual incremental deliverables. Thus, existing functionality does not evolve. Incremental development may be useful to shorten time-to-market by staging system functionality in incremental deliveries.

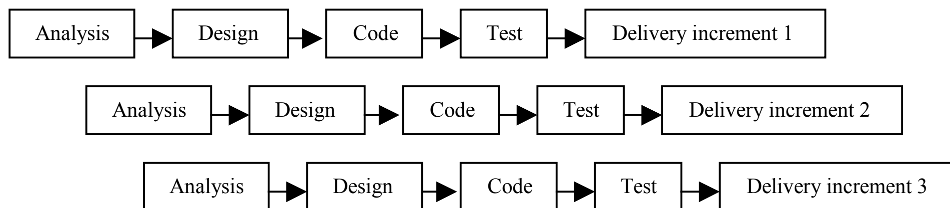

Fig. 4.3. Incremental development, adapted from (Pressmann, 1997)

4.1.3 Evolutionary Development

Evolutionary development is related to the "iterative enhancement" technique proposed in (Basili and Turner, 1975). This technique was proposed as a result of the realization that the problem (the requirements) and solution (the software) are often not understood at the inception of the project. An important objective of evolutionary development is to identify the "real" needs of the customer early, hence achieving improved customer satisfaction and avoiding expensive last-minute rework. In contrast to the linear sequential and incremental development, evolutionary development can be characterized by having unfinished incremental deliveries that *evolve* towards the final product. Each increment may be delivered to end users for feedback. Risks are assessed and plans are modified in response to the provided feedback. The following sections provide an overview of some common, evolutionary life-cycle models.

4.1.3.1 Evolutionary Prototyping

Prototyping may be regarded as techniques or process activities that can be incorporated in other process models, such as the waterfall model and the spiral model. At the point when a prototype has fulfilled its primary exploratory purpose, the prototype is often thrown away and a production system is developed. Throw-away prototypes are often built with little or no robustness, and serve only to evaluate parts of the system that are not well understood. When prototypes are *not* thrown away, the process is sometimes called evolutionary prototyping (Floyd, 1984; Bersoff and Davis, 1991; Lichter *et al.*, 1994).

> *Evolutionary prototypes are high-quality programs used to validate presumed requirements, to gain experience required to uncover new requirements, or to validate a possible design; and are repeatedly modified and re-deployed whenever new information is learned.*
>
> (Bersoff and Davis, 1991)

4.1.3.2 Evolutionary, Incremental Development

Evolutionary, incremental development can be seen as a mixed approach between the specifying approach (e.g., linear-sequential development and "pure" incremental development) and evolutionary prototyping. Some examples include Gilb's EVO (Gilb, 1988), HP Evolutionary Fusion (Cotton, 1996) and Rational Unified Process (RUP) (Jacobson *et al.*, 1999).

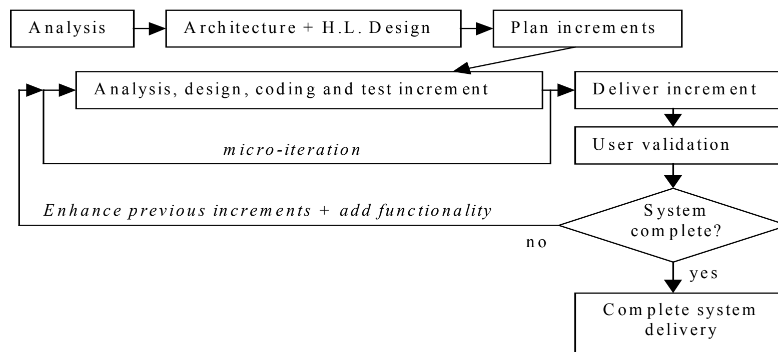

Fig. 4.4. The Genova Development Process (Arisholm *et al.*, 1999b)

Genova is another manifestation of an evolutionary and incremental development process (Arisholm *et al.*, 1999b). Figure 4.4 depicts the evolutionary delivery of increments, prescribed by the Genova process. The process prescribes the delivery of an initial architectural baseline and a high-level design. Each increment is developed by iteration of all major process activities, including analysis, design, coding and test. Note that each increment sub-project may itself be evolutionary, that is, each increment evolves through iteration of process activities until the increment is delivered. Hence, evolution occurs both within increments and between successive incremental deliverables. The Genova process prescribes up to three iterations per increment:

- Iteration 1: Implement the most important functional requirements of the increment. Iteration 1 serves as an evaluation of the design of the increment.
- Iteration 2: Implement the remaining functional requirements of the increment. Enhance functionality developed in iteration 1.
- Iteration 3: Stabilize increment.

The increment is then delivered to end-users for evaluation. The next increment will contain enhancements to the previous increments as well as new functionality. The system delivery is completed when all functionality has been delivered and no further enhancements are required.

Another example of evolutionary, incremental development is the spiral model (Boehm, 1988). The spiral model has a strong emphasis on the evaluation of requirements and the reduction of risks throughout the software life cycle. The spiral model differs from "pure" incremental development in that delivered increments are not "frozen", but may change over time based on evaluation of risk. Prototyping is primarily used to identify and resolve risks. The prototype does not evolve into the final product as in evolutionary prototyping. The risk assessment may in turn change system requirements and development plans. Thus, the spiral model for software development is evolutionary and incremental. Each increment is an enhancement of the previous increment, and the development of new increments uses feedback from the previous increment to converge towards the final system.

4.2 Consequences for Changeability

Does evolutionary development result in software with poor changeability? If so, what are the typical causes of decay, and how can it be reduced? Or perhaps the constant change in evolutionary development means that the structure can evolve and become better than if a sequential process had been used?

Some existing work has studied (or made claims regarding) how evolutionary development projects may affect the quality of the code, primarily focusing on usability. Only a few papers have made claims regarding quality aspects related to changeability. Even fewer papers have *empirically* investigated how evolutionary development may affect the changeability of the resulting software. In this section, an overview of existing claims and empirical studies related to changeability in evolutionary development is given.

4.2.1 Sommerville

According to Sommerville, following an evolutionary development process results in a product that is often difficult and expensive to change, because the constant changes to the software degrade its structure (Sommerville, 1996; Sommerville, 2001).

4.2.2 Lehman *et al.*

Studies regarding the effect of changes have been made for software evolution in general, most notably by Lehman's law of increasing complexity (Lehman and Belady, 1985). This 'law' of software evolution was also tested empirically in (Chong Hok Yuen, 1987). Using response times and number of outstanding fixes as dependent variables, the law of increasing complexity was confirmed. An overview of this and other related studies of software evolution is given in (Eick *et al.*, 1999; Kemerer and Slaughter, 1999). However, these studies are related to the maintenance of old and large legacy systems, not the initial development of software. Clearly, changes may eventually lead to decay unless work is done to avoid it. It is, however, unknown whether the results from studies of the evolution of legacy systems generalize to initial development of software using an evolutionary development process.

4.2.3 Parnas

Parnas introduced the concept of *software aging* (Parnas, 1994), which has two primary causes:

- The failure of the product owner to meet the users' changing needs.
- Changes made by people who do not understand the original design concept almost always cause the structure to decay.

To avoid software aging, Parnas recommends the principle of "designing for change" by means of information hiding and encapsulation, ensuring consistent documentation, and performing design reviews.

In the context of evolutionary development, the first cause implies that evolutionary development is also a way to avoid (or delay) software aging because more of the users' needs might be met during initial development.

4.2.4 Boehm *et al.*

Boehm states that evolutionary development is generally difficult to distinguish from the 'code-and-fix' model, resulting in "spaghetti" code (Boehm, 1988).

In (Boehm *et al.*, 1984), a prototyping approach was compared with a specification approach. The study involved an experiment where seven teams developed their own versions of the same product, an interactive version of COCOMO (Boehm, 1981). Four teams used a specification approach and three teams used an evolutionary prototyping approach. The experiment attempted, among other things, to assess what effect prototyping had on a software project's effort distribution, schedule distribution, and productivity; and on the product's size, quality and maintainability. The results are summarized below:

- The development teams using the prototyping approach used less overall effort, less design effort and more testing effort compared with the development teams using the specification approach.
- Based on a subjective assessment of the quality of the resulting systems, the results indicated that a specification approach produced more coherent designs and more robust software that was easier to integrate than did the prototyping approach. Consequently, Boehm suggests that prototyping should be followed by a reasonable level of specification of the product and its internal interfaces, especially for larger products.
- The systems developed using the prototyping approach were smaller and the ease of learning and ease of use was better than for the systems developed with a specification approach.
- The prototyping approach had a reduced "deadline effect" at the end of the project, because it always had something that "worked".

Thus, the experiment indicated that the prototyping projects produced code with lower changeability but better usability. However, the resulting code was also smaller, and required less total effort to develop. This is reiterated in another paper, in which Boehm states that a primary source of difficulty with the waterfall model has been its emphasis on fully elaborated documents as completion criteria for early analysis and design phases:

> *Document-driven standards have pushed many projects to write elaborate specifications of poorly understood user interfaces and decision-support functions, followed by the design and development of large quantities of unusable code.*
>
> (Boehm, 1988)

Furthermore, the most significant influence on software costs is the number of source instructions one chooses to program (Boehm and Papaccio, 1988). Thus, it seems that prototyping projects have time to spare – which could be used on improving the design.

4.2.5 Royce

Royce (Royce, 1990) describes the results of using the Ada Process Model, which is an evolutionary and incremental development process focusing on early design integration, design reviews and risk management. According to Royce, an evolutionary and incremental process results in a higher probability of producing a higher quality product:

> *Given a complex software system, there are far too many subtle interactions, miscommunications, and complex relationships to predictably achieve quality design verification without actually building subsets of the product and getting factual feedback. The real evaluation of goodness occurred very late in conventional programs when components were integrated and executed in the target environment together for the first time. This usually resulted in excessive rework and caused late "shoehorning" of less than desirable solutions into the final product. These late, reactive changes resulted in added fragility and reduced product quality.*
>
> (Royce, 1990)

Royce points out that evolutionary development may shift *rework* to an earlier phase of the software life cycle. Many studies have shown that the rework effort is much smaller in the earlier phases of software development (Fagan, 1976; Boehm and Papaccio, 1988). Royce also describes the concept of a software architecture skeleton (SAS) as being fundamental to evolutionary development.

4.2.6 Brownsword *et al.*

Lessons learned from three case studies using evolutionary, incremental development processes are described in (Brownsword and McUmber, 1991). Their experiences are summarized as follows:

- Some of the iterations focused on resolving requirements issues, while others improved the software design.
- Evolutionary, incremental development was the most efficient means for validating the proper layers of abstraction in an object-oriented design.
- An object-oriented design proved most resilient when adding system capabilities to the design in each successive iteration.
- Projects that leveraged the use of modern software design techniques had the best experiences with evolutionary, incremental development.
- Adding or changing the system requirements typically impacted only on the top-level design properties, not the entire design.

The results in (Brownsword and McUmber, 1991) suggest that evolutionary, incremental development may have a positive effect on the changeability of the resulting software. However, the experiences also imply what is also stated by Gilb: the design of an open-ended architecture is critical for the success of evolutionary development processes (Gilb, 1988).

4.2.7 May and Zimmer

Case studies on the experiences on the use of the Hewlett-Packard Evolutionary Fusion model have been reported (Cotton, 1996; May and Zimmer, 1996). The following summarizes the experiences in (May and Zimmer, 1996):

- The teamwork with end-users was improved and more time was spent by the developers thinking of alternative solutions to the given problems.
- After gaining some experience with the process, both developers and management felt that it was easier to focus on the right things in the development process and to uncover key issues early.
- The selection of users, and the management of the selected users, were important tasks when using the process. Selecting the right users had a major impact on the quality of the feedback. The closer the users are to real external users, the higher the quality of the feedback, but the more external they are, the harder they are to manage.
- The management focus in traditional software development was 95% on deployment. When using the evolutionary process, one third of the management effort was spent on getting feedback from the users and to make decisions based on the feedback. This may be seen as an indicator that the user feedback was regarded as a cost-effective means to improve the quality and usability of the software system.

4.2.8 Zamperoni *et al.*

This section reports some of the results from a case study where evolutionary prototyping (also referred to by the authors as 'incremental and iterative') was used within the TNO Institute of Applied Geoscience (Zamperoni *et al.*, 1995). Evolutionary prototyping projects were compared with projects using a waterfall life cycle. The projects were compared in pairs of very similar sequential and evolutionary projects or a total duration between 18 and 30 months and team sizes ranging from 4 to 15. Early development used the *Objectory* method (Jacobson *et al.*, 1992). Some of the highlights of the study are summarized below:

- Design activities had a considerably bigger share of the total percentage of the total project man-hours in projects with the evolutionary approach (18.1% compared with 10.8% for sequential projects). This was not primarily an indication of more complex systems, but was mainly due to the fact that evolutionary prototyping required a better-designed system architecture, which did not only aim at an optimal final product, but also facilitated reconstruction and change of system subcomponents during development.
- The evolutionary prototyping projects required much less time for testing (7.4% compared with 12.9% for sequential projects).

Thus, the case study reported in (Zamperoni *et al.*, 1995) observed a completely opposite effect of the experiment reported in (Boehm *et al.*, 1984). Other qualitative observations regarding the success of evolutionary prototyping projects were also described in (Zamperoni *et al.*, 1995). These are summarized below:

- Communication between users and developers was an important prerequisite for capturing the real system requirements. Sometimes users or customers had a certain, very specific perception of the future system, but were unable to communicate or formulate this perception in an adequate manner (for the developers to build the system).
- Evolving prototypes acquire two crucial roles: (1) The means of communication with users and domain experts, i.e., the mediator to reach agreement with the users and their expectations, and (2) the central project repository about acquired specification knowledge, design decisions, and development solutions, and therefore also the primary source for evaluation of development progress.
- If users are involved in the development process, the likelihood of acceptance increases. Furthermore, if future users are identical to the test users, this causes a significant decrease of the learning expenses concerning the introduction of the final system at the users' site because the users have already been accustomed to the system during its development.

However, the success of evolutionary prototyping has the following prerequisites:

- Commitment of users and short communication channels between users and the development team are important prerequisites for an effective application of evolutionary prototyping.
- Selection of the appropriate types of users for evaluation of prototypes is important. This selection depends on the state of completion of the system. Obviously, not every potential user should be burdened with testing partial or unstable prototypes. In the early stages of the software, it may be more practical to recruit members of the development team to evaluate prototypes. The actual users may be recruited once the functionality and stability of the prototypes increase.

4.2.9 Ehn

In the UTOPIA project, requirement specifications and systems descriptions based on information from interviews were not very successful:

> *What is it that the users know, that is, what have they learned that they can express in action, but not state explicitly in language?*
>
> (Ehn, 1993)

Prototyping design artifacts make it possible for ordinary users to use their practical skill when participating in the design process (Ehn, 1993).

4.2.10 Emam *et al.*

Emam *et al.* (Emam *et al.*, 1996) conducted an empirical study of user participation in the requirements engineering process. The results indicate that as uncertainty increases, greater user participation alleviates the negative influence of uncertainty on the quality of requirements. Increased user participation seems to be conductive towards greater user consensus, and also helps them reason about what their business- or work-processes should be like and what they want the software system to do. However, as uncertainty *decreases*, the beneficial effects diminish. When uncertainty

is low, user participation has no impact on the quality of requirement. Furthermore, users resent participation when they feel that they are unable to contribute substantially. This resentment may bring about *reductions* in quality of service as user participation increases.

These empirical results support the common belief that user participation is primarily useful for projects where requirements uncertainty is high, cf. (Sommerville, 1996). However, as pointed out in the following case studies (Section 4.2.11), there are many potential problems in obtaining the benefits of user participation.

4.2.11 Lichter *et al.*

In (Lichter *et al.*, 1994), a critical view of the prerequisites for successful prototyping and end-user involvement is presented. The results are based on interviews with developers in several industrial software projects, most of which used an evolutionary prototyping approach. Two remarks regarding the quality of the resulting systems are given in the paper:

- In one project, system components that became redundant due to modifications of the system design could still be found in the final system.
- In general, developers may make the mistake of encouraging the users to voice all the ideas and wishes that come into their minds when evaluating a prototype. This may result in the incorporation of absolutely every conceivable function or design option. This may increase the complexity and reduce the usability of the final system.

In addition to the above points, the general assessment given by Lichter *et al.* is that there are substantial problems in getting the users involved. Often, user management is reluctant to allow the actual end-users of an application system to participate in the evaluation of prototypes, let alone in discussions on design options. Even when end-users are involved, there are other problems that may limit the benefits of user participation in evolutionary development projects:

- The effort necessary for explicit evaluation of prototypes by end-users are often ignored or underestimated.
- Many developers expect too much from the users concerning creativity and innovative ideas about the technical design of a prototype. Since the users lack experience in everyday use of the system, they seldom can make suggestions for the design of a technical aspect that does not yet exist. Their suggestions are generally confined to criticism of what already exists.

4.3 Summary

In this chapter, common life-cycle models were described in some detail. Furthermore, an overview of existing claims and empirical results related to the consequences for changeability in evolutionary development was provided.

The results from several of the studies suggest that attention must be placed on the design of an open-ended architecture, that is, a design that can easily incorporate new

or changing requirements. Evolutionary software development may also result in software systems of smaller size, since the exploratory approach focusing on user participation may ensure that unnecessary functions are not developed. It is likely that both size and structure affect the changeability of software. Thus, the smaller size of the developed software may counteract some of the potentially negative influences of the frequent changes on the structure. Evolutionary development may also result in systems with high usability and user acceptance, given effective user participation. However, an important prerequisite is that the user participation is *effective*. Empirical studies have shown that achieving this is not straightforward.

Existing empirical studies constitute insufficient data to either support or refute the claim that evolutionary development often result in poorly structured designs. The presented studies are contradicting. Some of the studies suggest that evolutionary development result in poor structure. Other studies suggest the opposite. Perhaps more importantly, there is insufficient knowledge regarding what potentially *causes* decay, or how it can be avoided. The existing papers that have assessed consequences related to changeability are either based on a subjective assessments of the resulting code, e.g., (Boehm *et al.*, 1984; Brownsword and McUmber, 1991), or the results are based on the author's experience, e.g., (Parnas, 1994; Sommerville, 1996).

The questions outlined in this chapter are fundamental in software engineering. For example, *how* do we design for change? How do we determine whether a structure is actually deteriorating? Such questions have motivated the proposed measurement framework described in Chapter 6 and the empirical studies described in Chapter 7. As pointed out in Chapter 3, the accumulation of results from carefully designed empirical studies may increase our knowledge leading to a better understanding of the complex questions discussed.

5 Empirical Studies of Object-Oriented Software

Object-oriented technologies are becoming pervasive in many software development organizations. However, many methods, processes, tools, or notations are being used without thorough evaluation. Empirical studies aim at investigating the performance of such technologies and the quality of the resulting object-oriented software products. In other words, the goal is to provide a scientific foundation for the engineering of object-oriented software.

Some of the material presented in this chapter is based on the results of a working group at the Empirical Studies of Software Development and Evolution (ESSDE) workshop in Los Angeles in May 1999, in which this author participated. The results were published in (Briand *et al.*, 1999b).

Section 5.1 provides an overview of representative existing empirical studies, focusing primarily on studies assumed to be relevant for changeability assessment. Section 5.2 describes success factors or guidelines for conducting empirical studies of object-oriented software. Important directions for future research are described in Section 5.3. Section 5.4 summarizes.

5.1 Overview of Existing Empirical Studies

Do we have support for the claims that object-oriented technologies improve the quality of software product deliverables, support reuse, and reduce the effort needed to develop and maintain the software product? Which of the available technologies are more likely to yield benefits? This section attempts to give an overview of empirical validation methodologies and existing empirical studies to shed some light on some of these questions.

Existing research of the object-oriented paradigm can be classified according to the following (non-comprehensive and overlapping) research areas:

- Methodology for formal definition, theoretical, and empirical validation of object-oriented product measures.
- Empirical evaluation of object-oriented quality and productivity models, such as fault proneness, testability, and maintainability.
- Empirical evaluation of object-oriented technologies, such as programming languages, tools, methods, and processes.

These three broad categories of research provide the basis for the research outlined in this thesis: The measurement framework (Chapter 6) describes object-oriented product measures and methods for their empirical validation. The measurement framework is validated empirically by building a (quality) prediction model for changeability (Chapter 7). After careful validation, the framework may eventually be used for technology assessment, more specifically, for the assessment of consequences of evolutionary development on the changeability of object-oriented software.

5.1.1 Measurement Validation Principles

Several authors have suggested that measures should adhere to measurement theoretical principles (Zuse, 1991; Fenton, 1992; Fenton, 1994) as a means for evaluating software measures and to qualify the use of certain statistical techniques (depending on the measurement level of the measures). Briand *et al.* (Briand *et al.*, 1996a) argue, however, that a more pragmatic approach is likely to provide the software engineering community with more practical results. Several guidelines and frameworks for theoretical and empirical validation of measures have been proposed (Weyuker, 1988; Briand *et al.*, 1995; Kitchenham *et al.*, 1995b; Briand *et al.*, 1996b). These frameworks argue that measures should obey certain fundamental properties, that compound measures must be justified by an explicit theory, and that internal measures should be validated empirically against external (quality) attributes. Methodologies for the definition and construction of object-oriented coupling and cohesion measures are proposed in (Churcher and Shepperd, 1995; Briand *et al.*, 1998b; Briand *et al.*, 1999c). Finally, there are many different statistical approaches that may be used for establishing empirical validation of relationships between internal product measures and external software quality attributes, cf. (Kitchenham and Pickard, 1987; Briand *et al.*, 1992; Briand *et al.*, 1995; Kitchenham *et al.*, 1995b; Briand *et al.*, 1998c; Khoshgoftaar and Allen, 1998).

5.1.1.1 Structural Attribute Measures for Object-Oriented Software

Several measures for object-oriented designs have been proposed (Li and Henry, 1993; Chidamber and Kemerer, 1994; Brito e Abreu and Melo, 1996; Briand *et al.*, 1997b; Bieman and Kang, 1998) and validated (Basili *et al.*, 1996b; Brito e Abreu and Melo, 1996; Briand *et al.*, 1998b; Harrison *et al.*, 1998a; Harrison *et al.*, 1998b; Briand *et al.*, 1999c; Briand *et al.*, 1999d; Briand *et al.*, 2000). The "CK metrics suite" proposed by Chidamber and Kemerer (Chidamber and Kemerer, 1994) are the most frequently referenced object-oriented design measures.

5.1.2 Empirical Assessment of Object-Oriented Technologies

Object-oriented technologies (programming languages, tools, methods, and processes) are claimed to improve the quality of software product deliverables, to support reuse and reduce the effort of developing and maintaining a software product. However, little evidence exists to support these claims (Jones, 1994; Daly *et al.*, 1996). Surprisingly few papers exist that empirically compare OO technologies and processes with traditional structured techniques. Jones has identified lack of empirical evidence for several claims related to gains in productivity and quality of OO technologies (Jones, 1994). For example, inheritance and polymorphism are claimed to provide benefits such as greater extensibility and reusability of OO systems (Booch, 1994). However, (Jones, 1994; Daly *et al.*, 1996) showed results suggesting that, beyond a certain level, inheritance was a serious hindrance to maintainability.

In (Henry *et al.*, 1990), an experiment is described that supports the claim that systems developed with object-oriented languages are more maintainable than those developed with procedural languages. In this empirical study, student subjects determined the maintainability of systems developed with two languages by performing maintenance tasks on two functionally identical large programs, one

written in an object-oriented language and the other written in a procedural language. Maintenance times, error counts, change counts, and programmers' impressions were collected. The analysis of the data from this experiment showed that systems using object-oriented languages were indeed more maintainable than those built with procedural languages.

In a study described in (Basili *et al.*, 1996a), object-oriented reuse techniques were applied to eight medium-sized management information systems, using the waterfall life-cycle model. The study indicated *"significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity"*.

Results described in (Briand *et al.*, 1997a) strongly suggest that quality guidelines based on Coad and Yourdon's design principles have a significant beneficial effect on the maintainability of OO design documents. However, preliminary results showed that there was no strong evidence that OO designs were easier to maintain than structured designs.

Houdek *et al.* (Houdek *et al.*, 1999) conducted an experiment comparing structured and object-oriented methods applied to embedded software systems. The results identified only minor differences in development time and quality of the developed systems.

In (Sharble and Cohen, 1993), one of the few experiments comparing alternative object-oriented design methods is reported. The authors conducted an experiment where they compared a data-driven and a responsibility-driven design method. Two systems were developed based on the same requirements specification – using the data-driven and the responsibility-driven design method, respectively. Structural attribute measures of the two systems were collected and compared. Based on the measured values, the authors suggested that responsibility-driven design produces higher quality software than data-driven design. However, whether the design measures used in this experiment actually measured "quality" was not empirically validated. In other words, the experiment did not involve any direct measurement of external quality attributes.

5.1.3 Quality and Productivity Models for Object-Oriented Software

A large portion of empirical research in the OO research arena has been involved with the development and evaluation of quality models for OO software. The immediate goal of this research is to relate structural attribute measures intended to quantify important characteristics of object-oriented software, such as size, polymorphism, inheritance, coupling, and cohesion to external quality indicators such as fault proneness, change impact, reusability, development effort and maintenance effort.

The main motivations are to be able to assess quality early on in the software life cycle and to be able to use structural attribute measures as surrogates for external software quality attributes. This would greatly facilitate technology assessment and comparisons, e.g., in studies such as (Sharble and Cohen, 1993). Indeed, this is also the main motivation for the measurement framework proposed in Chapter 6.

5.1.3.1 Fault Proneness

Briand *et al.* (Briand *et al.*, 1999d; Briand *et al.*, 2000) report that highly accurate predictive models of fault-prone classes can be developed based on various

dimensions of coupling in OO systems. The same data suggests that current measures of cohesion, including Lack of Cohesion in Methods (LCOM) (Chidamber and Kemerer, 1994), are poor indicators of fault-proneness.

In (Basili *et al.*, 1996b), all CK measures except LCOM seem to be useful for predicting class fault-proneness during high- and low-level design phases.

In an investigation of a large object-oriented software system, Cartwright and Shepperd (Cartwright and Shepperd, 2000) found that classes participating in an inheritance structure was approximately three times more defect-prone than classes that did not.

Benlarbi and Melo (Benlarbi and Melo, 1999) conclude that polymorphism may increase the fault-proneness of OO software. However, as discussed below, these results should be interpreted with care as fundamental flaws undermine the analysis.

5.1.3.2 Change Impact

A few studies have shown that object-oriented design measures can be used to predict the impact of changes (Briand *et al.*, 1999e; Chaumun *et al.*, 2000). Both studies show high correlation between change impact (for example, quantified as the number of classes that require some sort of modification as a result of a change) and various types of coupling between classes. Furthermore, the results in (Chaumun *et al.*, 2000) do not support the hypothesis that the depth of the inheritance tree influence the change impact.

5.1.3.3 Effort

One frequently referred early paper investigating how structural attributes of object-oriented software affect maintenance effort is (Li and Henry, 1993). In this study, the number of lines of code changed per class was the dependent variable. Thus, an assumption was made that the number of lines of code changed is an indicator of the maintainability of the class. However, the validity of this assumption, i.e., the actual relationship between the change volume and change effort, has not been established empirically. Most of the CK measures were the independent variables. The study showed that the addition of CK measures other than size measures improved the prediction accuracy for the number of lines of code changed per class.

More recent studies have attempted to evaluate development effort prediction models at the class level (Nesi and Querci, 1998; Briand and Wust, 1999). In both studies, measures of class size are significant explanatory variables of development effort. Only limited improvements in effort estimation accuracy was obtained by also including coupling measures (Briand and Wust, 1999).

Binkley and Schach collected maintenance data from four development projects written in COBOL, C, C++ and Java, respectively (Binkley and Schach, 1998). The results suggest that a significant impediment to maintenance is the level of interaction (i.e., coupling) between modules. Modules with low coupling were subjected to less maintenance effort and had fewer maintenance faults and fewer run-time failures.

In (Chidamber *et al.*, 1998), an exploratory case study indicated that high coupling between objects (high value of *CBO*) and low cohesion (high value of *LCOM*) were associated with lower productivity, greater rework and greater design effort.

In (Daly *et al.*, 1996), an experiment was conducted to evaluate the effects of inheritance depth on the maintainability of OO software. The results suggest that systems with approximately three levels of inheritance may result in reduced time required to perform maintenance tasks by 20% compared with no use of inheritance. However, results from the same study indicate *decreased* maintainability at five levels of inheritance depth.

Harrison *et al.* (Harrison *et al.*, 2000) describe an empirical investigation into the modifiability and understandability of object-oriented software. The results indicate that the systems without inheritance were easier to modify than the corresponding systems containing three or five levels of inheritance. Also, it was easier to understand the system without inheritance than a corresponding version containing three levels of inheritance. The results also indicate that larger systems are equally difficult to understand whether or not they contain inheritance.

A model for effort prediction of the adaptive maintenance of object-oriented software was presented in (Fioravanti *et al.*, 1999). A selection of metrics for effort estimation was applied to a general model for evaluating maintenance effort. The validation showed that some (rather complex) object-oriented metrics might be profitably employed for effort estimation.

5.2 Success Factors for Empirical Studies of Object-Orientation

A large number of empirical studies have been undertaken in the past. The need for meta-analysis of the results of such studies underlies a number of inherent problems in the way that empirical studies have been carried out. For example, if an experiment is carried out, and the results are interesting, it would be useful for other researchers to replicate that experiment as closely as possible. The research reaches a dead end if this is impossible, thereby blocking the path to what may be more interesting and insightful research. Consequently, a number of what could be called *success factors* (broad-based factors) which create the conditions for a successful empirical study should be identified.

5.2.1 Nature of the Data

By its nature, an empirical study requires the collection, dissemination and analysis of data. In many OO studies, collection of the data alone is problematic. As an example, consider state-of-the-art in cohesion measures (Briand *et al.*, 1998b). The emphasis on cohesion measures seems to have been based on the distribution and use of instance variables of a class. Hence, a class with a single attribute used in all methods of that class is considered cohesive. Alternatively, a class with ten attributes, each of which is used in only one method is considered lacking in cohesiveness.

The problem with such proposed metrics is the following: most are theoretically flawed either because they produce meaningless and incomparable values or there are counter-examples that render the metric inadequate. Many measures that are theoretically sound are computationally intensive or unsupported by tools to aid the collection.

One success factor is therefore to ensure in any empirical study that quality of data collection is maintained. This entails ensuring reliability, completeness and efficiency

of the data collected. Furthermore, although no measurement is perfect, it must be clearly justified and its underlying assumptions must be made explicit to aid the interpretation of the results. Since the replication of studies is the key to successful empirical research, it is also crucial that any measurement reported is defined in an operational and unambiguous manner.

5.2.2 Consistent Terminology

A common problem in the software engineering community is the inconsistent use of terminology and notation. A good example of this is in the use of the terms *analysis* and *design* for OO development. These two terms are used interchangeably. The empirical studies community is no different in this respect; arguments on the correct approach to, and use of, measurement theory in empirical research have only started to abate recently. There have also been questions raised as to the exact meaning of case-studies vs. experiments, the conditions that must hold for research to be claimed to be based on either, and hence whether the results are credible.

Another example of misuse of terminology in the OO world regards the term coupling. Many forms of coupling can arise in OO systems. Establishing what the different forms of coupling are and which are most harmful, is still an open research question, although some work has already been done in this area (Briand *et al.*, 1999c).

5.2.3 Nature of the Research

In many cases, empirical research is undertaken without establishing beforehand whether the problem is either worth investigating, is too complex at this stage of knowledge or addresses a different underlying issue. A good example is the study of program complexity, where the goals and the concepts under study have lacked clarity (Briand *et al.*, 1996b). There were confusion regarding the practical software engineering problems to be addressed by the research and regarding the meaning of complexity. Successful empirical research seems to require a clearly defined problem, the results of which are useful to the community and that addresses the problem head-on.

5.3 Research Directions

This section first discusses objectives for further research in the field of empirical studies of object-oriented software development. Then, a non-exhaustive list of open research questions matching those research objectives are described.

5.3.1 Objectives

The overall objective of empirical studies of object-oriented technologies and products is to gather tangible evidence about its properties and gain deeper insights into the nature of the object-oriented paradigm and its relationship to other approaches. More precisely, we have identified four interdependent goals.

Identify important productivity and quality factors. The performance and quality of object-oriented technologies and products may depend on many factors, e.g.,

training and support tools. It is a prerequisite for valid empirical research to know about these factors and control for them. Without such control, research results can only be questionable in the sense that they may be due to other causes than the ones hypothesized. Relevant factors may be related to human factors (e.g., staff experience and training), development processes (e.g., time pressure and methodology) and the product itself (e.g., class coupling and depth of inheritance hierarchy). However, there may also be dependencies between factors (e.g., staff experience may influence the use of inheritance and therefore the depth of inheritance hierarchy). A lot of the work on object-oriented measures can be seen as contributions to this goal (see Section 5.1) as it helps understand what product characteristics make the software fault-prone or expensive to develop.

Evaluate OO technologies: Alternative OO technologies (e.g., methods and tools) should be assessed and compared. In order to determine the external validity of the obtained results, the studies should specify very clearly the context in which such an evaluation is performed, i.e., characterizing and controlling influential factors.

Building quality and productivity models. Important factors may be used as independent (explanatory) variables in quality or productivity models. Quality and productivity, regardless of the specific way they are measured, are then the dependent variables of the models. In other words, the relationships between independent and dependent variables can be explored and modeled. The resulting models form an essential input to plan, control, or evaluate processes and products. For example, these models can be used to trigger defect-detection activities on specific parts of a system, support impact analysis during maintenance, or devise design guidelines.

Meta-analysis. In the current literature, there are many instances of research contributions that do not specify clearly the goal of the research. It is often difficult to draw any useful, tangible conclusion. In addition, the setting of the study and the characteristics of the data collected are most of the time superficially described. These problems limit our capability to generalize conclusions to other settings.

5.3.2 Research Questions

The list of potentially relevant research questions seems endless. We will provide some of the most important ones below. They are based on our experience and the discussions that took place during the workshop. We will group the questions according to the goal structure proposed above.

5.3.2.1 Identify Important Factors

- One important part of determining influential factors is to perform proper measurement. Several useful measurement frameworks have been provided in the recent past. However, the proposed metrics are mainly related to static aspects of object-oriented systems. Measurement of dynamic attributes (e.g., coupling at run-time) has not yet been considered in depth.
- Measurement of single classes has been investigated to a great extent (see references in Section 5.1). But typical systems are not built as a collection of

single, independent classes but from a collection of class clusters (e.g., when using COTS products). Therefore, measuring only single classes is not sufficient; clusters of classes are also of interest. To date, it is unclear how to transfer the existing measurement techniques from single classes to class clusters.

- The number of measures that have been proposed for object-oriented products is very large. And yet every conference in this field proposes new ones. At this point, there needs to be a shift of effort from defining new measures to investigate their properties and applications on replicated studies. We need to better understand what these measures are really capturing, whether they are really different, and whether they are useful indicators of quality or productivity properties of interest. The need for new measures will then arise from, and be driven by, the results of such studies, cf. Sections 7.2 and 7.3.
- The application domain is usually seen as a major factor determining the usefulness of measures or the effectiveness of technologies. Application domains may be defined in different ways, but we mean here to characterize the type of functionality delivered, the type of development technologies used, the scale of development, the level of complexity of products, and any other characteristic that is typically associated with a domain.

5.3.2.2 Evaluation of Object-Oriented Technologies

- The Unified Modeling Language (Booch *et al.*, 1998) (UML) is now becoming a de-facto standard; it is important to investigate its use more thoroughly. It is, for example, possible that a subset of the notation could be used more efficiently (the UML being a quite large set of notations at the moment). Also, certain parts of the formalism may lead to confusion and need more precise semantics.
- Different tools are available to support UML-based development. They need to be evaluated and their prerequisite for successful use must be investigated. Many studies have shown that the success of the use of CASE tools is driven by extraneous factors, e.g., training (Bruckhaus *et al.*, 1996).
- OO development processes are also proposed such as the Rational Unified Process (Jacobson *et al.*, 1999). The introduction of such processes should be carefully monitored since it is likely that they will require some degree of tailoring in each organization.

5.3.2.3 Building Quality and Productivity Models

- Most quality models reported in the literature are based on measurements that can only be obtained at late stages, e.g., detailed design or coding. Quality models need to be available earlier in the life cycle in order to, for example, drive inspections and ensure early built-in quality.
- Certain aspects of quality have been subject of very little research. For example, although testability is recognized as an issue in object-oriented development, it rarely been addressed in the research community. Different aspects of maintenance have also been subject of little attention, for example, factors that affect design and code comprehension (von Mayrhauser *et al.*, 1997), or impact analysis (Briand *et al.* 1999e).

- There are studies trying to relate productivity or cost to product characteristics (Chidamber *et al.*, 1998; Nesi and Querci, 1998). But these studies are scarce and use information that can only be obtained in the late stages of design. We need productivity models that can be used earlier on in the course of development, so that accurate project planning and risk analysis can be performed. This may not be solved only through new measurement models and empirical studies, but also by the selection of a development process that documents requirements, specifications, and design decisions early on.

5.3.2.4 Meta-level issues

- Most studies of OO processes and products are exploratory in nature. In order to interpret quantitative results coming out of exploratory empirical studies, qualitative methods should be used. Without such techniques, the conclusions based on the quantitative data may be invalid (cf. Section 7.1.1). For example, it is rarely the case that quantitative results are interpreted by performing structured interviews of the study participants, administering a debriefing questionnaire, or organizing feedback sessions with the development teams.
- In order to build a body of evidence from a set of empirical studies, results need to be combined and conclusions need to be generalized. This is the field of meta-analysis, which is well developed in other fields such as medicine. However, to allow for meta-analysis, software engineering studies need to be better reported. Important details are often missing from research papers or case study reports. A typical example is that, although significance levels of the effect of an independent variable on the dependent variables are usually reported, the size effect is almost systematically missing (Pickard *et al.*, 1998).

5.4 Summary

There is no simple answer regarding the use and performance of object-oriented technologies. Although they are changing the face of software development, they do not bring the unconditional improvements that were promised or hoped for. From a more general perspective, the software engineering community seems to have followed, once again, the traditions of the past: technology adoption is mostly the result of marketing forces, not scientific evidence.

In addition, many different choices can be made in terms of which processes to follow, which tool support to employ, or which languages and notations to use. It is very likely that various OO technologies will show different properties, advantages and drawbacks. An overview of existing empirical studies aimed at empirically evaluating such processes was provided in Chapter 4. We need to better understand them.

Object-oriented code and design measures have been used to empirically assess how internal characteristics of object-oriented systems affect developer productivity and product quality. A large amount of work has focused on understanding how the quality of OO artifacts (e.g., design and code) could be assessed. External quality attributes, such as maintainability and reliability, can only be measured late in the

software life cycle. We therefore need to identify early indicators of such qualities based on, for example, the structural properties of artifacts. Existing data suggests that there are important relationships between structural attributes and external quality indicators. For example, there is some evidence that some forms of coupling have a negative impact on fault proneness. However, in general there exists insufficient empirical evidence supporting the usefulness of a vast number of proposed object-oriented measures.

The goals or hypotheses underlying many of the proposed measures are often not clearly stated. The measures are often not defined in a fully operational form, making it difficult to replicate the studies by other researchers. Even among the measures that *have* been properly defined and validated empirically, external factors (such as architectural design, developer experience, tools, software engineering processes, organizational maturity, etc.) may limit the external validity of the results. Consequently, work is required to replicate existing studies in different development environments. This would allow us to better understand how to build quality models under a variety of circumstances, i.e., understand what relationships between early indicators and external quality attributes are stable and what makes them vary across environments.

When the internal product measures are properly defined and implemented, and after extensive empirical validation against externally observable quality indicators, the use of the internal product measures may eventually provide a common, validated measurement framework, which would allow researchers and industry

- to evaluate and improve the efficiency of OO technologies, and
- to evaluate and improve the quality of the resulting OO software products.

6 Measuring Changeability

This chapter describes a measurement framework for assessing changeability in object-oriented software. Some of the material has been published in (Arisholm and Sjøberg, 1999; Arisholm and Sjøberg, 2000). However, the measurement framework described in this chapter has matured as a result of the experiences with its use and validation (Chapter 7). The intended use of the measurement framework is:

- To compare the changeability of alternative implementations of a software system, *s1* and *s2*.
- To assess *changeability decay* of a software system, i.e., predicting whether the changeability of a later version (e.g., *v2*) of a software system is decayed compared with an earlier design (e.g., *v1*) of a software system.

Both of these potential applications of the framework were motivated by the discussion of evolutionary development processes (Chapter 4). The comparison of alternative implementations of a software system may be important to choose an implementation with the most "open ended" design, supporting the frequent changes during evolutionary development of software systems. The assessment of changeability decay may, for example, be used as an instrument to determine factors of changeability decay in evolutionary development, such that preventive guidelines can be developed.

