Towards a Framework for Empirical Assessment of Changeability Decay[1]

Erik Arisholm and Dag I.K. Sjøberg

Industrial System Development
Department of Informatics, University of Oslo
PO Box 1080 Blindern, N-0316 Oslo, Norway
{erika,dagsj}@ifi.uio.no

Abstract. Evolutionary development allows early and frequent adaptations to new or changed requirements. However, such unanticipated changes may invalidate design documentation and cause structural degradations of the software, which in turn may accelerate *changeability decay*. Our definition of changeability decay focuses on the increased effort required to implement changes. We have identified three approaches to the assessment of changeability decay: (1) Structure measurement, (2) change complexity measurement and (3) benchmarking. Our research aims to evaluate and compare these approaches in order to develop an empirical assessment framework.

In this paper we propose a set of change complexity measures (2) and compare them with structural attribute measures (1) using detailed process and product data collected from a commercial object-oriented development project. The preliminary results indicate that the change complexity measures capture some dimensions of changeability decay not accounted for with structural attribute measures. However, the current findings also suggest that many aspects of changeability decay cannot be accounted for by the indirect measures utilized in approach (1) and (2). As an alternative approach, we therefore propose using benchmarks (3) where change effort can be measured more directly. A research methodology for the development of benchmarks and benchmarking procedures are described.

1 Introduction

Handling *change* is one of those fundamental problems in software engineering. Evolutionary development has been proposed as an efficient way to deal with risks such as 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. For this reason, the design and maintenance of an "open-ended architecture" is 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 (Kruchten and Royce, 1996). While 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). In our experience, the frequent changes may also lead to inconsistent and outdated

[1] To appear in Journal of Systems and Software, September 2000.

requirements specifications and design documentation. These factors are likely causes of *changeability decay*.
The goals of our research are:

- to define changeability decay,
- to develop a framework for assessing changeability decay,
- to identify factors causing decay, and
- to develop and evaluate preventive guidelines in the context of evolutionary development of object-oriented software.

This paper focuses on the first two goals. Our definition of changeability decay focuses on the difference in effort to implement a given change in two different versions of a software system. Three alternative approaches to measuring changeability decay are identified: (1) *Structure measurement,* (2) *change complexity measurement* and (3) *benchmarking.*

The motivation for *structure measurement* (1) is that the (external) changeability attribute of the software system is very difficult to quantify directly on 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 complexity measurement* (2) 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 complexity measurement may be a more accurate indicator of changeability decay than structure measurement, because, unlike structure measurement, it accounts for how changes propagate through the software structure.

In this paper, the change complexity measures are compared with structural attribute measures using detailed process and product data collected from a commercial object-oriented development project. Our preliminary results indicate that the change complexity measures may be more useful indicators of changeability decay than structural attribute measures. However, the current findings also suggest that many aspects of changeability decay cannot be accounted for by the indirect measures utilized in approach (1) and (2).

Therefore, as an alternative approach, we propose using *benchmarks* (3) 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 different versions of a software system. Implementing the same changes on different versions of the software provides the necessary baseline that ensures that change efforts can be compared. In addition to the impact of deteriorating structure, other aspects (e.g. inconsistent documentation, 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 of benchmarks and benchmarking procedures are therefore described.

Further validation work is required to determine the limitations and usefulness of the assessment framework. Since the proposed assessment framework essentially is independent of the underlying software engineering process, we believe there may also be other interesting applications for this work.

The remainder of this paper is organized as follows. Section 2 provides the definition of changeability decay and describes the three approaches to changeability decay assessment. Section 3 describes the relationships between the approaches and discusses validation issues. Section 4 attempts to empirically evaluate aspects of the assessment framework. Finally, Section 5 concludes and describes on-going and future research.

2 Measurements of Changeability Decay in Object-Oriented Software

A prerequisite for collecting meaningful measures is to have a clear understanding of the attributes in the empirical relational system (Zuse, 1991; Fenton, 1994). Consequently, in this section we first define changeability decay. We then describe three alternative approaches to measuring it. Tradeoffs and relationships between these approaches are discussed in Section 3.

2.1 Changeability Decay

The changeability of a software system characterizes the ease of implementing changes to the system. Intuitively, changeability decay is the decrease in changeability between two versions of a software system, more formally:

Definition of 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* iff $e2 > e1$.

We emphasize that in this definition, we do not regard the "given change" in isolation, but also include 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 new errors (the ripple effect). One study found that more than 50% of all errors were due to previous changes (Collofello and Buck, 1987).

