UNIVERSITY OF OSLO
Department of informatics

Measuring change – Creating and validating a tool for extracting change-level measures from version control systems.

Master thesis
30 credits

Simon R. Andresen

18th December 2006

Abstract

An underlying assumption for this thesis is that quantitative assessment of the ability of a software project to perform software changes can be a valuable complement to qualitative methods for evaluating and improving project processes and products. A software change is a coherent and independent unit of work that improves a functional or non-functional quality of a software system, and may affect one or more system modules. By measuring properties of software changes over time, software projects may improve their ability to identify what influences the cost and quality of the changes, giving them better cost estimates and helping them identify ways to improve

The goal of this thesis was to provide such information by identifying measures of software change that are possibly related to change-level effort and quality, provide exact definitions of the measures, to create a tool to extract the measures, and finally, to validate the measures using available empirical maintenance effort data. Many software projects today use some kind of issue or bug tracking tool to manage changes that need to be applied to a system. Likewise, some sort of version control system (VCS) is often used to keep track of the actual modifications to the body of software itself. In this thesis, we focus on retrieving change-level measurements from information resident in such tools.

We reviewed literature on measures that have been conjectured to correlate with effort and quality of changes, as well as literature on how such changes could be extracted. Based on this, we created a UML based domain model that provided a basis for consistent and precise definitions of change measures. The same model constituted the basis for the design of a measurement tool, the VCSMiner, which can be used by researchers or practitioners to extract change level measures. The tool extracts measurements of size and complexity of changes, as well as measurements of the experience of the developers implementing it

The correctness of the measurements provided by the tool was then verified by comparing them to that of similar measurements provided by manual extractions and existing scripts. The measurements were validated by conducting a correlation analysis to see if they correlated to the cost of change. Identification of possibly redundant measures were performed by investigating the internal correlations between the measures provided by the tool. The usefulness of such measurements was evaluated in a group discussion with the developers in the projects providing the empirical data.

We conclude that quantitative, change-level measurement can indeed prove useful for software projects, as well as research into the evolution of software. The creation of an extendible, VCS-impendent tool demonstrated that the extraction of measurements can be fully automated, which is essential for resource constrained software projects. In the future, the tool created as part of the thesis can be extended with regards to available measurements, and possibly becoming part of a larger framework for automated analysis and presentation.

Acknowledgements

First, I need to thank my supervisors: My main supervisor Bente Anda; her patience, guidance, good advice and feedback have helped me through this process. And my co-supervisor Hans Christian Benestad; for letting me use the results of his case study on effort and his scripts to validate and verify my results and for discussing various aspects of the work with me. His helpfulness and ability to communicate difficult concepts in an understandable manner has been invaluable.

I would also like to thank my family; they have been very supportive and helpful. I would especially like to thank my wonderful mother, Trine Røyrvik, who has just turned 50. I owe her a great deal and I love her dearly.

Oslo, December 2006
Simon R. Andresen

ABSTRACT **3**

ACKNOWLEDGEMENTS **5**

1. INTRODUCTION **9**

1.1 MOTIVATION 9
1.2 OBJECTIVES 11
1.3 RESEARCH CONTEXT AND EMPIRICAL DATA 11
1.4 STRUCTURE OF THESIS 11

2. METHODOLOGY **12**

3. RELATED WORK **13**

4. MEASURING CHANGE **14**

4.1 CHANGE MEASURES 14
4.2 USEFULNESS OF MEASUREMENTS 16
4.3 IDENTIFYING THE MEASURES 16
4.3.1 Definition of Change 16
4.3.2 The change measures 17

5. THE CHANGE MODEL **20**

5.1 PURPOSE AND TAXONOMY 20
5.2 THE MODEL 21
5.3 THE ACTORS 22
5.4 THE ENTITIES 23

6 THE VERSION CONTROL SYSTEM MINER **29**

6.1 ASSUMPTIONS 31
6.1.1 Implementation of other VCS 31
6.1.2 Remote Connections 31
6.1.3 Runtime Environment 31
6.2 DESIGN 31
6.3 IMPLEMENTATION OF MEASURES 33
6.3.1 Extracting Data from CVS 33
6.3.1.1 Data provided by CVS log: 34
6.3.1.2 Data provided by diff: 35
6.3.1.4 Parsing and capturing the data 36
6.3.3 AST representations of code 36
6.3.4 Size comparison (difference) 36
6.3.4 SSD: Getting Measurements 37
6.4 IMPLEMENTING SUPPORT FOR SVN 39