In both cases, the framework attempts to determine which of the software systems has better changeability. Thus, in the following descriptions, we will simply refer to software systems *s1* and *s2,* regardless of whether *s2* is a later version or an alternative implementation of *s1*. When comparing the changeability of *s1* and *s2*, the framework predicts the *difference* in *change effort* between *s1* and *s2* for a given set of changes. This predicted difference in change effort is used as an indicator of the difference in the changeability of the systems. Thus, the framework attempts to operationalize the definitions of changeability (Chapter 2).

The remainder of this chapter is organized as follows. Section 6.1 gives a short overview of the measurement approaches. Sections 6.2, 6.3 and 6.4 describe the three different approaches to measuring changeability in more detail. Section 6.5 compares the approaches in terms of accuracy, cost and practical use. Section 6.6 describes issues and methods for empirical validation. Section 6.7 summarizes and describes related work.

6.1 Overview of the Measurement Approaches

The proposed framework has three alternative approaches to measuring changeability: *Structural Attribute Measurement (SAM),* C*hange Profile Measurement (CPM)* and *Benchmarking.* Figure 6.1 depicts an overview of the measurement approaches.

Structural Attribute Measurement (SAM) quantifies the structure of the software and uses the obtained values in a prediction model, which can be used as an indicator

of changeability. The proposed *Change Profile Measurement* (CPM) combines structural attribute measures with measures of the actual changes on the software. It attempts to quantify some dimensions of "complexity" of the actual changes carried out instead of the "complexity" of the overall system structure. We believe that change profile measurement is a more accurate indicator of changeability than structure measurement, because, unlike structure measurement, it accounts for how changes propagate through the software structure.

Both CPM and SAM only indicate how structural properties affect changeability. However, in addition to the impact of the structural properties, other aspects (e.g. inconsistent documentation) may affect changeability. Thus, as an alternative approach, we propose using *benchmarks* where change effort can be measured more directly. Benchmarking can be used to determine the total effort to implement a given collection of "benchmark changes" on *s1* and *s2*. Implementing the same changes on *s1* and *s2* provides the necessary baseline that ensures that change efforts can be compared directly.

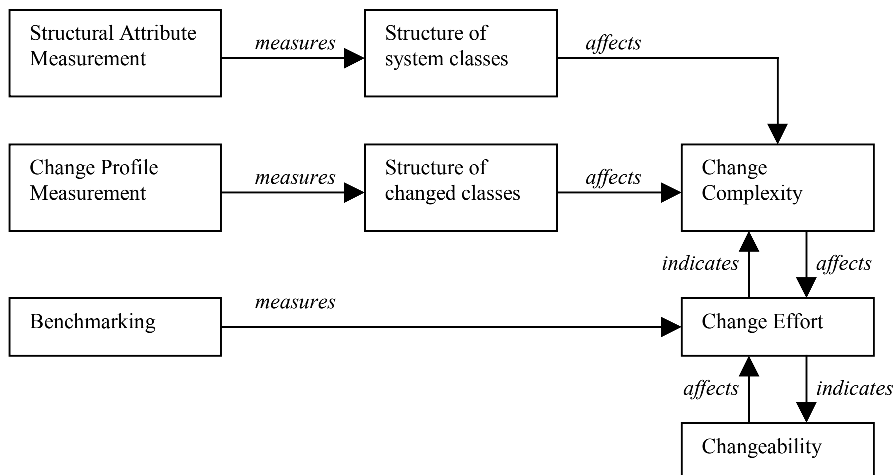

Fig. 6.1. Simplified view of relationships between the assessment framework and changeability

6.2 Structural Attribute Measurement (SAM)

It is commonly believed that a deteriorated structure has a significant negative impact on changeability. There is a growing body of results indicating that measures of structural attributes such as class size, coupling, cohesion, inheritance depth, etc. can be reasonably good predictors of development effort and product quality (Li and Henry, 1993; Chidamber and Kemerer, 1994; Basili *et al.*, 1996b; Brito e Abreu and Melo, 1996; Harrison *et al.*, 1998b; Briand *et al.*, 2000). Further details were provided in Chapter 5. Thus, it is conceivable that such structural attribute measures can be used as indicators of changeability.

6.2.1 Selection of Structural Attributes

Only a few and relatively simple measures that capture some important and intuitive dimensions of an object-oriented structure have been selected as changeability indicators: "coupling" quantifies interclass dependencies; "class size" and "method count" are supposed to indicate the amount of functional responsibility of a class. In theory, low coupling and small class size may reflect an object-oriented design with good functional responsibility alignment among classes, which in turn may affect the changeability of the software system.

It is commonly believed that size is a major contributor of "complexity". Two dimensions of the overall system size are measured: System Size (*SS*) and Class Count (*CC*). Furthermore, two measures of the class size are measured: Class Size (*CS*) and Method Count (*MC*).

There are several good reasons for using existing structural attribute measures instead of inventing new ones (Chapter 5). However, the current state-of-the-art indicates that it is premature to select only one type or dimension of coupling (Briand *et al.*, 1999b). Consequently, we are investigating several dimensions of coupling, in particular the static, class level coupling measures defined in (Briand *et al.*, 1997b) (at present adapted to Java and Visual Basic), as well as dynamic import and export coupling measures (at present adapted to SmallTalk). Note that the coupling measures in Table 6.1 (*IC* and *EC*) refer to all of these import coupling and export coupling measures, respectively.

Other structural properties related to, for example, cohesion and inheritance could also have been considered. However, based on the existing empirical results, size and coupling seem to be the most consistent predictors of changeability. Class size and coupling affect fault proneness, productivity, development costs, and change impact (Li and Henry, 1993; Briand *et al.*, 1997b; Binkley and Schach, 1998; Chidamber *et al.*, 1998; Briand *et al.*, 1999a; Briand and Wust, 1999; Chaumun *et al.*, 2000). The impact of cohesion is less understood and it is more difficult to measure precisely (Briand *et al.*, 1996b; Briand *et al.*, 1998b). Several of the studies reported in Chapter 5 showed that inheritance may be a serious hindrance to aspects related to changeability, such as fault proneness and maintainability (Daly *et al.*, 1996; Cartwright and Shepperd, 2000; Harrison *et al.*, 2000). However, the use of inheritance in many object-oriented systems is limited (Harrison *et al.*, 2000). The same observations were made in the systems studied in this thesis. Consequently, we have chosen to focus our investigation on coupling and size. Future extensions of the framework should clearly also consider cohesion and inheritance.

Table 6.1. Summary of Proposed Structural Attribute Measures

Name	Definition	Description
Class Count	CC	Total number of implemented (non-library) classes in the system
Class Size	CS(c)	Class size is measured as the number of Source Lines Of Code (SLOC) for the class c
System Size	$SS = \sum_{i=1}^{CC} CS(c_i)$	System size is defined as the sum of the class sizes for the total number of implemented (non-library) classes in the system
Method Count	MC(c)	Method count is defined as the number of implemented methods in a class c. A formal definition is provided in (Briand *et al.*, 1999c)
Import Coupling	IC(c)	The class level import coupling measures defined in Tables 6.2 and 6.5
Export Coupling	EC(c)	The class level export coupling measures defined in Tables 6.3 and 6.5

6.2.2 Specification of Static Coupling Measures

The selected static coupling measures distinguishes between many dimensions of coupling. A precise mathematical definition and justification for these dimensions is given in (Briand *et al.*, 1997b). Tables 6.2 and 6.3 summarize the measures. The dimensions and the notation of the measures are given below:

- Type of class: Other classes (O*xxxx*), Ancestors (A*xxxx*) or Descendents (D*xxxx*)
- Type of interaction: Method-Method interactions (*x*MM*xx*) or Method-Attribute (*x*MA*xx*) interactions
- Direction: Import Coupling (*xxx*IC) or Export Coupling (*xxx*EC)
- Stability of server: non-library (*xxx*IC) versus library (*xxx*IC_L)

The coupling measures count every statically distinct class-level interaction. For example, if a method *X* is called twice (in two different places) from class *A* to class *B*, the method-method import coupling of class *A* is incremented by 2 and the method-method export coupling of class *B* is incremented by 2.

Table 6.2. Summary of static import coupling measures

Measure Name	Description
OMMIC(c)	Number of static method invocations from a class c to non-library classes not within the inheritance hierarchy of c
OMMIC_L(c)	Number of static method invocations from a class c to library classes
OMAIC(c)	Number of direct accesses by class c to attributes defined in non-library classes not within the inheritance hierarchy of c
OMAIC_L(c)	Number of direct accesses by class c to attributes defined in library classes not within the inheritance hierarchy of c
AMMIC(c)	Number of static method invocations from a class c to non-library ancestor classes of c
AMMIC_L(c)	Number of static method invocations from a class c to library ancestor classes of c
AMAIC(c)	Number of direct accesses by class c to attributes defined in non-library ancestor classes of c
AMAIC_L(c)	Number of direct accesses by class c to attributes defined in library ancestor classes of c

Table 6.3. Summary of static export coupling measures

Measure Name	Description
OMMEC(c)	Number of static method invocations to a class c from non-library classes not within the inheritance hierarchy of c
OMAEC(c)	Number of accesses to attributes defined in class c by non-library classes not within the inheritance hierarchy of c
DMMEC(c)	Number of static method invocations to methods implemented in class c from descendants of c
DMAEC(c)	Number of accesses to attributes defined in class c by descendants of c

The method-attribute (MA) coupling measures are not described in Briand's coupling framework. In one of the case studies conducted in this thesis, a considerable amount of coupling due to the direct access to public attributes in other classes (e.g., OMAIC) and access to protected attributes of ancestor classes (e.g., AMAIC) were found. Thus, the proposed method-attribute interaction measures extend Briand's framework.

6.2.3 Specification of Dynamic Coupling Measures

The current knowledge regarding dynamic coupling in general, and object-oriented systems in particular, is very limited. Results from the empirical studies in Chapter 7 motivated the specification and preliminary validation of dynamic coupling in object-oriented systems. Amongst others, the dynamic coupling measures may be better predictors of the understandability in object-object oriented software than the static coupling measures.

The dynamic coupling measures proposed in this section quantify different dimensions of dynamic collaboration between entities. A convenient way to describe these dimensions is through the concept of role-models (Reenskaug *et al.*, 1995). A *scenario* is a part of the system implementing a given function or task. A *role* is an abstract representation of the functional responsibility of an entity (i.e., a class or an object) in a given scenario. A *role-model* is a representation of the interaction between roles in a functional scenario. Thus, an entity can have many roles because it may participate in many scenarios. Within one scenario, the role-model reflects the dynamic coupling between the roles along several orthogonal dimensions. These dimensions are *direction*, *mapping* and *strength*.

6.2.3.1 Direction of Coupling: Import and Export Coupling

One may distinguish between import coupling and export coupling. Dynamic import coupling counts the messages sent from a role (in which the role acts as a client) whereas dynamic export coupling counts the messages received (in which the role acts as a server). For example, if role *A* sends a message to role *B*, then the message contributes to import coupling for role *A* and export coupling for role *B*.

6.2.3.2 Mapping: Object-level and Class-level Coupling

Roles are only an abstract representation of the responsibilities of the entities collaborating to implement a given functional scenario. Roles are ultimately mapped to object-oriented code. Messages may be "understood" through methods defined within an object's class itself as well as through reference to methods inherited from ancestor classes. Thus, a role is mapped to only one *object* but (potentially) many

classes: a distinction is made between the *objects* sending and receiving the messages and the *classes* that actually implement the methods.

Dynamic, object-level coupling quantifies the extent to which messages are sent and received between the objects in the system. Dynamic, class-level coupling quantifies the extent of method interactions between the classes implementing the methods of the caller object and the receiver object. Due to inheritance, the class of the object sending or receiving a message may be different from the class implementing the corresponding method. For example, let object *a* be an instance of class *A,* which is inherited from ancestor *A′*. Let *A′* implement the method *mA′*. Let object *b* be an instance of class *B,* which is inherited from ancestor *B′*. Let *B′* implement the method *mB′*. If object *a* sends the message (i.e., calls the method) *mB′* to object *b*, the message may have been sent from the method source *mA′* implemented in class A' and processed by a method target *mB′* implemented in class *B′*. Thus, in this example, the message passing caused two types of coupling: object-level coupling between class *A* and class *B,* and class-level coupling between class *A′* and *B′*.

6.2.3.3 Strength of Coupling

The strength of coupling quantifies the amount of association between the roles. The amount of association between roles may be quantified in at least three levels of granularity:

Number of dynamic messages. Within a run-time session, it is possible to count the total number of times each message is sent from one role to another to implement a certain functional scenario. In the scenario depicted in Figure 6.2, *A* sends a total of four messages {*m1B, m1B, m1C, m2B*} and receives one message {*m1A*}. Thus, at the *dynamic message* granularity level, *A* has import coupling 4 and export coupling 1 (Table 6.4).

Number of method invocations. An alternative is to count the number of distinct method invocations between two roles. In Figure 6.2, *A* sends four messages using three different methods {*m1B, m1C, m2B*}. Thus, at the *method invocation* granularity level, *A* has import coupling 3 (Table 6.4).

Number of associations. Two roles are associated if they exchange one or more messages to implement the given scenario. In Figure 6.2, *A* sends messages to *B* and *C* and receives messages from *C*. Thus, at the *association* granularity level, *A* has import coupling 2 and export coupling 1 (Table 6.4).

Table 6.4. Summary of coupling measures from the example scenario in Figure 6.2

	Role A		Role B		Role C	
	Import	Export	Import	Export	Import	Export
Dynamic Messages	4	1	1	3	1	2
Method Invocations	3	1	1	2	1	1
Associations	2	1	1	1	1	1

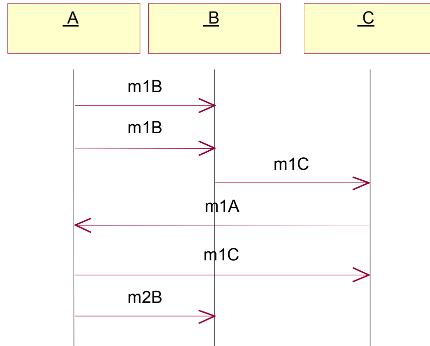

Fig. 6.2. Example message interaction diagram between roles *A*, *B* and *C*

6.2.3.4 Resulting Coupling Measures

In the preceding sections, three orthogonal dimensions of dynamic coupling were described:

- *Direction of coupling:* Import Coupling (*IC_xx*) versus export coupling (*EC_xx*)
- *Mapping:* Object-level (*xx_Ox*) versus class-level (*xx_Cx*) coupling
- *Strength of coupling:* Number of dynamic messages (*xx_xD*) versus number of distinct method invocations (*xx_xM*) versus number of distinct associations (*xx_xA*)

These dimensions define 12 different dynamic coupling measures, summarized in Table 6.5. A procedure for collecting the dynamic coupling measures is described in Section 7.3.

Table 6.5. Summary of the dynamic coupling measures

Direction	Mapping	Strength	Name
Import **C**oupling	**O**bject-level	Number of **D**ynamic messages	IC_OD
		Number of **M**ethod invocations	IC_OM
		Number of **A**ssociations	IC_OA
	Class-level	Number of **D**ynamic messages	IC_CD
		Number of **M**ethod invocations	IC_CM
		Number of **A**ssociations	IC_CA
Export **C**oupling	**O**bject-level	Number of **D**ynamic messages	EC_OD
		Number of **M**ethod invocations	EC_OM
		Number of **A**ssociations	EC_OA
	Class-level	Number of **D**ynamic messages	EC_CD
		Number of **M**ethod invocations	EC_CM
		Number of **A**ssociations	EC_CA

6.3 Change Profile Measurement (CPM)

The proposed *change profile measurement* is a combination of structure measurement and measures of the actual changes carried out during a development project. It measures properties of the change itself, as well as structural attributes of those parts of a software system that are affected by that change. Thus, it attempts to measure some dimensions of "complexity[2]" of the actual changes carried out instead of the "complexity" of the overall system structure. The hypotheses underlying this approach are:

H1. Changes affecting large classes or classes with high coupling result in higher change complexity (which in turn affects the change effort) than changes affecting small classes or classes with low coupling.
H2. Classes that are *not* changed contribute *less* to the change complexity than classes that *are* changed.
H3. Not all classes are changed the same amount. In particular, there may be a trend towards some classes becoming more change prone whereas other classes become less change prone. An example is the development of a combination of classes that eventually become a "framework" of stable classes. Thus, although the overall structural attributes of the software (as reflected by the SAM approach) may, for example, remain more or less constant, there may still be a positive or negative trend in changeability.

For hypothesis *H1*, the studies outlined in Chapter 5 described many instances in which class-level structural attributes, such as coupling and size, affected fault proneness, ripple effects, and development effort. Thus, it is plausible that the structural attributes of classes being changed also affect the change effort. Hypotheses *H2* and *H3* state that the CPM approach is a better way of measuring the *impact* of structural attributes on change complexity at the *system* level than the SAM approach. However, it is important to note that the SAM approach may also be useful: *H2* does not rule out that classes that are *not* changed may also contribute to change complexity. A quite obvious example is that changing one class with coupling=*X* and size=*Y* in a system involving 100 classes may still be more difficult than changing a class with the same level of coupling (*X)* and size (*Y)* in a system with only 10 classes. Thus, assuming that H1–H3 are valid, SAM and CPM are complementary: the combined set of SAM and CPM measures may provide a better indication of the changeability of object-oriented software than when using only one approach, i.e., SAM *or* CPM.

Table 6.6 describes the proposed measures in some detail. The main idea is to consider how changes propagate through the various classes in the software structure. For each class affected by a change, the proportion of work carried out on that class is recorded. This measurement is called the "change profile" (*CP*). The structural attributes "class size" (*CS*), "import coupling" (*IC*), "export coupling" (*EC*) and "method count" (*MC*) for those classes affected by the change are also measured. By

[2] According to (Fenton, 1992), *"It is counter-productive to insist on equating measures of specific (and often important) structural attributes with the poorly understood attribute of complexity".*

using the class level change profile as a weighting factor on the structural attribute measures, we obtain the "change profile measures" *CSCP*, *MCCP*, *ICCP* and *ECCP* for a given change.

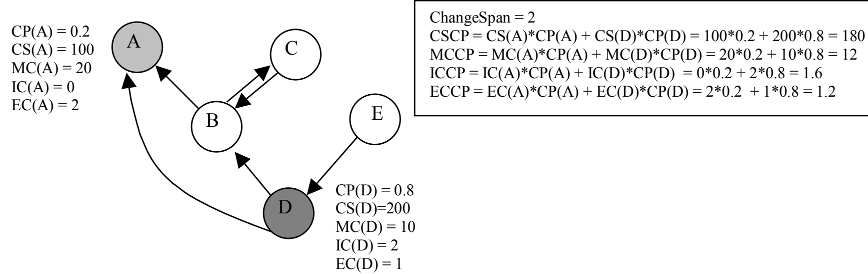

Fig. 6.3. Change profile measures for a given change affecting classes *A* and *D*.

Figure 6.3 depicts a hypothetical change affecting classes *A* and *D* and the resulting change complexity measures. In this figure, the nodes represent classes and the edges represent static method invocations from a client class to a server class. Example values for the structural attribute measures (*CS*, *MC, IC* and *EC*) and the change profile (*CP*) for class *A* and *D* are provided together with the resulting change complexity measures *CSCP*, *ICCP*, *ECCP* and *MCCP*.

Table 6.6. Summary of Proposed Change Profile Measures

Name	Definition	Description
Change Profile	$CP(c) = \dfrac{SLOCAdd(c) + SLOCDel(c)}{\sum_{i=1}^{CC} SLOCAdd(c_i) + SLOCDel(c_i)}$	The proportion of the total amount of changes (in SLOC added and deleted) on class c for a given change to the software system. CC is the Class Count measure defined in Table 6.1
Change Span	ChangeSpan	Number of classes changed for a given change to the software system
Class Size Change Profile	$CSCP = \sum_{i=1}^{CC} CS(c_i) \times CP(c_i)$	Class size weighted by the change profile for a given change to the software system. CS is the Class Size measure defined in Table 6.1
Method Count Change Profile	$MCCP = \sum_{i=1}^{CC} MC(c_i) \times CP(c_i)$	Method count weighted by the change profile for a given change to the software system. MC is the Method Count measure defined in Table 6.1
Import Coupling Change Profile	$ICCP = \sum_{i=1}^{CC} IC(c_i) \times CP(c_i)$	Import coupling weighted by the change profile for a given change to the software system. IC is any of the Import Coupling measures outlined in Tables 6.2 or 6.5
Export Coupling Change Profile	$ECCP = \sum_{i=1}^{CC} EC(c_i) \times CP(c_i)$	Export coupling weighted by the change profile for a given change to the software system. EC is any of the Export Coupling measures outlined in Tables 6.3 or 6.5

6.4 Benchmarking

An intuitively appealing approach for the assessment of changeability is *benchmarking*. A given collection of "representative changes" *c* are implemented on different versions of the software, *s1* and *s2*. The resulting change efforts, *e1* and *e2,* respectively*,* are recorded. Hence, our operational definition of changeability is reflected in this approach.

Some related work exists where benchmarking was used to evaluate the efficiency of different development tools by implementing the same changes with different tools (Jørgensen *et al.*, 1995; Sjøberg *et al.*, 1996). The effort required to implement the small, generic changes using the different tools was then used as an indicator of tool efficiency. In the approach proposed in this paper, the changes, the tools, the developers and the software system are fixed; only the software version varies.

6.4.1.1 Design of a Benchmarking Procedure

Performing a benchmark requires a specific benchmarking procedure to ensure accurate and reliable results. In our case, one must particularly deal with questions related to the learning curve and the skill level of the individuals who perform the benchmark.

There are at least two aspects of learning that need to be considered:

- *Learning the system* – if the versions of the software system have many similarities, most of the development team's initial system comprehension effort will be spent on the version first subjected to the benchmark assessment.
- *Learning the changes* – if a developer implements the same change on two versions of the software system, it is likely that the developer will be more efficient during implementation of the change on the second version.

To deal with this situation, we suggest an experiment where the developer implements the same change only once, while controlling for the differences in individual skill levels, as follows:

Step 1 (skill level assessment). The developers implement a small change on a fictive software system. The effort to implement the change is recorded for each developer.

Step 2 (division in groups).[3] The developers are divided in two groups (*g1, g2*), such that the mean and variance of the change effort data obtained in Step 1 of each group are approximately equal.

Step 3 (benchmarking). All members of group *g1* implement the benchmark on version *s1* of the software system. All members of *g2* implement the benchmark on version *s2*. The individual effort required by each developer to implement the benchmark is recorded.

[3] It is also possible to use randomization without blocking. Then, the skill level assessment can be used to adjust for differences *after* the experiment has been conducted. An example of this alternative approach is described in Section 7.2.

Step 4 (statistical analysis). Version *s1* has better changeability than version *s2* if the mean change effort for group *g1* is significantly smaller than the mean change effort for group *g2*. Assuming a normal distribution, this test can be performed using a two-sample Student's T-test. Otherwise, a non-parametric test such as the mean rank Kruskal-Wallis test can be used.

However, other, simpler experimental designs may be appropriate if one can ignore the learning effect – for example, when the time span between performing the benchmark is large, or when the software systems are sufficiently different. In those cases, each individual developer could implement the *same* benchmark on versions *s1* and *s2*. This design would eliminate the need for the skill level assessment (Step 1), and a paired Student's T-test or a paired Wilcoxon test could be used where each observation is the difference in individual effort, $d = e(s2) - e(s1)$, for each developer.

Although the main dependent variable is the difference in change effort, other dependent variables may also be used to provide a more comprehensive set of changeability indicators. For example, the SAM measures may be used in conjunction with the benchmarking to assess the impact on structural properties by implementing the benchmark tasks (i.e., structural stability). Other examples are provided in Section 7.2.9.

6.4.1.2 Composition of Benchmarks

The benchmark results are only valid for the particular collection of changes given by the benchmark. Thus, it is important that the changes prescribed by the benchmark represent *typical* changes performed on the software product. If benchmarking is performed on the same system from which actual change statistics have been collected, we can use this change statistics to compose a dedicated benchmark that is representative of the actual changes performed on that system. It is obviously a greater, long term, challenge to compose more *general* benchmarks that are representative of changes to different software systems in different application domains.

As a means to collect empirical data to develop representative benchmarks for a specific system, a change data collection process has been developed. It ensures that the developers

- classify all changes and assign a change ID,
- tag each file-level check-in with the correct change ID, and
- report process data (change effort, subjective change complexity, number of discovered faults, etc.) per change.

A change logger tool has been implemented to support this process (Figure 6.4). Note that the resulting data may also be used as a source of actual change effort data used to build prediction models using SAM and CPM. Further details of this change data collection process is provided in Section 7.1.2.

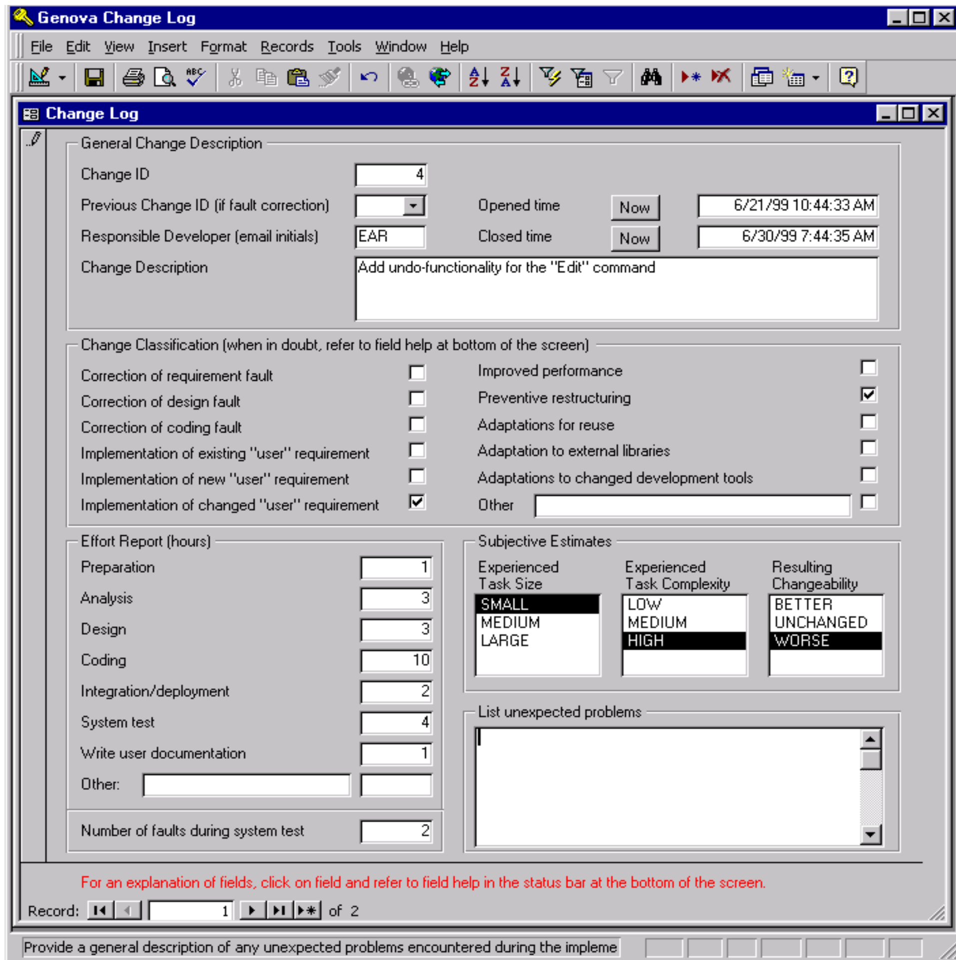

Fig. 6.4. The user interface of the change logger tool, developed by this author

6.5 Relationships between Changeability Measures

This section describes the relationships among the three approaches (structural attribute measurement (SAM), change profile measurement (CPM) and benchmarking) regarding the measurement of changeability. These relationships are explained according to assessment accuracy (Section 6.5.1), assessment cost (Section 6.5.2) and practical use (Section 6.5.3). Validation issues are described in detail in Section 6.6.

6.5.1 Accuracy of the Measurement Approaches

Benchmarking is the most direct way to assess changeability. Unlike SAM and CPM, benchmarking does not rely on an underlying theory relating changeability to structural attributes of software; it measures change effort directly. Therefore, it may account for other factors affecting the changeability of software, such as inconsistent or outdated documentation. One requirement for accurate benchmarking results is that benchmark changes are representative of typical changes. Otherwise, the results may be biased. Selecting such changes is not trivial. However, we believe that the data collection procedure described in Section 6.4.1.2 will provide important insight for the composition of benchmarks.

Our hypothesis is that structure measurement can be used to *indicate* changeability. A common belief is that the structure affects the changeability of software. SAM is intended to measure the structure. Increasing values of the structural attribute measures are thus intended to be indicators of changeability decay. The accuracy of these indicators depends on to what extent structural deterioration, as measured by the structural attribute measures, actually affects changeability.

Change complexity measurement is intended to measure the *complexity* of implementing changes to the software, where "complexity" is reflected by the structural attributes of the parts of the system actually being affected by a given change. We believe that change complexity measurement may be a more accurate indicator of changeability than structure measurement, because, unlike structure measurement, it accounts for how changes propagate through the software structure. This hypothesis must of course be tested empirically.

6.5.2 Cost of the Measurement Approaches

Benchmarking is by far the most expensive approach, as it requires implementation of changes by developers. Change complexity measurement and structure measurement are inexpensive in comparison, since the measures can be collected by semi-automatic data-collection tools if the software system has been subject to version control. Thus, from a cost perspective, structure measurement and change complexity measurement are superior to benchmarking.

6.5.3 Practical Use

Benchmarking does not provide much insight into the *cause* of differences in changeability. Increases in benchmark change effort may be due to, for example,

inconsistent or outdated documentation, or a deteriorated structure. Validating structure measurement and change complexity measurement using benchmarking results may provide further insight into the cause of the observed difference in benchmark change effort. Thus, the approaches can be combined to give complementary indicators of changeability. For example, in Section 7.2, benchmarking is used to assess change effort, whereas SAM is used to assess the structural stability of the designs by implementing the change tasks. Using benchmarking in conjunction with CPM may be used to assess the extent to which the structural attributes of the changed classes explain the observed differences in benchmark change effort.

6.6 Validation Issues

Before the measurement framework can be used to assess changeability, it needs to be validated empirically. There are two important aspects of such a validation:

- A qualitative evaluation of the *practicality* of the proposed methods (Section 6.6.1).
- A quantitative validation of the *accuracy* of the measurement approaches. There are two aspects of this validation. As a first step, the ability of the prediction models to explain the variation in change effort needs to be validated (Sections 6.6.2 and 6.6.3). Furthermore, there is a need to evaluate how good the predictions really are. Our long-term goal is to conduct field experiments in industry, where the predictions using various approaches are compared with actual project change data over long periods of time (Chapter 8).

6.6.1 Evaluation of Practical Issues

The evaluation of the practical issues regarding the proposed framework is qualitative in nature. For example, how easy is it to collect actual logical change data from real development projects? What are the problems involved in building prediction models? What types of statistical techniques are appropriate? When are the different approaches applicable? How difficult or costly are the different approaches when applying them in practice? The research methodology used for such an evaluation is rather *ad hoc*, based on the subjective experience and observations of using the framework. Many such practical issues are described in conjunction with the empirical studies (Chapter 7), and summarized in Chapter 8.

6.6.2 Building Prediction Models Using SAM and CPM

The first step in the validation of the accuracy of the SAM and CPM approaches can be performed by building, for example, regression models (Draper and Smith, 1981; Bølviken and Skovlund, 1994; Christensen, 1996). Alternative modeling techniques, such as pattern recognition, may be more appropriate in some circumstances (Briand *et al.*, 1992; Jørgensen, 1995), but this has not been considered in this thesis. To validate the structural attribute measures with regression, the response variable may be the differences in change effort. The structural attribute measures (Table 6.1) are used (either individually or in combination) as explanatory variables. For example,

using the *differences* in measurement values between *s1* and *s2* as explanatory variables one obtains the following candidate regression model:

$$Effort_{s2,b} - Effort_{s1,b} = b0 + b1*(ChangeSize_{s2,b} - ChangeSize_{s1,b}) + b2*(CC_{s2} - CC_{s1}) + b3*(SS_{s2} - SS_{s1}) + b4*(AvgCS_{s2} - AvgCS_{s1}) + b5*(AvgMC_{s2} - AvgMC_{s1}) + b6*(AvgIC_{s2} - AvgIC_{s1})$$

The subscript *b* on the *Effort* and *ChangeSize* measures refers to the change tasks used in the model validation. The corresponding example regression model for validation of change profile measures (Table 6.6) is:

$$Effort_{s2,b} - Effort_{s1,b} = b0 + b1*(ChangeSize_{s2} - ChangeSize_{s1}) + b2*(ChangeSpan_{s2,b} - ChangeSpan_{s1,b}) + b3*(CSCP_{s2,b} - CSCP_{s1,b}) + b4*(MCCP_{s2,b} - MCCP_{s1,b}) + b5*(ICCP_{s2,b} - ICCP_{s1,b}) + b6*(ECCP_{s2,b} - ECCP_{s1,b})$$

Both models include the difference in *ChangeSize* (total SLOC added and deleted) as a candidate explanatory variable. The SAM and CPM measures are included to determine the extent to which the measures explain additional variance in change effort, beyond what can be explained by *ChangeSize*. Note that the coupling measures can be any number of the ones described in Section 6.2. Interaction terms, higher order terms or other non-linear relationships are not considered at present; the consequential increase in the complexity of the models make interpretation difficult and increases the chance of overfitting the models.

The validation of the SAM and CPM measures can use either actual changes from real projects or benchmark changes as the source of the change effort data for building the regression models. Using benchmark changes has the advantage that the changes are identical (or very similar) on the two designs being compared. Furthermore, confounding factors such as differences in individual skill level can be controlled. When no actual change data is available, e.g., for the evaluation of initial design alternatives, the only option is to use benchmarks. However, using the benchmarks to validate SAM and CPM also assumes that the tasks are representative, which may be difficult if no change history is available. By using actual changes, the changes are certainly representative, but confounding factors may influence the results of the validation attempts. Thus, using actual change effort data as the source of the validation attempts will require a careful analysis of the development project in an attempt to identify factors that may have affected (and somehow biased) the results of the validation. An example is provided in Section 7.1.

By evaluating the explanatory power of the regression models (using, for example, cross-validated R-squared) in combination with the p-values for each regression coefficient (e.g., *b1*, *b2*, *b3*), one can determine to what extent each of the explanatory variables can explain the variance in the difference in change effort. Unfortunately, interpreting software engineering data with regression is far from trivial. The following subsections attempt to describe some of the most relevant issues of the regression analysis that should be considered. Further details can be found in many books discussing regression, cf. (Christensen, 1996).

6.6.2.1 *Stepwise Multiple Linear Regression*

The models in the previous section represent the complete regression model where all measures are used as explanatory variables. In practice, however, some measures may be strongly correlated with each other. Other measures may be uncorrelated with change effort. Thus, variable selection procedures such as stepwise regression can be used to determine a subset of the measures that

- explains a large portion of the variance in the difference in change effort, and
- ensures that the coefficients of each variable are significantly different from zero.

Thus, stepwise regression can be a valuable technique for data analysis, particularly in the early stages of building a model. At the same time, this procedure presents certain dangers:

- Since the procedures automatically "snoop" through many models, the model selected may fit the data "too well." That is, the procedure can look at many variables and select those that, by pure chance, happen to fit well. The stepwise regression procedure is actually testing a large number of hypotheses (depending on the number of candidate variable).
- Stepwise regression often works very well but may not select the model with the highest R-squared value.
- Automatic procedures cannot take into account special knowledge the analyst may have about the data. Therefore, the model selected may not be the best from a practical point of view.

For these reasons, stepwise regression should be used with caution. In particular, to account for the multiplicity of tests, it is important to ensure that the p-values of each coefficient are very low before they are included in the final model. This reduces the risk of committing type I errors (i.e., falsely rejecting the null-hypotheses). For example, with 10 candidate variables, one may consider using the Bonferroni adjustment, and select *alpha/10* as an acceptable level of significance. With the commonly used *alpha=0.05*, the acceptable level of significance could be *alpha/10=0.005*. If many variables are correlated, the individual hypotheses tests are also correlated, in which case the less conservative Holm's multiple test procedure (Holm, 1979) may be used.

6.6.2.2 *Principal Component Analysis*

The mathematical formulation of multiple linear regression assumes that explanatory variables are independent. Thus, preferably, only one variable among variables that are either strongly correlated or can be expressed as linear combinations of other explanatory variables should be included in the final model. One way to determine these subsets of variables is through Principal Component Analysis (PCA).

PCA can be used to analyze the covariance structure of the measures. Based on the PCA, the number of underlying dimensions of the data can be quantified. The number of principal components is usually decided based on the amount of variance explained by each component, using the rule of thumb of eigenvalues (variances) larger than 1.0. To ease the interpretation of the PCA, the Varimax rotation is often used. Based

on the resulting principal components, one may choose to select only one variable from each component as a candidate for the regression model. Such an approach will ease the interpretation of the confidence intervals and the p-values of the coefficients.

6.6.2.3 Cross-Validation

The cross-validated R-squared should be used to evaluate the prediction ability of the model. The usual multiple regression estimate of R-squared increases whenever more parameters are added to the prediction equation. This problem does not occur with the cross-validated R-squared. To calculate the cross-validated estimate of R-squared, the data is split in I subsets. For $i = 1, 2, ..., I$, the least squares fit and the mean are calculated for all cases but the i-th subset. The regression model and the mean are each used to predict the observations in the i-th subset. Note that it is possible for the cross-validated estimate of R-squared to be negative, especially when overfitting, because the regression model is competing with the mean. This indicates that the mean is a better predictor than the regression. For the studies conducted in this thesis, the cross-validated R-Squared was calculated in BLSS, using the number of subsets I equal to the number of data-points n.

6.6.2.4 Checking the Model Assumptions

Regression is a quite robust statistical technique, and small deviations from the underlying assumptions are not critical (Bølviken and Skovlund, 1994). However, gross violations from the assumptions should be detected.

For linear regression, the hypothesis tests on the coefficients are based on a number of conditions:

- The expected error mean must be zero
- Homogeneous error variance
- Uncorrelated errors (e.g., no serial correlation)
- Normally distributed errors

For the resulting models, these assumptions should be checked. One way of performing such model diagnostic is through plots of the residual errors. Examples are provided in Sections 7.1 and 7.3.

6.6.3 Validation of the Accuracy of Benchmarking

The benchmarking technique also needs to be validated. There are several aspects of this validation:

- It involves an assessment of threats to validity caused by the experimental design. For example, does the group assignment affect the results?
- It involves an assessment of threats to validity caused by the experimental material. For example, how representative are the selected change tasks?

Such investigation would ideally involve meta-level experiments where benchmarking is evaluated using different experiment designs, varying the number of subjects, the way subjects are assigned to groups, and the benchmark composition (e.g., the number, type and size of benchmark change tasks).

6.7 Summary and Related Work

This chapter proposed a comprehensive measurement framework describing how object-oriented design measures, change data collection and analysis, measure validation and experimental designs can be combined and used to assess changeability. Furthermore, the concept of changeability was defined in an operational form. CPM and the required change data collection process have, to our best knowledge, not been described before. Furthermore, very little is known about dynamic coupling in general. The empirical validation in Section 7.3 shows that dynamic coupling may be a very useful concept for changeability assessment.

As with any measurement framework, a fundamental requirement for a useful assessment framework is its empirical validity. Many aspects of the framework have been validated in terms of practical aspects and accuracy (Sections 7.1, 7.2 and 7.3). However, much more work is required before one can claim to know exactly what is the most accurate, practical and cost effective way to measure changeability.

The next subsections describe work that is related to the proposed framework described in this chapter.

6.7.1 Object-Oriented Metrics

Within the object-oriented "metrics" community, there is a large amount of work that has proposed object-oriented measures and empirically validated them against external quality characteristics, such as fault proneness or development effort (Chapter 5). Many of these studies are certainly related to different aspects of the framework presented in this chapter. In particular, the SAM approach is based on existing results and many existing measures. As discussed in Chapter 5, there are many reasons to reuse and extend existing measures and results.

6.7.2 SAAM

An alternative approach is the scenario-based methods, such as Software Architecture Analysis Method, SAAM (Kazman *et al.*, 1996). These methods are intended primarily for supporting architecture design decisions during the early phases of development. SAAM consists of four major steps: (1) identify scenarios, (2) describe candidate architectures, (3) evaluate scenarios and (4) carry out an overall evaluation. SAAM is a systematic qualitative method where experts play a central role. An important aspect of the method is the scenario elicitation, aimed to identify those changes that may affect the system. Based on the identified scenarios, the effects of the change scenarios are analyzed. Several aspects of SAAM could probably be integrated into the framework described in this chapter. For example, when change history is not available, the composition of benchmarks could be based on a scenario elicitation process.