The proposed definition of changeability decay exhibits important differences from related work. Our research is primarily 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", our definition is concerned with *"deteriorating structure"*, but only to the extent to which such deterioration actually affects changeability.

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 the software system. Thus, unlike the definition of "Code Decay", our 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 reflected in our definition since a "change" includes the work required to ensure consistency and reliability. However, we feel that *INTERVAL* is not a good indicator of changeability decay as the time schedule may depend on other external factors than the system attribute "changeability". Thus, *INTERVAL* is not reflected in our definition of changeability decay.

We recognize the importance of providing an operational definition of changeability decay as a prerequisite for meaningful measurement. However, the concept of changeability decay is not fully understood to the extent that one can claim to know exactly how it should be defined and measured. For example, as pointed out in (Eick et al., 1999), the actual change effort depends on the ability of the developer implementing the change. Thus, changeability decay may be viewed not only as an attribute of the software system but also as an attribute of "people". In principle, when we refer to "the total effort" in our definition of changeability decay, we could have added *"with respect to a given developer"*. However, our goal is to measure the *system* attribute changeability decay in an abstract context where the individual skill level of a given developer is measurement noise. Thus, experimental designs and statistical techniques must be used to control for the external factors such as the individual skill level of developers. We are in the process of conducting several empirical case studies and experiments that may provide valuable insight to improve our understanding of the empirical relational system and hence – to improve the definition of changeability decay. In the following, we will discuss the three approaches to measuring changeability decay according to our current understanding of the underlying concepts, and defer other important measurement theoretical issues to Section 3.

2.2 Structure measurement

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 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., 1996; Harrison et al., 1998; Briand et al., 1999b; Briand et al., 1999d). Thus, it is conceivable that such structural attribute measures can be used in a prediction model of change effort where increasing values in the model output indicate changeability decay.

2.2.1 Selection of Structural Attributes

We have selected a few and relatively simple measures that we believe capture some important and intuitive dimensions of an object-oriented structure: "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 architecture with good functional responsibility alignment among classes, which in turn may affect the changeability of the software system.

There are several good reasons for using existing structural attribute measures instead of inventing new ones (Briand et al., 1999a). We base the structural attribute measures on existing measures described in the literature (Briand et al., 1997; Briand et al., 1999b; Briand et al., 1999c; Briand et al., 1999d). However, the current state of practice indicates that it is premature to select only one type or dimension of coupling (Briand et al., 1999a). Consequently, we are investigating several dimensions of coupling, in particular the static, class level coupling measures defined in (Briand et al., 1997) adapted to C++, Java and Visual Basic, as well as dynamic import and export coupling measures for SmallTalk systems. Note that the coupling measures in Table 1 (*IC* and *EC*) refer to all of these import coupling and export coupling measures, respectively.

It is commonly believed that size is a major contributor of "complexity". We measure two dimensions of the overall system size: lines of code and class counts.

Table 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 (Briand et al., 1997) adapted to C++, Java and Visual Basic, as well as dynamic import coupling measures for SmallTalk.
Export Coupling	*EC(c)*	The class level export coupling measures defined in (Briand et al., 1997) adapted to C++, Java and Visual Basic, as well as dynamic export coupling measures for SmallTalk.

2.3 Change Complexity Measurement

The proposed *change complexity measurement* is a combination of structure measurement and measures of the actual changes carried out during the development process. It measures properties of the change itself, as well as structural attributes of those parts of the software system 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:

- Changes affecting "complex" structural components require more development effort than changes affecting "less complex" structural components.
- Although the overall structural attributes of the software may remain more or less constant, the change complexity may still vary substantially.

Table 2 describes the proposed measures in some detail. The main idea is to consider how changes propagate through the various components (i.e. classes) in the software structure. For each component affected by a change, the proportion of work carried out on that component 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 components affected by the change are also measured. By using the class level change profile as a weighting factor on the four structural attribute measures, we obtain the "change complexity measures" *CSCP*, *ICCP*, *ECCP* and *MCCP* for a given change.

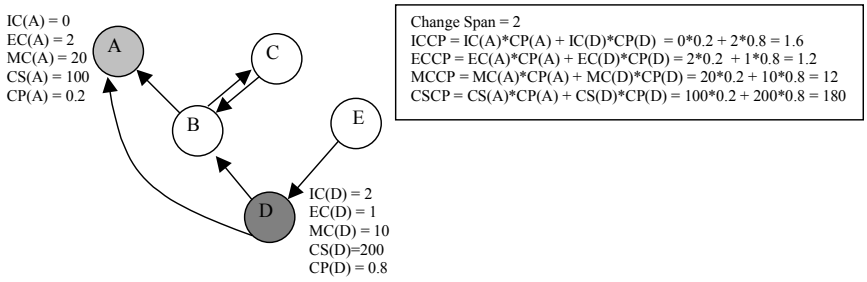

Fig. 1. Change complexity measures for a given change affecting classes *A* and *D*.