7. VERIFICATION AND VALIDATION **40**

7.1 VERIFICATION OF MINER 40
7.1.1 Correctness 40
7.1.2 OS Independence 40
7.1.3 VCS Independence 40
7.2 VALIDITY OF MEASURES 41
7.2.1 Correlation analysis 41
7.2.1.1 Correlation to Effort 43
7.2.1.2 Correlation between Measurements 44
7.3 USEFULNESS 45

8. DISCUSSION **46**

8.1 THE CHANGE MEASURES 46
8.2 THE VCSMINER 47
8.2.1 AST-based measures 47
8.2.2 CVSParser 47
8.2.3 Measurements on Multiple Changes 47

9. CONCLUSIONS AND FURTHER WORK **48**

REFERENCES 51

APPENDIX 55

ABBREVIATIONS, TERMS AND ACRONYMS 55
ATTACHMENT 1: CLASS DIAGRAM OF DOMAIN LAYER WITH ATTRIBUTES AND METHODS.......... 57
ATTACHMENT 2: CORRELATION BETWEEN MEASUREMENTS.......... 61

TABLE 1: SIZE MEASURES 17
TABLE 2: STRUCTURAL MEASURES 18
TABLE 3: ARCHITECTURAL MEASURES 19
TABLE 4: AUTHOR-BASED MEASURES 19
TABLE 5: MEASUREMENTS OF CHANGEREQUEST 24
TABLE 6: MEASUREMENTS OF CHANGEFILE 27
TABLE 7: MEASUREMENTS OF REVISION 28
TABLE 8: RESULTS OF CORRELATION ANALYSIS.......... 43

FIGURE 1: THE BASIC APPROACH TO BOTH EFFORT AND QUALITY PREDICTION, FENTON AND NIEL [8] P. 363 10
FIGURE 2: SOFTWARE METRICS CLASSIFICATION TABLE, FENTON AND NIEL [8] P. 361 14
FIGURE 3: MODIFICATION UNAWARE-METRICS, GERMAN AND HINDLE [2] P. 4 15
FIGURE 4: MODIFICATION AWARE-METRICS, GERMAN AND HINDLE [2] P. 5.......... 16
FIGURE 5: THE CHANGE MODEL 21
FIGURE 6: THE AUTHORS IMPLEMENTING A CHANGE REQUEST 25
FIGURE 8: VCSM CLASS DIAGRAM WITHOUT ATTRIBUTES AND METHODS SHOWING 32
FIGURE 9: SYSTEM SEQUENCE DIAGRAM: DOMEASURES().......... 38
FIGURE 10: CURVE RESULTING OF NORMAL DISTRIBUTION OF VARIABLES.......... 41
FIGURE 11: DISTRIBUTION OF DATA FOR EFFORT 42
FIGURE 12: DISTRIBUTION OF DATA FOR ADDED LOC 42
FIGURE 13: RETROSPECTIVE STEPS AS PART OF AN ITERATIVE CYCLE. LARSEN AND DERBY [5] P. 6.......... 45

1. Introduction

This section introduces the thesis with respect to motivation, context and objectives.

1.1 Motivation

Software development can be seen as a set of changes to a body of software, this is true for initial development of new projects, where the body of software is empty, as well as during evolution and maintenance of projects where code and documentation already exists [1]. Many types of changes can be made to a system such as introduction of new features, bug fixes and modifications such as restructuring to improve performance, reliability, changeability or other non-functional qualities [1]. We define a software change to be a coherent and independent unit of work that improves some functional or non-functional property of the software system.

There are several reasons as to why the properties of software change are of interest. First, it can help software practitioners understand the nature of the work being done on a system, and which direction a project is taking. Second, it can help them identify cost and quality drivers, which in turn may be monitored for the purpose of improving cost estimates, and for identifying needs for process or product improvements. It can also help researchers understand the nature of software evolution from a general perspective [2].

Several authors have noted the importance of understanding software change. Lehman and Ramil [3] points out that the effort required for understanding, changing, adding, deleting and replacing source code usually is a significant fraction of the total software evolutionary effort. They also emphasize the usefulness of being able to estimate the effort needed to perform software evolutionary tasks. Fluri and Gall [4] argue that some of the negative effects of aging in software can be combated by setting change in the centre of the development process. They argue that understanding the nature of fine-grained changes to source code is essential to understand how software evolves over time.

We believe that quantitative assessment of the ability of a software project to perform software evolutionary tasks is a valuable complement to qualitative methods. For example, the goal of “Agile Retrospectives”[5] is to:

…reflect at the end of every increment and identify changes and improvements that will increase the quality of the product and the work life of team members.