6.7.3 COMPARE

A combination of SAAM and quantitative assessment of the architecture was proposed in (Briand *et al.*, 1998a). A recent elaboration of this approach proposes the Change Difficulty Index (CDI), which combines scenario elicitation with structural attribute measures (Briand and Wust, 2000). The specification of the CDI measures

corresponds to the CPM approach described in this chapter. The main difference is that while CDI uses impact analysis techniques based on likely future change scenarios to determine the amount of change to each class, CPM uses the change history as a prediction of future changes. Clearly, the approaches are complementary and could probably be combined to produce better predictions.

6.7.4 TAC++

TAC++ is a method and a tool for the assessment and control of object-oriented projects (Fioravanti and Nesi, 2000). One interesting aspect of TAC++ is the use of histograms and other plots to assess the product at various stages of development. A problem with the SAM approach is the need to compute summary measures at the system-level for the structural attribute measures, by calculating totals or averages. Consequently, a lot of the underlying variability at the class-level is lost. Such information would be available if histograms as in TAC++ could be utilized in the prediction models. One idea could be to use both the average and median values in the prediction models proposed in this chapter. Thus, for each measure, two numbers would be calculated, to capture more of the class-level data distribution.

7 Empirical Studies of Changeability

This chapter describes the empirical studies that have been conducted. The goal of these studies was

- to validate the measurement framework described in Chapter 6, and
- to evaluate evolutionary development projects, focusing on changeability.

The studies consist of case studies and experiments. The data collection and analyses are both quantitative and qualitative.

Section 7.1 describes the results of two case studies. The primary goal of these studies was to validate the SAM and CPM measures proposed in Chapter 6. However, a comprehensive evaluation of the evolutionary development processes used in the studied projects is also provided. This evaluation suggests interesting improvements of evolutionary development processes, focusing on improving the changeability of the software. Furthermore, the detailed evaluations of the processes are important to understand the results of the SAM/CPM measure validation.

Section 7.2 describes the results of using benchmarking to assess the changeability of a given responsibility-driven design versus an alternative control-oriented ("mainframe") design. According to Coad and Yourdon's OO design quality principles (Coad and Yourdon, 1991a; Coad and Yourdon, 1991b), the responsibility-driven design represents a "good" design, with low coupling and small classes. The mainframe design represents a "bad" design, with higher coupling and large classes. To investigate which of these designs have better changeability, we conducted two controlled experiments – a pilot-experiment and a main experiment.

Section 7.3 presents initial ideas and preliminary results on using dynamic coupling measures to assess the effort to understand and implement changes to a scenario. A preliminary empirical validation is provided. The coupling measures are used to build reasonably accurate models for predicting *hot-spots* and *ripple effects* within a given functional scenario.

Section 7.4 describes the results from an interview with four developers. The purpose of this interview was to gain insights into which factors experienced developers consider important to reduce change effort in object-oriented software. The results may be used as an empirical basis for formulating theories from which hypotheses can be tested more formally in future studies, for example through case studies and controlled experiments.

Section 7.5 describes the results from a case study in which evolutionary development was used to develop a telecommunications support system. The study provides insight into the role of end-user participation, documentation and technology risks in evolutionary development projects. These aspects of evolutionary development may influence the changeability of the developed software.

7.1 Changeability in Evolutionary Development Projects

This section describes case studies involving two evolutionary development projects. The goal of the case studies was two-fold:

- One goal was to identify opportunities for process improvements in evolutionary development projects, focusing on improving changeability. To assess the evolutionary development projects, the distribution of effort for various process activities was studied in conjunction with an assessment of rework and distribution of changes.
- Another goal was to validate the Structural Attribute Measures (SAM) and the Change Profile Measures (CPM), proposed in Chapter 6. To validate the measures, SAM and CPM measures were calculated for (1) *weekly* changes in the Braathens case study, and (2) *logical* changes in the Genera case study. These measures were the independent variables in addition to the size of each change (in SLOC added or deleted). The dependent variable was change effort, measured in person-hours for each change.

Results from the development project for the Norwegian airline Braathens are reported in Section 7.1.1. Results from the Genera case study are reported in Section 7.1.2. Section 7.1.3 summarizes. The results of the Braathens case study have been published in (Arisholm and Sjøberg, 1999; Arisholm *et al.*, 1999b; Arisholm and Sjøberg, 2000). The Genera case study incorporates many methodological improvements resulting from the experiences from the Braathens case study.

7.1.1 The Braathens Case Study

7.1.1.1 System under Study

The Braathens case study used an evolutionary development process called the Genova Process, which is quite similar to the Rational Unified Process (Jacobson *et al.*, 1999). A detailed description of the process can be found in (Arisholm *et al.*, 1998; Arisholm *et al.*, 1999b). The development team consisted of four developers and one experienced project manager. The experience level of the developers varied from 1 year to 5 years. The system being studied implemented an automated customer service for Braathens frequent flyer program, "Wings". The system is a three-tier application consisting of Java/HTML clients, a middle-tier component for transaction processing and business logic, and a mainframe database server. The middle-tier module was implemented as classes in Visual Basic 6 and bundled in ActiveX components running on a Microsoft Transaction Server. Data from this middle-tier module was collected based on weekly versions of the software through a 22-week period. After week 22, the system became operational. Three increments were delivered during these 22 weeks, at week 6, 11 and 22, respectively. Weekly effort data in person-hours was available for different activities (analysis, design, code, test and administration) to implement the module and provided the effort data reported in this section.

7.1.1.2 Data Collection

The process was instrumented with a small number of process and product measures. Weekly effort data (in person-hours) for the various activities was recorded by each team member. Coding effort data was reported individually for each module of the system, and was used as the dependent variable in the validation of the collected SAM and CPM measures. Using the configuration management tool and a code parser for Visual Basic, a few SAM and CPM measures were collected from the middle-tier module based on weekly versions of the software throughout the 22-week period. The internal product measures consisted of system size (*SS*), class count (*CC*), class size in SLOC (*CS*), number of methods per class (*MC*) and method coupling between classes. We implemented a parser to collect the *OMMIC* import coupling and the *OMMEC* export coupling measures described in Chapter 6.

Unfortunately, we encountered a problem with the collection of the coupling measures for Visual Basic. In this particular project, many variables were declared as a generic base class "object", and the class constructors were implemented in a way that the actual type of the variable could not be determined from static code parsing. This means that the target of the method could not be determined accurately. Consequently, the *OMMEC* export coupling measure collected from this module is inaccurate. Hence, data for export coupling is not reported. For the same reason, it is impossible to distinguish between coupling to library and non-library classes. Thus, the import coupling measure *IC* really represents message import coupling to non-library and library classes, *OMMIC* + *OMMIC_L*. This limits the scope of the validation of SAM and CPM. Fortunately, the Genera case study provided a better opportunity to validate a much larger portion of the measures proposed in Chapter 6.

7.1.1.3 Evaluation of Evolutionary Development

The summary data in Table 7.1 indicates that, during the development of the middle-tier module, there was a significant amount of rework. This rework is indicated by the ratio between net productivity and gross productivity for implementation of this module. For example, during the five weeks of the second increment, 3191 source lines of code (SLOC) were added or deleted from the module but the module grew by only 1128 SLOC. Only about 35% of the total amount of coding on the module contributed to increased module size (yielding a *rework ratio* of 65% for the second increment).

Table 7.1. Summary process data for the middle-tier module

Process/product measure	Incr.1 (week 1-6)	Incr.2 (week 7-11)	Incr.3 (week 12-22)
Coding effort per incr. (person-hours)	149	209	517
Changes per incr. (SLOC added + deleted)	1120	3191	5203
Gross productivity (SLOC added + deleted per hour)	7.5	15.3	10.1
Net productivity (SLOC growth per hour)	4.6	5.4	4.8
Rework ratio (% of effort not contributing to code growth)	39%	65%	52%
System size (SLOC)	687	1815	4307

Interviews with the development team indicate that the amount of rework experienced on the development project may in part be explained by uncertainties caused by the new technology used in the project. In particular, there was a mismatch between the promised and actual quality of certain development tools and libraries. A significant amount of coding effort was spent on trying alternative "work-around" solutions to compensate for flaws in the development tools and libraries. The rework is probably also a result of the evolutionary and incremental way in which the module was developed.

A certain amount of rework is a natural part of evolutionary development – it is necessary in order to produce a good product. However, too much rework may result in unacceptably low productivity. Furthermore, the rework may cause a gradual deterioration of the initial design structure, i.e., changeability decay. For rework to be a useful and valid process performance indicator, it needs to be used in conjunction with other quality indicators, such as customer satisfaction, the number of change requests from users after system delivery, etc. This balanced view of initial rework versus customer satisfaction and later change requests may provide a meaningful baseline for improvement activities in future evolutionary development projects.

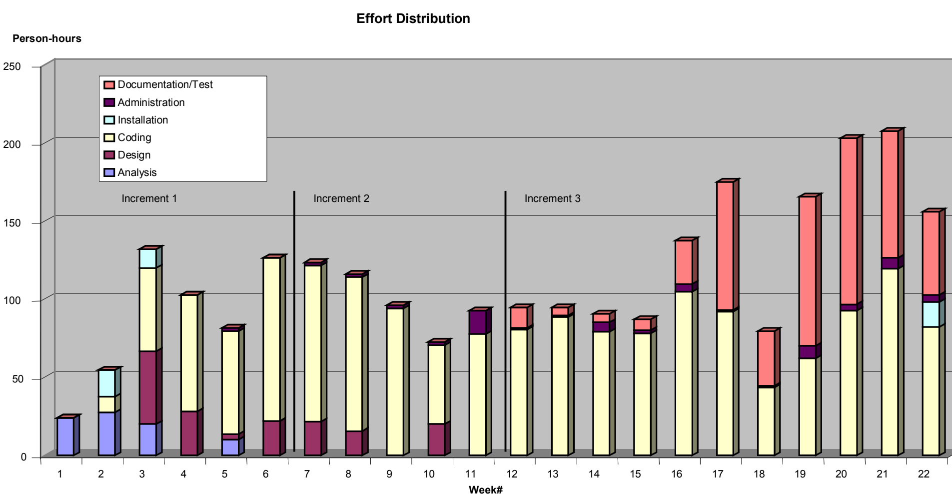

Fig. 7.1. Effort distribution for activities during the Braathens project

Figure 7.1 depicts the distribution of effort (in person-hours) for various process activities (analysis, design, coding[4], documentation/test, administration and installation) during the 22 weeks. While there is some overlap in the process activities, there is still a somewhat "phased" distribution of activities over time – to some extent resembling that of the traditional waterfall development process. For example, formal testing was only conducted in the third increment, not in the first two increments as prescribed in the Genova process (Arisholm *et al.*, 1999b). *Informal* testing was done by each developer throughout the coding activity. However, the

[4] The coding effort in Figure 7.1 reports the total coding effort for all modules in the system; not only for the middle-tier module as in Table 7.1.

separate, *formal* test activity including, for example, deployment in a dedicated test environment, writing test cases and applying test-logging tools, was not initiated before towards the end of the last increment.

Process Conformance

The degree of process conformance determines whether the defined and actual process coincide. Ensuring process conformance is important

- to ensure a stable process execution, that is, achieving a predictable process, and
- to ensure the validity of the data, information, experiences and knowledge that are acquired throughout the development projects (Sørumgård, 1997).

Without a "reasonable" degree of process conformance, it may be difficult to determine the effect of process improvement activities. In the Braathens case study, testing was not performed as prescribed by the defined process. Interviews with the developers indicate that the delayed testing contributed to many costly last-minute changes to the software. For example, many of the detailed requirements of the client-tier of the application were not discovered before the detailed write-up of the test cases, resulting in late rework. This provides a partial explanation for the large volume of coding effort towards the end of the final increment (Figure 7.1).

One explanation for this lack of process conformance was that the initiation and execution of the Genova process at Braathens were quite informal. However, insufficient guidelines for initiation and execution of the testing activities may also have contributed to this "flaw". In the Braathens case, one problem was that machine resources for deployment and load testing were made available very late by the software customer. Thus, it is uncertain whether the resulting lack of process conformance could have been avoided by the development team. Both the software vendor and the customer would have benefited if test facilities had been made available from the beginning of the project, allowing early testing according to the prescribed evolutionary life cycle. This aspect should have been addressed explicitly in the initial contract between the software vendor and customer. The result of this experience is thus a suggestion for improvement of one aspect of the defined Genova process: contractual guidelines regarding test facilities should be incorporated in the process description.

7.1.1.4 Validation of SAM and CPM

To validate the SAM and CPM measures, the weekly coding change effort for the module was used as the dependent variable. The *ChangeSize*, SAM and CPM measures were candidate explanatory variables.

Table 7.2 shows the correlation between class-level Import Coupling (*IC*), Method Count (*MC*) and Class Size (*CS*), based on the operational software system from week 22. Although the correlation coefficients with *CS* are high, there is sufficient variance in *IC* not explained by *CS* to consider both of them as candidate explanatory variables in a regression model for change effort. In a similar comparison, less correlation (r=0.59) was found between a similar import coupling measure and a similar size measure (Briand *et al.*, 2000).

Table 7.2. Pearson correlation between class-level structural attribute measures

Correlation (p-value)	IC	CS
CS	0.760 (0.000)	
MC	0.442 (0.031)	0.742 (0.000)

Figure 7.2 compares change profile measures (*ICCP* and *MCCP*) with the corresponding structural attribute measures (*Avg. IC* and *Avg. MC*). The results illustrate that more work is often done on classes with higher than average import coupling and higher than average message counts. Average structural attributes do not reflect the variation in weekly "change complexity", as such variation depends on which parts of the software structure that happens to be affected by the changes implemented during a given week. Hence, change profile measurement is useful for assessing structural properties of actual changes, whereas structural attribute measurement is useful for assessing trends in the overall structural properties of the software system.

For the Braathens case study, ChangeSize, ChangeSpan, SS, CC, Avg. IC, Avg. CS, Avg. CS, Avg. MC, ICCP, CSCP and MCCP were the candidate explanatory variables in a linear multiple regression model for weekly change effort. Unfortunately, the weekly data from this study was quite noisy as a result of irregularities in file check-in times. Consequently, weeks 1 and 15 were not included in the regression because no changes occurred. Weeks 14 and 17 were not included because the changes were very small compared with the rest (12 SLOC). Note that there was still a substantial amount of *total* coding effort allocated to weeks 14, 15 and 17 (Figure 7.1). In part, this is a result of noisy data, although the coding effort in Figure 7.1 depicts the total coding effort including *all* modules, not only the studied middle-tier module. The results from the linear regression are given in Table 7.3.

The explanatory power of the model in Table 7.3 is given by *R-Sq* = 68.2%. As explained in Chapter 6, it may also be necessary to assess the cross-validated R-Sq (cross), as the "normal" *R-Sq* may be too optimistic. The cross-validated *R-Sq(cross)* of the model is 52.8%, indicating that the model has limited usefulness for prediction purposes; a large portion of the variance is still not accounted for. None of the CPM measures contributed to a significant increase in the explanatory power of the regression.

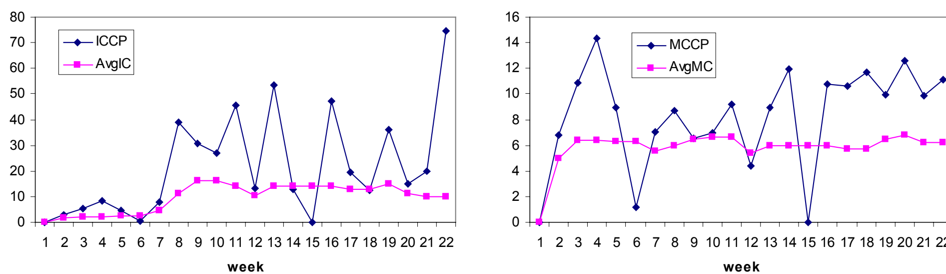

Fig. 7.2. Weekly plot of ICCP vs Avg. IC, and MCCP vs Avg. MC

Table 7.3. Preliminary regression model of change effort in the Braathens case study

```
The regression equation is
Effort = - 5.7 + 0.0163 ChangeSize - 1.78 AvgIC + 10.3 AvgMC

Predictor        Coef       StDev          T        P
Constant        -5.71       26.90      -0.21    0.835
ChangeSize   0.016255    0.004471       3.64    0.002
Avg.IC        -1.7773      0.3948      -4.50    0.000
Avg.MC         10.262      4.523        2.27    0.038

S = 8.508      R-Sq = 68.2%   R-Sq(adj) = 61.8%  R-Sq(cross) = 52.8%

Analysis of Variance

Source           DF          SS          MS         F        P
Regression        3     2323.92      774.64     10.70    0.001
Residual Error   15     1085.74       72.38
Total            18     3409.66

Source       DF      Seq SS
ChangeSize    1      756.32
Avg.IC        1     1194.96
Avg.MC        1      372.64
```

There is another possible problem with the above model, however. The obtained model poses a question of why an *increase* in average import coupling results in a *decrease* in change effort. Intuitively, one might expect a positive correlation, not a negative correlation as in this case. One plausible explanation is indicated by Figure 7.2. For the first increment (weeks 1–6), the average import coupling is low. At the beginning of the second increment, the import coupling rises quickly to a much higher level, and remains high for the rest of the project. Thus, one explanation of the negative sign of the coefficient for *Avg.IC* in Table 7.3 is simply that the productivity was considerably lower than average during the initiation of the project, before the coding had really started. Low values of the *Avg.IC* measure may simply serve as a substitute indicator of "increment 1". High values of the *Avg.IC* measure may serve as a substitute indicator of "increment 2 or 3". This suggests that import coupling actually may not affect changeability in the way indicated by the model.

To investigate this further, the first six weeks were removed from the data-set. Repeating the regression, none of the candidate variables entered the model except *ChangeSize* (Table 7.4). Thus, in the revised analysis, neither SAM nor CPM measures seem to explain any significant amount of the variation in change effort after accounting the "time" explanation of *Avg.IC* by removing the first six rows. These conflicting results make it difficult to conclude.

The results illustrate one of the difficulties associated with the validation of structural attribute measures in case studies. As opposed to a controlled experiment, a case study does not provide control of other factors that may be the *real* cause of observed statistical relationships. A careful analysis of the development project suggested an alternative (more conservative) interpretation of the results; other factors were more important for the variation in the change effort measure than structural attributes of the code. Indeed, the main risks for the project were related to design decisions and the incorporation of new and unfamiliar technology.

Table 7.4. Alternative regression model for change effort in the Braathens case study

```
The regression equation is
Effort = 33.6 + 0.0177 ChangeSize

Predictor         Coef        StDev          T        P
Constant        33.590        3.853       8.72    0.000
ChangeSize    0.017710     0.005240       3.38    0.006

S = 8.869        R-Sq = 50.9%     R-Sq(adj) = 46.5%

Analysis of Variance

Source            DF          SS          MS         F        P
Regression         1      898.42      898.42     11.42    0.006
Residual Error    11      865.16       78.65
Total             12     1763.58
```

However, the validation of the measures would probably be more reliable if effort and product data were collected from logical changes (i.e., implementing a given function), rather than from weekly changes (i.e., the changes that occurred within a weekly time span). For this case study, change data for logical changes were not available. Instead, we accumulated weekly changes until approximately 1000 SLOC had been changed (added or deleted). In this manner, we tried to obtain more accurate effort data by reducing potential noise caused by irregularities in file check-in times. Then, we calculated the total effort for the changes that occurred within each time span (e.g., from week 1 to week 5, and from week 6 to week 7). The resulting data is shown in Table 7.5. No statistically significant correlation was found between the structural attribute measures and the measured change effort, nor the change profile measures and the measured effort.

Table 7.5. Accumulated Change Effort, Change Size, SAM and CPM measures for the Braathens Wings development project

Week #	1–5	6–7	8	9–11	12–13	14–19	20	21–22
Effort (hours)	209	132	55	120	83	227	65.5	108.5
Change Size	1057	1086	1150	1018	1473	1187	1556	987
SAM:								
SS (System Size)	645	1202	1864	1815	2543	2894	4023	4305
CC (Class Count)	13	12	12	10	14	15	22	25
Avg. IC	2.7	3.1	11.3	14.1	14.1	14.0	11.4	10.1
Avg. CS	50	67	155	182	182	181	183	172
Avg. MC	6.3	4.8	6.0	6.6	5.9	6.1	6.8	6.2
CPM:								
ChangeSpan	13	12	8	9	13	12	12	15
ICCP	6.0	7.5	38.9	31.5	39.4	34.2	14.8	23.2
CSCP	75	133	316	320	307	334	311	268
MCCP	9.9	6.8	8.7	7.5	7.4	10.5	12.6	10.0

7.1.2 The Genera Case Study

The results from the Braathens case study motivated a follow-up case study using the proposed Change Log data collection process and tool (Chapter 6). Using the tool, it was possible to collect change data based on logical changes instead of weekly changes.

7.1.2.1 System under Study

The Genera case study was initiated in conjunction with an internal product development project (of the Genova tool itself) in Genera AS. The programming languages used were C++ and Java. This development project was larger than the Braathens case, consisting of about nine developers. The project was expected to last for several years, and this internal project was in many respects more stable and "laid back" than the Braathens project. The module studied was an independent part of the Genova tool providing run-time support for automatically generated source code based on the UML and Dialog Models specified in Genova (Arisholm *et al.*, 1998). The development of the module was organized as a separate development project, lasting approximately five months and consisting of three developers (one assigned to C++ and two assigned to Java) in addition to project management. For most of the changes, the two Java programmers cooperated in the implementation. For this project, no defined process was followed but the project was still evolutionary in nature. During the months this system was studied, existing, changed and completely new requirements were incorporated in an incremental fashion, with iterative analysis, design, coding and test activities. Two versions of the module were released within the five month time span.

7.1.2.2 Data Collection

The developers logged each logical change in the change log. For each logical change, the following change data was provided:

Change ID: A unique number identifying the change. This number was used to identify the classes changed. Each time a class was "checked in" to the configuration management system, the change ID was attached to the comment field of the change record. This information was subsequently used to select files for the calculation of SAM and CPM.

Description of the change: A short textual description of the change, such as *"Write a Condition Parser to evaluate GOAL condition expression to strings"*.

Classification of the change: Each change was classified into the following categories. Some changes may be classified into several categories:

- Correction of requirement fault, i.e., fixing a *bug* originating from the requirement specification.
- Correction of design fault, i.e., fixing a *bug* that may be classified as a design fault.

- Correction of code fault, i.e., fixing a coding bug for which the design was correct but there was an actual error in the code.
- *Implementation of existing user requirement*, i.e., implementation of functionality specified in the initial requirements.
- *Implementation of new user requirement*, i.e., implementation of new functionality not in the initial requirements.
- *Implementation of changed user requirement*, i.e., modification of code as a result of changes to existing functionality requirements.
- *Improved performance*, i.e., changes that were intended to improve the execution speed.
- *Preventive restructuring*, i.e., changes that were intended to restructure the code, to improve the design in some way (e.g., improved changeability).
- *Adaptations for reuse*, i.e., changing the design so that parts of the module could be reused in some way (a special case of preventive restructuring).
- *Adaptations to external libraries*, i.e., changes in external libraries on which the module depends, requires changes in the code.
- Adaptations to changed development tools, i.e., changes in development tools that requires changes in the code.
- *Other (specified in a text field),* i.e., a logical change that cannot be adequately classified into any of the other categories.

Effort Data: For each change, the following effort data (in hours) was reported:

- *Preparation:* Number of hours spent on preparing for the change
- *Analysis:* Number of hours spent on analysis
- *Code:* Number of hours spent on the actual coding of the change
- *Test:* Number of hours spent on testing the change
- *Documentation:* Number of hours spent on documenting the change

Subjective Task Assessment: For each change, the change was assessed by the developer as follows:

- *Task Size:* (Low, Medium, High)
- *Task Complexity:* (Low, Medium, High)
- *Resulting Changeability:* (Better, Unchanged, Worse)

Collection of Structural Attribute Measures and Change Profile Measures:
The Structural Attribute Measures (SAM) and Change Profile Measures (CPM) based on the description provided in Chapter 6 were collected for each logical change, which was written either in Java or in C++. Only changes to the Java module have been parsed. The SAM measures were collected based on the software all software files belonging to a particular change ID had been checked in. All other files were selected based on the timestamp of the last check in for a given change ID, using file selection scripts in the configuration management system (ClearCase). ClearCase was queried to give a list of all changes belonging to a given change ID. The result was a list of files. For each file, a version stamp and a time stamp were provided.

Table 7.6. Example ClearCase "config spec" file selection macro *before* a given logical change

```
element \s-rt\no\genova\goal\wrappers\GoalDbConnection.java \main\1
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\12
element \s-rt\no\genova\goal\gas\ClientGoalHelperGas.java \main\4
element \s-rt\no\genova\goal\ejb\ClientGoalHelperEjb.java \main\12
element * \main\LATEST -time 7-Jul.14:18
```

Table 7.7. Example ClearCase "config spec" file selection macro *after* a given logical change

```
element \s-rt\no\genova\goal\wrappers\GoalDbConnection.java \main\2
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\20
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\19
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\18
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\16
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\15
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\14
element \s-rt\no\genova\goal\helpers\GoalHelper.java \main\13
element \s-rt\no\genova\goal\gas\ClientGoalHelperGas.java \main\6
element \s-rt\no\genova\goal\gas\ClientGoalHelperGas.java \main\5
element \s-rt\no\genova\goal\ejb\ClientGoalHelperEjb.java \main\14
element \s-rt\no\genova\goal\ejb\ClientGoalHelperEjb.java \main\13
element * \main\LATEST -time 7-Jul.14:18
```

Based on this information, a file selection macro (a "config spec" in ClearCase) was written to collect the versions of the files as they were immediately *before* the change, and immediately *after* the change. All files that were not attached a given change ID were selected based on the timestamp of the last check in for a given change ID. An example "config spec" is provided in Tables 7.6 and 7.7. The instructions in the file selection macros have the following semantic:

element <file> <version-selector>: For the file specified in the <file> field, select the version of the file specified in <version-selector>. For example, the first line in Table 7.6, *(element \s-rt\no\genova\goal\wrappers\GoalDbConnection.java \main\1) w*ill select version *\main\1*, i.e., version 1 of the class *GoalDbConnection.java* on the "main" branch. Note that the same file may be checked in several times on the same change ID. Thus, several versions of the same file are specified in the config spec, but only the *last* version is selected. For example, for the *GoalHelper* class in Table 7.7, only version *\main\20* is selected. The version-selectors in Table 7.6 are equal to the version prior to the smallest version of the same file in Table 7.7. For the *GoalHelper* class, this corresponds to version *\main\12*.

*element * -time <time-stamp>:* This file selector selects the remainder of the files in the database based on the *<time-stamp>* field.

7.1.2.3 Evaluation of Evolutionary Development

Figure 7.3 shows a histogram of the distribution of change effort among activities throughout the development project. The histogram shows the iterative nature of the process activities. In a waterfall process, one would expect to find much less overlap (in time) of the various activities. Figure 7.4 shows how the 39 changes are distributed among the various change categories. Note the large number of changes classified as restructuring. Most of this restructuring occurred early, that is, during the first two months of the development project.

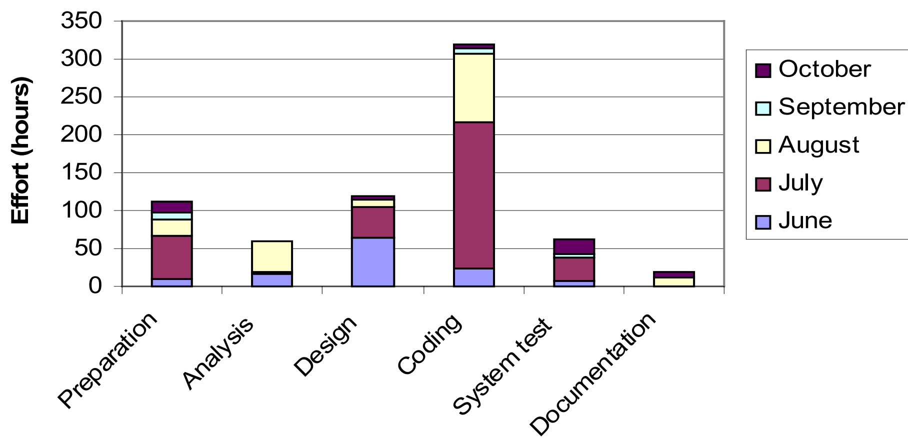

Fig. 7.3. Distribution of change effort for different activities throughout the 5-month period

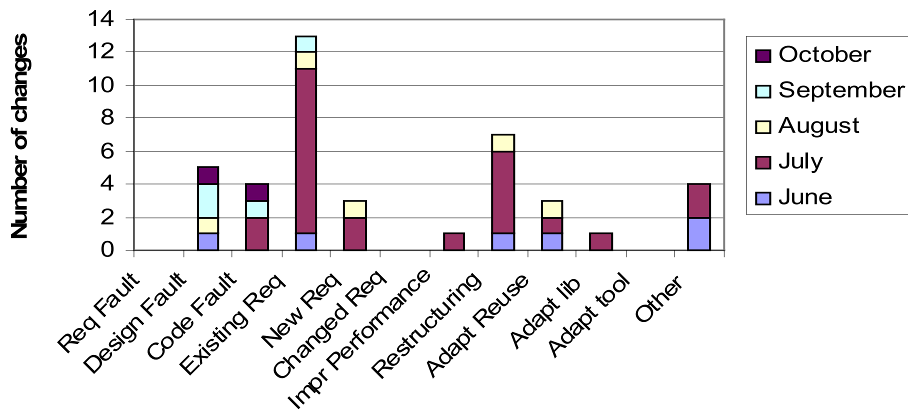

Fig. 7.4. Distribution of number of changes for different change categories throughout the 5-month period

There are at least two plausible interpretations of these results:

- Each change contained too small amount of analysis and design activities. Consequently, the changes resulted in increasingly unstructured code that needed frequent restructuring to avoid changeability decay.
- An insufficient amount of initial analysis and design was conducted before the main (existing) requirements were implemented.

It is difficult to accurately determine the relative significance of these possible explanations for the large number of restructuring changes in this project. However, there are reasons to believe that the second interpretation is more likely than the first one. Figure 7.5 shows that a large portion of the total design effort was spent during restructuring. Thus, the project is perhaps best characterized as a "code-and-fix" project. Too little analysis and design was performed for the non-restructuring changes.

These results suggest several opportunities for process improvements. Restructuring does not produce more functionality. The restructuring is primarily consequential rework from other changes. The question is whether the need for restructuring can be reduced, for example by increasing the amount of initial analysis and design. In this development project, it might have been possible to reduce the number of restructuring changes, because most of the other changes are implementations of existing requirements. There were only three *new* requirements, and no *changes* to the existing requirements (Figure 7.4). Thus, by increasing the amount of initial analysis and design based on the existing requirements, a more stable design could have been produced. In addition to more *initial* analysis and design, more analysis and design could clearly be incorporated for each change, too. To be cost-beneficial, it is assumed that the required increase in analysis and design effort would be less than the obtained decreases in restructuring efforts at later stages.

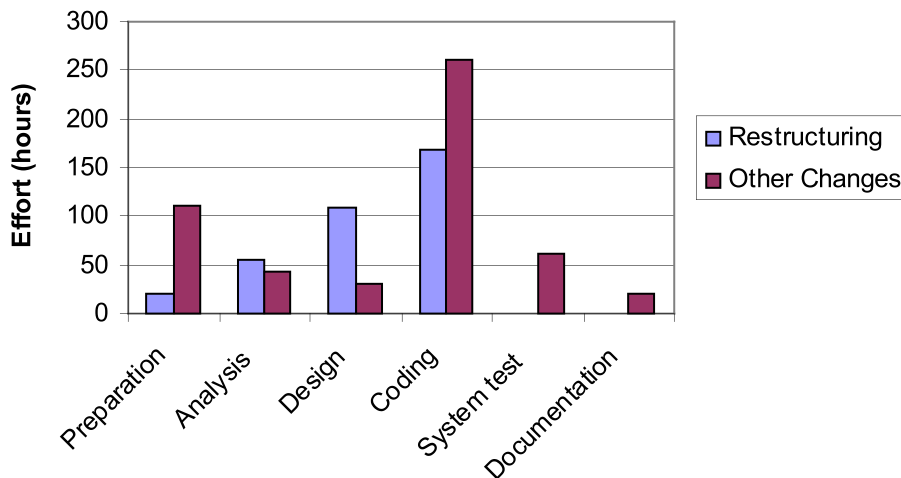

Fig. 7.5. Distribution of effort for restructuring versus other changes

A related idea is to prioritize the changes in a better *order*, such that the requirements that define the largest portion of the design would be implemented early. Less critical requirements could be implemented once a stable design had been developed.

7.1.2.4 Validation of SAM and CPM

Selection of Changes

For the validation purposes, 10 changes were selected among the available 39 logical changes reported in the change log. The remainder of the changes were not selected for the validation purposes for one of the following reasons:

- they were primarily related to documentation or administration (e.g., makefiles or configuration management)
- they included C++ code (no C++ parser was implemented)
- they had not been tagged correctly (or consistently) in the change log.
- they were mixed with other changes, making it difficult to determine the contribution of a specific change to the SAM and CPM measures

For the Genova module, a large number of measures were calculated for the selected changes (Table 7.8).

Measuring the Effect of Restructuring

Among the changes in Table 7.8, the first six changes occurred during the first increment. The remaining four changes occurred during the second increment. The first change in increment two (change number 7) was a large restructuring change. This change gives valuable information on how the restructuring affected the SAM and CPM measures. The changeability of the module was assessed by the developers as "better" after the change. Assuming the subjective assessment of the developers are correct and hence the restructuring was "successful", the SAM and CPM measures after change 7 should be indicative of a better structure. For change 7, the SAM and CPM measures that changed noticeably and consistently as a result of the restructuring are marked in bold (for an increase) and in italic (for a decrease).

Table 7.8. Change Effort, ChangeSize, Total SAM, Avg. SAM and CPM for the selected logical changes (ordered by check-in time) for the Genera project. Note that for Total SAM and Avg. SAM values, the import coupling value is always equal to the corresponding export coupling value. For example, TotOMMIC equals TotOMMEC.

Change number	1	2	3	4	5	6	**7**	8	9	10
Effort (hours)	1	2	31	26	7	10	150	18	6	13
ChangeSize	153	42	308	9	209	164	1421	111	106	72
SAM:										
CC	33	35	35	35	35	35	**47**	47	47	47
SS	399	406	531	701	610	662	**946**	961	932	965
TotMC	118	121	139	168	155	167	**200**	200	193	203
TotOMMIC, TotOMMEC	30	34	54	78	66	75	66	72	67	76
TotOMMIC_L	67	66	81	102	90	94	**162**	167	167	174
TotOMAIC, TotOMAEC	66	66	106	168	119	150	178	190	164	199
TotOMAIC_L	5	5	5	5	5	5	**15**	15	15	15
TotAMMIC, TotDMMEC	2	5	5	15	15	15	***0***	0	0	0
TotAMMIC_L	1	1	1	1	1	1	1	1	1	1
TotAMAIC, TotDMAEC	0	0	0	0	0	0	0	0	0	0
AvgCS	12.1	11.6	15.2	20.0	17.4	18.9	20.1	20.4	19.8	20.5
AvgMC	3.6	3.5	4.0	4.8	4.4	4.8	4.3	4.3	4.1	4.3
AvgOMMIC, AvgOMMEC	0.9	1.0	1.5	2.2	1.9	2.1	***1.4***	1.5	1.4	1.6
AvgOMMIC_L	2.0	1.9	2.3	2.9	2.6	2.7	**3.4**	3.6	3.6	3.7
AvgOMAIC, AvgOMAEC	2.0	1.9	3.0	4.8	3.4	4.3	3.8	4.0	3.5	4.2
AvgOMAIC_L	0.2	0.1	0.1	0.1	0.1	0.1	**0.3**	0.3	0.3	0.3
AvgAMMIC, AvgDMMEC	0.1	0.1	0.1	0.4	0.4	0.4	***0.0***	0.0	0.0	0.0
AvgAMMIC_L	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
AvgAMAIC, AvgDMAEC	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
CPM:										
ChangeSpan	4	3	6	2	4	3	22	3	6	10
CS_CP	44.3	12.0	96.9	148.3	88.0	153.0	***65.3***	70.8	50.5	49.7
MC_CP	12.3	2.9	22.3	35.4	23.7	38.6	***11.6***	13.2	10.8	7.7
OMMIC_CP	4.3	2.6	16.6	23.0	16.5	25.9	***5.3***	6.7	4.9	3.0
OMMEC_CP	0.3	0.3	1.8	0.4	0.3	0.0	**1.8**	1.9	1.3	2.1
OMMIC_L_CP	4.0	5.4	11.9	17.0	13.1	17.8	16.1	12.5	9.9	9.6
OMAIC_CP	17.0	0.0	42.7	75.8	34.8	78.0	***8.8***	31.1	7.0	19.5
OMAEC_CP	0.1	0.0	1.8	0.0	0.0	0.0	0.4	0.6	0.0	1.1
OMAIC_L_CP	0.6	0.0	0.0	0.0	1.1	0.5	1.6	0.0	0.4	0.3
AMMIC_CP	0.3	2.0	0.1	0.0	3.1	1.2	***0.0***	0.0	0.0	0.0
DMMEC_CP	1.5	0.0	3.4	11.7	8.0	12.4	***0.0***	0.0	0.0	0.0
AMMIC_L_CP	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
AMAIC_CP	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
DMAEC_CP	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

The changes in the measures caused by the restructuring can be summarized as follows.[5] The restructuring resulted in more code, and several new classes. Furthermore, there was more coupling to library classes after the change. The *total* non-library message coupling remained unchanged, but the restructuring resulted in considerably lower *average* message coupling to non-library classes. Furthermore, although the *average* class size (Avg. CS) remained unchanged, the CS_CP measure detected that classes actually being changed after the restructuring was smaller. The decrease in many of the CPM measures (e.g., MC_CP and OMMIC_CP), indicate a trend towards changing smaller classes with fewer methods and lower import coupling. From this analysis one may conclude that SAM and CPM are complementary, and may be used to capture different aspects of changes in structure: the CPM measures are sensitive to trends in the design that are "invisible" in the SAM measures (e.g., Avg. CS versus CS_CP).

Another interesting observation is made when comparing the changes in structural attributes seen in Table 7.8 with another restructuring described in Section 7.2. The changes in the structural attributes seen in Table 7.8 correspond very closely to the restructuring done on the coffee-machine design described in Section 7.2.

Principal Component Analysis

Table 7.9 shows the results of a PCA based on the class-level measures based on the 47 classes of the final release of the system. The even distribution of the variance among the components suggests that the measures capture many distinct dimensions of the software structure. The proposed method-attribute coupling measures *OMAIC* and *OMAEC* belong to different components than the method-method coupling measures *OMMIC* and *OMMEC*. Thus, they may represent a useful extension to Briand's coupling framework (Briand *et al.*, 1997b).

Table 7.9. Rotated Principal Components of the class-level SAM measures

Variable	PC1	PC2	PC3	PC4	PC5	PC6
SLOC	0.383	0.337	**-0.745**	-0.044	0.382	-0.009
MC	**0.608**	-0.028	-0.447	**-0.528**	0.164	-0.164
OMMIC	**0.960**	0.086	-0.196	-0.065	0.111	-0.053
OMMEC	0.079	-0.040	0.029	**-0.980**	0.048	-0.064
OMMIC_L	0.286	**0.647**	**-0.651**	0.091	0.187	-0.067
OMAIC	0.130	-0.046	-0.206	-0.076	**0.963**	-0.063
OMAEC	-0.075	-0.048	0.034	0.083	-0.055	**0.990**
OMAIC_L	0.004	**0.977**	-0.148	0.024	-0.083	-0.037
Variance	1.549	1.501	1.283	1.267	1.159	1.023
% Var	0.194	0.188	0.160	0.158	0.145	0.128

[5] Not all of these structural changes can be attributed to restructuring. There may be some more functionality as well.

Regression of Change Effort

To validate the measurement approaches, regression models were built according to the guidelines given in Chapter 6. The restructuring change (change 7) was considerably larger than the other changes, and was considered as an outlier in the regression analysis (Figure 7.6). Change Effort (in hours) was used as the dependent variable. The ChangeSize, SAM and CPM measures were the independent variables.