Figure 1 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*, *IC*, *EC* and *MC*) and the change profile (*CP*) for class *A* and *D* are provided together with the resulting change complexity measures *CSCP*, *ICCP*, *ECCP* and *MCCP*.

[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".*

Table 2. Summary of Proposed Change Complexity Measures

Name	Definition	Description
Change Size	$ChangeSize(c) = SLOCAdd(c) + SLOCDel(c)$	The number of source lines of code (SLOC) added to or deleted from class *c* for a given change to the software system.
Change Profile	$CP(c) = \frac{ChangeSize(c)}{\sum_{i=1}^{CC} ChangeSize(c_i)}$	The proportion of the total amount of changes done on class *c* for a given change to the software system. *CC* is the Class Count measure defined in Table 1.
Change Span	$ChangeSpan = \sum_{i=1}^{CC} \begin{cases} 0 \text{ if } ChangeSize(c_i) = 0 \\ 1 \text{ if } ChangeSize(c_i) > 0 \end{cases}$	The total number of classes modified in a given change to the software system.
Class Size Change Profile	$CSCP = \sum_{i=1}^{CC} CS(c_i) \times CP(c_i)$	Average class size weighted by the change profile for a given change to the software system. *CS* is the Class Size measure defined in Table 1.
Import Coupling Change Profile	$ICCP = \sum_{i=1}^{CC} IC(c_i) \times CP(c_i)$	Average 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 Table 1.
Export Coupling Change Profile	$ECCP = \sum_{i=1}^{CC} EC(c_i) \times CP(c_i)$	Average 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 Table 1.
Method Count Change Profile	$MCCP = \sum_{i=1}^{CC} MC(c_i) \times CP(c_i)$	Average method count weighted by the change profile for a given change to the software system. *MC* is the Method Count measure defined in Table 1.

2.4 Benchmarking

An intuitively appealing approach for the assessment of changeability decay is *benchmarking*. A given collection of "representative changes" *c* are implemented on different versions of the software *v1* and *v2*. The resulting change efforts *e1* and *e2,* respectively*,* are recorded. Hence, our operational definition of changeability decay 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 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.

2.4.1 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, in particular if the benchmark is implemented on different versions of the software system by the same developers within a short time period:

- *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 consecutive 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). 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 *v1* of the software system. All members of *g2* implement the benchmark on version *v2*. The individual effort required by each developer to implement the benchmark is recorded.

Step 4 (statistical analysis). The changeability of version *v2* is decayed with respect to version *v1* if the mean change effort for group *g2* is significantly larger than the mean change effort for group *g1*. 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 *v1* and *v2*. 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, $d = e(v2) - e(v1)$, for each developer.

2.4.2 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 are representative of *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 the 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, we have defined a data collection process to ensure that the developers

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

A data reporting tool has been implemented to support this process (Figure 2). The tool can also be used to collect actual benchmark results.