- “Agile Retrospectives: Making Good Teams Great” Derby and Larsen [5], p. 16

However, if measurements are to be used to give insights and provide decision support for practitioners, extraction of data and measurements about change needs to be made easier and appropriate measures need to be identified.

The goal of this work was therefore to create a tool for extracting measurements that can be used by software practitioners to assess their software evolutionary tasks. Mockus and German [6] proposed, on the basis of their experience with analyzing numerous open source and commercial projects, the development of tools for the extraction and validation of project data. The argued that such tools would streamline empirical investigation, facilitate the testing of new theories and give us better insights into the projects.

Our focus is therefore on the following: First, we develop a domain model in order to structure and understand change-related concepts. Based on the model, we identify feasible change measures. Then, based on this model, we create a tool to extract measurements. The measures rely on data available in an extended change management system (ECMS). An ECMS consists of a bug/issue tracking tool, e.g., JIRA[7], and a version control system (VCS), e.g., CVS and Subversion. We focus on identifying change measures that are hypothesized to correlate with change effort and change quality.

Developer effort is usually considered the most influential factor when determining the cost of evolutionary tasks performed on software systems [3, 8, 9] and formulas such as

effort = f(changesize)

has been the traditional way of estimating effort. By identifying other factors influencing change effort, such models could become more accurate, e.g.

effort=f(change size, change complexity, changed system complexity, skill)

Size along with measurements for complexity has also traditionally been used to make predictions about the quality[8]. The following model illustrates this:

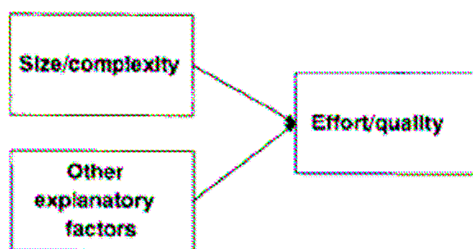


Figure 1: The basic approach to both effort and quality prediction, Fenton and Niel [8] p. 363

According to Fenton and Niel, quality predictions has usually been based on estimates in the form of

f(complexity metric) = defect density (defects pr thousand lines of code)

We validated the measures by analysing them with regards to how much they correlate to the effort of change in an industrial case study. We also verified the correctness of results extracted by our measurement tool (which we call "VCSMiner") by performing semi-automated measurement implemented in the research group, and then investigating whether the results were similar. Finally, to investigate whether assessments based on these measurements were useful for identifying cost drivers and assessing the ability of a software project to apply changes in a real world context, we discussed their use with the members of the development team in the case study.

1.2 Objectives

The main objective for this thesis is to develop a VCS-independent tool for extracting measurements about change. This objective is divided into three sub-objectives:

1. The creation of a domain model for identifying measures that are candidates for correlating with change-level effort and quality, concentrating on measures of change size and change complexity. Size and complexity of existing code is not considered.

2. Develop a VCS-independent tool for extracting change-level measurements.

3. Validate the measures and the tool using real world data.

1.3 Research Context and empirical data

This thesis is part of a research initiative at Simula Research Laboratory that aims at providing industrial software projects with concrete guidelines on how to conduct in-process measurement of maintenance performance. As part of this, empirical data is available for validation purpose: The developers in an ongoing project are asked to comment modifications submitted into the VCS with an identifier corresponding to the change as it is identified in the bug/issue tracking tool. This is a common heuristic for tracing changes from change-requests in a tracking tool down to a VCS [10]. Furthermore, the developers record the effort expended on each change. The researchers in this project then tries to related change measures to change effort, by techniques such as statistical regression, AI-methods and analogy based reasoning.

1.4 Structure of thesis

In the following section (section 2), we will describe the methodology used when working on this thesis. The next section, section 3, will briefly describe similar and related work we have reviewed during our research. In Section 4 we first describe how change can be measured in general, and then specifically which measurements we chose and why. Section 5 describes the model that was created to get a better understanding of changes in the ECMS domain. Section 6 describes how this understanding was used to create a tool for extracting measurements, and how this tool was designed and implemented. Section 7 describes how the correctness of the tool was verified, and how we validated the measurements with respect to usefulness. Section 8 discusses our findings and experiences. Section 9 presents our conclusions as well as our vision for further work and improvement of model, tool and measures. References are located at the end of the thesis, followed by appendices containing an explanation of terms and abbreviations as well and more detailed results of the analysis conducted as part of the validation.