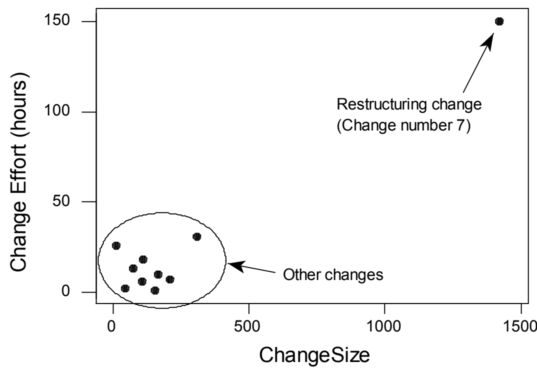

Fig. 7.6. Plot of Change Effort versus Change Size. The restructuring change is treated as an outlier in the regression model validation.

Table 7.10. Stepwise regression using ChangeSize, SAM and the CPM measures as candidate explanatory variables of change effort

```
F-to-Enter:      4.00    F-to-Remove:      4.00

 Response is  Change Effort (hours) on 25 predictors, with N =    9

    Step         1        2        3
Constant     8.477   -2.021    1.890

OMAEC_CP      10.2     10.5      8.5
T-Value       2.26     3.47     3.25

CS_CP                 0.131    0.132
T-Value                3.06     3.90

OMAIC_L_CP                     -10.1
T-Value                        -2.12

S             8.50     5.74     4.57
R-Sq         42.27    77.46    88.11
```

Table 7.10 shows the results of the stepwise regression using ChangeSize, system-level and class-level SAM measures and the CPM measures as candidate explanatory variables. None of the SAM measures were significant predictors of change effort. The final model consisted of *CS_CP*, *OMAEC_CP* and *OMAIC_L_CP*. *OMAIC_L_CP* was removed because the coefficient was not significant (p=0.08). Using backward selection, a better model was found, using *OMMEC_CP* and *OMMIC_CP* as explanatory variables. Regardless of the chosen variable selection heuristics, only CPM measures were included. The two best models are summarized in Table 7.11.

Figure 7.7 shows the residual model diagnostic for one of the resulting models. No serious violations of the conditions for valid interpretation of the regression model (Chapter 6) were found. According to the Anderson-Darling normality test, the residuals seem to be normally distributed (p=0.45). There may be some serial correlation. This is discussed further in Section 7.1.3.3.

OMMEC_CP (or alternatively *OMAEC_CP*) and *OMMIC_CP* (or alternatively *CS_CP*) are significant explanatory variables of change effort in the data set, and explain almost 80% of the variance of the change effort. Higher values of export coupling and import coupling of the classes being changed result in higher change effort. None of the SAM measures could explain any significant amount of variation in change effort. This indicates that the CPM measures may be better indicators of changeability.

Furthermore, for the small logical changes investigated, the CPM measures explain the variance in change effort considerably better than the number of lines of code added or deleted (ChangeSize). Although the models are significant, the predictive power of these models are limited, as indicated by the relatively small value for the cross-validated *R-Sq(cross)*. For larger changes, ChangeSize would probably play an important part in explaining change effort, as it did in the Braathens case study. This is also indicated by the plot in Figure 7.6.

Table 7.11. Resulting regression models for change effort in the Genera case study

Variables	Coefficient	Coefficient p-value	R-Sq	R-Sq (cross)
Intercept ChangeSize	9.083 0.027	0.210 0.536	5.7%	negative
Intercept OMAEC_CP CS_CP	-2.021 10.546 0.131	0.641 **0.013** **0.022**	77.5%	51.5%
Intercept OMMEC_CP OMMIC_CP	-7.404 10.072 0.936	0.164 **0.008** **0.008**	78.5%	56.8%

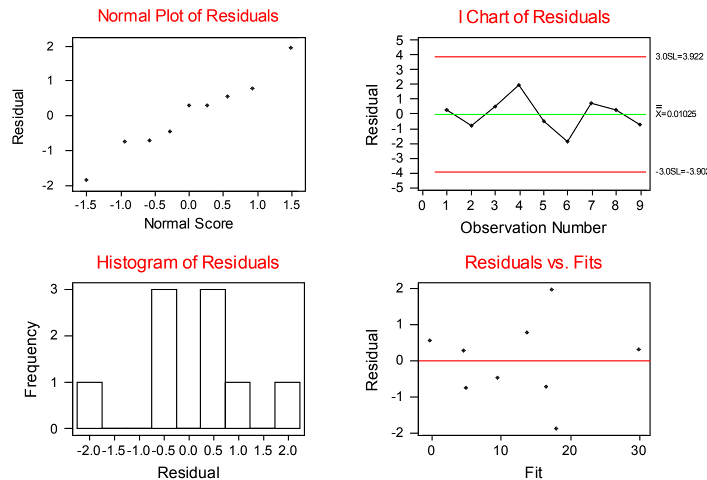

Fig. 7.7. Residual diagnostics for ChangeEffort = - 2.02 + 10.5 OMAEC_CP + 0.131 CS_CP

Regression on the difference in Change Effort

Recall from Chapter 6 that the purpose of the validation is not primarily to build effort prediction models, but to determine how structural attributes affect changeability. Thus, one may also attempt to build models that attempt to explain the *difference* in change effort between any two pair of changes, as explained in Chapter 6. The nine logical changes were grouped into distinct pairs (i.e., changes {*x, y*} 1<=*x*<=9; *x*>*y*) forming N=36 pairs of logical changes. Then, the *differences* in change effort for each distinct pair, i.e., *ChangeEffort(x) – ChangeEffort(y)*, and the *differences* in each SAM/CPM measure, e.g., *OMMIC_CP(x) – OMMIC_CP(y)*, were calculated.

Table 7.12. Regression models for the *difference* in change effort between logical changes

Variables	Coefficient	Coefficient p-value	R-Sq	R-Sq (cross)
Intercept	1.607	0.521	6.4%	negative
DiffChangeSize	0.029	0.135		
Intercept	-0.668	0.590	77.6%	74.7%
DiffOMAEC_CP	10.620	**0.000**		
DiffCS_CP	0.131	**0.000**		
Intercept	-5.106	**0.000**	87.5%	85.7%
DiffOMMEC_CP	12.379	**0.000**		
DiffOMMIC_CP	0.994	**0.000**		

Table 7.13. Details of the regression model using the method export and import coupling change profile measures

```
The regression equation is
DiffHours = - 5.11 + 12.4 DiffOMMEC_CP + 0.994 DiffOMMIC1_CP

Predictor          Coef        StDev          T         P
Constant         -5.106        1.047      -4.88     0.000
DiffOMME        12.3793       0.9641      12.84     0.000
DiffOMMI        0.99421      0.07811      12.73     0.000

S = 5.462       R-Sq = 87.5%    R-Sq(adj) = 86.7%  R-Sq(cross) = 85.7%

Analysis of Variance

Source             DF           SS          MS         F         P
Regression          2       6863.6      3431.8    115.05     0.000
Residual Error     33        984.4        29.8
Total              35       7848.0
```

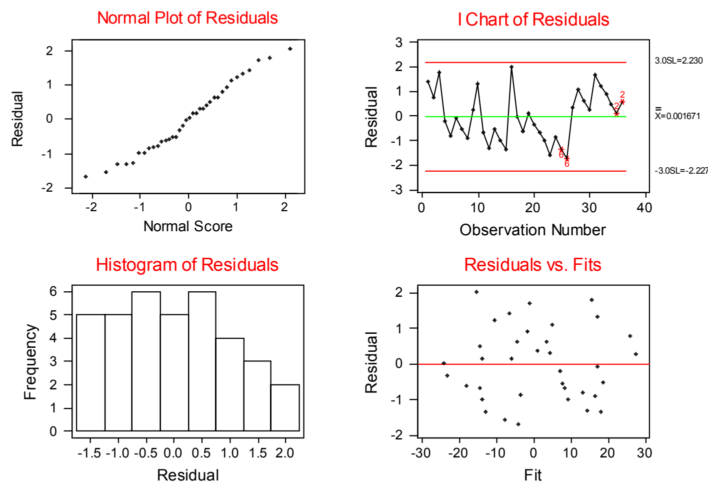

Fig. 7.8. Checking the model assumptions for DiffHours = - 5.11 + 12.4 DiffOMMEC_CP + 0.994 DiffOMMIC1_CP

Two of the resulting prediction models are shown in Tables 7.12 and 7.13. In this case, a cross-validated R-Sq(cross)=85.7% suggests that the resulting model is quite useful for predicting the difference in change effort. The p-values for each coefficient are clearly significant. Other alternative CPM measures also yield good models, but when restricting the models to only two explanatory variables (to reduce the chance of overfit), the best variables seem to be the same as those predicting change effort (Table 7.11).

There are some indications of serial correlation (Figure 7.8). This is discussed further in Section 7.1.3.3.

7.1.3 Summary

In the described studies, CPM and SAM were validated using change data from two case studies. Furthermore, the evolutionary development processes of the projects were studied in depth. The following subsections summarize the findings.

7.1.3.1 Using the Change Log

The Genera case study illustrated a practical method for analysis of logical changes using the change log coupled with a configuration management system. Each individual change reported in the log can be traced in the source code using configuration management file selection macros. This traceability allows us to collect internal product measures related to each change, coupled to external indicators such as change effort and defect data. The data may be used to validate SAM and CPM. The change log can also be used to assess evolutionary development projects.

Our long-term goal is to implement this data collection process in all internal product development projects at Genera AS, enabling many opportunities for evaluation of products and processes. An initial meeting with the developers in Genera suggests that they in general are positive towards the tool. The programmers using the tool so far expressed some concern regarding the use of the tool as a way to monitor each individual in terms of their individual productivity, fault rates, etc. The tool could easily be used for such purposes, although, as pointed out by this author during the meeting, such monitoring of individuals is not the purpose of the tool. Furthermore, there is some overhead using the change log, and there were a few suggestions for improvements. For example, the tool should be better integrated with the configuration management system, so that the developers do not have to manually attach the change ID each time a file is checked in. Using the ClearCase macro language (e.g., "triggers"), this is a straightforward extension of the change log. In general, the developers recognize the potential long-term benefits of using the tool.

7.1.3.2 Evolutionary Development

A preliminary evaluation of the Genova process was conducted in an industrial development project at Braathens in Norway. The case study provided one instance of an evolutionary development project that succeeded. However, based on quantitative and qualitative data, we identified improvements related to the distribution of test effort: the late initiation of formal testing contributed to unnecessary rework. We believe that less rework would have been required if formal testing had been conducted in each increment according to the prescribed process. Thus, more accurate contractual guidelines will be incorporated in the process description to ensure better process conformance for the test activity in future development projects. The effect of the suggested process changes still needs to be evaluated. Such effect measurement will use the change log for empirical assessment of the cost of implementing changes.

A case study at Genera, in which changes on the Genova tool were recorded, showed that a considerable amount of effort was spent on restructuring. The results raise challenging issues regarding how analysis and design may be distributed throughout the development project in an attempt to reduce the need for restructuring.

7.1.3.3 *Validation of SAM and CPM*

Change Profile Measurement was empirically evaluated against Structural Attribute Measurement. In the Braathens case study, there was some evidence that the SAM measures may be used as indicators of changeability. However, in an alternative analysis of the Braathens change data, neither structural attribute measures nor the change profile measures seemed to explain a significant amount of variation in change effort. Thus, it is difficult to interpret the results from the Braathens case study. However, we believe the collection of the measures in the Braathens case study would have been more reliable if effort and product data were collected from logical changes, as in the Genera case study.

The results from the Genera study indicate that CPM may account for some dimensions of the changeability of object-oriented software not provided with the SAM approach. A reasonably accurate model of change effort was developed based on the import coupling and export coupling change profile measures.

The investigation of the restructuring change reveals that SAM and CPM are in many ways complementary because they are "sensitive" to different dimensions of the changes to a design. This is exemplified by the results of the restructuring change summarized in Table 7.8: the overall structural attributes (i.e., SAM) of the software system remained unchanged while the CPM measures showed a clear trend towards changing smaller classes with lower coupling after the restructuring.

The Genera study also illustrates a potentially serious problem with the SAM approach. Measuring the overall structural attributes of a system assumes that all the parsed files are actually "live" code. After the large restructuring change (change number 7 in Table 7.8), several of the classes were no longer part of the module, but they were not removed from the file system. Thus, they were "hanging around" causing erroneous values for the SAM measures. Fortunately, we were able to remove the "dead" files after studying the comments in the change log and after talking with the developers. Thus, the reported SAM measures are correct. However, this means that it may be difficult to obtain reliable SAM measures if the measure collection is completely automated, unless the file structure is always up-to-date and consistent. The CPM measures were not sensitive to the dead files because only structural attributes of code being changed were accounted for.

In summary, the case studies provide some support for our hypothesis that the change profile measures may be good predictors of changeability, and better predictors than the structural attribute measures. Of course, the exploratory nature of this research cannot rule out that this apparent relationship between CPM and Change Effort is not due to "shotgun correlation" (Courtney and Gustafson, 1993). Thus, the validation should be interpreted with caution. One can often find statistical relationships between variables. This is not the same as proving a cause-effect relationship. Furthermore, it seems to be some serial correlation in the residuals of the models. The logical change data was collected in a specific time order. Serial correlation is due to some phenomenon not captured by the model, affecting the data over several consecutive observations. The consequences of such effects are that the error assessment may be too optimistic. Fortunately, the results from cross-validation indicate that this violation of the regression assumptions may not be that serious, at least for the prediction of the *difference* in change effort.

The problem discussed regarding serial correlation is just one manifestation of a more general threat when attempting to validate product quality measures using case study research. It is not possible to control for factors that may influence the results without our knowledge. In the Braathens case study, one such factor was found, which made the validity of the regression model, in our opinion, questionable. We believe that several of the problems discussed above can be addressed by a benchmark approach. If measurements were performed on benchmarks instead of *actual* changes, the validation may have avoided important threats to validity caused by:

- Inaccuracies in reported effort data (since a controlled benchmark experiment may allow better report and control of time expenditure).
- Differences in inherent change difficulty (since benchmarking prescribes the implementation of the same, given change – as described in Chapter 6).
- Differences in individual skill levels of the developers (since the benchmarking experimental design may control for individual ability – as described in Chapter 6).

7.2 Assessing the Changeability of two Object-Oriented Design Alternatives – a Controlled Experiment

The goal of the study reported in this section was

- to get a better understanding of how design decisions influence changeability, and
- to gain experience regarding the experimental design of benchmark experiments.

This study is also published in (Arisholm *et al.*, 2001). Chapter 4 suggested that the design of an open-ended object-oriented structure that easily supports change is critical for the success of evolutionary development. This section attempts to improve our understanding of the changeability of object-oriented software by studying the impact of identical changes on two alternative designs.

The changeability of a given responsibility-driven design was compared with an alternative control-oriented ("mainframe") design. According to Coad and Yourdon's OO design quality principles (Coad and Yourdon, 1991a; Coad and Yourdon, 1991b), the responsibility-driven design represents a "good" design. The mainframe design represents a "bad" design. To investigate which of the designs has better changeability, we conducted two controlled experiments – a pilot-experiment and a main experiment. In both experiments, the subjects were divided in two groups in which the individuals designed, coded and tested several identical changes on one of the two design alternatives.

The results clearly indicate that the "good" responsibility-driven design requires significantly more change effort for the given set of changes than the alternative "bad" mainframe design. This difference in change effort is primarily due to the difference in effort required to *understand* how to solve the change tasks. Consequently, reducing class-level coupling and increasing class cohesion may actually *increase* the cognitive complexity of a design. With regards to correctness and learning curve, we found no significant differences between the two designs. However, we found that structural attributes change less for the responsibility-driven design than for the mainframe design. Thus, the responsibility-driven design may be less prone to structural deterioration. A challenging issue raised in this study is therefore the tradeoff between change effort and structural stability.

The remainder of this section is organized as follows. Section 7.2.1 describes the design of the study, including the chosen design alternatives, the change tasks and the dependent variables. Section 7.2.2 describes the results of the pilot experiment used to formulate the hypotheses. Section 7.2.3 describes the results of the main experiment. Section 7.2.4 summarizes the results and relates them to existing research. Section 7.2.5 discusses validity issues. Section 7.2.6 describes future work.

7.2.1 Design of the Study

An important goal of our research is to investigate how design characteristics affect the changeability of object-oriented software. However, changeability can also be affected by other characteristics of the software, such as programming style and quality of documentation. Thus, to study how design decisions affect changeability, it

is necessary to restrict our study to software systems where only software characteristics directly related to the structural attributes (e.g. coupling, class count) of the software are varied. In this study, both systems implemented the same functionality and had similar programming style, naming conventions and documentation. Subjects of similar skill level designed, coded and tested a given set of changes on one of the two alternative software designs. The study consisted of two experiments:

1. the pilot experiment – to evaluate experimental design and material, and formulate the hypotheses, and
2. the main experiment – to replicate the pilot experiment with different subjects and test formal hypotheses on a larger scale.

7.2.1.1 Treatments: The Coffee-Machine Design Problem

We wanted to find alternative designs for the same system, in which one alternative adhered to Coad and Yourdon's quality design principles (the "good" design) and one did not (the "bad" design). The coffee-machine designs seemed to be good candidates for the experiment. These designs have been discussed at a workshop on object-oriented design quality at OOPSLA'97 and are described in two articles in C/C++ User's Journal (Cockburn, 1998):

> *This two-article series presents a problem I use both to teach and test OO design. It is a simple but rich problem, strong on "design," minimizing language, tool, and even inheritance concerns. The problem represents a realistic work situation, where circumstances change regularly. It provides a good touch point for discussions of even fairly subtle designs in even very large systems…*
>
> (Cockburn, 1998)

The initial problem statement was as follows:

> *You and I are contractors who just won a bid to design a custom coffee vending machine for the employees of Acme Fijet Works to use. Arnold, the owner of Acme Fijet Works, like the common software designer, eschews standard solutions. He wants his own, custom design. He is, however, a cheapskate. Arnold tells us he wants a simple machine. All he wants is a machine that serves coffee for 35 cents, with or without sugar and creamer. That's all. He expects us to be able to put this little machine together quickly and for little cost. We get together and decide there will be a coin slot and coin return, coin return button, and four other buttons: black, white, black with sugar, and white with sugar.*
>
> (Cockburn, 1998)

7.2.1.2 Description of the Design Alternatives

The MainFrame Design

According to (Cockburn, 1998), the type of design that most students come up with when faced with the problem of designing the given coffee-machine software is a so-called mainframe (MF) design. The MF design, adapted from "Design 3" in (Cockburn, 1998), consists of seven classes:

- CoffeeMachine: Initiates the machine, knows about the hardware components.
- CashBox. Knows amount of money put in; gives change; answers whether a given amount of credit is available.
- FrontPanel. Captures selection; knows price of selections, and materials needed for each; asks Cash Box if enough money has been put in; knows how to talk to the dispensers.
- Dispensers (cup, coffee powder, sugar, creamer, water). Knows how to dispense a fixed amount; knows when it is empty.
- Output. Knows how to display text to the user.
- Input. Knows how to receive command-line input from the user.
- Main. Initializes the program.

The Responsibility-Driven Design

The alternative responsibility-driven design was a result of a restructuring effort after the "customer" had requested several changes to the coffee-machine. For example, the coffee-machine was extended to make bouillon. For this reason, we thought that the restructured, responsibility-driven (RD) design would be an interesting design alternative to compare with the initial mainframe design. The RD design, adapted from "Design 4" in (Cockburn, 1998), consists of twelve classes:

- CoffeeMachine. Knows how the machine is put together; handles input.
- CashBox. Knows how much credit is available; handles money.
- FrontPanel. Knows products and selection; coordinates payment and drink making; knows the price of coffee.
- ProductRegister. Knows what products are available.
- Product. Knows its recipe.
- Recipe. Tells dispensers to dispense ingredients in sequence.
- DispenserRegister. Acts as a librarian for the dispensers; controls nothing.
- Dispenser. Controls dispensing; tracks amount it has left.
- Ingredient. Knows its name only.
- Output. Knows how to display text to the user.
- Input. Knows how to receive command-line input from the user.
- Main. Initializes the program.

Message sequence charts of the main functional scenario for the two designs were given to help clarify the flow of messages between the objects of the designs (Appendix B.4). The two designs were coded using similar coding style, naming conventions and amount of comments. Variable names and method names were long and reasonably descriptive. Two small code fragments from the MF and RD designs are given in Appendix B.5.

Comparing the Designs Against Coad and Yourdon's Quality Principles

According to Coad and Yourdon's design principles (Coad and Yourdon, 1991b; Coad and Yourdon, 1991a), a "good" design adheres to (among others) the following guidelines, based on (Briand *et al.*, 1999a):

Coupling. Interaction coupling between classes should be kept low, by decreasing the number of messages that can be sent and received by an individual object.

Cohesion. A class should carry out one, and only one, function. The attributes and services should be highly cohesive, i.e., they should all be descriptive of the responsibility of the class.

Clarity of design. The names in the model should closely correspond to the names of the concepts being modeled. Second, the responsibilities of a class should be clearly defined and adhered to. Furthermore, the responsibilities of any class should be limited in scope.

Keeping objects and classes simple. First, avoid excessive numbers of attributes in a class. A class should map to a type of entity in the problem description.

Although these design principles require a certain degree of subjective interpretation (Briand *et al.*, 1999a), clearly the RD coffee-machine design adheres significantly better to these design principles than does the MF design. The RD design has lower class-level coupling, more cohesive classes, better clarity of design and simpler classes. The MF design is assessed as follows:

> *Although the trajectory of change in the mainframe approach involves only one object, people soon become terrified of touching it. Any oversight in the mainframe object (even a typo!) means potential damage to many modules, with endless testing and unpredictable bugs. Those readers who have done system maintenance or legacy system replacement will recognize that almost every large system ends up with such a module. They will affirm what sort of a nightmare it becomes.*
>
> (Cockburn, 1998)

Furthermore, Cockburn assessed the RD design as follows:

> *The design we come up with at this point bears no resemblance to our original design. It is, I am happy to see, robust with respect to change, and it is a much more reasonable "model of the world." For the first time, we see the term "product" show up in the design, as well as "recipe" and "ingredient." The responsibilities are quite evenly distributed. Each component has a single primary purpose in life; we have avoided piling responsibilities together. The names of the components match the responsibilities.*
>
> (Cockburn, 1998)

For this experiment, we had to do certain modifications to the designs presented by Cockburn, so that they delivered the same functionality. Primarily, we removed the modifications done on the RD design in order to make bouillon, since we thought this functionality to be a particularly good candidate for a change task. This also motivated leaving the price of coffee where it was originally – in the front-panel class. However, it would be a simple task to move the price attribute from the front-panel class to the product class in order to provide differentiated pricing (to make bouillon). Otherwise, the main concepts underlying the two designs have been kept as far as possible. Although the modified RD design may represent a slightly less "pure" design than the one presented by Cockburn, we believe his assessment is also applicable to the MF and RD design alternatives.

Structural Attributes of the Design Alternatives

Table 7.14 shows the values of coupling (*OMMIC*, *OMMIC_L* and *OMMEC*) and size (*MC* and *CS*) for the two designs. The RD design has about 40% lower class-level coupling to non-library classes (*OMMIC and OMMEC*). *OMMIC_L* quantifies the number of method invocations to library classes, which in this case are *String* and *Vector*. Because the RD design uses vectors to represent products and dispensers, the class level *OMMIC_L* measure is slightly higher for the RD design (mean = 1.3) than for the MF design (mean = 1.1). The MF design has larger classes and fewer methods per class than the RD design.

At the *system* level, however, Table 7.14 (the "Sum" column) shows that the overall non-library coupling remains almost unchanged, whereas the coupling to library classes, the total number of methods and the total system size have *increased* for the RD design. Thus, to 1) reduce coupling, 2) increase cohesion, 3) improve the clarity of the design, and 4) keeping classes simple, the values for some system-level measures have increased. These findings are consistent with the restructuring change reported in Section 7.1.2. Thus, it would seem that, at least to some extent, experienced developers apply similar heuristics for restructuring a design.

Table 7.14. Descriptive statistics of structure and size attributes for the MF and RD designs

Measure	Description	Design	Median	Mean	Sum
OMMIC(c)	The number of static method invocations from a (client) class *c* to non-library classes	MF	2	4.7	33
		RD	1	2.8	34
OMMIC_L(c)	The number of static method invocations from a (client) class *c* to library classes	MF	0	1.1	8
		RD	0	1.3	16
OMMEC(c)	The number of static method invocations to a (server) class *c*	MF	3	4.7	33
		RD	2	2.8	34
MC(c)	The number of implemented methods in a class *c*.	MF	1	1.6	11
		RD	2	1.8	22
CS(c)	The size (in SLOC) of each class. Note that the sum corresponds to system size.	MF	9	11.0	77
		RD	7	8.9	107

7.2.1.3 *The Mocca Programming Language*

We had to make some decisions regarding the choice of programming language. It should be easy to understand for subjects with some prior experience with common OO programming languages – to minimize the learning curve. However, the programming language should also contain sufficient OO constructs and flexibility to allow the development of realistic code for the change tasks. Thus, we created a scaled-down version of Java, called *Mocca*. Mocca has the same syntax as Java, but is restricted in several ways:

- It does not contain inheritance mechanisms or constructs for interfaces.
- It has no explicit type-casting.
- It has a globally available (static) INPUT and OUTPUT class.

- It contains only the elementary types void, int and boolean.
- It contains only two library classes: String and Vector.

A relatively complete documentation of the Mocca language was written in eight pages. While the restrictions in Mocca may be too limiting as a general purpose, experimental OO programming language, we believe that Mocca was a reasonable tradeoff between realism and simplicity for the given change tasks on the coffee-machine designs.

7.2.1.4 The Programming Tasks

The programming tasks consisted of one *calibration task* and three *change tasks* (*c1*, *c2* and *c3*) for the coffee machine. For practical reasons, the changes were coded with pen and paper. For small designs and change tasks, we believe this may be a better choice than using a computer to reduce the possibility of errors caused by technical- and tool-oriented problems. Each task description contained a test case that each subject used to manually "test" the solution. Of course, this is not a real test, which would require running the program on a computer. The main purpose of the test was to motivate the subjects to produce solutions of good quality before starting on the next change task. Judging from the actual correctness score of the solutions (Table 7.19), this strategy seems to have worked quite well.

The Calibration Task

The first programming task to be completed by all subjects was the calibration task. The calibration task consisted of adding transaction log functionality in an automatic teller machine, and was not related to the coffee-machine designs. Since all subjects implemented the same change on the same design, the calibration task provided a common baseline for comparing the programming skill level of the subjects. The calibration task was almost the same size as the change tasks *c1*, *c2* and *c3* combined. The size of the calibration task ensured that most aspects of the Mocca programming language (e.g. class constructors, vectors, strings, input and output) were exercised, thus reducing the influence of the programming language learning curve.

The Change Tasks

Each change task was coded by the students directly on the coffee-machine code printout for the given design. The change tasks consisted of three changes to the coffee-machine, to be implemented in the given order. The actual change task descriptions are given in Appendix B.2.

c1. *Implement a coin return-button*. The actual solution was identical for the RD and the MF design. In both cases, it involved a modification to the menu handling routine (to include the "Return Coins" menu choice) and the addition of corresponding event handling routine. In addition, the developers had to call the "ReturnCoins" method in the CashBox class.

c2. *Make bouillon.* Extend the machine with a menu choice and the functionality to make bouillon in addition to coffee. Bouillon costs more than coffee. The solution involved making a menu-choice and event handling routine for bouillon

by modifying the front panel. It also involved making a new dispenser for bouillon, and checking whether the customer had deposited sufficient funds.

c3. *Fix a bug: Check whether all ingredients are available for the selected drink.* If one or more dispensers are empty, the user should get an error message and can try another drink or get his money back. This change task was motivated by a "bug" found in both of the code listings for the original design alternatives presented in (Cockburn, 1998). If the machine was empty of a required ingredient (e.g. creamer), the machine would still produce the "drink" using only the remainder of the ingredients, i.e., the customer would receive black coffee when asking for white coffee. The solution involved checking whether all required ingredients were available before making the drink.

For subjects that managed to complete all change tasks within the allocated time, an "extra assignment" was given. This change task was included to ensure that none of the subjects finished before the end of the allocated time of the experiment. We did not want the subjects to leave early, disturbing the other subjects. Furthermore, without change task *c4*, subjects may have been inclined to use more time on change task *c3* than they would otherwise. Change task *c4* is not included in any subsequent analysis since only very few subjects managed to complete the task:

c4. *Add the option "make your own drink"*, by selecting among any meaningful combination of the available ingredients.

7.2.1.5 The Change Task Questionnaire

After completing each programming task (including the calibration task), the participants reported the effort to understand, code and test each task. In addition, the subjects reported on subjective task difficulty, solution strategy (explorative or systematic) and confidence in the correctness of their solution. The questionnaire is given in Appendix B.3.

7.2.1.6 Experimental Design

To ensure accurate and reliable results we had to deal with issues related to the learning curve and the skill level of the individuals who participated in the experiment. To control for the differences in skill level of the individuals, we considered a cross-over design where each developer implements the same (or similar) changes on both design alternatives. However, this experimental design does not control for the following learning effects:

- *Learning the system* – if the design alternatives have many similarities, most of the developer's initial system comprehension effort will be spent on the first design.
- *Learning the changes* – if a developer implements the same change on two alternative designs, it is likely that the developer will be more efficient during implementation of the change on the second design.

Thus, we used a design where each developer implements the same change only once, while still controlling for the differences in individual skill levels by assigning the

developers in two groups by means of randomization and blocking. This is described further in the following sections.

7.2.1.7 Design of the Pilot Experiment

The subjects consisted of twelve graduate students and professionals enrolled in a course in software process improvement at University of Oslo. The experiment was divided in three separate, one-hour sessions consisting of:

Session 1. *Experience level assessment and training.* During this session, the students completed the experience questionnaire (Appendix B.1). Then, we trained the subjects in Mocca and distributed the programming language documentation.

Session 2. *Skill level assessment and group assignment.* All students implemented the calibration task and completed a change task questionnaire (Appendix B.3). Based on the results of the calibration task, the students were divided into blocks and then assigned at random (within each block) into two groups, one for each design alternative.

Session 3. *Coffee-machine experiment.* The subjects to the first group implemented the change tasks on the MF design. The subjects assigned to the other group implemented the change tasks on the RD design.

7.2.1.8 Design of the Main Experiment

Subjects of the main experiment were mainly undergraduate students in computer science at University of Oslo. Unlike the pilot-experiment, the subjects volunteered for the experiment and were paid to participate. The experiment took place within one 3-hour session. The subjects were introduced to the experimental procedures and trained in Mocca during the first hour. During the next two hours, the subjects first implemented the calibration task and then the change tasks.

With regards to the group assignment, we were unable to use blocking based on the calibration task because the experiment consisted of only one session. Furthermore, results from the pilot-experiment suggested that it was not useful to use the reported experience level data (Appendix B.1) to create blocks. Consequently, the students were assigned at random into two groups of equal size, one for each design alternative. Some students did not show up for the experiment, while some other students that had failed to register for the experiment showed up just prior to the session. They were assigned at random when they arrived. The resulting group assignment consisted of 17 subjects on the MF design and 19 subjects on the RD design. In the unlikely event that the randomization would fail to provide approximately equal groups, we could use the results from the calibration task in subsequent analyses to adjust for such differences (Section 7.2.5.5).

7.2.1.9 Dependent Variables

Figure 7.9 depicts the dependent variables of the study. They are explained further in the following sections.

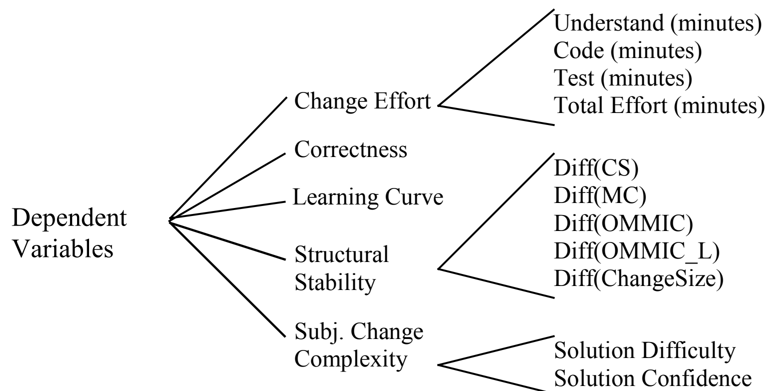

Fig. 7.9. Summary of the dependent variables of the study

Change Effort

Before starting on a task, the subjects wrote down the current time. When the subjects had completed the task, they reported the total effort (in minutes) to complete the change task. The primary dependent variable of the study was the combined total effort to complete all change tasks. After completing each change task, the subjects also estimated how much time was spent to

- *understand* the task (analysis and design of the solution),
- *code* the solution, and
- *"test"* the paper solution against the test case.

Correctness

It is possible that one design is more error-prone than another design, resulting in errors that are not discovered and subsequently corrected by each subject in the experiment. Furthermore, the reported change effort for the change tasks may contain lower values for solutions that contain errors. This may bias the change effort results if one design is more error-prone than the other design. Consequently, each change task solution was reviewed and given a (subjective) correctness score by this author. Table 7.15 gives the coding scheme.

Table 7.15. Coding scheme of the correctness measure

Correctness Score	Interpretation
6	Correct solution, passes the test case
5	Small deviations from the test case, but no logical errors
4	Small logical errors that are estimated to be very simple to fix
3	Some errors that are estimated to take some time to fix
2	Incomplete solution that are estimated to take a long time to fix
1	Very incomplete solution

Learning Curve

The experiment contains only a small number of change tasks. Thus, the recorded *total* change effort for the combined change tasks may fail to reflect trends in change effort caused by the system learning curve. For example, a given design may be difficult to change until the developer understands the intricate structural properties and abstraction mechanisms of the design. However, the subsequent changes may be easy to implement, hence resulting in a trend towards less change effort compared with another design.

To perform statistical tests on differences in the learning curve we need to quantify it such that the measure is normalized and hence comparable for different subjects. We measure the learning curve with respect to the given change tasks as the normalized difference in effort to understand the last change (*c3*) versus the first change (*c1*) for each subject, for design RD and MF, respectively:

$$\text{LearningCurve}(d) = \frac{\text{Understand}(d, c1) - \text{Understand}(d, c3)}{\text{Understand}(d, c1) + \text{Understand}(d, c3)}, \; d \in \{\text{RD}, \text{MF}\}$$

A larger number indicates a stronger learning effect. The measure is not meaningful as an absolute measure of the learning curve since change task *c3* is probably more difficult to solve than change task *c1*. Thus, in this case one may even get "negative" learning. The measure is only meaningful when comparing the relative *difference* in the learning curve on MF versus RD. The measure also assumes that most of the learning occurs early.

Subjective Change Complexity

We also asked the subject two questions that may reflect the perceived complexity of each change task:

- *Solution Difficulty* – how difficult did the subjects think it was to solve each change task (1 = very simple; 6 = very difficult).
- *Solution Confidence* – how confident were the subject that the solution of a change task did not contain serious errors (1 = very unsure; 6 = very confident)

Structural Stability

When studying the changeability of an object-oriented design, it may be appropriate to assess the impact changes have on the design. Consider Lehman & Belady's "law of increasing complexity":

> *As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.*
>
> (Lehman and Belady, 1985)

In general, changes in structural attributes do not necessarily indicate decay; there could be restructuring or re-engineering going on. However, in the coffee-machine experiment, there is no "restructuring" of the design. The change tasks represent functional additions (*c1* and *c2*) and bug-fixes (*c3*). Thus, when assessing the changeability of design alternatives it may also be appropriate to measure trends in

structural attributes (i.e. structural stability) that may indicate changeability decay (Arisholm and Sjøberg, 2000). The *differences* in the average values of the measures before and after each change (e.g., from change task *c1* to change task *c2*) may be used to assess whether the structural attributes of one design change faster than an alternative design.

A summary of the structural stability measures is given in Table 7.16. To measure the structural change of the solutions, five paper solutions were selected at random for each of the two design alternatives, for a total of ten solutions to each of the change tasks. To ensure accurate structural attribute measures, only solutions with a correctness score of five or six (i.e., "correct" solutions) for all three change tasks were considered. The selected paper solutions were coded into a computer by one of the authors, and subsequently compiled and tested to ensure that the solutions actually were correctly implemented. A Java parser was used to collect the measures. In addition to the change in structural attributes, the size of each change was calculated. For the *ChangeSize* measure, we manually counted (based on the paper solutions) the number of lines of code added, deleted or modified for *all* solutions that were correctly implemented.

Table 7.16. Summary of Structural Stability Measures

Definition	Detailed explanation
Diff(CS)	The difference in average class size before and after a change task
Diff(MC)	The difference in average number of implemented methods in a class c
Diff(OMMIC)	The difference in average import coupling for a class c to non-library classes
Diff(OMMIC_L)	The difference in average import coupling to library classes
ChangeSize	SLOC added+deleted+modified for each change task

7.2.2 Results of the Pilot Experiment

The goals of the pilot experiment were

- To evaluate and improve the quality of the experimental materials (e.g. questionnaires, change tasks and programming language). This is described further in conjunction with analysis of threats (Section 7.2.5).
- To evaluate the usefulness of different blocking strategies to reduce random errors, i.e., blocking on the results from the calibration task and blocking based on data from the experience level questionnaires.
- To formulate hypotheses and to develop meaningful dependent variables through an exploratory analysis of the preliminary results from the pilot-experiment.

7.2.2.1 Evaluation of Blocking Strategies

The pilot-experiment used the correctness score of the calibration task to create blocks on skill level. An equal number of subjects were assigned at random to each design (MF and RD) from each block. Unfortunately, only eight of the twelve subjects attended session 3. This resulted in the average skill level being slightly higher for subjects assigned to the RD design, despite the randomized block scheme (Figure 7.10). Clearly, the usefulness of blocking may be limited unless one can be sure that the assigned subjects will attend the experiment.

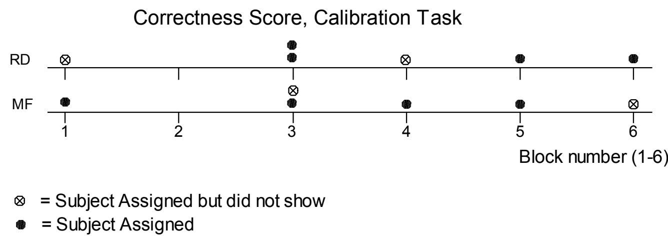

Fig. 7.10. Dot-plot of group assignment with the correctness score (1 to 6) from the calibration task as the blocking factor

We also evaluated whether data from the experience level questionnaire could be used to create blocks for the main experiment. We found no significant correlation between the experience level data and the results of the calibration task in the pilot-experiment. This suggests that it may be ineffective to use the experience level data to create a randomized block design.

7.2.2.2 Preliminary Assessment of Change Effort for MF and RD

Figure 7.11 depicts the difference in total effort to change tasks *c1* and *c2* (most subjects did not have time to complete *c3* within the 1-hour session of the pilot experiment). Although subjects assigned to design RD had performed better on the calibration task in the previous session, they still needed on average 30 percent more time to complete change tasks *c1* and *c2* than the subjects assigned to design MF.

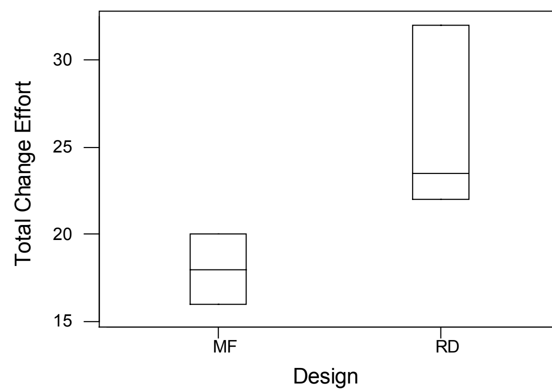

Fig. 7.11. Box-plot of the total change effort (in minutes) to complete change tasks *c1* and *c2* for design MF and RD, respectively. A line is drawn across the box at the median. The box represents the 95% confidence interval for the median.

7.2.3 Results of the Main Experiment

The explorative analysis of the results from the pilot experiment was used to formulate the hypotheses of the main experiment. According to design principles such as Coad and Yourdon's, we would expect that the RD design enables a more efficient and correct implementation of changes. However, the results of the pilot experiment do not support this theory. The theory underlying the formulation of the hypotheses is that the more fine-grained delegation of responsibilities of the RD design results in added complexity that more than outweighs its theoretical advantages with regards to change effort and correctness (H1 to H4). However, the improved delegation of responsibilities of the RD design should result in a more stable design (H5).

7.2.3.1 Formal Hypotheses

H1. *Change Effort:* The RD design requires more change effort than the MF design.

H2. *Learning Curve:* The RD design has a stronger learning effect than the MF design. We regard H2 as a validity-check on H1. If both H1 and H2 are accepted, it will be difficult to determine whether H1 would have been valid if we had included even more change tasks.