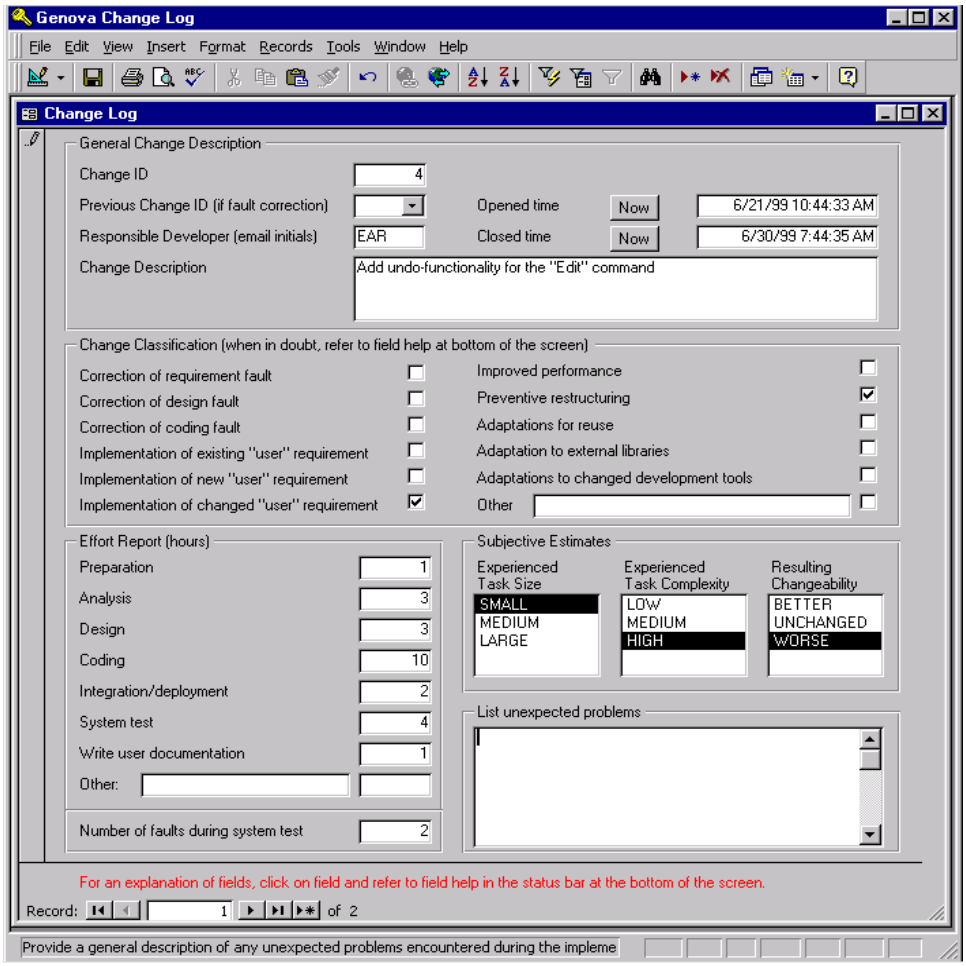

Fig. 2. The user interface of the change logger tool

3 Relationships between Changeability Decay Measures

The goal of our research is to measure changeability decay such that causes of decay can be identified and preventive guidelines developed. This section describes the relationships between the three approaches (structure measurement, change complexity measurement and benchmarking) to the measurement of changeability decay. Figure 3 depicts some important relationships between the empirical relational system and the proposed measurement approaches. These relationships are explained further, according to assessment accuracy (Section 3.1) and assessment cost (Section 3.2).

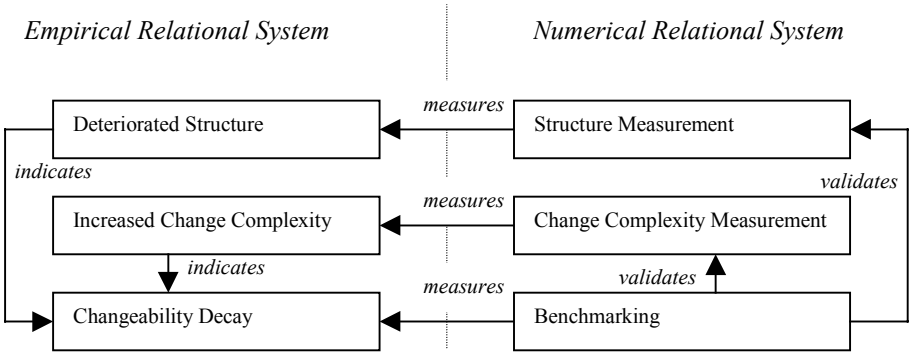

Fig. 3. Relationships between the empirical relational system and the assessment framework

3.1 Accuracy of the Measurement Approaches

We believe that benchmarking is the most accurate way to assess changeability decay. Unlike structure measurement and change complexity measurement, benchmarking does not rely on an underlying theory relating changeability decay to structural attributes of software; it attempts to measure changeability decay 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 2.4.2 will provide important insight for the composition of benchmarks.

Our hypothesis is that structure measurement can be used to *indicate* changeability decay. A common belief is that a deteriorated structure has a negative impact on the changeability of software. Structure measurement is intended to measure deteriorated 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 decay 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.

3.1.1 Validation Issues

Before structure measurement and change complexity measurement can be used as indicators of changeability decay, they must be validated. Furthermore, benchmarking does not provide much insight into the *cause* of decay. Increases in benchmark change effort may be due to, for example, inconsistent or outdated documentation, or a deteriorated structure. Thus, validating structure measurement and change complexity measurement using benchmarking results may provide further insight into the cause of the observed trend in benchmark change effort.