H3. *Correctness:* The solutions for the RD design contain more errors than the MF design.

H4. *Change Complexity:* The RD design has higher change complexity than the MF design.

H5. *Structural Stability:* The RD design has better structural stability than the MF design.

The statistical tests will attempt to reject the null-hypotheses, which are just the opposites of H1 to H5. For H1 and H2, a one-sided two-sample T-test (assuming unequal variances) on the difference in means was used. Before using the T-tests, the samples were checked for normality using the chi-square based Kolmogorov-Smirnov normality test. No significant deviations from the normal distribution were found. For H3 and H4, the tests were performed using Mood's median test, which is a robust, non-parametric sign scores test for ordinal scale measures such as the Correctness Score and Subjective Task Difficulty. H5 was not tested formally, but was assessed based on a subjective interpretation of the results.

The more tests are performed on the same dataset, the more likely is it that one will find significant results occurring by chance. Thus, to make a scientific statement with a reasonable degree of confidence, the significance level for the hypotheses tests were initially set to $alpha = 0.1$, and subsequently reduced to account for multiplicity using Holm's multiple test procedure (Holm, 1979). Holm has shown that, for K statistical tests, the adjusted significance level must be set equal to $alpha/(K-i+1)$, where $(i = 1,\ldots,K)$ is the index of each test ordered by the p-value ($p_1 <= p_2 <= \ldots <= p_K$). This means that p_1, the smallest p-value, must be compared with $alpha_1 = alpha/K$. The largest p-value p_K must be compared with $alpha_K = alpha$. In our case, adjusting the significance level using Holm's procedure is more appropriate than using the even more conservative Bonferroni adjustment, i.e., $alpha/K$, since the Bonferroni adjustment ignores the correlation between tests. A practical discussion of the power of tests and presetting the level of significance is provided in (Briand *et al.*, 1999a).

7.2.3.2 Change Effort (H1)

Hypothesis H1 is supported (p=0.0072, two-sample T-test on the difference in mean total change effort to implement *c1*, *c2* and *c3*). On average, the total change effort on RD was about 20% higher than the total effort on MF (Table 7.17). Most of the difference in change effort is due to differences in time to understand how to implement the change tasks (p=0.0006). There are smaller differences in the coding and testing effort, but all results are in favor of the MF design. Figure 7.12 depicts the average change effort for the change tasks.

Only eight out of nineteen subjects assigned to the RD design reported that they completely finished change task *c3,* whereas sixteen out of seventeen subjects assigned to the MF design completed change task *c3*. The analysis of the effort data on the RD design therefore includes data points for those subjects that *almost* finished, i.e., some testing remained but they still reported effort data. All subjects completed the first two change tasks. When only counting change tasks *c1* and *c2*, the results show more than 40% difference in change effort (p=0.0004).

Based on the results of the pilot experiment, we had estimated that more subjects would have completed all change tasks within the allocated time. It is interesting that the professional programmers and graduate students of the pilot experiment implemented *c1* and *c2* significantly faster than the undergraduate students of the main experiment, even though the undergraduate students had more experience with Java. On average, the graduate students/professionals used about 40% less time than the undergraduate students. In retrospect, we should have allocated more time for the change tasks.

Table 7.17. Summary of change effort (in minutes) for the change tasks *c1*, *c2* and *c3*

Changeability Indicator	**Group**	**N**	**Mean**	**StDev**	**SE Mean**	**H1: mu(MF)<mu(RD) (p-value)**	**Holm's alpha 0.1/(15-i+1)**
Total *c1+c2*	MF	17	26.88	8.28	2.0	**0.0004**	0.0067 (i=1)
	RD	19	38.30	10.20	2.3		
Total *c1+c2+c3* (H1)	MF	16	49.20	12.60	3.1	**0.0072**	0.0083 (i=4)
	RD	18	59.22	9.29	2.2		
Understand *c1+c2+c3*	MF	16	16.03	7.31	1.8	**0.0006**	0.0071 (i=2)
	RD	17	26.06	8.88	2.2		
Code *c1+c2+c3*	MF	16	27.13	9.26	2.3	0.42	0.0143 (i=9)
	RD	15	27.77	7.95	2.1		
Test *c1+c2+c3*	MF	16	6.09	4.07	1.0	0.43	0.0200 (i=11)
	RD	14	6.36	3.77	1.0		

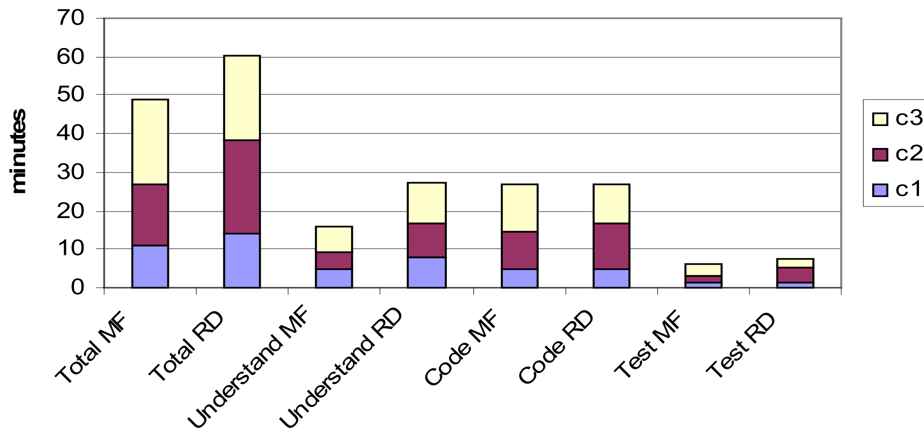

Fig. 7.12. Avg. change effort (in minutes) for each change task (*c1, c2, c3*) for MF and RD

7.2.3.3 Learning Curve (H2)

The hypothesis H2 is not supported. The results indicate that there is no significant difference in the relative learning effect for designs MF and RD (Table 7.18). The average values for the effort to understand each change is shown in Figure 7.13. The results show that the time to understand each change task on MF is lower than the time to understand the same tasks on RD. Furthermore, there is no visible difference in the trend in the learning curve (from *c1* to *c3*).

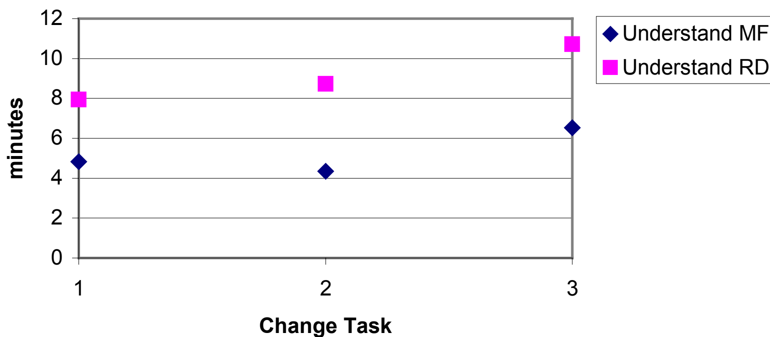

Fig. 7.13. Trend in the comprehension effort from change task *c1* to change task *c3*

Table 7.18. Two-sample T-test on difference in learning curve

Changeability Indicator	Group	N	Mean	StDev	SE Mean	H1:mu(RD)< mu(MF)	Holms alpha= 0.1/(15-i+1)
LearningCurve (c1,c3)	MF	16	-0.096	0.354	0.088	p = 0.52	0.0250 (i=12)
	RD	17	-0.103	0.370	0.090		

7.2.3.4 Correctness (H3)

Hypothesis H3 is not supported. There is no significant difference in correctness (Table 7.19). On average, the solutions had high quality. This suggests that the change effort results are reliable – they are not confounded by low correctness or differences in correctness.

Table 7.19. Summary of Mood's median test on correctness on MF and RD

Changeability Indicator	Popul. median	Group	N	Group Median	N<	N>=	Chi-Sq	p-value	Holms alpha= 0.1/(15-i+1)
Correctness *c1*	6	MF RD	17 19	6 6	4 5	13 14	0.04	0.847	0.1000 (i=15)
Correctness *c2*	4	MF RD	16 19	6 4	6 12	10 7	2.29	0.130	0.0111 (i=7)
Correctness *c3*	5	MF RD	16 9	5 6	9 4	7 5	0.32	0.571	0.0500 (i=14)

7.2.3.5 Subjective Change Complexity (H4)

Hypothesis H4 is partially supported. There are no significant differences for change tasks *c1* and *c2* (Table 7.20). For change task *c3*, subjects assigned to the RD design were less confident about the correctness of the solution. There is some evidence that subjects assigned to the RD design also thought it was more difficult to solve change task *c3* (p=0.033). However, this result is not significant with respect to Holm's adjusted alpha-value, which in this case requires a p-value less than 0.0091.

Table 7.20. Summary of Mood's median tests on subjective task difficulty

Changeability Indicator	Popul. median	Group	N	Group Median	N<	N>=	Chi-Sq	p-value	Holms alpha= 0.1/(15-i+1)
Subj. Task Difficulty *C1*	1	MF RD	17 19	1 1	12 11	5 8	0.63	0.429	0.0167 (i=10)
Subj. Task Difficulty *C2*	3	MF RD	17 19	2 3	10 7	7 12	1.74	0.187	0.0125 (i=8)
Subj. Task Difficulty *C3*	3	MF RD	16 16	2 3	10 4	6 12	4.57	0.033	0.0091 (i=5)
Confidence *C1*	5	MF RD	17 19	5 5	11 14	6 5	0.34	0.559	0.0333 (i=13)
Confidence *C2*	4	MF RD	17 19	5 4	7 13	10 6	2.70	0.101	0.0100 (i=6)
Confidence *C3*	3	MF RD	16 16	4 3	3 14	13 2	15.2	**0.000**	0.0077 (i=3)

7.2.3.6 Structural Stability (H5)

Figure 7.14 suggests that the structure of the MF design is affected more than the RD design when subjected to the changes *c1*, *c2* and *c3*. The data is based on ten randomly selected solutions to the change tasks, five for the MF design and five for the RD design. In particular, the average class size (*CS*) and the average import coupling to non-library classes (*OMMIC*) change much more for the MF design than for the RD design.

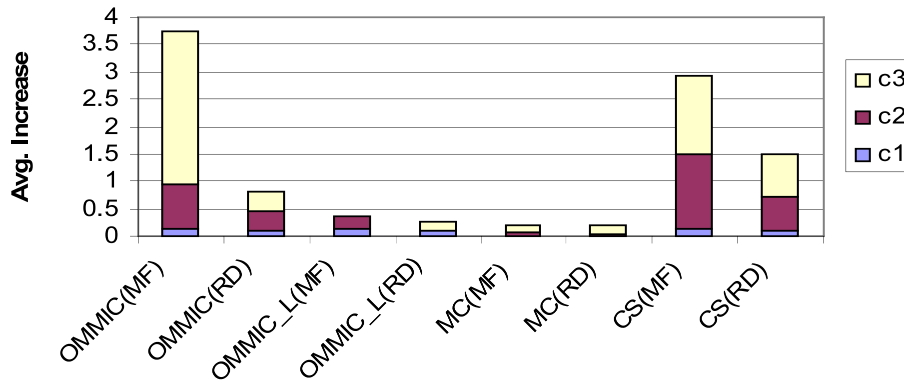

Fig. 7.14. Changes in structural attribute measures of the MF and RD designs after implementing change task *c1*, *c2* and *c3*

The results also indicate that the RD design requires smaller changes (in SLOC added+deleted+modified) for the third change task (Figure 7.15). More interestingly, the changes in these measures are not reflected by corresponding changes in change effort, correctness and subjective change complexity. However, a qualitative assessment of the resulting MF design after changes *c1*, *c2* and *c3* suggests that the *FrontPanel* class of the MF design is indeed becoming a "maintenance nightmare", piling up with more and more responsibilities and high class-level coupling.

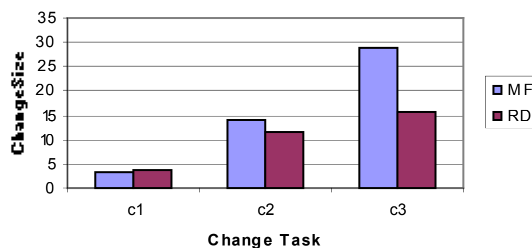

Fig. 7.15. Comparison of the size of each change task for the MF and RD designs

7.2.3.7 Attempting to Explain the Results

Investigating the delivered solutions and the comments given by the subjects on the change task questionnaires, we attempt to explain why there is a significant difference in change effort for the design alternatives.

One difference between the designs is size. The RD design is larger than the MF design. Although size in general may be an important contributor to complexity, we do not believe that the difference in size is that important for the MF and RD designs. Both designs are "small". Furthermore, most of the difference in size is due to simple initialization code in the RD design (e.g., declaration of identifiers and construction of

the objects). In our opinion, this initialization code is quite simple compared with the remainder of the RD design.

For change task *c1* (the "return button"), the solution is *identical* for the two design alternatives, involving two small changes to the *CoffeeMachine* class after determining that the *CashBox* class already contains a *returnCoins* method. Still, it required less effort to understand how to solve *c1* on the MF design compared with the effort to understand the same solution for the RD design. With regards to the RD design, and in particular for change tasks *c2* and *c3*, we found comments such *as "I keep nesting through the classes, but it is too complicated – I give up".* Although the MF design has classes with higher coupling, the dynamic depth of the message interactions among classes to implement a given functional scenario is significantly smaller than for the RD design. Thus, it may be more difficult to perform a systematic trace of the RD design. The fact that the subjects had access to a message sequence chart (Appendix B.4) did not seem to help much. Furthermore, for change task *c3*, the RD design involved changing four classes whereas only two classes had to be changed in the MF design. The same change task was also reported to have higher subjective complexity for the RD design than for the MF design. Thus, we believe that the number of classes changed and the depth of the message interactions among classes are important contributors to the complexity of a design.

These results are supported by existing theory. The first mental representation programmers build to understand completely new code is a control flow abstraction of the program (von Mayrhauser *et al.*, 1997). Some developers use a *systematic* approach (e.g., line-by-line) to build the control-flow abstraction. Other developers used a more *opportunistic* approach, studying code in an as-needed fashion based on hypotheses guided by clues in the code (von Mayrhauser *et al.*, 1997). According to this theory, one may expect an increase in the time required to understand how to implement a change when the amount of collaboration between objects participating in the implementation of a functional scenario increases. Furthermore, the effect may be larger for programmers with a systematic approach.

To assess whether the solution approach affected the change effort for each change tasks, we performed a one-way analysis of variance using "solution approach" as the explanatory factor with three levels, and the total time for a given change task (e.g., *c2*) as the response variable. The solution approach was given by the subjects on the change task questionnaire (Appendix B.3), and coded as follows for the analysis.

1. explorative: Subject characterized the solution approach as 1 or 2
2. mixed: Subject characterized the solution approach as 3 or 4
3. systematic: Subject characterized the solution approach as 5 or 6

The solution approach does not necessarily correspond directly to "opportunistic" versus "systematic" program understanding using von Mayrhauser's terminology. However, it seems plausible that explorative programmers are also more "opportunistic" than the systematic programmers.

For change task *c1* there was no difference in the change effort depending on the solution approach. The change task may be too small to uncover differences in change effort due to the solution approach. For the larger change task *c2*, the exploratory programmers were significantly faster than the systematic programmers on the RD design (Table 7.21). For change task *c2*, the exploratory programmers were actually

slower than the systematic programmers on the MF design (Table 7.22), although the difference in change effort is not significant. For change task *c3*, the effect of the solution approach was similar to change task *c2*. However, there were too few subjects completing the *c3* task for the RD design to give reliable results. The results based on change tasks *c1* and *c2* can be summarized as follows.

- The solution approach may have a significant impact on the change effort.
- The effect of the solution approach on the change effort depends on the size of the change and on the design approach. The RD design seems to be better suited for an explorative solution approach than the MF design.

Table 7.21. ANOVA for the solution approach on change task c2 for the RD design

```
Analysis of Variance for Total c2
Source     DF        SS        MS        F        P
Strategy    2     663.7     331.9     5.59    0.014
Error      16     950.3      59.4
Total      18    1614.0
                                   Individual 95% CIs For Mean
                                   Based on Pooled StDev
Level      N      Mean     StDev  -------+---------+---------+---------
expl.      9    19.111     5.442   (----*-----)
mixed      8    25.875     9.015         (-----*-----)
syst.      2    38.500    12.021                (-----------*----------)
                                  -------+---------+---------+---------
Pooled StDev =    7.707                 20        30        40
```

Table 7.22. ANOVA for the solution approach on change task c2 for the MF design

```
Analysis of Variance for Total c2
Source     DF        SS        MS        F        P
Strategy    2      99.0      49.5     1.26    0.314
Error      14     551.0      39.4
Total      16     650.0
                                   Individual 95% CIs For Mean
                                   Based on Pooled StDev
Level      N      Mean     StDev  ---------+---------+---------+-------
expl.      7    18.857     6.594                  (---------*---------)
mixed      7    14.286     5.851         (----------*---------)
syst.      3    13.333     6.506  (---------------*--------------)
                                  ---------+---------+---------+-------
Pooled StDev =    6.273                 10.0      15.0      20.0
```

7.2.4 Summary of Results

In the coffee-machine study, cohesion was effectively increased by splitting the "mainframe" class, and delegating some of its functional responsibilities to several smaller classes, resulting in the RD design. The RD design also had significantly lower class-level coupling. The RD design adhered better to Coad and Yourdon's design principles than the MF design. However, the RD design contained twice as many classes and slightly more code (in SLOC) compared with the initial, mainframe design. With respects to the given change tasks, the results can be summarized as follows:

- The responsibility-driven (RD) design requires significantly (20-50%) more change effort than the alternative mainframe (MF) design.
- The RD design is harder to understand for the "average" programmer than the MF design, and the learning curve is not better for the RD design than for the MF design.
- The RD design does not result in fewer errors than the MF design.
- The RD design may have higher structural stability than the MF design.

Although one must be careful when generalizing results based on a single study, the results indicate that using delegation of responsibilities to reduce class-level coupling and increase class cohesion may not necessarily improve the changeability of a design. On the contrary, the resulting structure may contain deeply nested class interactions, which average programmers may find difficult to understand. Thus, decreasing coupling and increasing cohesion may *increase* complexity and the costs of changes.

The only indicator that shows potential benefits of the RD design is structural stability. In the RD design, new functionality is divided among the collaborating classes. In the MF design, most subjects piled the code onto the already overloaded "mainframe" class. However, what are the practical consequences of the potentially increased stability? In our study, the changes in structural attributes are not reflected by external quality attributes (e.g., increased change effort and decreased correctness). Thus, when does the added "stability" of the responsibility-driven design justify the increased complexity and costs of changes? For the coffee-machine, it is difficult to envision a sufficient number of future changes to justify the more complex responsibility-driven design. The mainframe approach works well for the types of changes likely to occur. Furthermore, we believe that a prerequisite for achieving the potential advantage of the RD design is that the programmers are confident with the design, and understand the abstract delegations of responsibilities so that they do not break the underlying structure. Our results indicate that this understanding may be difficult to achieve for the average programmer.

7.2.4.1 Comparing the Results with Related Research

Current research in object-oriented design quality often concludes, based on empirical data, that classes with low coupling and/or high cohesion are less error-prone, easier to maintain, etc. There is a growing body of results indicating that measures of structural attributes such as coupling, cohesion, inheritance depth, etc. can be reasonably good predictors of development effort and product quality (Li and Henry, 1993; Chidamber and Kemerer, 1994; Basili *et al.*, 1996b; Daly *et al.*, 1996; Briand *et al.*, 1999d; Briand *et al.*, 2000). Thus, it seems conceivable that such measures can be used to compare the changeability of alternative designs. However, for practical reasons, many of these studies have validated the measures by comparing *different* systems or different classes within one system. For example, in (Briand *et al.*, 1997a; Briand *et al.*, 1999a), they investigated whether a "good" design (adhering to Coad and Yourdon's design principles) was easier to maintain than a "bad" design. The results strongly suggest that Coad and Yourdon's design principles have a positive effect on the maintainability of object-oriented designs. However, as pointed out by the authors, the designs represented two different systems – a temperature controlling

system and an automatic bank teller machine, respectively. Thus, it is difficult to determine what the practical consequences of the results are. For example, *how* can coupling be reduced without increasing other attributes that also contribute to the complexity of the software? In this paper, we compare alternative designs of the *same* system. While this approach introduces new problems (e.g., group assignments, how not to bias the designs and the change tasks), it enables us to assess how different design tradeoffs of the same system actually affect the overall complexity.

In (Sharble and Cohen, 1993), one of the few experiments comparing alternative OO technologies was reported. The authors conducted an experiment where they compared a data-driven and a responsibility-driven design method. Two systems were developed based on the same requirement specification – using the data-driven and the responsibility-driven design method, respectively. Structural attribute measures of the two systems were collected and compared. Based on the measured values, the authors suggested that responsibility-driven design produced higher quality software than data-driven design, because the responsibility-driven method resulted in designs with less coupling and higher cohesion than the data-driven method. We believe it may be premature to draw such conclusions. Whether the design measures used in the experiment actually measured "quality" was not empirically validated. In other words, the experiment did not involve any direct measurement of external quality attributes.

The combined results of (Briand *et al.*, 1997a; Briand *et al.*, 1999a), (Sharble and Cohen, 1993) and the results presented in this section can be summarized as follows:

- A system adhering to Coad and Yourdon's design quality principles is easier to maintain than *another* system not adhering to those principles (Briand *et al.*, 1997a; Briand *et al.*, 1999a).
- Responsibility-driven design may result in lower coupling between classes and higher class cohesion (Sharble and Cohen, 1993).
- However, a practical concern is *how* to adhere to design quality principles such as those proposed by Coad and Yourdon. Reducing coupling and increasing cohesion of the *same* system may result in changing other aspects of the design that contribute to an *increase* in system complexity.

7.2.5 Threats to Validity

The external validity of this study depends on, for example, the choice of design alternatives, the choice of change tasks, and how representative the sample is of the population. We cannot eliminate these threats within the context of this study. The ultimate means to improve the validity of the study is by replication, using other subjects, other design alternatives and other change tasks. The main experiment may be viewed as a replication of the pilot experiment with different population samples and group assignments. However, it may be more important to use different design alternatives and other change tasks. In addition, internal validity may be threatened by, for example, skewed group assignments, ambiguous questions and otherwise unclear experimental materials. This is elaborated in the following sections.

7.2.5.1 *Experimental Materials*

Conducting a pilot experiment effectively results in "throwing away" data, but such an investment may significantly reduce threats and, hence, improve the validity of the study. For example, we used the pilot experiment to evaluate and improve the quality of the experimental materials before the main experiment took place.

Mocca

With regards to the Mocca programming language, the subjects of the pilot-experiment reported that Mocca was very easy to learn. Two of the subjects had problems understanding how to program in Mocca, but they had no previous experience with object-oriented programming. Furthermore, informal discussions with the subjects indicate that Mocca did not restrict their choice of solutions for the given change tasks on the coffee-machine designs. Among the subjects of the main experiment, all subjects had previous experience with Java and similar OO programming languages.

Written Materials

The evaluation of the pilot experiment resulted in important improvements of the design descriptions and the change task descriptions. Some subjects had misunderstood certain aspects of the change task descriptions. Furthermore, one subject had misunderstood how and where the code was supposed to be written. Although the process had been explained in detail during session 1 of the pilot-experiment, we discovered that there was a need to be *extremely* clear and explicit in the written materials to avoid confusion and misunderstandings.

One problem we found after the main experiment was that one of the messages in the message sequence chart for the RD design had an incorrect sequence number. For developers relying on the MSC for understanding the design, this may have influenced the time to implement the tasks, in particular for task *c2*. Although this threat cannot be ruled out, it is in our opinion very unlikely that this "bug" can explain the large difference in change effort. Otherwise, we believe that the written materials of the main experiment were of high quality.

7.2.5.2 *Size and Choice of Design Alternatives and Change Tasks*

The coffee-machine designs and the changes to them are small. The RD design may support "opportunistic" programmers better than the MF design. As the size of the programs increases, memory limitations may eventually result in that it becomes too difficult to use a systematic approach, even for the "MF" type of designs. Thus, different results may have been obtained if the programs were larger. This threat to external validity should be considered in future experiments.

With regards to internal validity, it is possible that adding more than three change tasks would have produced different results:

- Adding a fourth change task may have resulted in a total "breakdown" of the MF design.

- It is possible that most of the system learning occurred during change task *c3* for the RD design, after which subsequent changes would have been simple to understand.

7.2.5.3 Pen and Paper

The changes were coded with pen and paper. This represents another important threat to the external validity. Using a computer one has access to advanced editors, multiple windows, class browsers, etc. Some subjects preferred an exploratory approach to changing the program, which may be difficult to do with pen and paper compared with using a computer. For this experiment, the designs and the change tasks were small. Furthermore, there was a quite even distribution of subjects characterizing their solution approach as exploratory for the MF and RD designs. Finally, the students are accustomed to working with pen and paper programs on their written exams. This means that the advantage of using a computer is probably not that great.

Using a computer would have introduced many new problems regarding training, learning effects and biases towards certain solution approaches depending on the available tool functionality. In this particular experiment, it was in our opinion a better approach to use pen and paper rather than a computer. Still, the only way to eliminate the resulting threats is to replicate the experiment using computers instead of pen and paper.

7.2.5.4 Subject Selection

One important question regarding external validity is whether the subjects form a representative sample of the population. The subjects (mostly undergraduate students, but also some graduate students and professional developers) of the experiment may not be representative of the "general programmer". Furthermore, according to Cockburn, the MF design is typical of the initial designs most students propose. Thus, it is possible that the MF design has an unfair advantage when using students as experimental subjects. The results may have been quite different if the subjects were OO design experts. Thus, we cannot rule out that the subject selection may have biased the results.

Another threat is whether some subjects actually have read Cockburn's article series or otherwise knew the details of the designs prior to the experiment. Because of randomization and the number of subjects involved in the experiment, we believe it is very unlikely that this have affected the results of the experiment.

7.2.5.5 Group Assignment

A serious threat to the validity of the results of *between-subject* experiments is the group assignment (Briand *et al.*, 1999a). It is difficult to ensure that the skill levels of the two groups are approximately equal. In our case, we used the results of the common calibration task as an indicator of the skill level of each subject. The randomized block group assignment in the pilot experiment became skewed towards higher skills for subjects assigned to the RD design, because some subjects did not attend the third session of the experiment (Figure 7.10). Still, the average change effort for the RD design was significantly higher than for the MF design (Figure

7.11). Thus, for the pilot-experiment, the uneven group assignment actually *strengthens* the results.

In the main experiment, we checked the skill level of the two randomized groups by calculating confidence intervals for the mean effort to implement the common calibration task:

```
                   Calib.                    95% CIs For Mean (minutes)
Group        N      Mean      StDev  --+---------+---------+---------+----
MF          17     46.24      14.22    (------------*-------------)
RD          19     50.37      13.70             (------------*------------)
                                     --+---------+---------+---------+----
                                        40.0      45.0      50.0      55.0
```

The confidence intervals show that the 17 subjects assigned to the MF design on average performed slightly better on the calibration task than the 19 subjects assigned to the RD design, that is, the opposite of what was the case in the pilot experiment. The difference in means is not significant, however. We also checked whether more even group assignments might have produced results that are inconsistent with the results of the main experiment. First, we created two blocks based on the calibration task effort, using the median (49 minutes) as a boundary. We then "balanced" the group assignment by randomly removing subjects from the initial groups such that an equal number of subjects (seven, in our case) remained in each block for each group (Table 7.23).

Table 7.23. Initial and adjusted group assignment cross-tabulated on skill level

Block (minutes)	Initial Group Assignment (count)		Adjusted Group Assignment (count)	
	MF	RD	MF	RD
<49	10	7	7	7
>=49	7	12	7	7
Total	17	19	14	14

Furthermore, we tested whether the mean calibration task efforts for the adjusted groups were different, using a two-sided T-test on the difference in means (*H1,* Table 7.24). Finally, we tested whether the mean total effort to implement change tasks *c1+c2+c3* were lower on the MF design than on the RD design, using a one-sided T-test on the difference in means (*H2*). This process was repeated six times (Run 1–6)[6]. The results are shown in Table 7.24. The sub-samples of the initial groups have very even mean effort to implement the calibration task (p-values from 0.66 to 0.87). Furthermore, the differences in mean total effort to implement *c1+c2+c3* are consistent with the results presented in Section 7.2.3 (p-values from 0.0012 to 0.024). Thus, we have no reasons to believe that the group assignment threatens the results of this study.

[6] There are (12!/7!5!)(10!/7!3!)=95040 ways to select such balanced sub-samples from the initial group assignment. We selected only six of these possible samples at random.

Table 7.24. Adjusted results based on sub-samples of the original group assignment

Run	Group	N	Effort Calib.	H1:Unequal groups?	Effort c1+c2+c3	H2: mu(MF)<mu(RD)?
1	MF	14	46.6	p = 0.84	47.4	p = 0.0023
	RD	14	47.6		59.6	
2	MF	14	44.9	p = 0.66	49.9	p = 0.0087
	RD	14	46.7		60.5	
3	MF	14	45.3	p = 0.86	50.5	p = 0.0071
	RD	14	46.0		61.3	
4	MF	14	44.8	p = 0.85	50.9	p = 0.024
	RD	14	45.6		59.7	
5	MF	14	46.6	p = 0.66	47.4	p = 0.0012
	RD	14	45.1		60.4	
6	MF	14	45.6	p = 0.87	47.1	p = 0.0024
	RD	14	46.2		59.1	

7.2.6 Future Work

To further explain the results of this experiment, we are in the process of conducting a follow-up experiment with professional programmers in industry where the subjects "think aloud" while trying to understand the change tasks. The comments and actions made by the subjects are carefully recorded and subsequently analyzed. The preliminary results from that experiment suggest that instead of creating "hypotheses" about how the design works (i.e., a more opportunistic solution approach), many subjects perform a systematic trace of the functionality related to a given change task. Thus, the deeply nested interactions among classes to implement a given functional scenario of the RD design may, to some extent, explain the increase in change effort compared with the MF design. For this reason, we are also investigating whether *dynamic* coupling measures (to measure the scenario depth) are useful in building predictive models of the changeability of object-oriented designs (Section 7.3).

7.3 Definition and Evaluation of Dynamic Coupling

This section presents initial ideas and preliminary results on using dynamic coupling measures to assess the effort to understand and implement changes to a scenario. To our knowledge, very little research has been done on the investigation of dynamic coupling, in particular with regards to its relationships to changeability.

Dynamic coupling measures based on the underlying concept of *role-models* were proposed in Chapter 6. In this section, a preliminary empirical validation is provided. An important aspect of such a validation is to ensure that the measures actually capture what they are intended to capture. Unlike many static coupling measures, the proposed dynamic coupling measures are not surrogate size measures. The results show that the measures capture several distinct dimensions of dynamic coupling. Furthermore, the coupling measures are used to build reasonably accurate models for predicting *hot-spots* and *ripple effects* within a given functional scenario.

The remainder of this section is organized as follows. Section 7.3.1 describes a case study used to evaluate the measures. Dynamic coupling data and change data from ten versions of the Ooram case tool is used to provide a preliminary validation the measures. Section 7.3.2 illustrates how the changes to classes identified by the dynamic coupling measures can be assessed with change complexity measurement. Section 7.3.3 evaluates whether the dynamic coupling measures can explain the change proneness of classes. Such models can be useful to identify "hot-spots" and unstable classes in a design. Section 7.3.4 uses the change data and the coupling measures to build prediction models for common changes, which in turn may be an indicator of ripple effects in a given change scenario. Section 7.3.5 summarizes and describes future research.

7.3.1 The Case Study

To evaluate the dynamic coupling measures, a case study was conducted. The software system studied is a commercial object-oriented analysis and design CASE tool – the Ooram system (Reenskaug *et al.*, 1995). The collected data was based on nine maintenance releases (version *g01 – g09*) of the system, which in turn were based on the major system version called Ooram Version *g*. The nine maintenance releases were produced within a time span of approximately one and a half years. The system is implemented in VisualWorks SmallTalk and consists of more than 1000 classes and close to 300 KSLOC.

7.3.1.1 Collection of the Change Data

The dependent variables of the study were collected from change data for the system. For each of the 10 versions of the Ooram system, versions *g00* to *g09*, the SmallTalk "image" was dumped to ASCII-files. Each file corresponded to one class, and was machine formatted using a standard VisualWorks utility. Using some shell-scripts, the size of each class (in SLOC) and the number of lines of code added to and deleted from each class between two successive versions were calculated using Unix *diff*. These elementary measures were used to calculate change profile measures, change

proneness and common changes, as described in Sections 7.3.2, 7.3.3 and 7.3.4. Based on the elementary measures, a number of system-level summary measures were also calculated (Table 7.25):

- *System Size* is the total number of SLOC for the system
- *SLOC Add* is the number of SLOC added compared with the previous version
- *SLOC Del* is the number of SLOC deleted compared with the previous version
- *CC* is the total class count
- *Change Span* is the number of classes changed (i.e., at least one line added or deleted) from one version to the next
- *Class Add* is the number of new classes
- *Class Del* is the number of classes deleted
- *AvgCS* is the average size (in SLOC) of the classes

Table 7.25. Summary measures for the minor releases *g01–g09* of Ooram

Version	Build date (dd/mm/yy)	System Size	SLOC Add	SLOC Del	CC	Change Span	Class Add	Class Del	AvgCS
g01	20/08/97	272840	4793	3383	1107	164	8	10	246
g02	14/11/97	277030	7169	2979	1120	173	13	0	247
g03	28/11/97	277030	9	9	1120	3	0	0	247
g04	26/01/98	277031	44	43	1120	27	0	0	247
g05	18/03/98	281843	9759	4947	1133	219	19	6	249
g06	12/05/98	284959	8284	5168	1143	142	11	1	249
g07	25/09/98	287796	5033	2196	1152	208	10	1	250
g08	08/10/98	288971	1902	727	1157	66	5	0	250
g09	22/12/98	291503	4688	2156	1164	144	7	0	250

7.3.1.2 Collection of the Coupling Measures

The coupling measures were collected using a reverse-engineering utility implemented on version *g09* of the Ooram-system. This utility generates role models corresponding to a run-time session. The actual implementation is specific to the VisualWorks programming environment and the Ooram code. In principle, the implementation modifies source code at run-time (using the VisualWorks "doit"-command) such that each object is associated with a "shadow object". These shadow objects intercept all messages sent and received at run-time. A repository is updated in real time with information about the sender object, receiver object, sender class and receiver class for each message. At any time during the run-time session, the repository can be dumped to an ASCII-file (Table 7.26). The ASCII-file contains a row for each pair of interacting roles (client and server) at the object-level and at the class-level. In Table 7.26, *OM* is the number of distinct methods used in the interaction between the client object and the server object. *OD* is the total number of messages sent between the objects. *CM* is the number of methods used in the interaction between the client class and the server class. *CD* is the total number of messages sent between the client class and the server class. The ASCII-file was

imported into a relational database providing a convenient way to calculate the 12 dynamic coupling measures from the raw coupling data.

A given, well-defined functional scenario limited to one important GUI dialog of the Ooram-system was selected as the target of the investigation. The scenario was executed while the dynamic coupling parser intercepted each run-time message. The algorithm used was as follows.

1. Start the Ooram system
2. Load the reverse-engineering utility into memory
3. Start the functional scenario
4. Iterate: Perform some (new) sub-function within the scenario
5. Dump ASCII-file
6. If new classes are added or values (OM, CM) in the ASCII-file are different from the previous iteration, goto 4. Otherwise, goto 7
7. End run-time session

As will be pointed out in Section 7.3.5, there is considerable room for improvement of this algorithm, depending on the intended use of the measures.

Table 7.26. ASCII-dump (edited) showing raw coupling data. In the selected scenario, the object-level coupling between *{listview2, rmrolemodelwithscenarios2}* is reflected as class-level coupling between *{listview2, rmnode2}* and *{listview2, rmrolemodel2}*.

```
'From VisualWorks®, Release 2.5.2 of September 26, 1995
{IMAGE: g09.im3} on October 5, 1999 at 2:38:15 pm'!

Object-level Coupling
Client role         Server role                  OM      OD
----------------------------------------------------------------
listview2           rminteraction                1       4
listview2           rmport2                      1       18
listview2           rmrolemodelwithscenarios2    2       3
<other role pairs>
…
Class-level Coupling
Client role         Server role                  CM      CD
----------------------------------------------------------------
listview2           rminteraction                1       4
listview2           rmnode2                      1       1
listview2           rmport2                      1       18
listview2           rmrolemodel2                 1       2
<other role pairs>
```

7.3.1.3 Descriptive Statistics of the Coupling Measures

The descriptive statistics of the coupling and size measures based on version *g09* for the identified classes are shown in Table 7.27. Note that the mean values for a given import coupling measure (e.g., *IC_OA*) is always equal to the mean values for the corresponding export coupling measure (e.g., *EC_OA*) because the total number of messages sent is always equal to the total number of messages received. There are large differences between the lower 25th percentile, the median, and the 75th percentile, however.

Table 7.27. Descriptive statistics for the measures collected from the given scenario

Variable	Max	75%	Median	25%	Min	Mean	StDev
CS	2590	1499	638	340	21	924.0	736.0
IC_OA	7	4	2.5	1	0	2.5	2.0
IC_OM	35	13.75	7	2	0	8.7	8.8
IC_OD	5962	267	42	7	0	715.0	1642.0
IC_CA	9	4	2.5	2	1	3.1	2.0
IC_CM	23	9	5	3	1	6.9	5.6
IC_CD	5513	737	60	12	5	712.0	1397.0
EC_OA	8	5.5	0	0	0	2.5	3.2
EC_OM	39	17.5	0	0	0	8.7	12.2
EC_OD	4383	760	0	0	0	715.0	1390.0
EC_CA	13	6	1	0	0	3.1	3.6
EC_CM	23	12.75	1.5	0	0	6.9	8.2
EC_CD	6250	1090	6	0	0	712.0	1412.0

7.3.1.4 Principal Component Analysis

Principal Component Analysis (PCA) was used to analyze the covariance structure of the measures. Based on the PCA, the number of underlying dimensions of the data can be quantified. The number of principal components is usually decided based on the amount of variance explained by each component, using the rule of thumb of eigenvalues (variances) larger than 1.0. In this case, PCA with four components should be used for the interpretation. Table 7.28 shows the results from PCA using the Varimax rotation to ease the interpretation.

The results from the principal component analysis show that most of the dynamic coupling measures are not surrogate size measures; they represent a significant amount of variance in the data set not accounted for by the size measure. None of the measures are correlated with *CS*. Only the class-level export coupling measures belong to the same principal component as size, but there is also some overlap with class-level import coupling and *CS*. The dimension *strength of coupling* is not represented in different principal components. Thus, it may be sufficient to measure either association coupling or method coupling or dynamic coupling. Based on the coefficients of the rotated components, the dimensions are interpreted as follows:

- **PC1:** Object-level Export Coupling
- **PC2:** Class-level Import Coupling
- **PC3:** Class-level Export Coupling/Size
- **PC4:** Object-level Import Coupling

Table 7.28. Rotated Principal Components

Variable	PC1	PC2	PC3	PC4
CS	-0.351	0.465	**0.503**	-0.156
IC_OA	0.195	0.117	-0.192	**0.893**
IC_OM	0.361	0.309	-0.011	**0.848**
IC_OD	0.048	**0.526**	0.098	**0.793**
IC_CA	0.098	**0.854**	0.017	0.374
IC_CM	0.390	**0.885**	-0.002	0.122
IC_CD	0.106	**0.890**	0.058	0.314
EC_OA	**0.895**	0.144	0.104	0.297
EC_OM	**0.927**	0.128	0.119	0.212
EC_OD	**0.838**	0.151	0.223	0.070
EC_CA	0.284	0.052	**0.925**	-0.052
EC_CM	**0.529**	0.100	**0.818**	-0.021
EC_CD	0.008	-0.059	**0.937**	-0.010
Eigenvalue	3.191	2.985	2.781	2.564
% Variance	0.245	0.230	0.214	0.197
%Cumulative	0.245	0.475	0.689	0.886

7.3.2 Assessing Changeability with the Change Profile Measures

The goal of this section is to illustrate how change profile measurement (CPM) can be used in practice – to assess trends in how changes propagate through the structure of the selected scenario. This, in turn, may be useful to assess changeability decay within the selected scenario. Based on the 24 identified classes, change profile measures for the scenario were calculated for version *g01* to *g09* according to the descriptions provided in Chapter 6.