The validation can be performed by building, for example, regression models (Draper and Smith, 1981). To validate the structural attribute measures with regression, the response variable may be the differences in benchmark change effort. The structural attribute measures (Table 1) are used (either individually or in combination) as regressor variables. For example, using the *differences* in measurement values from *v1* to *v2* as regressor variables one obtains the following candidate regression model:

$$Effort_{v2,b} - Effort_{v1,b} = b0 + b1*(CC_{v2} - CC_{v1}) + b2*(SS_{v2} - SS_{v1}) + b3*(AvgMC_{v2} - AvgMC_{v1}) + b4*(AvgIC_{v2} - AvgIC_{v1}) + b5*(AvgCS_{v2} - AvgCS_{v1})$$

The corresponding example regression model for validation of change complexity measures (Table 2) is:

$$Effort_{v2,b} - Effort_{v1,b} = b0 + b1*(ChangeSpan_{v2,b} - ChangeSpan_{v1,b}) + b2*(CSCP_{v2,b} - CSCP_{v1,b}) + b3*(MCCP_{v2,b} - MCCP_{v1,b}) + b4*(ICCP_{v2,b} - ICCP_{v1,b}) + b5*(ECCP_{v2,b} - ECCP_{v1,b})$$

By evaluating the explanatory power of the regression models (using, for example, cross-validated R-square) one can determine to what extent the regressor variables explain the variation in differences in benchmark change effort. Unfortunately, interpreting software engineering data with regression models is far from trivial. Hence, alternative modeling techniques such as pattern recognition may be more appropriate in some circumstances (Briand et al., 1992; Jørgensen, 1995).

The benchmarking technique also needs to be validated. This involves an investigation of experimental errors. Such investigation would involve a meta-level experiment where benchmarking is evaluated using different benchmark experiment designs, varying the number of subjects, and varying the benchmark composition (e.g., the number, type and size of benchmark changes).

3.2 Cost of the Measurement Approaches

Benchmarking is by far the most expensive approach, as it requires implementation of changes by system 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.

4 Empirical Evaluation

The proposed assessment framework is currently being evaluated in four industrial case studies (Ericsson, Numerica-Taskon, Genera and Braathens) in Norway. These case studies differ in both size (12 person-months to 100 person-months) and application domains (large telecommunication application in Java, CASE-tool development projects in SmallTalk, Java and C++, and Web-application in Visual Basic/Java/HTML). Results from the development project for the Norwegian airline Braathens are reported below.

4.1 Description of the Case Study

The Braathens case study used an evolutionary development process called the Genova Process (Arisholm et al., 1998; Arisholm et al., 1999), which is quite similar to the Rational Unified Process (Kruchten and Royce, 1996). The development team consisted of four developers and an 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".[3] 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 module was collected based on weekly versions of the software through a 21-week period. After week 21, the system became operational. Three increments were delivered during these 21 weeks, at week 5, 10 and 21, 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. A more thorough analysis of the development process is provided in (Arisholm et al., 1999).

For the *IC* and *EC* coupling measures (Table 1), we implemented a parser for one of the import coupling measures called *OMMIC,* and for one export coupling measure called *OMMEC* (adapted from (Briand et al., 1997) to Visual Basic). Unfortunately, we encountered a problem with the collection of the export coupling measure 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 *OMMEC* export coupling measure collected from this module is inaccurate. Hence, data for export coupling is not reported in this paper.

4.2 Results

Table 3 shows the correlation between class-level Import Coupling (*IC*), Method Count (*MC*) and Class Size (*CS*), based on the operational software system from week 21. Although the correlation coefficients with *CS* are high, the variance of *IC* explained by *CS* is only 57% (*R-Sq=0.57*). Only 55% of the variance of *MC* is explained by *CS* (*R-Sq=0.55*). Hence, despite the definite correlation, one cannot claim that *IC* and *MC* both essentially capture size. In a similar comparison, less correlation (r=0.59) was found between the same import coupling measure and a similar size measure (Briand et al., 1999b).

[3] The system can be viewed at *http://www.braathens.no.*

Table 3. Pearson correlations between important 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 4 compares selected change complexity measures (*ICCP* and *MCCP*) with the corresponding structure measurements (*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. Structure measurement obviously does 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 complexity measurement may be useful for assessing trends in actual change complexity, whereas structure measurement may be useful for assessing trends in the overall structural properties of the software system.