Figure 7.16 shows the resulting change complexity measures. No changes were performed on the scenario for version *g03* and *g04,* hence they are excluded from the graph. The graph of *Change Span* shows the percentage of classes changed for the scenario (i.e., number of classes changed in version *g0x* divided by 24) versus the percentage of classes changed for the total system (i.e., number of classes changed in version *g0x* divided by the total number of classes *CC* for each version of the Ooram system). These measures show that the relative change ratio is higher for the selected scenario than for the remainder of the system.

Thus, having poor design for this part of the system may have bigger consequences on the total change effort from version *g01* to *g09* than for other parts of the design. However, there is no trend towards an *increase* in change span for the scenario compared with the rest of the system. Furthermore, the graph of *the Class Size Change Profile* shows that the average size of the 24 classes (*AvgCS*) increased slightly from g01 to g09, but during the same period, the weighted average size of the classes *changed* (*CS_CP*) decreased quite dramatically from *g01* to *g09*. Thus, there is a trend towards changing the smaller classes of the scenario more than the larger classes.

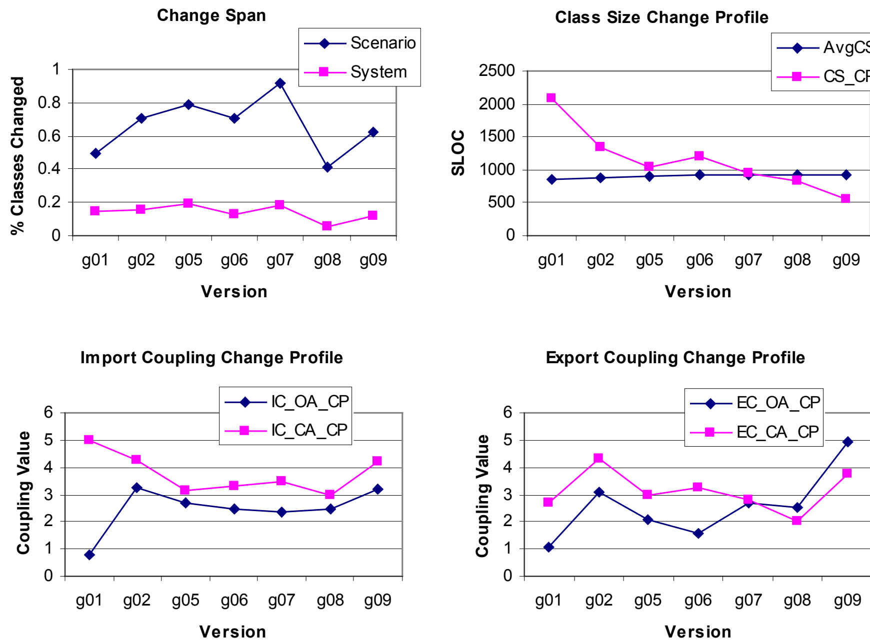

Fig. 7.16. Change Complexity Measurement for the change scenario from version *g01* to *g09*

The change profile measures *IC_OA_CP*, *IC_CA_CP*, *EC_OA_CP* and *EC_CA_CP* are also plotted (Figure 7.16). To calculate these measures, the association level coupling measures *IC_OA*, *IC_CA*, *EC_OA* and *EC_CA* collected from release *g09* were used in conjunction with the change profile (*CP*) for each release (*g01* to *g09*) of the scenario. Thus, an assumption is made that the coupling data at the association level have been stable across the releases. Based on the resulting measures, trends in the coupling of the classes being *changed* are visualized. Studying the relative differences in the class-level versus object-level measures, there seem to be a trend towards changing large classes with high class-level coupling for the early releases (e.g., *g01*) and a trend towards changing small classes with high object-level coupling in the later releases (e.g., *g09*).

The relative magnitude in class-level versus object-level coupling can be used to determine *where* the changed classes are located in the inheritance hierarchy:

- Large classes with low object-level coupling and high class-level coupling were changed in the first two or three releases. These classes are ancestor (framework) classes that are not instantiated directly, because they have low objects-level coupling.
- Once they stabilized, the classes further down in the inheritance hierarchy (that is, classes that are instantiated as objects, and hence having higher relative object-level coupling compared with the class-level coupling) were changed.

According to the developers in Numerica-Taskon, changing framework classes are much more difficult than changing other classes (Section 7.4). Furthermore, results from the Genera case study (Section 7.1.2) indicate that larger classes are also more difficult to change than smaller classes. Thus, this analysis suggests a positive trend in changeability of the selected scenario from *g01* to *g09*. The results presented in this section only illustrate the use of the CPM approach in conjunction with dynamic coupling data of a selected scenario. A more in-depth analysis would be required to improve the validity of the results by, for example, combining the results with expert opinion.

7.3.3 Change Proneness

If class *A* is changed more often than class *B*, then class *A* is more change prone than class *B*. Change proneness have been used in other studies as an indicator of effort (Li and Henry, 1993). This section investigates whether the number of changes to a class depends on the dynamic coupling of the class. This knowledge could subsequently be used to aid in design refactoring[7] (e.g., removing "hot-spots"), when choosing among design alternatives or when assessing changeability decay.

7.3.3.1 Hypotheses and Statistical Analysis

The dependent variable in this study is the total number of changes (*NumChanges)* to each of the 24 classes participating in the scenario from version *g01* to *g09*. None of these classes were deleted or added during the changes in the maintenance releases from *g01* to *g09*. Thus, the number of changes to these classes from version *g01* to *g09* reflects the change proneness of these classes. The independent variables are the class-level size measure *CS* and the twelve dynamic coupling measures of the scenario based on version *g09*. Raw data is provided in Appendix C.

Many of the coupling measures show a positive correlation with *NumChanges*. An important part of the validation is to determine whether the measures may be used to build *better* models than when using only simple measures, such as *CS*. Consequently, it may be appropriate to first test whether the class size measure affects change proneness. Then, one may test whether the dynamic coupling measures are significant *additional* explanatory variables, above what has already been accounted for by size.

Hypothesis formulation for class size:

- *H0*: The number of changes to a class does not depend on the size of the class.
- H_{CS}: The number of changes to a class depends on the size *CS* of the class.

Hypotheses formulation for the dynamic coupling measures:

- *H0*: The number of changes to a class does not depend on the dynamic coupling of the class when the size of the class has been accounted for.
- $H_{CS,x}$: The number of changes to a class depends on the dynamic coupling *x* of the class, above what can be explained by the size *CS* of the class alone.

[7] This assumes that there is a *cause-effect* relationship between coupling and change proneness. By reducing coupling for a class one would expect a reduction in change proneness for that class. However, showing a covariance between two variables is not sufficient to show such causality.

To test hypotheses involving one dependent variable and more than one independent variable, and assuming the independent variables are not correlated (as is the case for the selected variables), multiple linear regression may be used. There are twelve coupling measures and one size measure, resulting in a total of 13 hypotheses to be tested. With that many tests, it is more likely that one discovers empirical relationships by chance, i.e., shotgun correlation. Consequently, the alpha-level is set to $\alpha = 0.05/13 = 0.004$, according to the Bonferroni procedure. The null-hypotheses are rejected when the p-value for H0: $\beta_{CS,X} = 0$ is smaller than 0.004. However, the reader may choose to be less strict by interpreting the actual p-values directly.

7.3.3.2 Results

The results of the tests are shown in Table 7.29. Just because there is a *significant* relationship between some variables it does not mean that the relationship is very useful in building predictive models. Thus, the table also includes the R-Sq to assess the explanatory power of each model and the adjusted R-Sq to assess the prediction ability of each model.

At the pre-determined alpha-level, none of the class-level export coupling measures (*EC_CA, EC_CM, EC_CD*) are significant explanatory variables of *NumChanges* after taking into account the class size measure *CS*. This is perhaps not that surprising since the class-level export coupling measures belong to the same principal component (*PC3*) as *CS.* Regardless of the choice of level of significance α, it is clear that *EC_OA* and *EC_OM* explain considerably more of the data variability of *NumChanges* than the other coupling measures when size has been accounted for.

Table 7.30 shows the results of the best linear regression model for the *NumChanges* independent variable. The same model was also the result when forward, backward and stepwise variable selection heuristics were applied. For this data-set, 72% of the variability of the number of changes to a class is explained by the measures *CS* and *EC_OA*.

Table 7.29. Linear regression used to test hypothesis H_x: $\beta_x \neq 0$

Hypothesis $H_{CS,X}$	Principal Component	p-value H0: $\beta_{CS}=0$	p-value H0: $\beta_x=0$	R-Sq	R-Sq (adj)
H_{CS}	**PC3**	**0.003**	**N/A**	**34.0%**	**31.0%**
$H_{CS,\ IC_OA}$	PC3,PC4	0.001	0.159	40.1%	34.4%
$H_{CS,\ IC_OM}$	PC3,PC4	0.001	0.016	50.2%	45.4%
$H_{CS,\ IC_OD}$	PC3,PC4	0.003	0.120	41.4%	35.8%
$H_{CS,\ IC_CA}$	PC3,PC2	0.007	0.101	42.1%	36.6%
$H_{CS,\ IC_CM}$	PC3,PC2	0.006	0.051	45.2%	40.0%
$H_{CS,\ IC_CD}$	PC3,PC2	0.006	0.154	40.3%	34.6%
$H_{CS,\ EC_OA}$	**PC3,PC1**	**0.000**	**0.000**	**72.0%**	**69.3%**
$H_{CS,\ EC_OM}$	**PC3,PC1**	**0.000**	**0.000**	**64.6%**	**61.2%**
$H_{CS,\ EC_OD}$	PC3,PC1	0.001	0.016	50.3%	45.6%
$H_{CS,\ EC_CA}$	PC3	0.012	0.048	45.5%	40.3%
$H_{CS,\ EC_CM}$	PC3	0.006	0.015	50.5%	45.8%
$H_{CS,\ EC_CD}$	PC3	0.009	0.437	36.0%	29.9%

Table 7.30. Regression model for *NumChanges* using *EC_OA* and *CS* as explanatory variables

```
The regression equation is
NumChanges = 1.84 + 0.403 EC_OA + 0.00197 CS

Predictor          Coef        StDev           T         P
Constant         1.8391       0.4539        4.05     0.001
EC_OA           0.40304      0.07560        5.33     0.000
CS            0.0019701    0.0003316        5.94     0.000

S = 1.151       R-Sq = 72.0%     R-Sq(adj) = 69.3%

Analysis of Variance

Source              DF          SS          MS         F         P
Regression           2      71.491      35.746     26.96     0.000
Residual Error      21      27.842       1.326
Total               23      99.333

Source         DF       Seq SS
EC_OA           1       24.704
CS              1       46.787
```

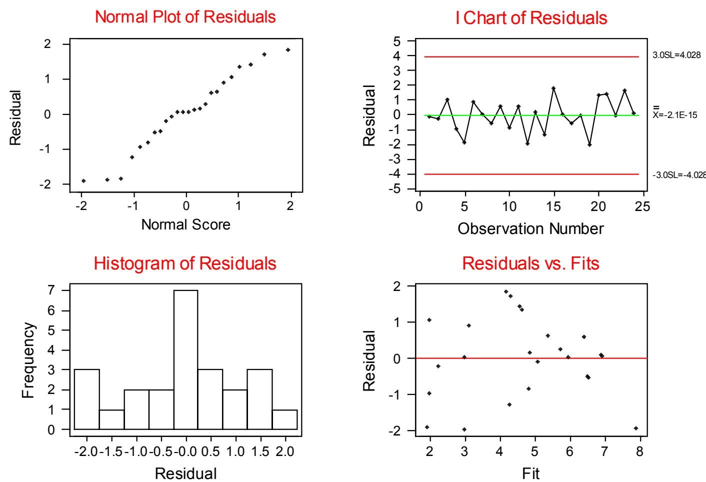

Fig. 7.17. Checking the model assumptions for NumChanges = 1.84 + 0.403 EC_OA + 0.00197 CS

As with any type of hypothesis test, there are a number of model conditions that, if not met, may invalidate the results. For linear regression, the hypothesis tests on the coefficients are based on a number of conditions. These are: (1) that the expected error mean is zero, and assumptions of (2) homogeneous error variance, (3) uncorrelated errors and (4) normally distributed errors. For the resulting model, these assumptions were checked and no serious deviations were found (Figure 7.17).

The results are interpreted as follows. The larger the class (*CS*), the more functionality is allocated to it, and hence it is more likely to be affected by changes.

The higher object-level export coupling of a class, the more objects are dependent on services provided by the object, and hence it is more likely that the class of which the object is an instance will be changed. Furthermore, whether the services provided are implemented by methods in the actual class of the object or in an ancestor class of the object is less relevant for the change proneness of the class.

7.3.4 Using Dynamic Coupling for Impact Analysis

The section investigates whether the dynamic coupling measures can help perform impact analysis. The *ripple effect* refers to the phenomenon that changes made to one part of a software system ripple throughout the system. As pointed out in (Kung *et al.*, 1994), the complex relationships between the object classes due to OO features such as inheritance, polymorphism and dynamic binding make it difficult to anticipate and identify the ripple effect of changes. This means that changes may be prone to errors. Hence, reducing the amount of ripple changes may improve the changeability of the software. In order to reduce ripple effects one needs to identify factors causing ripple effects. However, a simpler task may be to *predict* ripple changes, without necessarily having to reduce the *amount* of ripple changes.

One way to support impact analysis is through formal dependency-analysis of the source code (Kung *et al.*, 1994). However, in (Briand *et al.*, 1999e), a simpler approach was proposed, in which static coupling measures were used to predict *common changes* between pairs of classes. A class pair {*A, B*} has a common change if both classes are changed within the same logical change. Such common changes may in turn be a result of ripple effects. This study investigates whether the *dynamic* coupling measures can be used to identify classes with common changes similar to that outlined in (Briand *et al.*, 1999e), but using dynamic coupling instead of static coupling. More precisely, the goal of our study is to

- investigate whether dynamic coupling affect the probability of common changes, which in turn may be used as an indicator of ripple effects, and
- evaluate a prediction model for common changes. The model is intended to predict the answer to the following question: if class *A* is changed, how likely is it that class *B* is also changed?

Note that common changes (and hence ripple effects) may also be caused by non-functional dependencies such as common programming style, documentation, performance requirements, user interface look-and-feel, etc. This investigation focuses on how *functional* dependencies affect the probability of common changes.

7.3.4.1 Collection of the Measures

To determine which measures are useful indicators of common changes, and subsequently to build and evaluate prediction models, change data and coupling measures were collected for version *g09* of the Ooram system. The following calculations were carried out:

Calculation of Dynamic Coupling Between Pairs of Classes

The dynamic measures described in Chapter 6 were modified to count the coupling between each individual *pair* of classes, using the data exemplified in Table 7.26. With 24 classes in the scenario, there are 276 class pairs. For prediction of common changes in this scenario, the direction of coupling is not relevant, because the order in which the classes are changed is to a large extent a random process (Briand *et al.*, 1999e). Thus, for each class pair {*A, B*}, the coupling measure was calculated as the coupling from class *A* to class *B* summed with the coupling from class *B* to class *A,* as suggested in (Briand *et al.*, 1999e). This results in the six dynamic coupling measures in Table 7.31. In addition to the coupling measures, the size of the class pair (*CS_P*) was calculated. This is just the sum of the *CS* measure for the two classes in the class pair.

Table 7.31. Dynamic coupling measures at the class-pair level

Mapping	Strength	Scope	Name
Object-level	Number of Dynamic messages	Between Pair	OD_P
	Number of Method invocations	Between Pair	OM_P
	Number of Associations	Between Pair	OA_P
Class-level	Number of Dynamic messages	Between Pair	CD_P
	Number of Method invocations	Between Pair	CM_P
	Number of Associations	Between Pair	CA_P

Calculation of Common Changes

The dependent variable is a binary response variable taking the value 1 when there was a common change within a class-pair among the 24 classes in the scenario and the value 0 if there was no common change between a class-pair. In this preliminary investigation, only common changes occurring in version *g09* of the selected scenario were considered. Because these 24 classes collaborate in the implementation of the same functional scenario, the common changes are likely to belong to the same logical change.

7.3.4.2 Identification of Prediction Models for Common Changes

To identify useful dimensions of explanatory variables, principal component analysis was first used to identify the covariance structure of the data (Table 7.32). It shows that object-level coupling and class-level coupling capture different dimensions of coupling. Furthermore, there may be problems with collinearity if several covariates from the same component are used within one regression model. This is considered when interpreting the regression models. The variables selected were determined by the logistic regression model that had the best fit (smallest deviance) between model and data.

Table 7.33 shows the results from stepwise logistic regression on common changes. The column $G - G'$ is the deviance reduction for *nested* models, that is, models where one model is contained in the other such as when doing forward variable selection. By computing the deviance reduction and comparing with percentiles from the chi-square distribution with 1 degrees of freedom, the

significance of the slope of the logistic regression can be evaluated. Referring to Table 7.33, the best univariate model with significant deviance reduction over the constant model is *{OM_P}*. Both *OM_P* and *OA_P* are significant explanatory variables. However, the model including *{OA_P*, *OM_P}* does not yield better fit than *{OM_P}* because these measures belong to the same principal component. The best nested model with significant deviance reduction over *{OM_P}* is *{OM_P, CA_P}*. No further models have significant deviance reductions.

The resulting model is shown in Table 7.34. There are more complicated models providing additional improvements in fit, such as models including interaction terms and indicator variables. However, such models are difficult to interpret and may also overfit the data.

Table 7.32. Rotated Principal Components

Variable	PC1	PC2	PC3	PC4
CS_P	-0.064	-0.121	0.046	**0.988**
OA_P	**0.889**	-0.310	0.021	-0.071
OM_P	**0.871**	-0.314	0.287	-0.035
OD_P	**0.518**	0.038	**0.787**	0.039
CA_P	0.297	**-0.880**	0.069	0.141
CM_P	0.334	**-0.822**	0.294	0.060
CD_P	-0.054	-0.474	**0.804**	0.047
Eigenvalue	2.024	1.885	1.443	1.009
% Variance	0.289	0.269	0.206	0.144
% Cumulative	0.289	0.558	0.764	0.908

Table 7.33. Logistic regression on ripple-effects using forward stepwise variable selection heuristics using the deviance reduction G - G' as selection criterion

x1	x2	G	G - G'	p-value H0: $\beta_{x1}=0$	p-value H0: $\beta_{x2}=0$
OA_P	N/A	**39.477**	**39.477**	**0.000**	N/A
OM_P	N/A	**43.228**	**43.228**	**0.000**	N/A
OD_P	**N/A**	**40.210**	**40.210**	**0.005**	N/A
CA_P	N/A	2.335	2.335	0.126	N/A
CM_P	**N/A**	**6.164**	**6.164**	**0.023**	**N/A**
CD_P	N/A	0.170	0.170	0.678	N/A
CS_P	N/A	2.882	2.882	0.093	N/A
OM_P	OA_P	43.874	0.586	0.109	0.405
OM_P	OD_P	45.654	2.366	0.031	0.291
OM_P	**CA_P**	**48.086**	**4.798**	**0.000**	**0.040**
OM_P	CM_P	45.705	2.417	0.000	0.149
OM_P	CD_P	46.552	3.264	0.000	0.130
OM_P	CS_P	44.813	1.525	0.000	0.212

Table 7.34. Resulting logistic regression model for the prediction of common changes

```
Logistic Regression Table
                                                          Odds
Predictor        Coef       StDev         Z      P      Ratio
Constant     -0.8060      0.1397    -5.77 0.000
OM_P          0.6820      0.1747     3.90 0.000      1.98
Log-Likelihood = -161.726
Test that all slopes are zero: G = 43.228, DF = 1, P-Value = 0.000
```

7.3.4.3 Model Evaluation – Prediction of Common Changes

To determine the accuracy of the prediction models, the data was split such that 75% of the data (207 rows) was selected at random to build prediction models for common changes. Then, the models were evaluated on the *remainder* 25% of the data (69 rows). For the evaluation, a threshold of 0.5 was picked, such that any prediction above that threshold was classified as a common change. If the prediction was below 0.5, the given class pair is predicted to have no change in common. The results can be displayed in a contingency table showing correct positives, false positives, false negatives and correct negatives. Alternative prediction models using the K Nearest Neighbors technique (KNN) and PCA logistic regression were also considered. These techniques may sometimes provide better results for non-linear data and when using correlated independent variables, respectively.

The results for the simplest model using univariate logistic regression with *OM_P* as predictor, is shown in Table 7.35. On the evaluation data, 73.9% of the class pairs were correctly predicted as either having a common change or not, using only *OM_P* as predictor. Among the class pairs predicted to have a common change, 83.3% actually did have a change in common. Thus, if the model *predicts* a common change between class pair *{A, B}*, it is very likely that there *will* be a common change between these classes.

However, one problem with the model is that it is quite conservative. It only finds 10/26 = 38.5% of the actual common changes, that is, 61.5% are false negatives. Using both *OM_P* and *CA_P*, the overall accuracy of the model increases to 75.4%, but there are still 57.7% false negatives (Table 7.36). The models using the KNN technique perform no better than logistic regression. The model that finds the largest number of actual common changes is the PCA logistic regression model, at the expense of slightly more false positives. Which model is "better" depends on the chosen tradeoff between the accepted level of false positives versus false negatives. A higher portion of false positives means that more classes with common changes are found but also that more classes may be investigated for ripple effects in vain.

It is difficult to compare the results in Table 7.36 directly with the results using static coupling in (Briand *et al.*, 1999e) because they used a different evaluation method and the prediction model was developed based on a different data set. To determine whether dynamic coupling is better than static coupling or vise versa, one would at least need to use the same underlying change data. However, to give a preliminary *indication* of the relative merits of dynamic versus static coupling, the main results in (Briand *et al.*, 1999e) are compared with the results reported here. Using static coupling, Briand *et al.* found on average about 50% of the common

changes, which is slightly better than the best results reported here (using PCA regression, 46.1% of the common changes where found). However, the results in (Briand *et al.*, 1999e) had a substantially higher ratio of false positives (50% to 60%) compared with between 4.7% to 9.3% for the best models reported here (Table 7.36). Thus, using dynamic coupling may provide a significant improvement in the *overall* accuracy of the models compared with using static coupling, at the expense of slightly lower sensitivity.

Table 7.35. False Positives and False Negatives Contingency Table using Logistic Regression on OM_P. Overall Accuracy = 51/69 = 73.9%. Correct Positive Ratio = 10/12 = 83.3%. False Positives = 2/43 = 4.7%. False Negatives = 16/26 = 61.5%.

	Actual Common Change	Actual No Common Change	Total
Predicted Common Change	10	2	12
Predicted No Common Change	16	41	57
Total	26	43	69

Table 7.36. Model Evaluation Summary

Model type	Variables	Overall Accuracy	Correct Positive	False Positive	False Negative
Logistic regression	OM_P	73.9%	83.3%	4.7%	61.5%
Logistic regression	**OM_P, CA_P**	**75.4%**	**84.6%**	**4.7%**	**57.7%**
KNN	OM_P	73.9%	83.3%	4.7%	61.5%
KNN	OM_P, CA_P	73.9%	83.3%	4.7%	61.5%
KNN	all variables	55.1%	38.1%	30.2%	69.2%
PCA regression	all variables	73.9%	75.0%	9.3%	53.9%

7.3.5 Summary and Future Work

Dynamic coupling measures based on the concept of role-models have been proposed. A preliminary validation of the measures was conducted using change data from ten versions of the Ooram system. The investigation has demonstrated how dynamic coupling measurement can be used to

- identify classes collaborating in the implementation of a given scenario,
- assess trends in "change complexity" of a given scenario,
- identify hot-spots within an important scenario, and
- build prediction models supporting impact analysis at the scenario-level.

The preliminary results show that it is probably worthwhile to continue the investigation into dynamic coupling and its relationship to changeability. The work poses interesting challenges for future research, presented in the following sub-sections.

7.3.5.1 Comparing Dynamic Coupling with Static Coupling

There are important aspects of coupling that probably cannot be quantified using static code parsers. With *static* coupling measurement, class-level coupling measures quantify the coupling to every other class in the system. *Dynamic* coupling measurement allows us to identify the classes involved in the implementation of a given functional component or scenario of a system. Furthermore, the resulting coupling measures only quantify coupling based on the actual messages sent from and received to the *role* played by the individual classes collaborating in the execution of the given functional scenario. This may explain why the proposed dynamic coupling measures are not just surrogate size measures, unlike many static coupling measures. Finally, dynamic coupling allows the distinction between object-level and class-level coupling. Problems of static coupling measures, such as attempting to determine the target class of polymorphically invoked methods are thereby solved with dynamic coupling measures. So clearly, there are important theoretical advantages with dynamic coupling.

However, an important question is whether the potential benefits of dynamic coupling measures outweigh the cost of collecting them. In the Ooram case study, it was not possible to investigate this question because SmallTalk is a dynamically typed language in which there is no type information associated with identifiers, but rather with the objects themselves. Consequently, it was not possible to quantify coupling from static code parsing. With other object-oriented programming languages such as Java, it may be possible to compare static coupling with dynamic coupling, to assess whether the collection of dynamic coupling data is cost-effective compared with static coupling.

7.3.5.2 Within-Object Coupling

The coupling measures investigated in this case study use messages sent between distinct objects to quantify object-level and class-level dynamic coupling, i.e., *between-object* coupling. A possible extension of the proposed coupling measures could be to also measure *within-object* coupling. To illustrate *within-object* coupling, let object *a* be an instance of class *A,* which is inherited from class *A'*. Let *A* implement the method *mA* and let *A'* implement the method *mA'*. If object *a* sends the message *mA'* to itself from the method source *mA*, the message caused two types of coupling: within-object, object-level coupling for class *A* and *A*, and within-object, class-level coupling between class *A* and *A'*.

For each of the 12 *between-object* coupling measures described in Section 6.2.3, there would be a corresponding within-object coupling measure. Thus, in total there would be 24 dynamic coupling measures. By comparing within-object, object-level coupling (e.g., *IC_OM_**W***) with the corresponding within-object, class-level measures (e.g., *IC_CM_**W***) one may be able to assess the relative amount of functional dependency to classes higher up in the inheritance hierarchy.

7.3.5.3 Using Dynamic Coupling Measures to Support Impact Analysis

In this case study, 24 classes that collaborate in implementing the scenario were identified out of more than 1000 classes. These 24 classes are a good starting point for

impact analysis when changing the scenario. However, changes may ripple through a large variety of dependencies. One cannot rule out that no other classes participate in the scenario in ways that have not been detected by the dynamic coupling parser. The within-object dynamic coupling measures could potentially cover even more of such dependencies, and hence be used to identify more classes potentially affected by ripple effects. However, there are of course *static* dependencies in code as well. It may be interesting to investigate how dynamic coupling based prediction models of the type evaluated here can be combined with traditional static dependency analysis techniques of object-oriented software, e.g., (Kung *et al.*, 1994). Such hybrid techniques could for example use dynamic coupling measures to identify the classes with run-time dependencies within the scenario. Among those classes, the classes with the highest predicted probability of ripple effects may be selected as a starting point for a detailed dependency analysis.

Another use of dynamic coupling is to use the data to generate models of the dynamic behavior of the classes in a selected scenario. Such models may be used to support impact analysis and code comprehension by providing a graphical model of the interactions between collaborating classes. Figure 7.18 gives an example of such a model, generated using the modified Ooram tool, which collects the dynamic coupling measures and then generates the corresponding collaboration model.

7.3.5.4 Using Dynamic Coupling Measures to Assess Understandability

According to von Mayrhauser, the first mental representation programmers build to understand completely new code is a control flow abstraction of the program (von Mayrhauser *et al.*, 1997). Some developers use a *systematic* approach to build the control-flow abstraction. Other developers use a more *opportunistic* approach, studying code in an as-needed fashion based on hypotheses guided by clues in the code (von Mayrhauser *et al.*, 1997). According to this theory, one may expect an increase in the time required to understand how to implement a change when the amount of collaboration between objects participating in the implementation of a functional scenario increases. Furthermore, the effect may be larger for programmers with a systematic approach.

The theories proposed by von Mayrhauser and the results of the coffee-machine experiment indicate that a large portion of the cognitive complexity of object-oriented designs are related to the way the objects collaborate. One way to represent the object collaboration of a functional scenario is in a role-model. The message flow between the roles is quantified using the proposed dynamic coupling measures. However, there is still a need to determine how the dynamic coupling measures can be combined to provide a useful model of cognitive complexity based on the message-flow of a functional scenario. This is an interesting venue for future research.

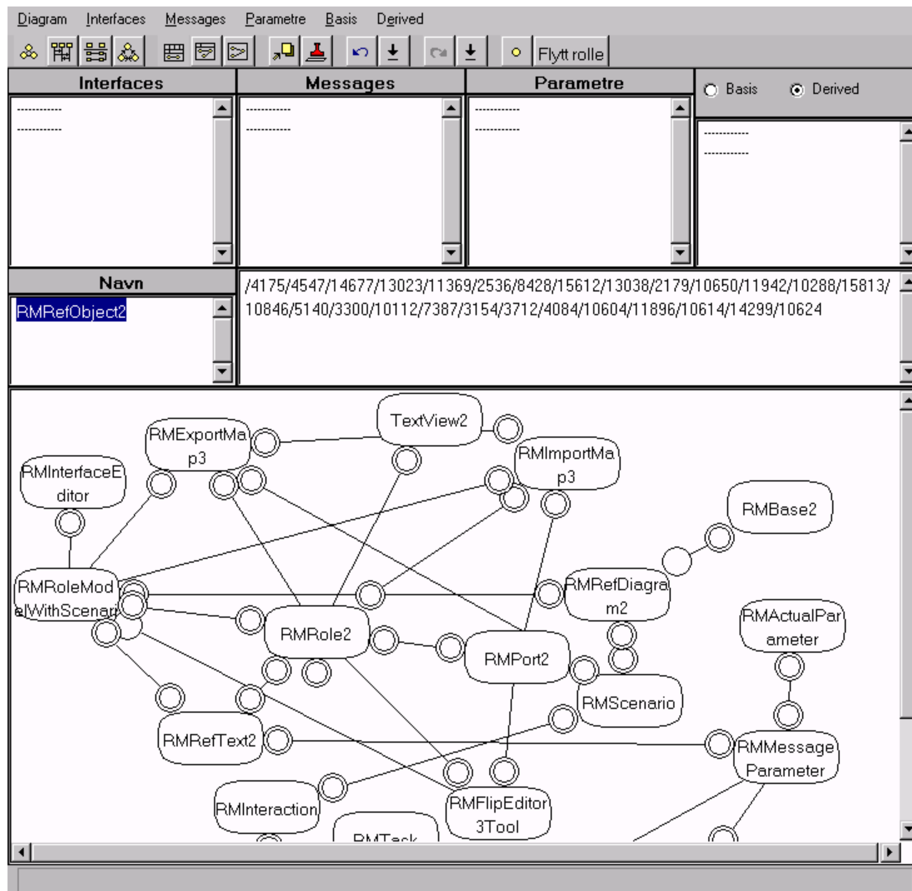

Fig. 7.18. A role-model generated from the dynamic coupling data from a run-time session, using the modified Ooram tool. Such models may support impact analysis and code comprehension for a selected scenario. Whether the models are useful needs to be evaluated. One obvious problem is related to the size and layout of the models.

7.3.5.5 Data Collection Algorithms

The accuracy of the dynamic coupling measures depends on being able to execute as many sub-functions as possible within a selected scenario. If one misses a small sub-function, say a user-input error condition, then the coupling measures will not reflect the dependencies to the objects implementing that sub-function. If the purpose of the measures is impact analysis, running the scenario to completion may be crucial in order to detect as many dependencies as possible. The problem is that it may be very difficult to guess *when* the scenario has been run to completion unless one knows the detailed functionality of the system extremely well. However, running only a portion of the complete scenario may also be quite useful: one may distinguish between

"happy-day" scenarios and "error" scenarios. For example, to understand how to implement a change, it may be unnecessary to have an overview of all possible error conditions; initially it may be more useful to get an overview of the flow of messages within the happy-day case.

Clearly, the proposed data collection algorithm can be improved. It does not guarantee that all sub-functions have been executed at least once. Perhaps more importantly, the algorithm does not reflect how users actually *use* the system. Depending on the intended use of the coupling measures, a better data collection "algorithm" may be to let actual *users* run the system over a long period of time while collecting the measures. The resulting data may be used to assess the system in ways that have not been discussed so far. For example:

- How much are different parts of the system actually being used?
- Functionality (e.g., classes) that is never or seldom used are candidates for deletion. By deleting classes implementing unused functionality one may improve execution speed, usability and reliability. Furthermore, the system will become smaller and hence probably easier to maintain.
- Functionality that is used often are also candidates for improvements. For example, can the execution speed, reliability or usability of this functionality be improved?
- Are the parts of the system that is used often also *changed* often? If so, one may focus restructuring efforts on those parts of the system.

7.3.5.6 Implementation Issues

In this case study, extensive modifications of source code of the Ooram system was carried out in order to collect the dynamic coupling measures. This is a time-consuming, error-prone, language-dependent and clearly not very practical approach. In the Java programming language, it may be possible to modify the Java Virtual Machine to trace messages between objects for any application written in Java. In compiled programming languages such as C++, it may be possible to use the debuggers in integrated development environments (e.g., Microsoft Visual Studio) for collecting the measures. Alternatively, if the system to be evaluated has not been implemented yet, the object-level coupling measures could be collected from object interaction diagrams (e.g., UML sequence diagrams and collaboration diagrams). However, these are initial ideas that may prove very difficult to implement in practice.

7.4 Causes of Increased Change Effort and Project Delays

This section describes results from an interview with four experienced developers on matters related to the changeability of object-oriented systems. The purpose of the interview with developers in Numerica-Taskon was to gain insights into what factors experienced developers consider important to reduce change effort in object-oriented software. This qualitative study was primarily intended to identify factors that may affect the changeability of object-oriented software. The results may be used as an empirical basis for formulating theories from which hypotheses can be tested more formally in future studies, for example through case studies and controlled experiments. Using triangulation in this way may improve the validity of the conclusions of the studies, as discussed in Chapter 3.

7.4.1 Design of the Study

A choice had to be made between conducting a semi-structured interview and a questionnaire-based survey. Because the time of the developers were a limited resource, using an interview approach meant that we would not get a sufficient number of data points for statistical analysis. On the other hand, we believe that we could obtain higher *quality* data using a semi-structured interview approach:

- We could ensure that the subjects interpreted questions in approximately the same way, and we could do a subjective assessment of the quality of responses.
- We could follow up interesting issues raised by the subjects by asking new questions.

The quality of questionnaire-based surveys may be reduced because of the factors mentioned above (Jørgensen, 1994). Thus, we believe that, given the time-limitations and exploratory nature of the study, a structured interview approach was appropriate.

We developed a simple questionnaire that guided the interview. However, the questionnaire was modified slightly during the interview sessions based on issues raised by the interviewed developers. Answers were typed directly into the questionnaire data form in cooperation with the subjects. Each interview took approximately 3 hours. The subjects were selected based on the experience level in cooperation with the department manager, but the selection was also limited by the availability of the developers.

7.4.2 Results

Numerica-Taskon has been one of the leading consulting companies for object-oriented processes, methods and tools in Norway during the past several years. They have developed a concept called role-modeling in the OORAM method (Reenskaug *et al.*, 1995) and have been an important consulting resource for Norwegian companies wanting to use object-oriented tools (e.g., OORAM, Rational Rose) and processes (e.g., Rational Unified Process). Table 7.37 gives an overview of the project

experience of the interviewed developers. The subjects cover important functions within object-oriented development, e.g., product management, development management, key account manager and system developer. They have varied work experience and a high level of education.

Table 7.37. Overview of project experience for the interviewed subjects

Question	Subject 1	Subject 2	Subject 3	Subject 4
Current work title	Systems Developer	Product Manager	Key Account Manager	Development Manager
Work Experience (years)	3.5	12	3	12
Education level	Master	Master	Bachelor	Master
Typical team Size	6	1–5	1–5	3–15
Project Size (person-years)	4	1–10	1	2–90
Number of projects	N/A	10	2	6
Number of delayed projects	N/A	4	2	4
Typical delay	40%	50–100%	50%	20%–150%
Reasons for delays (1)	N/A	Requirement analysis was not correct	Waited for development product	Poor analysis & design resulted in rework
Reasons for delays (2)	N/A	Technology risk, availability of key resources	Improper planning/ resource allocation	Failure to evaluate technology risks
Reasons for delays (3)	N/A	Too little design	Delays associated with external vendor dependencies	Untimely change requests, communication with customer
Per-hour consulting	x	x	x	x
Product development	x	x		x
In-house development	x		x	x
Mentoring	x	x	x	x
Research		x		x
C++	x			
Smalltalk	x	x		x
Java	x			
Jasmine (OO database)		x	x	
Taskon Integrator		x	x	
OOAD	x	x	x	x
OORAM	x	x	x	x
Rational Unified Process	x			
Windows	x	x	x	x
Unix	x			x

Table 7.38. Most influential factors of change effort

Subject	Factors
Subject 1	N/A
Subject 2	1: Reusability, quality of component/interface design
	2: Flexibility/openness of design; Portability
	3: Availability of test resources; test environment
Subject 3	1: Complexity of analysis/design model
	2: Exists GUI standards
	3: Availability of resources
Subject 4	1: New functionality or bugfix requiring changes in framework classes
	2: How well resources know system
	3: Testability (equipment, test environment, human skill)

The subjects were asked to name the three most influential factors for the effort to implement changes in object-oriented software (Table 7.38). For subject 2, the first two responses are related to design quality; for subject three and four, the first response is related to design quality.

The subjects were also asked to grade the importance of various given factors that we believe influence change effort. The subjects ranked each factor between 1 (not important) and 5 (very important). The results are depicted in Figure 7.19, showing the minimum, median and maximum score for each factor ordered from left to right by decreasing importance. To our surprise, the subjects do not think that the size of object-oriented software (i.e., 'Code size') is particularly important for the required effort to implement changes. Among the given factors, only the educational level of the programmers is judged as less important than code size.[8] In a follow-up question, the developers answered that the design mechanisms of object-oriented languages often prevent code size from being a major contributor to "complexity". Design complexity, however, is rated among the most important factors along with knowledge of the code. Design complexity is as important as the programming experience of the developers implementing the change. Code size and code complexity are believed to affect change effort less than design level complexity.

Finally, we asked the subjects to judge how difficult it is to change various components of object-oriented software systems. The results in Figure 7.20 indicate that (reusable) framework classes are more difficult to change than, for example, user interface logic and database schema changes. This may be related to the higher design complexity of reusable framework components. Thus, the development of object-oriented frameworks can be regarded as a long-term investment at the expense of increased short-term change effort. This tradeoff in changeability is clearly an interesting research topic worth further investigations.

---

[8] Note that one should be careful in generalizing the results because of the small sample size and because the subjective assessment of the importance of individual factors may be biased by the developers.

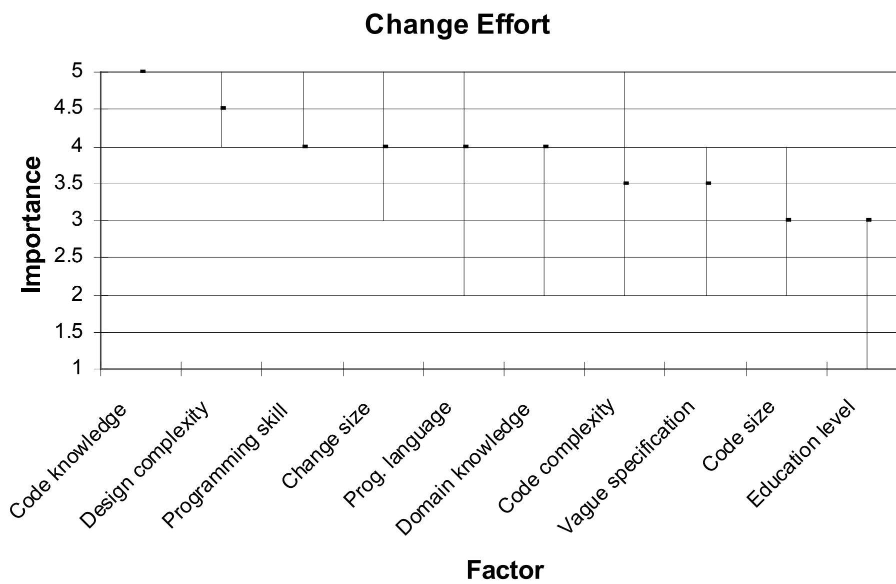

Fig. 7.19. Relative importance of factors affecting change effort in object-oriented development, sorted by importance from left to right based on the median, maximum and minimum values of the ordinal importance score (1=not important; 5=very important) by the subjects.

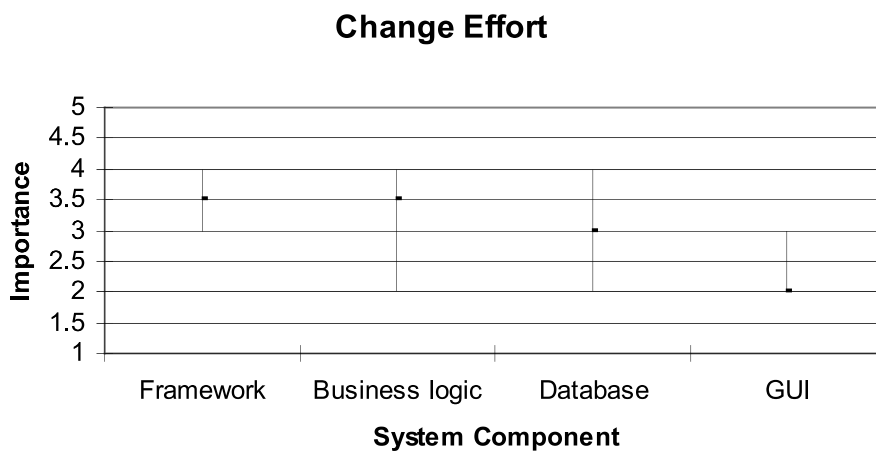

Fig. 7.20. The subjects reported that the change effort in object-oriented development depends on the component being changed. According to the developers, it is more difficult to change framework logic and business logic than the graphical user interface logic (GUI).

7.4.3 Summary of Results

The results of this qualitative study was primarily intended to serve as an empirical basis for formulating theories from which hypotheses can be tested more formally in future studies.

The results of the interviews presented in this section identify code knowledge and design complexity as the two most influential factors for change effort. These factors are rated as more important than code size and code complexity. The results also motivate further investigation into the understandability aspect of object-oriented designs.

Furthermore, framework classes and business logic are more difficult to change than databases and GUI components. This knowledge may be utilized when assessing the changeability of designs using CPM, as illustrated in Section 7.3. Furthermore, the results also suggest that better indicators of changeability might be developed if design-level measures, such as SAM and CPM, differentiate between the various design layers of an object-oriented system, such as framework, business logic, databases and GUI.

Finally, design decisions and technology risks are important reason for project delays. These results are corroborated by the results of the case study presented in Section 7.5.

7.4.3.1 Threats to Validity

One must of course be careful drawing general conclusions based on interviews with four developers within one company. The small number of data points and the informal analyses are not sufficient to obtain statistically valid results. Furthermore, no 'post-mortem' data quality assurance was performed; we have assumed that the answers reflect the actual experience of the subjects and that they remember details from the development projects reasonably well. Furthermore, the selection of the questions and the interpretation of the answers may also be biased by the preconceived theories of the researcher (Hufnagel and Conca, 1994).

7.5 Evaluation of an Evolutionary Development Project

This section describes experiences from an evolutionary development project in Norway. The study provides insight into the role of end-user participation, documentation and technology risks in evolutionary development projects, and describes how these factors may influence the changeability of the software.

The described study is a part of a process improvement project funded by the national research project PROFIT. The goal of a sub-project of PROFIT is to develop guidelines for evolutionary development of web applications. Based on these guidelines, the Genova Process (Arisholm *et al.*, 1998; Arisholm *et al.*, 1999b) will be extended to provide specific support for web projects (Genova Web Process). The resulting process will subsequently be instantiated and evaluated on a new project. The guidelines are based on the experiences collected through interviews with developers, project management and end-users on several existing web development projects. This section reports the results from one of these projects, called TelMont[9].

7.5.1 Design of the Study

The study consisted of semi-structured interviews with subjects involved in the TelMont project. To reduce biases due to different perspectives and interpretations of the researchers, two researchers (this author and one of his master's students) were involved in most aspects of the study, consisting of

- subject selection,
- formulation of the interview guide and conducting the actual interviews,
- transcription, analysis and unification and
- post-mortem quality assurance.

This process is described further in the following sections.

7.5.1.1 Subject Selection

Several interviews were conducted with subjects selected to cover different roles on the development project. From the development organization, three persons were selected. Two of these were the two most central developers responsible for many different functions within the project, from database design, middle-tier development and web user-interface development. The third person was the project manager from the contractor. Interviews with the end-users and project management for the customer have also been conducted, but these results have not been analyzed at present.

7.5.1.2 Interview Technique

Before the interview sessions, several specific questions were formulated by each of the two researchers. The questions were combined and prioritized to form an interview guide. The questions were categorized into different topics such as "project

---

[9] All names have been altered for confidentiality reasons

initiation", "end-user contact", "experience", "CASE support", etc. As a starting point for data collection, the interview guide worked very well. Other, more open-ended questions were also asked. The number and type of open-ended questions depended on the responses given by a particular interviewee to the specific questions in the interview guide. For example, the project manager was more interested in questions related to project costs and process establishment, whereas the developers were more interested in technical aspects. Thus, the interviews were designed not only to elicit the information foreseen, but also unexpected types of information.

7.5.1.3 Transcription, Data Analysis and Unification

The interviews were recorded on a tape recorder in order to avoid loss of information. Subsequently, each question and answer from the recorded interviews were written down in detail. Although this transcription process is very time consuming, it is in our opinion essential to improve the accuracy and comprehensiveness of the analyses. It is particularly important for the unforeseen types of information, which may be scattered and intermixed within several answers to specific and open-ended questions. Each researcher used the transcribed material to perform the analysis in parallel. Each researcher used "copy and paste" to group questions and answers into categories determined by keywords or themes such as "customer contact". The keywords were determined during the exploratory search and analysis of the transcribed interview text. A given question/answer that spanned many categories was copied into each of the categories. Questions/answers that did not fit in an existing category resulted in the creation of a new category. Each researcher wrote an analysis report based on the data from the interviews. Finally, these reports were combined into a unified analysis report. The experiences from this analysis process clearly show the benefits of performing the analyses in parallel. Each of the two researchers analyzed the data using different perspectives resulting in distinct themes or categories of experiences. If only one researcher had analyzed the data, important information would have been lost.

7.5.1.4 Quality Assurance

The unified analysis report was given to the interviewees for quality assurance. The purpose was to uncover any errors or ambiguities in the analysis. The interviewees made modifications to the report using a change tracking system. The results reported in this section are based on the revised report.

7.5.2 The TelMont Project

TelMont is a support system used by a Norwegian telecommunication company, TeleX. The support system simplifies and automates the work processes required for performing compression and optimization activities for ISDN telephony hardware. The web-based system is accessible within the TeleX intranet. The users of the system are professional engineers.

The company InterDev developed the product for TeleX. The development phase of the project required 8400 person-hours over a nine-month period. The project consisted of a total of 10 to 15 persons from TeleX and InterDev, in addition to some

external consultants. A dedicated test team within InterDev was used for integration test and system test activities. Each developer in InterDev was responsible of one "module". In addition, most developers participated in common activities such as analysis, design and communication with project management and users in TeleX. Specialists in TeleX who knew the ISDN compression and optimization work processes contributed to the analysis and design. Furthermore, TeleX was also involved in the specification of user interface design and testing. The project was paid on a per-hour basis. The maintenance of the system is performed by InterDev.

7.5.2.1 Project Activities and Milestones

May 1999 - August 1999 (Inception)

A pilot project was initiated by TeleX in May 1999. The goal of the pilot project was to determine the need for an automated software solution. The pilot project involved one person from InterDev and 3-4 persons from TeleX. The deliverable from the pilot project was an analysis report.

August 1999 - October 1999 (Elaboration)

The requirement specification activities started in August. In this phase of the project, the main work processes that the TelMont system was intended to support were discussed. The work resulted in a requirement specification and an analysis model consisting of a work-flow model and some use cases, as well as a simple prototype of the user interface.

October 1999 – March 2000 (Iterative Elaboration and Construction)

In this phase of the project, a large evolutionary prototyping activity was initiated. The prototype had two purposes: 1) to evaluate the feasibility of the chosen technology platform, and 2) to elaborate requirements by providing a detailed user interface of the most important functionality. The product was modeled in UML. The model was made available on a local server such that the customer, and later the end-users, had access to it throughout the development project.

The first construction increment (or prototype) in the project was finished in March 2000. This increment served as an architecture release and as a requirements specification. However, there was also an intermediate delivery in January, used to evaluate the user interface and the work processes automated by the TelMont prototype. In October/November InterDev asked TeleX to provide end-users in order to evaluate the system iteratively. However, these end-users were very important engineering resources within TeleX, and there was considerable debate within different groups in TeleX regarding how and when to assign these resources to the TelMont project. Finally, in early February, end-users were assigned to the TelMont project.

The increment took much more time and effort than planned. The part of the prototype intended to evaluate the architecture and technology became very large. Based on the workshops with the end-users, the developers in InterDev discovered that there was too much complexity in the initial requirement specification. A considerable amount of rework occurred as a result of the end-user feedback provided

from January/February. These end-users had not been involved in the specification of the initial requirements, and did not feel any ownership to it. In addition to the workshops, the end-users had access to the actual development environment through the web, and could at any point in time evaluate the progress and latest additions to the system.

March 2000 (Changed Process)

From March 2000 the TeleX and InterDev agreed on changing from the evolutionary process (called the "Solution Delivery Process" in InterDev) used up to that point, to a waterfall process. The evolutionary construction phase had taken much longer time and effort than expected. The workshops and the continuous evaluation of the system by the end-users produced a large number of change requests that were not always handled through formal channels. Thus, the project management (especially on the customer side) felt that they were loosing control of the project. The customer was very unhappy about the continuous changes, the cost overruns and the schedule delays.

May 2000

A detailed system specification was accepted.

June 2000 (Transition)

The first main release was in June 2000, and consisted of a fully operational system. At this time, the project had used 8600 person-hours, whereas the initial estimate was 4300 person-hours. The release was delayed by three months.

7.5.2.2 Customer/Software Vendor Relationship

In the TelMont project, TeleX and InterDev assumed the role of "customer" and "software vendor", respectively. In earlier projects, the two companies had worked more as one team. In this project, there was a lot of discussion between TeleX and InterDev about progress plans and other project management activities.

7.5.2.3 Estimation

Function point estimation was used to estimate project effort. The estimates were based on other web projects within InterDev using function points. However, the data was not as relevant as assumed. Amongst others, the TelMont system contained considerably more business logic than the web application projects upon which the estimates were based. Furthermore, the business logic was implemented in new (and error-prone) multi-tier technology (COM/MTS) with which the developers had no prior experience. Integration between the Oracle and Microsoft products was also much more complicated than anticipated.

7.5.2.4 Process

The process used on the project is an evolutionary process called the "solution delivery process", consisting of three month time-boxes. The developers had experience using this process in previous projects. However, those projects were

internal projects. In general, the interviewees were quite negative to using an evolutionary development process for external development projects, such as TelMont. For external projects, evolutionary development may be a serious hindrance to project control. However, as pointed out by one of the interviewees, many of the experienced problems were not caused by the process itself, but were rather a result of a lacking formal change management process.

Both customer and end-user had access to the evolving system at all times. Thus, users could test solutions and suggest changes continuously. However, this access actually caused some frustration for the customer and the end-users. It resulted in changes in requirements even after requirement specifications had been "frozen". Consequently, project management on both sides lost control. Furthermore, the end-users were not accustomed to testing partially implemented functionality. Only the developers had an overview of the status of the different functional components of the system.

At some point (March) the evolutionary process was abandoned and replaced by a formal waterfall process, requiring formal acceptance of detailed specifications and code at the method level. This radical change in the project was determined necessary to get economy and time schedules under control. InterDev could no longer defend using the solution delivery process because of the delays and the complaints voiced by the customer caused by the continuously changing requirements. The project manager in TeleX felt that he lost control because the requirements changed continuously, without the formal documentation being changed accordingly. However, the developers strongly believe that the changes requested by the end-users were crucial in order to achieve a realistic and useful product.

7.5.2.5 User Participation

During the early construction phase, the developers wanted contact with the end-users. Up to some point, they were depending on the customer's understanding of the work processes. When the end-users finally were allocated to the project, they did not feel much ownership to the requirement specifications. This resulted in that it took some time before they actually started criticizing the initial specifications, even when they believed that they contained faulty, unnecessary or missing requirements. Before the end-users started to comment on the requirements, the developers spent considerable time implementing unnecessarily and complicated functionality into the product. The end-users wanted a simpler system that did not allow for several different ways of doing the same things. From January to March, three workshops were held with the end-users. The developers describe the end-users as positive and interested in spending time on the project. These workshops resulted in many important changes to the system. According to the interviewees, the contact with the end-users enabled the developers to make a very good product, tailored towards the way the end-users actually worked.

7.5.2.6 Prototyping

The prototype was developed to evaluate both the development technology and the functional requirements. By the time the prototype was "finished", it had become very large. Because of the large amount of effort spent on the prototype, it was decided to

base the actual product on the prototype. It was not cost-effective to throw away the prototype, even after it became clear that the chosen technology was error-prone and had obvious limitations. Most of the interviewees felt that a different technology platform would have been chosen if the prototype had not contained so much functionality and could have been thrown away.

7.5.2.7 Requirements Specification

The interviewees thought that the requirement specification took too much time. The representatives from TeleX initially involved in requirement specification were taken out of the project. The end-users took over their role, and these users had a completely different view of the requirements of the system.

InterDev ended up with having a mediator role, in which they had to defend the initial requirement specification to the end-users, and at the same time listen to their suggestions. This process took a lot of time, and it was perceived as very inefficient. Initially, the customer wanted a very complex and flexible system, and it was not before the end-users were allocated to the project that InterDev managed to define realistic requirements. However, the end-users also wanted more functionality, but different in content from the initial requirement specification. The customer was not very willing to pay for these changes requested by the end-users. Furthermore, the continuous change requests made it difficult to keep requirement specification documents consistent and up-to-date, which was demanded by the customer.

7.5.2.8 UML and Rational Rose

The system was modeled in UML, using use-cases, class diagrams and some sequence diagrams. The modeling was supported using the CASE tool Rational Rose. One of the interviewees regarded the use of UML as useful in the early phases of the project, but only as a means to evaluate and discuss design alternatives internally between the development team in InterDev. In general, none of the interviewees considered UML as successful for the project.

Code generation (i.e., from model to C++ code) for the COM/MTS architecture using Rational Rose did not work. Reverse engineering (i.e., from code to models) also had many flaws. Thus the UML documentation became inconsistent with the code, and the manual maintenance of the UML documents was very time consuming. Furthermore, the UML models (including the use-case model) were not useful as a means for communicating requirements and designs with the customer or end-user. It was too difficult to explain the semantics of the UML models. To communicate with the customer, simple process flow diagrams and user interface mockups, in addition to the available implementation, are far superior to UML.

7.5.2.9 Lessons Learned

The interviewees were asked what they would have done differently if they could start the project again. The results are summarized below:

- They should have made a simple prototype that only evaluated the technology, not technology *and* requirements.

- The choice of technology should have been considered more carefully. Unfortunately, the developers had insufficient experience with the alternatives to make an educated choice.
- Considerable cost savings could have been achieved if the end-users had been involved in the project from the inception.
- It is very important to determine the customer's expectations of the project early. Milestones must be clearly defined and agreed upon.
- In general, evolutionary development may be appropriate for internal development projects, but difficulties with project management activities such as cost control suggests that evolutionary development processes may be difficult to use in external development projects. When using an evolutionary development process with an external customer, the changes must at least be documented very thoroughly. A formal change management process most be followed and agreed upon by the customer and the developer. Otherwise, the customer may fail to recognize the importance of the changes, and may contest the project costs and time schedule.
- Web development technology is immature, time-consuming and error-prone. The development of web applications requires a considerable amount of new knowledge acquisition. This must be accounted for when estimating new web projects.

7.5.3 Summary

The TelMont project was a success in the sense that the customer received a very good product:

- The customer and its end-users are extremely happy with the provided functionality and the reliability of the product. The analyses suggest that the evolutionary development process resulted in a more usable product than what would have been obtained with waterfall development.
- A significant amount of the functionality initially specified was never developed as a result of frequent user feedback. Furthermore, the design did not have to account for future "extensions" since such flexibility was determined to be unnecessary. The design was better adapted to the final functionality of the product. Thus, the product became smaller and probably easier to change than if waterfall development had been used.

However, the project was delayed and project costs were doubled compared with initial estimates. This resulted in conflicts between the project management of the customer and the software developer. Based on the experiences, two simple but probably quite important process guidelines are proposed:

- An important prerequisite for success in evolutionary development is that detailed and formal change management guidelines are in place. Otherwise, the frequent changes may become a serious hindrance to efficient and reliable project management. Furthermore, the informal changes may have a negative effect on the changeability because design documentation is not updated. For external development projects, i.e., when the customer and software vendor represent separate companies, it may be necessary to restrict the informal discussions of

requirements between developers and end-users, and instead organize such communication in formal workshops.

- The project experiences emphasize the potential risks of cost and schedule overruns when applying new software development tools and libraries. New technology should be evaluated through early prototyping activities that should be kept small in size and schedule. The technology prototyping should be organized as an activity separate from the evolutionary elicitation and implementation of functional requirements. Otherwise, the technology prototype may incorporate too many functional requirements and consequently become too large and too costly to throw away – even if the chosen technology is determined inadequate (e.g., unstable or hard to change).

8 Conclusions and Future Work

A prerequisite for obtaining a better scientific knowledge regarding the consequences of evolutionary development on changeability is to assess the changeability of the developed software in an objective and accurate way. The current knowledge regarding how to quantify the changeability of object-oriented software is very limited, however.

The main contribution of this thesis is the development and validation of a measurement framework that may be used to assess changeability of object-oriented software. The proposed measurement framework, the associated data collection methods, and the empirical validation techniques were evaluated in several industrial development projects and controlled experiments.

The second contribution of this thesis is the identification of factors potentially causing changeability decay in evolutionary development of object-oriented software. At present, only a very limited number of empirical studies of evolutionary development projects exist. The results of the case studies described in this thesis extend the current state of knowledge regarding evolutionary development.

8.1 Summary of Results

The goal of this thesis was to

- to define changeability and changeability decay in a concise manner,
- to develop a measurement framework for assessing changeability, and
- to identify factors affecting changeability in evolutionary development.

This section describes the results of this thesis in more detail according to the above goal structure. Section 8.1.1 summarizes issues related to the definition of changeability. Section 8.1.2 summarizes the main contributions regarding the empirical validation of the proposed measurement framework. Section 8.1.2 summarizes the results of the empirical studies of evolutionary development projects.

8.1.1 Definition of Changeability

This thesis has attempted to define changeability and changeability decay in an operational form, focusing on the effort required to implement changes. The intention of the definitions has been to provide a sufficiently unambiguous fundament for the formulation of useful indicators of changeability. With this regard, the working definitions serve their purpose. However, the empirical studies have shown that the underlying concepts are more complex than what is reflected explicitly by the definitions. One challenging issue illustrated by the empirical studies is the tradeoff between changeability and changeability decay. As exemplified by the coffee-machine experiment, a design that has better initial changeability may still be more prone to changeability decay than another design. Furthermore, the changeability of a design is very much linked to the skill level and prior knowledge of the system by the

developers implementing changes. For example, it seems that the solution approach has a considerable impact on the change effort. In summary, it seems that changeability, and the tradeoff between changeability and changeability decay, depends on the developers that will implement the changes and on the expected number and type of future changes to the system. Further empirical work will hopefully allow us to develop more concise definitions that reflect these issues in a better way.

8.1.2 Empirical Validation of the Measurement Framework

The proposed Change Profile Measurement (CPM) approach can be used to develop accurate models for the difference in change effort. Consequently, the CPM measures may be useful indicators of changeability. However, it is difficult to generalize the models. Although an accurate model of changeability was developed for one particular development project, the resulting model is not necessarily valid for *other* development projects. At present, it may be necessary to calibrate the indicators for specific projects. It is highly plausible that the impact of the structural properties on changeability are influenced by many factors specific to a given project, such as developer experience, familiarity with the code, tool and library dependencies, and other project characteristics.

Studying the details of a large restructuring change in the Genera case study, the CPM measures seem more sensitive to the resulting changes in the design than the SAM measures. Similar results were found in the Ooram case study, in which there was a clear trend towards changing the smaller classes of a given module in later releases, whilst the corresponding average class size (i.e., SAM) increased. Apparently, large framework classes were changed in the early releases, but eventually they stabilized. According to the Numerica-Taskon interview, classes in object-oriented frameworks are much more difficult to change than other classes. Consequently, the CPM approach seemed to indicate a positive trend in changeability for the studied Ooram module.

The results of the empirical studies suggest that the CPM measures are better indicators of changeability than the SAM measures. The added cost of the CPM approach compared with the SAM approach may be worthwhile. However, since the SAM measures are included as components of the CPM measures, there are no practical reasons *not* to utilize the simpler SAM measures also. The two approaches offer complementary views of software structure evolution. CPM can be used to assess how changes actually propagate through the software structure, whereas the SAM approach can be used to quantify structural change at the system level. Consequently, by combining the SAM and CPM approaches, one may get a more complete assessment of the changeability of object-oriented software than when using only one of them.

The benchmarking approach was used to compare the changeability of two alternative object-oriented designs. The results show that object-oriented design decisions have a significant impact on the changeability of a system; the design may affect the change effort at almost the same order of magnitude as the differences in individual programmer productivity or skill level. These results are corroborated by the results of interviews with the developers in Numerica-Taskon. As expected, the design with *low* coupling and *small* classes had high structural stability. The changes

had less effect on the structure, indicating that it was more "open ended" than the alternative design with *high* coupling and *large* classes. However, surprisingly, the design alternative with high coupling and large classes required considerably less change effort than the design with low coupling and small classes. Most of the observed effect was due to the difference to *understand* how to implement changes. Furthermore, the results suggest that the observed effect depends on the solution approach (e.g., explorative or systematic) used by the developers.

The results of the coffee-machine experiment apparently contradict the results of the Genera case study. The Genera case study indicates that high coupling and class size have a *negative* impact on changeability, whereas the opposite effect is observed in the coffee-machine experiment. However, the two studies investigated different aspects of changeability. In the Genera case study, the developers were familiar with the software, hence understandability was less important. In the coffee-machine experiment, the developers had no prior knowledge of the software before implementing the change tasks. The preliminary results from a follow-up "think-aloud" experiment indicate that it might be useful to also consider how the *dynamic* aspects of code affect the comprehension effort, in particular when developers are unfamiliar with the code.

This motivated the proposal for a comprehensive set of dynamic coupling measures. These measures may be used to quantify the depth of the dynamic message interactions between objects in object-oriented software, which in turn may affect the cognitive complexity of a design. A preliminary validation of the measures was performed. Using dynamic coupling data, classes collaborating in the implementation of a given functional change scenario could be identified. This provides a useful starting point for change impact analysis. Furthermore, the results indicate that dynamic coupling measures can be used to build prediction models of common changes within the identified classes (i.e., if class *A* is changed, how likely is it that class *B* is changed). Thus, the proposed dynamic coupling measures and the resulting prediction models can support change impact analysis, which can be seen as a technique to improve the changeability of the software.

8.1.3 Changeability in Evolutionary Development

The case studies conducted in this thesis show that there are many important requirements for a successful application of evolutionary development. Several factors that may cause changeability decay have been identified. These factors need to be investigated further, as explained in Section 8.2.

One of the case studies illustrates that it is crucial that formal testing is performed for each increment to avoid last-minute expensive rework. Such untimely rework may otherwise be a major contributor to changeability decay. In another case study, a considerable amount of the total effort was spent on restructuring activities during the project. This particular project had characteristics of a "code-and-fix" process. The analysis of the change data suggests that the need for restructuring would probably have been reduced if more analysis and design had been performed based on the initial (and quite stable) requirements of the system.

These studies have also identified the need to take the evaluation of the chosen technology more seriously than what seems to be current practice. In several of the studied development projects, problems related to the incorporation of new

technology was a major contributor to schedule delays and cost overruns. The rework resulting from technology-dependent work-around solutions may cause decay. If new technology is used, considerable cost savings can probably be obtained if the first incremental delivery is a technology evaluation prototype. This prototype should be no bigger than that it can be thrown away if the technology is proven inadequate for the task at hand.

Another problem identified in this thesis is that frequent changes may result in outdated documentation. Up-to-date documentation is important for understanding how to change the code (Tryggeseth, 1997). On the TelMont project, a considerable amount of manual work was required to update the requirements and design documentation, even when a modern CASE tool was used. Consequently, the documentation was updated infrequently, resulting in documentation inconsistent with code. CASE tools such as Rational Rose may alleviate some of the documentation problems, but mainly for the *static* parts of the code, such as the class diagrams.

To document the dynamic aspects of a system, this thesis illustrated how a reverse-engineering tool producing models of the collaboration between run-time objects can be developed. Thus, CASE tool support may eventually solve a major potential cause of changeability decay in evolutionary development of software.

Fortunately, these studies show that users participating in the evolutionary development of software may contribute to the construction of software systems that reflect the requirements of the end-users better than what would otherwise be possible. A design that has evolved to reflect the requirements results in smaller software. UML-based user-interface prototyping tools, such as Genova, can support the user participation (Arisholm *et al.*, 1998). In the TelMont case study, user interface prototypes were considerably more useful for communication with the customer and end-user than "use cases" and other design descriptions. Thus, evolutionary development may *improve* the changeability of the resulting software if issues such as those addressed above are dealt with.

8.2 Future Work

This section outlines areas for future research related to the two main contributions of this thesis: the measurement framework (Section 8.2.1) and the empirical studies of regarding changeability in evolutionary development projects (Section 8.2.2).

8.2.1 Improvements of the Measurement Framework

The empirical validation studies performed in this thesis constitute, in our opinion, a useful first step towards a validated measurement framework for changeability. However, further validation is necessary. Furthermore, there are several potential improvements in the way changeability is measured. This is outlined in the following subsections.

8.2.1.1 *Industrial Evaluation*

The accuracy and practical use of the changeability measurement framework needs to be evaluated on longitudinal, industrial development projects. One such project is the Genova project. Further measurement on this project will provide a large amount of data for longitudinal validation of the SAM and CPM approaches. Furthermore, the collected change data may also be used to develop product-specific benchmark tasks necessary to evaluate the benchmarking approach in an industrial setting. The measures will also be validated on new evolutionary development projects in conjunction with a new national research project called PROFIT.

8.2.1.2 *Building Generic Benchmarks*

In the coffee-machine experiment, the composition of benchmarks was based on guessing likely change tasks. When change history is available, the distribution, size and types of earlier changes may be used to compose representative product-specific benchmarks. Our long-term goal, however, is to build more generic benchmarks, for example for a specific application domain. The assumption is that a given application domain has many similarities with respect to the types and distribution of changes. If so, pools of representative benchmark tasks corresponding to given application domains can be developed. These benchmark tasks are not specific to a given product; they must be more generic, such as "add input field to the most central user interface dialog" or "implement an *undo* functionality". To develop generic benchmarks that are sufficiently representative of a given application domain, a considerable amount of change data from many similar products needs to be collected. The change data collection could, for example, use the change log described in this thesis. The types of changes that occur frequently within the application domain can then be used as the basis for the composition of benchmark tasks.

8.2.1.3 *Combining CPM with Scenario Elicitation*

At present, the class-level Change Profile (CP) component of the CPM measures (i.e., the proportion of change to each class), is calculated from the change history. This approach seems to be appropriate because the change history fully describes how the changes propagate through the design right up to the point when the designs are compared. However, when comparing early design alternatives, the change history may be limited. One possible extension of the CPM approach is to *combine* change history with scenario elicitation and impact analysis as a basis for the calculation of the class-level Change Profile. In this way, the CP measure might reflect *future* changes better than when only considering the change history. Furthermore, when no change history is available, the CP measure could be estimated based solely on class-level impact analysis of likely change scenarios. The latter approach is similar to the one proposed in (Briand and Wust, 2000). However, using scenario elicitation and impact analysis as a basis for the CP calculation is certainly more costly and subjective than using actual change data.

Another potential use of such hybrid CPM approach (i.e., scenario elicitation and impact analysis as the basis for calculation of CPM) is for change effort estimation. Evolutionary development projects are characterized by frequent, incremental

changes. Thus, it is important to be able to predict the effort required to implement the changes, e.g., to support prioritization of changes within increments, to decide on how many changes can be implemented within a given time box, and to determine the costs of including new change requests. In the Genera case study, quite accurate models of change effort using the CPM measures as explanatory variables were developed. Although the primary purpose of the CPM measures is changeability assessment, it seems plausible that the hybrid CPM measures also may be used for effort prediction for *new* actual changes. Consequently, it may be worthwhile investigating the hybrid CPM approach further, because it expands the potential uses of the measurement framework.

8.2.1.4 Dynamic Coupling

Dynamic coupling measurement opens up a variety of interesting new opportunities for assessing (and improving) the changeability of object-oriented software. For example, better explanatory models of the effort required to understand how to implement changes can probably be built based on dynamic coupling. Dynamic coupling may also be used to support impact analysis in many different ways. For example, the coupling data can be used to identify collaborating classes and construct reverse-engineered collaboration diagrams for a functional scenario. Furthermore, prediction models can be built to help identify ripple effects when changing the scenario. However, as explained in Section 7.3, several data collection and implementation issues need to be resolved. For example, the collection of dynamic coupling data depends on running the system, and it is nontrivial to determine when the system has been exercised sufficiently to give useful coupling data. Another challenge is how to implement dynamic coupling parsers that are independent of the product source code, in particular for compiled languages, such as Java. Future research will investigate whether the Java virtual machine can be modified to track run-time messages between objects.

8.2.2 Evolutionary Development Processes

Several factors that may potentially cause changeability decay in evolutionary development were identified through the case studies. These factors are related to

- the distribution of analysis and design effort – to reduce the need for expensive restructuring,
- early technology prototyping and incremental testing – to reduce rework, and
- frequent updates of design documentation – to support understandability.

However, the studies have been quite exploratory in nature. These factors should thus be considered as preliminary theories that need validation. Unfortunately, it is difficult to validate the theories in case study research because there is no reliable baseline for comparisons. As a first step, each factor could be adjusted and evaluated separately in controlled experiments similar to Beohm's prototyping experiment (Boehm *et al.*, 1984). The proposed measurement framework (e.g., benchmarking) should be used to assess the impact of the adjustments on the resulting changeability. Cost issues dictate that such experiments might be rather small in scale. Small scale studies are often affected by threats to external validity, as exemplified by the

contradicting results of (Boehm *et al.*, 1984) versus (Zamperoni *et al.*, 1995). Thus, the results from small, controlled experiments need to be triangulated with further case study research of the type conducted in this thesis.

8.3 Concluding remarks

During the past years, there seems to be a trend towards more companies using evolutionary and incremental development processes, supported by object-oriented development methods and tools. Object-oriented software is claimed to be easier to change, hence supporting the frequent changes of an evolutionary life-cycle better than procedural software. Evolutionary development is claimed to be useful for reducing or controlling risks and to build software systems that better reflect the end-user requirements. Unfortunately, the combination of evolutionary development processes such as Rational Unified Process, object-oriented modeling notations such as UML, and object-oriented programming languages such as Java may be a new "silver bullet" for the software community. There is very little scientific evidence to support or refute claims regarding evolutionary development processes in general, and consequences regarding the changeability of resulting object-oriented software in particular.

Clearly, many of the proposed solutions for technology and product assessments, in this thesis and elsewhere, need further validation before it can deliver its potential benefits. After further empirical validation, we believe that the proposed measurement framework may facilitate technology assessment studies related to evolutionary development of object-oriented software:

- Potential causes of changeability decay in evolutionary development of object-oriented software can be identified and preventive guidelines can be evaluated. Consequently, evolutionary development processes can be tailored to improve the changeability of the software.
- Design decisions can be supported by quantitative assessments, e.g., to provide a more "open-ended" design that more easily can incorporate changes.

Although the awkward interaction between developers, users, organizations, products, tools and processes make empirical studies of software engineering a difficult task, it is not an *impossible* task. This thesis has proposed solutions that have been used to improve the current knowledge regarding evolutionary development and changeability.

Appendix A: Raw Data for the Genera Case Study

Most of the raw data for the Braathens and Genera case studies were presented in Section 7.1. The principal component analysis presented in Table 7.9 was based on the raw data given in Table A.1.

Table A.1. Raw data for the class-level coupling and size measures based on the final version of the system

Class name	OMMIC	OMMIC_L	OMAIC	OMAIC_L	OMMEC	OMAEC	MC	CS
EJBGoalHome	0	0	0	0	0	0	1	1
EqualInsertInSource	0	40	0	9	0	0	3	94
GDBC	0	0	0	0	6	0	16	16
GDBCodbc	0	1	0	0	0	0	0	1
GDBCoracle	0	1	0	0	0	0	0	1
GDBCsybase	0	1	0	0	0	0	0	1
Goal	0	0	0	0	0	0	0	0
GoalAppServerMisc	0	0	0	0	0	0	0	0
GoalAppServerMiscImp	0	0	0	0	0	0	0	1
GoalBaseCondition	0	0	0	0	0	19	0	15
GoalBean	0	0	0	0	0	0	5	0
GoalCondition	0	0	0	0	0	36	0	14
GoalConstants	0	0	0	0	0	0	0	38
GoalContext	0	0	0	0	17	0	12	12
GoalContextEjb	11	17	0	4	0	0	13	43
GoalContextGas	11	11	0	0	0	0	12	26
GoalDatabase	2	4	15	0	1	0	9	51
GoalDbConnection	0	0	0	0	0	8	0	2
GoalDbMisc	0	0	0	0	0	0	4	4
GoalDbMiscImpl	2	11	7	1	4	0	4	37
GoalDbMiscWrapper	0	0	0	0	0	12	0	5
GoalErrorHandler	0	0	0	0	0	0	0	0
GoalException	0	0	0	0	0	0	1	5
GoalHome	0	0	0	0	0	0	1	1
GoalImpl	15	0	0	0	11	0	15	29
GoalIterator	14	36	15	0	0	0	22	157
GoalLogWriter	0	0	0	0	0	0	1	0
GoalManipulation	0	0	0	0	0	0	4	4
GoalManipulationImpl	5	10	25	0	4	0	4	66
GoalManipulationWrap	0	0	0	0	0	51	0	8
GoalNavigation	0	0	0	0	0	0	5	5
GoalNavigationImpl	1	1	6	0	5	0	6	30

GoalNavigationWrappe	0	0	0	0	0	15	0	6
GoalQuery	0	0	0	0	0	0	2	2
GoalQueryImpl	2	1	14	0	2	0	2	32
GoalQueryWrapper	0	0	0	0	0	55	0	13
Manipulation	3	3	27	0	0	0	13	39
Query	8	9	35	0	2	0	13	61
AppServerContextObje	0	0	0	0	0	0	0	0
AppServerProxy	0	2	0	0	0	0	1	4
AppServerProxyBase	0	0	0	0	0	0	0	0
AppServerProxyWeblog	0	0	0	0	0	0	0	0
ArgParser	0	2	0	0	2	0	3	17
ConditionBuilder	0	0	16	0	8	0	7	19
CondParser	0	15	39	0	3	3	7	55
DocBasisScanner	2	9	0	1	0	0	2	35
EJBGoal	0	0	0	0	11	0	15	15

Appendix B: The Coffee-Machine Experiment

B.1 Experience Level Questionnaire

Name:

Total number of credits:

Total number of credits in programming:

Estimate the total number of lines of code you have written in the following programming languages:

Prog. Language	0	0, but knows some	<100	<1000	<10000	10000+
Java	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
C++	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
Simula	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
SmallTalk	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
C	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
Pascal	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
Other lang. ( )	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
Other lang. ( )	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]

Which design methods and notations do you know:

Method/Notation	No Experience	Some Experience	Experienced
UML/Rose	[ ]	[ ]	[ ]
OMT	[ ]	[ ]	[ ]
Responsibility-driven design	[ ]	[ ]	[ ]
CRC	[ ]	[ ]	[ ]
role-modeling	[ ]	[ ]	[ ]
Structured analysis and/or structured design	[ ]	[ ]	[ ]
Data-driven design	[ ]	[ ]	[ ]
Other ( )	[ ]	[ ]	[ ]
Other ( )	[ ]	[ ]	[ ]

B.2 Change Tasks

NAME: *<THIS FIELD WAS FILLED OUT BY THE AUTHORS BEFORE THE EXPERIMENT>*

IMPORTANT:

- THE CODE YOU WILL CHANGE IS ATTACHED AT THE END OF THIS DOCUMENT. DO YOUR CHANGES DIRECTLY IN THE SOURCE CODE LISTING.
- BEFORE YOU START THE CHANGE TASK, WRITE DOWN THE START TIME IN THE QUESTIONNAIRE THAT FOLLOWS THIS CHANGE TASK DESCRIPTION.
- WHEN YOU HAVE FINISHED THE CHANGE TASK, YOU COMPLETE THE REMAINDING QUESTIONS OF THE CHANGE TASK QUESTIONNAIRE, BEFORE YOU START THE NEXT ASSIGNMENT.

In this experiment you shall implement changes to a "virtual" coffee-machine. At the moment, the machine can make four different types of coffe (black, white, black w/sugar, white w/sugar). The customer must give textual commands to insert money and select coffee, and will subsequently receive the "coffee", given that he has deposited sufficient funds and assuming all necessary ingredients are available. At present, coffee costs 5 credits. The test run given below shows how the machine works at present:

Example Test Run:

```
Menu: I=insert S=select Q=quit
I
Amount>
4
    CashBox: Depositing 4
    You now have 4 credits.
Menu: I=insert S=select Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream)>
2
    FrontPanel: Insufficient funds
Menu: I=insert S=select Q=quit
I
Amount>
2
    CashBox: Depositing 2
    You now have 6 credits.
Menu: I=insert S=select Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream)>
2
    Dispensing cup
    Dispensing coffee
    Dispensing water
    Dispensing cream
    CashBox: Returning 1
```

CHANGE TASK 1[10]

In this assignment, you shall extend the coffee machine with "return button" functionality that returns the deposited funds. The menu choice is called "Return".

Test Case:

Menu: I=insert S=select R=return Q=quit
I
Amount>
4
    CashBox: Depositing 4
    You now have 4 credits.

Menu: I=insert S=select R=return Q=quit
R
    CashBox: Returning 4

Menu: I=insert S=select R=return Q=quit

CHANGE TASK 2

In this assignment, you shall extend the machine to make bouillon. Bouillon costs more than coffee. While coffee costs 5 credits, bouillon costs 6 credits.
HINT: You must, among others, make a "dispenser " for bouillon powder.

Test Case:

Menu: I=insert S=select R=Return Q=quit
I
Amount>
6
    CashBox: Depositing 6
    You now have 6 credits.

Menu: I=insert S=select R=Return Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream, 5=Bouillon)>
5
    Dispensing cup
    Dispensing bouillon
    Dispensing water
    CashBox: Returning 0

Menu: I=insert S=select R=Return Q=quit

[10] In the actual handouts, each change task was described on a separate printed page. The subjects were instructed not to look at the next change task before they had completed the change task questionnaire for the previous change task.

CHANGE TASK 3

Unfortunately, there is a quite serious problem with the coffee machine at present. If the user chooses for example "coffee with cream", and the cream dispenser is empty, the machine gives a small error message, after which it dispenses black coffee (without cream). If the machine does not contain any more cups, the machine dispenses the drink right into the drain… The user will of course get quite irritated over having to pay for this!

The simplest solution to this problem is that the user receives a message if the machine is out of a required ingredient of the selected drink. Then, the user is given the option to choose another drink. The following test case illustrates what should happen when the machine runs out of cream:

Test Case:

Menu: I=insert S=select R=Return Q=quit
I
Amount>
5
    CashBox: Depositing 5
    You now have 5 credits.

Menu: I=insert S=select R=Return Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream, 5=Bouillon)>
2
    Dispensing cup
    Dispensing coffee
    Dispensing water
    Dispensing cream <after this the machine is out of cream>
    CashBox: Returning 0

Menu: I=insert S=select R=Return Q=quit
I
Amount>
5
    CashBox: Depositing 5
    You now have 5 credits.

Menu: I=insert S=select R=Return Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream, 5=Bouillon)>
2
        Sorry, no more cream! Select another.

Menu: I=insert S=select R=Return Q=quit

B.3 Change Task Questionnaire

Time for start of the change task:

Time for completing the change task (not including answering this questionnaire):

Effort (in minutes) to solve the change task:

A. Effort to understand how to solve the change task:
B. Effort to code the change task:
C. Effort to evaluate/test the solution (run test-case):

How would you characterize your strategy to solve the task?

Very explorative (1) – Very systematic (6):
(Very explorative = "trial and error")
(Very systematic = "analysis, design, code, test")

What is your subjective assessment of your skill level as a programmer?[11]

Very poor (1) - Very skilled (6):

What is your subjective assessment of the quality of your solution?

Very poor (1) - Very good (6):

How confident are you that the solution does not contain serious faults?

Very unsure (1) - Very confident (6):

How difficult did you think the change task was?

Very easy (1) - Very difficult (6):

OTHER COMMENTS:

[11] The subjective skill-level question was asked only once, in the calibration change task questionnaire. The other questions were answered for all change tasks.