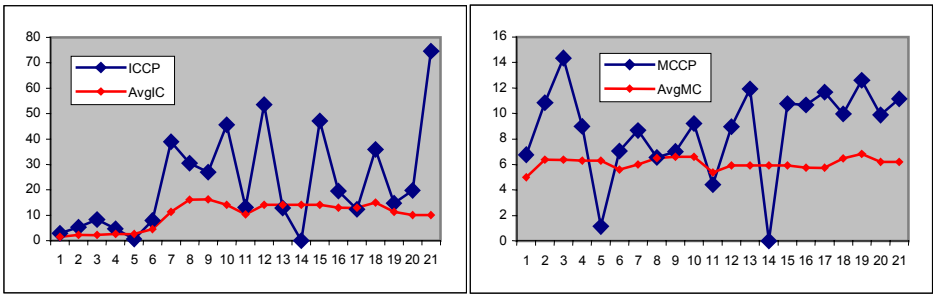

Fig. 4. Weekly plot of ICCP vs Avg. IC, and MCCP vs Avg. MC (no changes during week 14)

We still need to validate the structural attribute measures and the change complexity measures as indicators of changeability decay. To attempt validating the measures, we built regression models where the measures were candidate explanatory variables. A "productivity" measure was the response variable in the models. The productivity measure was calculated by dividing "change size" (measured in SLOC added + deleted) by reported effort data (SLOC added + deleted per person-hour). To obtain accurate productivity data, we had to reduce noise in change size caused by irregularities in file check-in times. Thus, we accumulated weekly changes until approximately 1000 SLOC had been changed (added or deleted). Then, we calculated productivity for the changes that occurred within each time span (e.g., from week 1 to week 4, and from week 5 to week 6). The resulting data is shown in Table 4.

In this case study, no statistically significant correlation was found between the structural attribute measures and the measured productivity, nor the change complexity measures and the measured productivity. The results suggest that, for this project, other factors were more important for the variation in the productivity measure than "structure" or "change complexity". Indeed, the main risks for the project were the incorporation of new and unfamiliar technology.

Table 4. Summary of process measures (Effort, Change Size, Productivity), structural attribute measures (SS, CC, Avg.IC, Avg.CS, Avg.MC) and change complexity measures (Change Span, ICCP, CSCP, MCCP)

Week #	1–4	5–6	7	8–10	11–12	13–18	19	20–21
Effort (hours)	209	132	55	120	83	227	65.5	108.5
Change Size	1057	1086	1150	1018	1473	1187	1556	987
Productivity	5.1	8.2	20.9	8.5	17.7	5.2	23.8	9.1
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
Change Span	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

Although we at present have been unable to validate the measures using regression models, we have still found interesting but inconclusive relationships between some of the change complexity measures and the productivity measure. Figure 5 shows an example of such a relationship. Values of the Import Coupling Change Profile (*ICCP*) and the corresponding structural attribute measure (*Avg. IC*) are plotted against the productivity measure. The resulting plot seems to indicate a positive correlation between *ICCP* and productivity during the main construction phase of the software (from week 1 to week 19). However, during the last three weeks before system delivery, the relationship changes. Interviews with the developers and examination of their time sheets reveal that weeks 19 to 21 correspond to the test phase. We have no further qualitative explanation of this irregularity. While there is a visible relationship between productivity and *ICCP*, there is no visual relationship between productivity and the import coupling measure *IC* (used in the structure measurement approach). This provides some very speculative support for our hypothesis that "change complexity measures" may be better indicators of changeability decay than the "structural attribute measures" (Section 3.1). Of course, the exploratory nature of this research cannot rule out that this apparent relationship between *ICCP* and productivity are not due to "shutgun correlations" (Courtney and Gustafson, 1993).

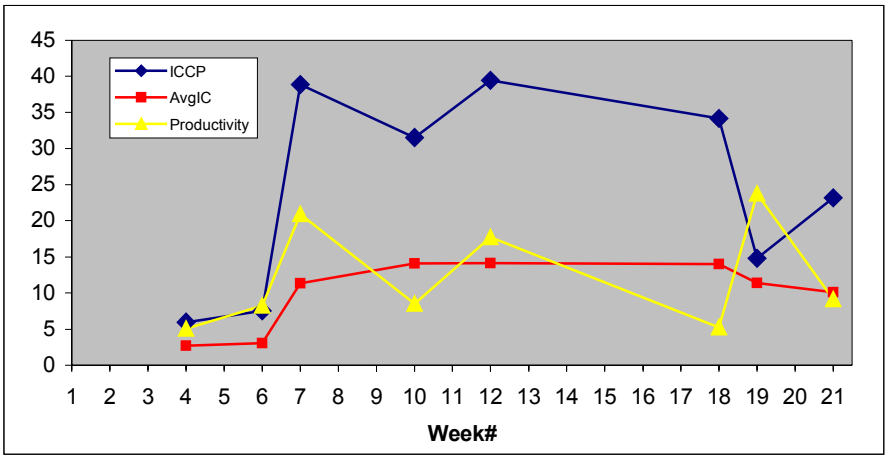

Fig. 5. Plot of productivity versus *ICCP* and *Avg.IC*[4]

The preliminary validation should be interpreted with caution because of the difficulties associated with the used validation method. The main problem with the validation method was the use of SLOC-based "productivity" to measure changeability. For example, it is well known that the number of source lines of code used to implement a certain function varies widely among individuals with different skill levels. Hence, using lines of code to obtain the productivity measure may be inappropriate. An alternative measure of change size is function points (Albrecht and Gaffney, 1983; Symons, 1988). However, using function points to estimate the size of small changes to software may be impractical (Jørgensen, 1995).

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 sources of measurement noise caused by:

- The need to normalize change effort (resulting in a productivity measure of questionable validity – as described in the previous paragraph); the validation model could, for example, have used the response variable described in Section 3.1.1.
- 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 Section 2.4).
- Differences in individual skill levels of the developers (since the benchmarking experimental design may control for individual ability – as described in Section 2.4.1).

[4] Note that the measurement scales on the Y-axis in Figure 5 are different for the different measures. The plot only intends to illustrate a co-variation between *ICCP* and productivity.

5 Conclusions and Future Work

This paper proposed a framework for empirical assessment of changeability decay. We defined changeability decay and identified three approaches to empirical assessment. In the preliminary case study described in this paper, change complexity measurement was empirically evaluated against structure measurement. The results indicate that change complexity measurement may account for some dimensions of the changeability of object-oriented software not provided with the structure measurement approach. However, both approaches are based on the hypothesis that structural attributes of object-oriented software significantly affect changeability. In this case study, neither structural attribute measures nor the proposed measures of "change complexity" seem to explain the variation in measured productivity. However, we believe the validation of the measures would be more reliable if effort and product data were collected either from logical changes (e.g., "implement password encryption for the login screen") or preferably from benchmark changes, rather than from weekly changes. Consequently, further investigation and development of the approaches are required.

At present, research is underway to evaluate the approaches through controlled experiments as well as case studies in different application domains, system sizes and programming languages. The code parsers needed to collect the structural attribute measures and the change complexity measures have been implemented for Java and SmallTalk. For Java, the parser collects the static import and export coupling measures described in (Briand et al., 1997), adapted to Java. Work is in progress to implement a similar tool for C++. For SmallTalk, we have implemented *dynamic* coupling measures by intercepting messages sent between objects at run-time since it is not possible to parse the SmallTalk programming language to obtain accurate static coupling. At present, the dynamic coupling parser is being tested on 10 versions of a CASE tool, which supports the OORAM software engineering method (Reenskaug et al., 1995). This SmallTalk software system consists of more than 1000 classes.

We are also doing the final preparations for a benchmarking experiment involving graduate students at University of Oslo. The purpose of the benchmarking experiment is to

- evaluate the proposed experimental design (Section 2.4.1),
- gain practical experience with the composition of benchmarks (Section 2.4.2), and
- provide a controlled environment for validation of the structural attribute measures and the change complexity measures (Section 3.1.1).

In addition to that experiment, we are continuing the data collection in several industrial case studies. In one of them, we analyze changes performed on a medium sized CASE tool called Genova (Arisholm et al., 1998). Genova is written in Java and C++, and is significantly larger than the Visual Basic system analyzed in Section 4. At present, approximately 10 developers are involved in the project. They use the change logger tool and the data collection process described in Section 2.4.2. So far, 34 logical changes to the Genova CASE tool have been recorded. These changes are then traced in the configuration management system (ClearCase) to collect structural attribute measures and change complexity measures, which in turn will be validated using the reported change effort data.

Acknowledgements

We thank Magne Jørgensen, Letizia Jaccheri, Harvey Siy, Todd Graves, Ray Welland, Frank Houdek, the ESSDE'99 workshop participants and the reviewers for providing valuable comments for improving contents and structure of this paper. We gratefully acknowledge the support from our industrial partners, Erik Amundrud, Jon Skandsen and Stein Grimstad at Genera AS, Lasse Bjerde and Anne-Lise Skaar at Numerica-Taskon AS and Pål Berg at Ericsson ETO/IR. The research project is funded by The Research Council of Norway through the industry-project SPIQ (Software Process Improvement for better Quality).