B.4 Message Sequence Charts for the Designs

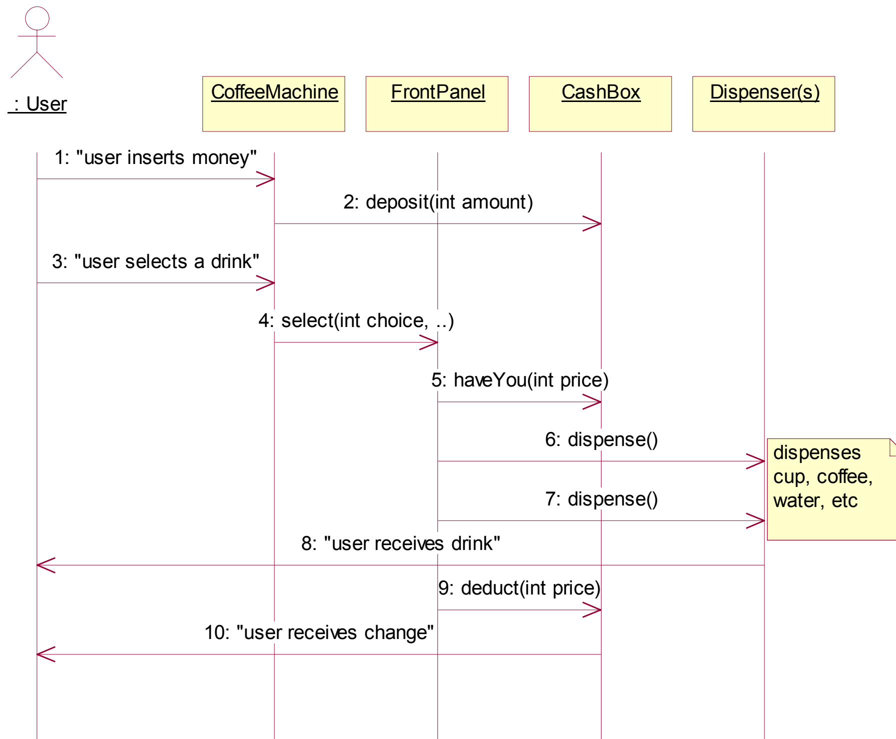

Fig. B.1. Message Sequence Chart for the MF design

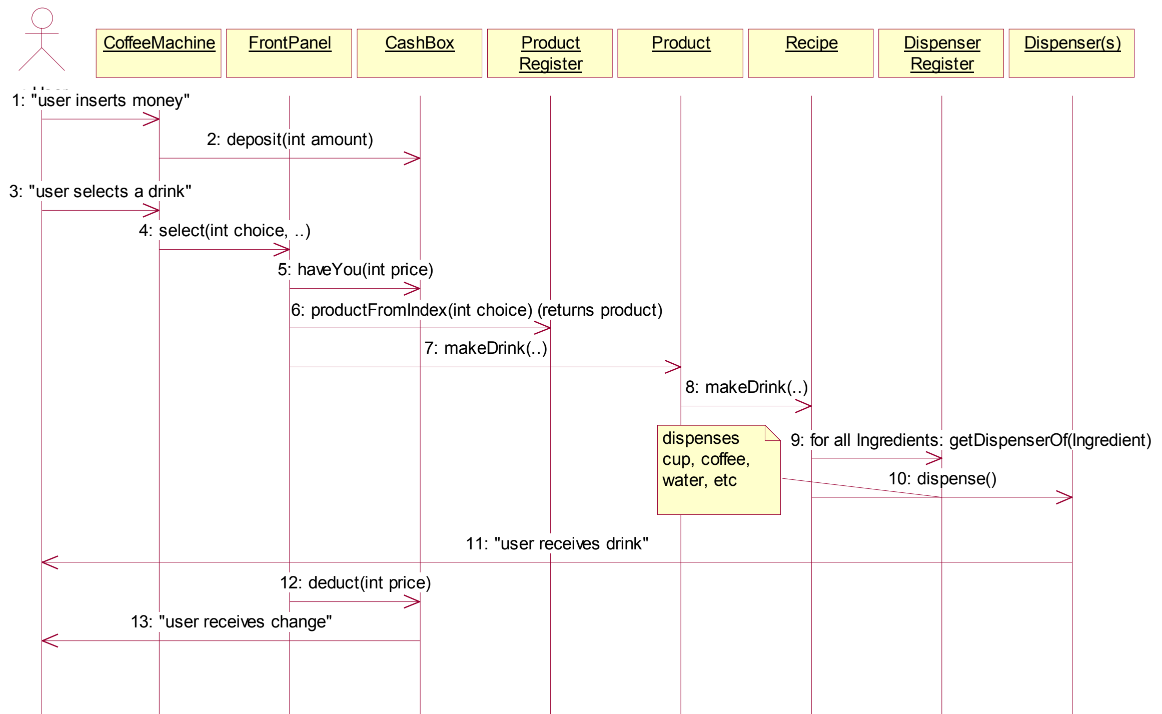

Fig. B.2. Message Sequence Chart for the RD design

B.5 Code Fragments from the Designs

Table B.1. Code fragment from the MF design

```
class FrontPanel
{
        // knows price of selection;
        // knows ingredients needed for each selection
        // asks CashBox how much money was put in
        // instructs dispensers
        <…snip…>

        // constructor method for the FrontPanel class
        FrontPanel() {
                cupDisp = new Dispenser("cup", 50); // 50 cups
                waterDisp = new Dispenser("water", 50);
               <…snip…>
        }

        // user selected a drink. Make drink!
        void select(int choice, CashBox cashBox)
        {
            if (cashBox.haveYou(drinkPrice))
            {
                //1 = Black Coffee, 2=Coffee w/Cream, 3 = Coffee w/Sugar,
                //4 = Coffee w/Sugar & Cream
                if (choice == 1)
                {
                        cupDisp.dispense();
                        coffeeDisp.dispense();
                        waterDisp.dispense();
                }
                <…snip…>
                else // cream & sugar
                {
                        cupDisp.dispense();
                        coffeeDisp.dispense();
                        waterDisp.dispense();
                        sugarDisp.dispense();
                        creamDisp.dispense();
                }
                cashBox.deduct(drinkPrice);
        }
        else
        {
            Output.print("\tFrontPanel: Insufficient funds");
        }
 }
```

Table B.2. Code fragment from the RD design

```
class FrontPanel
{
        // knows price,
        // knows products,
        // asks CashBox how much money was put in
        // instructs the correct product to make the drink
        // instructs CashBox to return change

        private int drinkPrice = 5;

        void select(int choiceIndex, CashBox cashBox, ProductRegister
                        productReg, DispenserRegister dispenserReg) {

            if (cashBox.haveYou(drinkPrice))
            {
                Product product;
                product = productReg.productFromIndex(choiceIndex);
                //1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar,
                //4 = Coffee w/Sugar & Cream

                product.makeDrink(dispenserReg);
                //make product using the dispensers

                cashBox.deduct(drinkPrice); //deduct price
            }
            else
            {
                Output.print("\tFrontPanel: Insufficient funds\n");
            }
        }
    }
```

B.6 Raw data from the main experiment

Table **B.3.** Summary measures of change effort

Subject	Design	Total c1+c2	Total c1+c2+c3	Understand c1+c2+c3	Code c1+c2+c3	Test c1+c2+c3	LearningCurve
1	RD	32	56	30	23	3	-0.520000
2	MF	26	48	15	23.5	9.5	0.200000
3	RD	42	58	29	26	3	-0.090909
4	RD	40	53	36	13	4	0.076923
5	MF	38	63	30	25	8	0.000000
6	RD	56	*	*	*	*	*
7	RD	26	61	12	40	9	0.333333
8	RD	28	42	18	19	5	-0.333333
9	RD	23	38	23	*	*	-0.500000
10	MF	43	79	13	54	12	-0.111111
11	RD	29.5	57.5	20	33.5	4	0.411765
12	RD	31	56	8	38	10	-0.200000
13	MF	30	56	23	28	5	-0.375000
14	RD	49	64	35	*	*	0.200000
15	MF	20	35	17	13	5	-0.166667
16	RD	39	48	25	21	2	0.000000
17	MF	21	36	14	18	4	0.111111
18	RD	54	69	34	26	9	0.142857
19	MF	17	42	8	27	7	-0.428571
20	MF	13	36	10	26	0	-0.250000
21	RD	45	65	35	24	6	-0.500000
22	MF	24	44	11	22	12	-0.666667
23	MF	25	40	11	25	4	0.250000
24	RD	51	67	21	29	*	0.384615
25	MF	24	48	8	34	6	0.000000
26	RD	33.5	61.5	*	*	*	*
27	MF	35	41	9.5	26.5	5	0.846154
28	RD	49	71	39	28	9	0.166667
29	MF	26	52	13	31	8	0.000000
30	MF	40	65	30	35	0	-0.200000
31	MF	31	61	20	29	12	-0.333333
32	RD	29	64	33	29	2	-0.818182
33	RD	30	70	20	42	8	0.000000
34	MF	24	41	24	17	0	-0.411765
35	RD	40	65	25	25	15	-0.500000
36	MF	20	*	*	*	*	*

Table B.4. Data for the calibration task

Subject	Total Cal	Understand Cal	Code Cal	Test Cal	Strategy Cal	Subj Qual. Cal	Confidence Cal	Correctness Cal	ChangeSize Cal
1	55	15	35	5	2	2	2	2	39
2	39	20	14	5	2	4	3	3	17
3	60	28	30	2	4	4	5	6	25
4	65	35	20	10	2	3.5	3	3	14
5	50	25	20	5	5	4	4	5	22
6	56	25	15	16	3	4	4	3	21
7	38	10	23	5	2	5	4	4	19
8	31	14	15	2	2	5	5	4	26
9	55	30	20	5	2	2	3	2	19
10	23	10	10	3	5	5	5	4	18
11	29	14	12	3	3	3	5	5	20
12	37	10	20	7	3	3	6	6	28
13	62	20	40	2	5	5	5	5	23
14	55	10	40	5	2	3	4	3	23
15	40	20	15	5	4	3	5	6	18
16	87	37	40	10	5	4	4	3	30
17	42	16	25	1	5	4	5	4	21
18	50	25	20	5	2	2	3	2	11
19	55	17.5	32.5	2.5	3	4	5	6	24
20	50	20	20	10	3	2	2	2	14
21	55	20	30	5	2	4	4	6	25
22	25	7	15	3	4	5	5	6	15
23	33	7	20	5	4	5	6	4	24
24	51	14	30	7	4	4	3	3	20
25	40	8	25	7	2	5	6	6	28
26	57	30	25	2	4	5	5	4	16
27	50	20	28	2	5	4	5	3	23
28	43	20	10	13	5	4	5	5	19
29	52	10	30	12	4	4	4	6	24
30	45	20	20	5	5.5	4.5	6	5	21
31	45	20	20	5	4	4	5	6	24
32	55	20	30	5	5	6	2	2	15
33	33	15	15	3	4	5	6	3	9
34	50	40	10	0	6	6	6	4	15
35	45	20	20	5	3	4	4	4	18
36	85	60	20	5	3	1	1	3	19

Table B.5. Data for change task c1

Subject	Total c1	Understand c1	Code c1	Test c1	Strategy c1	Subj Qual. c1	Confidence c1	Difficulty c1	Correctness c1	ChangeSize c1
1	12	6	5	1	4	5	5	2	6	3
2	10	6	3.5	0.5	2	4	5	2	6	3
3	13	10	2	1	2	4	3	1	4	2
4	20	14	5	1	4	5	3	1	4	2
5	16	10	5	1	4	5	5	1	6	3
6	9	7	1	1	2	4	2	1	4	2
7	14	4	8	2	1	6	5	2	6	3
8	8	3	4	1	2	5	6	1	6	6
9	10	5	5	0	2	1	6	1	6	3
10	10	4	3	3	2	6	6	1	6	3
11	20.5	12	8	0.5	5	5	5	1	6	8
12	10	2	7	1	6	5	6	1	4	2
13	12	5	6	1	2	4	4	2	6	4
14	21	15	4	2	2	5	5	2	6	3
15	10	5	3	2	5	5	5	2	6	3
16	8	5	2	1	3	3	3	2	6	3
17	8	5	1	2	5	5	5	1	6	3
18	26	12	10	4	4	4	4	2	6	3
19	10	2	7	1	4	6	6	1	4	2
20	7	3	4	0	4	5	5	1	4	3
21	10	5	4	1	4	5	5	1	6	3
22	4	1	2	2	1	5	6	1	6	3
23	10	5	5	0	2	6	6	1	6	3
24	14	9	2	3	3	5	5	1	6	3
25	7	3	2	2	1	6	6	1	4	2
26	13.5	10	3	0.5	4	5	5	1	6	3
27	18	6	10	2	3	3	3	2	6	3
28	19	14	3	2	4	4	4	3	6	3
29	12	5	5	2	3	4	4	1	6	6
30	20	10	10	0	6	5.5	6	1	6	3
31	11	5	4	2	4	5	5	2	4	2
32	10	3	6	1	3	6	6	1	6	3
33	13	10	2	1	5	6	6	2	4	2
34	10	5	5	0	2	6	4	1	6	3
35	20	5	10	5	4	4	3	2	6	3
36	10	2	5	3	4	5	5	1	6	3

Table B.6. Data for change task c2

Subject	Total c2	Understand c2	Code c2	Test c2	Strategy c2	Subj Qual.c2	Confidence c2	Difficulty c2	Correctness c2	Change-Size c2
1	20	5	14	1	2	3	4	3	4	12
2	16	5	8	3	2	5	4	3	6	13
3	29	7	20	2	4	5	5	3	4	17
4	20	10	7	3	1	3	2	4	3	5
5	22	10	10	2	3	4	4	2	6	10
6	47	30	5	12	5.5	3	2	4	4	6
7	12	6	4	2	1	5	5	2	4	8
8	20	9	9	2	2	5	5	3	4	7
9	13	3	8	2	1	4	4	2	4	6
10	33	4	26	3	1	1	5	5	6	27
11	9	3	5.5	0.5	3	5	5	1	3	4
12	21	3	16	2	3	3	5	2	4	14
13	18	7	10	1	2	4	4	2	4	8
14	28	10	15	3	4	5	4	3	6	8
15	10	5	3	2	3	3	4	3	6	11
16	31	15	15	1	2	2	1	4	3	8
17	13	5	7	1	5	6	5	1	4	11
18	28	13	10	5	3	5	4	3	4	7
19	7	1	5	1	5	6	6	1	3	7
20	6	2	4	0	4	5	5	1	3	11
21	35	15	15	5	3	4	4	3	6	15
22	20	5	10	5	2	4	5	3	6	14
23	15	3	10	2	3	5	5	1	6	11
24	37	8	15	14	4	5	3	3	6	17
25	17	2	13	2	2	5	5	2	5	16
26	20	*	*	*	2	5	5	1	5	10
27	17	3	12	2	4	4	3	3	6	12
28	30	15	16	4	5	3	3	4	6	8
29	14	3	8	3	2	4	4	2	6	16
30	20	5	15	0	5	4	6	1	4	25
31	20	5	10	5	3	4	5	4	6	9
32	19	0	18	1	1	5	4	2	6	9
33	17	0	15	2	2	4	3	3	5	14
34	14	7	7	0	2	5	5	2	3	14
35	20	5	10	5	3	3	2	2.5	4	6
36	10	2	8	0	4	4	4	4	*	*

Table B.7. Data for change task c3

Subject	Total c3	Understand c3	Code c3	Test c3	Strategy c3	Subj Qual. c3	Confidence c3	Difficulty c3	Correctness c3	Change-Size c3
1	24	19	4	1	2	3	3	4	2	*
2	22	4	12	6	2	3	4	2	5	15
3	16	12	4	0	4	4	3	4	*	*
4	13	12	1	0	3	*	*	*	*	*
5	25	10	10	5	4	3	3	3	6	67
6	*	*	*	*	5	*	*	*	*	*
7	35	2	28	5	3	3	3	5	6	16
8	14	6	6	2	1	2	3	3	4	11
9	15	15	*	*	6	1	1	1	*	*
10	36	5	25	6	*	3	3	5	5	22
11	28	5	20	3	3	4.5	4	3	6	18
12	25	3	15	7	3	4	3	2	4	10
13	26	11	12	3	1	4	4	2	2	60
14	15	10	*	*	3	*	*	3	*	*
15	15	7	7	1	2	1	3	3	6	70
16	9	5	4	0	1	1	1	*	*	*
17	15	4	10	1	5	4	5	2	4	62
18	15	9	6	0	4	3	3	3	*	*
19	25	5	15	5	2	5	5	2	3	28
20	23	5	18	0	4	4	4	2	3	60
21	20	15	5	0	2	2	3	3	*	*
22	20	5	10	5	4	4	5	4	6	13
23	15	3	10	2	2	3	4	2	3	56
24	16	4	12	*	3	2	2	2	6	14
25	24	3	19	2	1	5	4	2	6	24
26	28	*	*	*	1	4	3	2	6	16
27	6	0.5	4.5	1	5	5	4	1	1	2
28	22	10	9	3	5	4	4	3	5	9
29	26	5	18	3	*	3	4	4	6	16
30	25	15	10	0	6	4	6	2	5	11
31	30	10	15	5	2	3	4	4	6	29
32	35	30	5	0	2	2	1	4	*	*
33	40	10	25	5	3	3	3	5	6	20
34	17	12	5	0	5	6	5	2	6	20
35	25	15	5	5	3	2	1	3	*	*
36	*	*	*	*	*	*	*	*	*	*

Appendix C: Raw data from the Ooram Case Study

Table C.1. Raw data for the dynamic coupling measures and NumChanges for Section 7.3.3

Class	CS	IC_OA	IC_OM	IC_OD	IC_CA	IC_CM	IC_CD	EC_OA	EC_OM	EC_OD	EC_CA	EC_CM	EC_CD	NumChanges
1	210	4	13	46	1	3	5	7	26	113	1	1	2	5
2	193	1	2	8	2	2	8	0	0	0	0	0	0	2
3	53	1	4	158	1	4	158	0	0	0	0	0	0	3
4	60	1	5	62	1	5	62	0	0	0	0	0	0	1
5	21	2	2	7	1	1	7	0	0	0	0	0	0	0
6	642	4	8	10	3	5	10	0	0	0	0	0	0	4
7	2590	0	0	0	6	20	3763	0	0	0	1	1	10	7
8	2382	0	0	0	1	1	152	0	0	0	13	23	6250	6
9	1798	0	0	0	3	3	6	0	0	0	0	0	0	6
10	1515	0	0	0	4	6	23	0	0	0	6	10	35	4
11	1903	7	35	5962	6	9	2199	2	4	12	1	2	2	7
12	1449	2	15	104	2	12	84	8	39	3790	6	21	1220	6
13	1569	3	5	14	2	3	9	2	3	9	1	1	1	6
14	1235	6	10	107	2	4	107	0	0	0	0	0	0	3
15	1175	0	0	0	2	2	18	0	0	0	10	19	2673	6
16	936	3	16	959	3	12	882	8	30	4383	7	22	932	7
17	731	6	27	5531	9	23	5513	8	28	1200	6	16	1143	6
18	573	1	7	303	2	7	303	0	0	0	0	0	0	3
19	570	3	4	25	4	4	25	0	0	0	0	0	0	1
20	603	2	7	38	2	5	31	4	13	451	3	9	436	6
21	570	4	16	1573	5	13	1557	4	13	2438	4	10	2315	6
22	460	3	11	79	4	7	57	8	22	863	5	13	528	6
23	633	3	7	32	3	5	4	3	11	55	4	5	20	6
24	300	4	14	2139	5	9	2076	6	19	3843	6	12	1512	5

BIBLIOGRAPHY

Adrion, W.R. (1993). Research Methodology in Software Engineering. *ACM Software Engineering Notes*, 18 (1), 36–37.

Arisholm, E., Anda, B., Jorgensen, M. and Sjoberg, D. (1999a). Guidelines on Conducting Software Process Improvement Studies in Industry. In: *22nd IRIS Conference (Information Systems Research Seminar In Scandinavia),* Keuruu, Finland, pp. 87–102.

Arisholm, E., Benestad, H.C., Skandsen, J. and Fredhall, H. (1998). Incorporating Rapid User Interface Prototyping in Object-Oriented Analysis and Design with Genova. In: Proceedings of NWPER'98 Nordic Workshop on Programming Environment Research*,* Sweden, pp. 155–161.

Arisholm, E. and Sjøberg, D. (1999). Empirical Assessment of Changeability Decay in Object-Oriented Software. In: ICSE'99 Workshop on Empirical Studies of Software Development and Evolution*,* Los Angeles, CA, pp. 62–69.

Arisholm, E. and Sjøberg, D.I.K. (2000). Towards a Framework for Empirical Assessment of Changeability Decay. *The Journal of Systems and Software*, 53 (1), 3–14.

Arisholm, E., Sjøberg, D.I.K. and Jørgensen, M. (2001). Assessing the Changeability of two Object-Oriented Design Alternativers – a Controlled Experiment. *Empirical Software Engineering*, Accepted for publication.

Arisholm, E., Skandsen, J., Sagli, K. and Sjøberg, D.I.K. (1999b). Improving an Evolutionary Development Process – A Case Study. In: Proceedings of the EuroSPI'99 (European Software Process Improvement Conference)*,* Pori, Finland, pp. 9.40–9.50.

Basili, V., Briand, L. and Melo, W. (1996a). How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM*, 39 (10), 104–116.

Basili, V.R., Briand, L.C. and Melo, W.L. (1996b). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22 (10), 751–761.

Basili, V.R. and Turner, A.J. (1975). Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering*, 1 (4), 390–396.

Benlarbi, S. and Melo, W.L. (1999). Polymorphism Measures for Early Risk Prediction. In: 21st International Conference of Software Engineering (ICSE'99)*,* Los Angeles, CA, pp. 334–344.

Bersoff, E.H. and Davis, A.M. (1991). Impacts of Life Cycle Models on Software. *Communications of the ACM*, 34 (8), 104–118.

Bieman, J.M. and Kang, B.K. (1998). Measuring Design-Level Cohesion.

IEEE Transactions on Software Engineering, 24 (2), 111–124.

Binkley, A.B. and Schach, S.R. (1998). Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. In: 20th International Conference on Software Engineering (ICSE'98), pp. 452–455.

Boehm, B. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ., Prentice-Hall.

Boehm, B.W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, 21 (5), 61–72.

Boehm, B.W., Gray, T.E. and Seewaldt , T. (1984). Prototyping versus Specifying – A Multiproject Experiment. *IEEE Transactions on Software Engineering*, 10 (3), 290–302.

Boehm, B.W. and Papaccio, P.N. (1988). Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14 (10), 1462–1477.

Bølviken, E. and Skovlund, E. (1994). *Lectures in Applied Statistics*. Dept. of Mathematics, University of Oslo.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummins Publishing Company Inc.

Booch, G., Rumbaugh, J. and Jacobson, I. (1998). *The Unified Modeling Language Users Guide*. Addison-Wesley.

Braa, K. and Vidgen, R. (1999). Interpretation, intervention, and reduction in the organizational laboratory: a framework for in-context information system research. *Accounting Management and Information Technologies*, 1999 (9), 25–47.

Bradburn, N.M. (1982). "Question-wording effects in surveys". In: *New directions for methodology of social and behavioral science: Question framing and response consistency*. Hogarth (editors), Jossey-Bass, San Francisco, pp. 65–76.

Briand, L., Bunse, C. and Daly, J.W. (1999a). A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs. *IEEE Transactions on Software Engineering*, Accepted for publication. *(Also available as technical report ISERN-99-07).*

Briand, L.C., Arisholm, E., Counsell, S., Houdek, F. and Thevenod, P. (1999b). Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of The Art and Future Directions. *Empirical Software Engineering*, 4 (4), 387–404.

Briand, L.C., Basili, V.R. and Thomas, W.M. (1992). A Pattern Recognition Approach for Software Engineering Data Analysis. *IEEE Transactions on Software Engineering*, 18 (11), 931–942.

Briand, L.C., Bunse, C., Daly, J.W. and Differding, C. (1997a). An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. *Empirical Software Engineering*, 2 (3),

291–312.
Briand, L.C., Carriere, S.J., Kazman, R. and Wust, J. (1998a). COMPARE: a comprehensive framework for architecture evaluation. In: Object-Oriented Technology. ECOOP'98 Workshop Reader. ECOOP'98 Workshops, Demos, and Posters. Berlin, Germany, Springer-Verlag, pp. 48–49. Extended version available as ISERN Technical Report TR-98-29.
Briand, L.C., Daly, J. and Wust, J. (1998b). A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3 (1), 65–117.
Briand, L.C., Daly, J.W., Porter, V. and Wust, J. (2000). Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. *Journal of Systems and Software*, 51 (3), 245–273.
Briand, L.C., Daly, J.W. and Wust, J. (1999c). A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25 (1), 91–121.
Briand, L.C., Devanbu, P. and Melo, W.L. (1997b). An Investigation into Coupling Measures for C++. In: 19th International Conference on Software Engineering (ICSE'97), Boston, USA, pp. 412–421.
Briand, L.C., El Emam, K. and Morasca, S. (1995). Theoretical and Empirical Validation of Software Product Measures. ISERN Technical Report 95-03,
Briand, L.C., Emam, K.E. and Morasca, S. (1996a). On the Application of Measurement Theory in Software Engineering. *Empirical Software Engineering*, 1 (1), 61–68.
Briand, L.C., Morasca, S. and Basili, V.R. (1996b). Property-based Software Engineering Measurement. *IEEE Transactions on Software Engineering*, 22 (1), 68–85.
Briand, L.C., Morasca, S. and Basili, V.R. (1998c). Defining and Validating Measures for Object-Based High-Level Design. *IEEE Transactions on Software Engineering*, 25 (5), 722–743.
Briand, L.C. and Wust, J. (1999). The Impact of Design Properties on Development Cost in Object-Oriented Systems. ISERN Technical Report TR-99-16.
Briand, L.C. and Wust, J. (2000). Integrating Scenario-based and Measurement-based Software Product Quality. ISERN Technical Report 00-04.
Briand, L.C., Wust, J., Ikonomovski, S.V. and Lounis, H. (1999d). Investigating Quality Factors In Object-Oriented Designs: an Industrial Case Study. In: 21st International Conference of Software Engineering (ICSE'99), Los Angeles, CA., pp. 345–354.
Briand, L.C., Wust, J. and Lounis, H. (1999e). Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. In: International Conference on Software Maintenance (ICSM'99), IEEE Comput. Society, pp. 475–482.

Brito e Abreu, F. and Melo, W. (1996). Evaluating the Impact of Object-Oriented Design on Software Quality. In: Proceedings of the Third International Software Metrics Symposium (METRICS'96), Berlin, pp. 90–99.

Brownsword, L. and McUmber, R. (1991). Applying the Iterative Development Process to Large 2167A Ada Projects. In: TRI-Ada'91, New York, USA, ACM, pp. 378–386.

Bruckhaus, T., Madhavji, N., Janssen, I. and Henshaw, J. (1996). The Impact of Tools on Software Productivity. *IEEE Software*, 13 (5), 29–38.

Cartwright, M. and Shepperd, M. (2000). An Empirical Investigation of an Object-Oriented Software System. *IEEE Transactions on Software Systems*, 26 (8), 786–796.

Chaumun, M.A., Kabaili, H., Keller, R.K., Lustman, F. and Saint-Denis, G. (2000). Design Properties and Object-Oriented Software Changeability. In: Fourth Euromicro Working Conference on Software Maintenance and Reengineering, pp. 45–54.

Chidamber, S.R., Darcy, D.P. and Kemerer, C.F. (1998). Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering*, 24 (8), 629–637.

Chidamber, S.R. and Kemerer, C.F. (1994). A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20 (6), 476–493.

Chong Hok Yuen, C.K.S. (1987). A Statistical Rationale for Evolution Dynamics Concepts. In: Proc. Conf. Software Maintenance, Austin, Texas, IEEE, pp. 156–164.

Christensen, R. (1996). *Analysis of Variance, Design and Regression. Applied Statistical Methods.* Chapman & Hall.

Churcher, N.I. and Shepperd, M.J. (1995). Towards a Conceptual Framework for Object-Oriented Software Metrics. *Software Engineering Notes*, 20 (2), 69–76.

Coad, P. and Yourdon, E. (1991a). *Object-Oriented Analysis*. Prentice-Hall.

Coad, P. and Yourdon, E. (1991b). *Object-Oriented Design*. Prentice-Hall.

Cockburn, A. (1998). The Coffee Machine Design Problem: Part 1 & 2. *C/C++ User's Journal*, 1998 (May/June).

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

Cotton, T. (1996). Evolutionary Fusion: A Customer-Oriented Incremental Life-Cycle for Fusion. *Hewlett-Packard Journal*, 47 (4), 25–38.

Courtney, R.E. and Gustafson, D.A. (1993). Shutgun correlations in software measure. *Software Engineering Journal*, 1993 (January), 5–13.

Cunningham, J.B. (1997). Case study principles for different types of cases. *Quality and Quantity*, 31, 401–423.

Daly, J., Brooks, A., Miller, J., Roper, M. and Wood, M. (1996). Evaluating

Inheritance Depth on the Maintainability of Object-Oriented Software. *Empirical Software Engineering*, 1 (2), 109–132.

Draper, N.R. and Smith, H. (1981). *Applied Regression Analysis*. John Wiley & Sons, Inc.

Ehn, P. (1993). “Ch.4: Chandinavian Design: On Participation and Skill”. In: *Participatory Design: Principles and Practice*. Schuler, D.N., Aki (editors), Lawrence Erlbaum, pp. 41–77.

Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S. and Mockus, A. (1999). Does Code Decay? Assessing the evidence from Change Management Data.. *Submitted to IEEE Transactions on Software Engineering*.

Emam, K.L., Quintin, S. and Madhavji, N.Z. (1996). User Participation in the Requirements Engineering Process: An Empirical Study. *Requirements Engineering*, 1996 (1), 4–26.

Fagan, M.R. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15 (3), 182–211.

Fenton, N. (1992). When a software measure is not a measure. *Software Engineering Journal*, 1992 (September), 357–362.

Fenton, N. (1994). Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20 (3), 199–206.

Fenton, N., Pfleeger, S.L. and Glass, R.L. (1994). Science and Substance: A Challenge to Software Engineers. *IEEE Software*, 1994 (July), 86–95.

Fioravanti, F. and Nesi, P. (2000). A method and tool for assessing object-oriented projects and metrics management. *Journal of Systems and Software*, 53 (2), 111–136.

Fioravanti, F., Nesi, P. and Stortoni, F. (1999). Metrics for controlling effort during adaptive maintenance of object oriented systems. In: Proceedings IEEE International Conference on Software Maintenance 1999 (ICSM'99), Los Alamitos, CA, USA, IEEE Comput. Soc, pp. 483–492.

Floyd, C. (1984). “A Systematic Look at Prototyping”. In: *Approaches to Prototyping*. (editors), Springer-Verlag, pp. 105–122.

Garvin, D. (1984). What does 'Product Quality' Really Mean. *Sloan Management Review.*

Gilb, T. (1988). *Principles of Software Engineering Management*. Addison-Wesley.

Gilgun, J.F. (1992). *Definitions, Methodologies, and Methods in Qualitative Famility Research*. Qualitative Methods in Family Research, Thousand Oaks, Sage.

Glass, R.L. (1994). The Software Research Crisis. *IEEE Software*, 11 (6), 42–47.

Harrison, R., Counsell, S. and Nithi, R. (2000). Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software.*, 52 (2–3), 173–179.

Harrison, R., Counsell, S.J. and Nithi, R.V. (1998a). An Investigation into the

Applicability and Validity of Object-Oriented Design Metrics. *Empirical Software Engineering*, 3 (3), 255–273.

Harrison, R., Counsell, S.J. and Reuben, V.N. (1998b). An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Transactions on Software Engineering*, 24 (6), 491–496.

Henry, S., Humphrey, M. and Lewis, J. (1990). Evaluation of the maintainability of object-oriented software. In: IEEE Region 10 Conference on Computer and Communication Systems (TENCON'90), New York, NY, USA, pp. 404–409.

Holm, S. (1979). A Simple Sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 1979 (6), 65–70.

Houdek, F., Ernst, D. and Schwinn, T. (1999). Comparing Structured and Object-Oriented Methods for Embedded Systems: A Controlled Experiment. In: ICSE'99 Workshop on Empirical Studies of Software Development and Evolution (ESSDE), Los Angeles, USA, pp. 75–79.

Hufnagel, E.M. and Conca, C. (1994). User response data: The potential for errors and biases. *Information Systems Research*, 5 (1), 48–73.

ISO9126 (1992). Information Technology: software product evaluation: quality characteristics and guidelines for their use. International Organization for Standardization.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1992). *Object-Oriented Software Engineering*. Addison-Wesley.

Jarvinen, P. (1999). On Research Methods. ISBN 951-97113-6-8.

Jones, C. (1994). Gaps in the Object-Oriented Paradigm. *IEEE Computer*, 27 (6), 90–91.

Jørgensen, M. (1994). Empirical Studies of Software Maintenance. PhD Thesis, University of Oslo.

Jørgensen, M. (1995). Experience With the Accuracy of Software Maintenance Task Effort Prediction Models. *IEEE Transactions on Software Engineering*, 21 (8), 674–681.

Jørgensen, M. (1999). Software Quality Measurement. *Advances in Engineering Software*, 30 (12), 907–912.

Jørgensen, M., Bygdås, S.S. and Lunde, T. (1995). Efficiency Evaluation of CASE Tools – Methods and Results. TF R 38/95, Telenor FoU.

Kazman, R., Abowd, G., Bass, L. and Clements, P. (1996). Scenario-Based Analysis of Software Architecture. *IEEE Software*, 13 (6), 47–56.

Kemerer, C.F. and Slaughter, S. (1999). An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Software Engineering*, 25 (4), 493–509.

Kerlinger, F.N. (1988). *Foundation of behavioral research*. New York, Holt Rinehart and Winston Inc.

Khoshgoftaar, T.M. and Allen, E.B. (1998). Classification of Fault-Prone Software Modules: Prior Probabilities, Costs and Model Evaluation. *Empirical Software Engineering*, 3 (3), 275–298.

Kitchenham, B. (1996a). *Software Metrics. Measurement for Software Process Improvement.* Blackwell Publishers Inc.

Kitchenham, B. and Pickard, L. (1987). Towards a constructive quality model. Part II: Statistical techniques for modelling software quality in the ESPRIT REQUEST project. *Software Engineering Journal*, 2 (4), 114–126.

Kitchenham, B., Pickard, L. and Pfleeger, S.L. (1995a). Case Studies for Method and Tool Evaluation. *IEEE Software*, 12 (4), 52–62.

Kitchenham, B.A. (1996b). Evaluating Software Engineering Methods and Tools . Part 1: The Evaluation Context and Evaluation Methods. *ACM Software Engineering Notes*, 21 (1), 11–15.

Kitchenham, B.A., Fenton, N. and Pfleeger, S.L. (1995b). Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*, 21 (12), 929–944.

Kraemer, K.L. (1993). *The information systems research challenge: Survey research methods*. Boston, Harvard Busines School.

Kruchten, P. and Royce, W. (1996). A Rational Development Process. *CrossTalk*, 9 (7), 11–16.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y. and Chen, C. (1994). Change Impact Identification in Object-Oriented Software Maintenance. In: International Conference on Software Maintenance, IEEE, pp. 202–211.

Lee, A.S. (1989). A scientific methodology for MIS case studies. *MIS quarterly*, 13 (1), 33–50.

Lehman, M.M. and Belady, L.A. (1985). *Program Evolution: Processes of Software Change*. Academic Press.

Li, W. and Henry, S. (1993). Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23 (2), 111–122.

Lichter, H., Schneider-Hufschmidt, M. and Zullighoven, H. (1994). Prototyping in Industrial Software Projects - Bridging the Gap beween Theory and Practice. *IEEE Transactions on Software Engineering*, 20 (11), 825–832.

Lientz, B.P., Swanson, E.B. and Tompkins (1978). Characteristics of Application Software Maintenance. *Communications of the ACM*, 21 (6), 466–471.

Linger, R.C. (1993). Cleanroom Software Engineering for Zero-Defect Software. In: 15th International Conference on Software Engineering (ICSE'93), IEEE, pp. 2–13.

May, E.L. and Zimmer, B.A. (1996). The Evolutionary Development Model for Software. *Hewlett-Packard Journal*, 47 (4), 39–45.

Munson, J.B. (1981). Software Maintainability: a practical concern for life-

cycle costs. *IEEE Computer*, 14 (11), 103–109.

Nesi, P. and Querci, T. (1998). Effort estimation and prediction of object-oriented systems. *Journal of Systems and Software*, 42 (1), 89–102.

Parnas, D.L. (1979). Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5 (2), 128–138.

Parnas, D.L. (1994). Software Aging. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE94),* Sorrento, Italy, pp. 279–287.

Patton, B. (1983). Prototyping – a nomenclature problem. *ACM SIGSOFT Software Engineering Notes*, 8 (2), 14–16.

Peercy, D.E. (1981). A Software Maintainability Evaluation Methodology. *IEEE Transactions on Software Engineering*, SE–7 (4), 343–351.

Pfleeger, S.L. (1995). Experimental Design and Analysis in Software Engineering. Part 2: How to Set Up an Experiment. *ACM Software Engineering Notes*, 20 (1), 22–26.

Pfleeger, S.L. (1998). *Software Engineering: Theory and Practice*. Prentice Hall.

Pickard, L., Kitchenham, B. and Jones, P. (1998). Combining Software Engineering Results in Software Engineering. In: Proc. of the EASE'98 conference, Keele, UK.

Popper, K. (1968). *The logic of scientific discovery*. New York, Harper Torchbooks.

Pressmann, R.S. (1997). *Software Engineering. A Practitioner's Approach*. McGraw-Hill.

Reenskaug, T., Wold, P. and Lehne, O.A. (1995). *The OOram Software Engineering Method*. Manning/Prentice-Hall.

Royce, W. (1970). Managing the development of large software systems: Concepts and techniques. In: Proceedings of IEEE WESTCON, Los Angeles, pp. 1–9.

Royce, W. (1990). TRW's Ada Process Model for Incremental Development of Large Software Systems. In: 12th International Conference on Software Engineering (ICSE'12), Los Alamitos, CA, IEEE, pp. 2–11.

Seaman, C.B. (1999). Qualitative Methods in Emprical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25 (4), 557–572.

Sharble, R.C. and Cohen, S.S. (1993). The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods. *Software Engineering Notes*, 18 (2), 60–73.

Sjøberg, D.I.K., Welland, R. and Atkinson, M.P. (1997a). Software Constraints for Large Application Systems. *The Computer Journal*, 40 (10), 598–616.

Sjøberg, D.I.K., Welland, R., Atkinson, M.P., Jørgensen, M., Martinussen, J.P. and Maus, A. (1996). Evaluating Software Maintenance Technology.

In: Norwegian Conference in Informatics, Alta, Norway, 18–20 November, TAPIR, pp. 49–61.

Sjøberg, D.I.K., Welland, R., Atkinson, M.P., Philbrow, P. and Waite, C. (1997b). Exploiting Persistence in Build Management. *Software – Practice and Experience*, 27 (4), 447–480.

Sommerville, I. (1996). Software Process Models. *ACM Computing Surveys*, 28 (1), 269–271.

Sommerville, I. (2001). *Software Engineering*. Pearson Education Limited.

Sørumgård, L.S. (1997). Verification of Process Conformance in Empirical Studies of Software Development. PhD Thesis, Norwegian University of Science and Technology (NTNU).

Tryggeseth, E. (1997). Support for Understanding in Software Maintenance. PhD Thesis, Norwegian University of Science and Technology (NTNU).

von Mayrhauser, A., Vans, A.M. and Howe, A.E. (1997). Program Understanding Behaviour during Enhancement of Large-scale Software. *Journal of Software Maintenance: Research and Practice*, 9 (5), 299–327.

Walsham, G. (1995). Interpretive case studies in IS research: nature and method. *European Journal of Information Systems*, 4 (2), 74–81.

Weyuker, E.J. (1988). Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14 (9), 1357–1365.

Whyte, W.F. (1991). *Participatory Action Research*. Newbury Park, CA., Sage publications.

Yin, R.K. (1994). *Case Study Research, Design and Methods, 2nd edition*. Thousand Oaks, CA., Sage Publications.

Zamperoni, A., Gerritsen, B. and Bril, B. (1995). Evolutionary Software Development: An experience Report on Technical and Strategic Requirements. Technical Report TR-95-25, Leiden University, The Netherlands.

Zelkowitz, M.V. and Wallace, D.R. (1998). Experimental Models for Validating Technology. *Computer*, 31 (5), 23–31.

Zuse, H. (1991). *Software Complexity: Measures and Methods*. de Gruyter.