References

Albrecht, A.J. and Gaffney, J.E. (1983). Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, 639–648.

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., Skandsen, J., Sagli, K. and Sjøberg, D.I.K. (1999). Improving an Evolutionary Development Process – A Case Study. In: *Proceedings of the EuroSPI'99 Conference (European Software Process Improvement),* Pori, Finland, pp. 9.40–9.50.

Basili, V.R., Briand, L.C. and Melo, W.L. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, 751–761.

Boehm, B.W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, Vol. 21, No. 5, 61–72.

Boehm, B.W., Gray, T.E. and Seewaldt , T. (1984). Prototyping versus Specifying – A Multiproject Experiment. *IEEE Transactions on Software Engineering*, Vol. 10, No. 3, 290–302.

Briand, L.C., Arisholm, E., Counsell, S., Houdek, F. and Thevenod, P. (1999a). Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of The Art and Future Directions. *To be published in Empirical Software Engineering*.

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*, Vol. 18, No. 11, 931–942.

Briand, L.C., Daly, J.W., Porter, V. and Wust, J. (1999b). A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems. *To be published in Journal of Systems and Software*.

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*, Vol. 25, No. 1, 91–121.

Briand, L.C., Devanbu, P. and Melo, W.L. (1997). An Investigation into Coupling Measures for C++. In: *19th International Conference on Software Engineering (ICSE'97),* Boston, USA, pp. 412–421.

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, USA, pp. 345–354.

Chidamber, S.R. and Kemerer, C.F. (1994). A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, 476–493.

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

Cotton, T. (1996). Evolutionary Fusion: A Customer-Oriented Incremental Life-Cycle for Fusion. *Hewlett-Packard Journal*.

Courtney, R.E. and Gustafson, D.A. (1993). Shutgun correlations in software measure. *Software Engineering Journal*, January, 5–13.

Draper, N.R. and Smith, H. (1981). *Applied Regression Analysis*. John Wiley & Sons, Inc.

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. *To be published in IEEE Transactions on Software Engineering*.

Fenton, N. (1992). When a software measure is not a measure. *Software Engineering Journal*, September, 357–362.

Fenton, N. (1994). Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, Vol. 20, No. 3, 199–206.

Gilb, T. (1988). *Principles of Software Engineering Management*. Addison-Wesley.

Harrison, R., Counsell, S.J. and Reuben, V.N. (1998). An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, 491–496.

Jørgensen, M. (1995). Experience With the Accuracy of Software Maintenance Task Effort Prediction Models. *IEEE Transactions on Software Engineering*, Vol. 21, No. 8, 674–681.

Jørgensen, M., Bygdås, S.S. and Lunde, T. (1995). Efficiency Evaluation of CASE Tools – Methods and Results. TF R 38/95, Telenor FoU.

Kruchten, P. and Royce, W. (1996). A Rational Development Process. *CrossTalk*, Vol. 9, No. 7, 11–16.

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*, Vol. 23, No. 2, 111–122.

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, USA, pp. 1–9.

Sjøberg, D.I.K., Welland, R. and Atkinson, M.P. (1997a). Software Constraints for Large Application Systems. *The Computer Journal*, Vol. 40, No. 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 (NIK'96),* Alta, Norway, 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*, Vol. 27, No. 4, 447–480.

Symons, C.R. (1988). Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering*, Vol. 14, No. 1, 1–10.

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.
Zuse, H. (1991). *Software Complexity: Measures and Methods*. de Gruyter.

Biography

Erik Arisholm received a bachelor's degree in computer science from University of Oslo and a M.A.Sc. degree in electrical engineering from University of Toronto. He has seven years industry experience in Canada and Norway as a Lead Engineer and Design Manager. He is currently a PhD candidate in the research group Industrial System Development, Department of Informatics, University of Oslo. His research interests include empirical software engineering, software process improvement and object-oriented methods.

Dag Sjøberg received an MSc degree in computer science from University of Oslo in 1987 and a PhD degree in computing science from University of Glasgow in 1993. He has five years industry experience as consultant and Group Leader. He is now a Professor in software engineering and is the leader of the research group Industrial System Development in the Department of Informatics, University of Oslo. Among his research interests are software evolution, software process improvement, programming environments, object-oriented methods and persistent programming.