2. Methodology

The work performed as part of this thesis can be split into four stages:

1. Review into related literature to identify measures that have been hypothesized to correlate with effort and quality, as well literature on methods for developing a tool for extracting measurements.
2. Definition of the change domain, by development of a model of changes in an ECMS framework
3. Development of tool for extracting data and performing measurements.
4. Validation of appropriateness of measurements and verification of correctness of measurements extracted by the developed tool.

Review

In order to identify measures that have been hypothesized to correlate with quality and effort of change, research papers and appropriate literature within the field was reviewed. Likewise, to find out what kind of data could be extracted from the VCS, and how the data could be extracted, literature and research papers on this topic were examined. In examining research papers, the date of submission, number of citations by other authors, the authority of the authors within the field (based on previous work) and where the paper was published was considered, as well as appropriateness of content and findings for our objectives.

Definition of the change domain

To model how change in an ECMS framework can be structured and how and where measurements can be extracted, related attempts at visualizing change found as part of the literature review was studied and improved upon. A UML modelling tool, "Poseidon"[11], was then used to model the ECMS domain as we defined it.

Development

The development of the VCSMiner was done based on the results of the review and the created model. The implementation was done on the basis of previous development experience as well as literature on programming [12, 13]. The program was developed using the development environments provided by Eclipse (Eclipse) and Netbeans (Netbeans).

Validation and verification

To verify the correctness of the results, a comparison to semi-manual scripts by another author was performed, a so called *cleanroom* approach [6]. To validate the appropriateness of the measurements, a statistical correlation analysis was conducted on data from the industrial change study described in section 1.3 , investigating the correlation between each change measure and actual effort. Finally, to validate the usefulness of such measurements in a real world context, the results were discussed with the developers in the case study.

3. Related Work

Mockus and Weiss [1] attempted to use measures on the properties of change to create a framework for predicting the probability of failure for software updates. Although they were interested in larger, more wide reaching changes that make up software updates, their measures on the properties of change has been the basis for many of our candidate measures. However we have attempted to improve upon them and increase their level of detail, for instance: While Mockus and Weiss measure size by looking at added and deleted lines of code, we expand upon this to include fresh/new lines of code and modified lines of code.

German [14] extracted information about the modifications to files that were part of the same transaction to CVS by a specific author. His approach differs from ours in that he is grouping together modifications done as part of the same transaction calling them Modification Records (MRs), while we are grouping together modifications identified by an identifier in the comments as belonging to a record in the bug/issue tracking tool (which we identify as Change Records). His focus is on identifying and categorising the modifications part of an MR into types according to their nature. For example codeMR for code modifications, this in turn could consist of commentMR for modifications to commentation in code, and bugMR that are modifications part of a bug fix and so on. These were then used to identify how many MRs had been done and of which types to systems and how many files they touched.

Mockus and German [6] developed a tool for extracting change information from CVS as well as mailing lists and the bug tracking tool "Bugzilla". However, they did not trace changes in a ECMS perspective and down to a VCS by use of comments. Instead they grouped changes based on time interval of modifications and author, in the same manner as German [14]. Judging by their description their tool is tied to a specific VCS technology (CVS). In addition, the measurements they present are of a coarser granularity and have a different scope than ours.

Fluri and Gall [4] created a tool for extracting measurements about changes, based on changes to the abstract syntax trees of involved source code. An abstract syntax tree is the structure of the syntax used in code represented in a tree format (with nodes and leaf nodes). As the syntax and structure is different for each programming language, this can be seen as a language and structural specific approach, and their tool concentrated on the structure of java code. They also provided a taxonomy for such measurements and proposed a weighting system for the level of impact changes on specific parts of the AST has on other entities in the code. They focus on CVS as VCS technology and suggest very fine-grained measures of code change. They did not perform measurement at the logical change level, concerning themselves only with modifications on individual classes.

Mierle et al. [15] mined CVS student repositories for academic performance indicators, which has nothing to do with measuring software changes, but gives insights into data extraction from CVS.

Girba et al. [16] extracted data from a CVS to do measurements based on a notion of code ownership for use in ownership maps, to visualize which author interacted with what part of the system, and how.

Canfora and Cerulo [10] traced changes in an ECMS framework, represented by the bug tracking tool Bugzilla and CVS. They did this by tracing change requests identifiers in Bugzilla down to the comments recorded in CVS. They attempted to predict the impact of a change

request in the system before it is implemented. Their level of granularity was coarser than ours; their scope being almost exclusively on the number of files affected by a change.

4. Measuring Change

A common approach to answering questions and make predictions about external attributes of a system, like quality and complexity, is to collect data about internal attributes (such as size, structure and modularity). To collect data about the internal attributes in an attempt to assess size and complexity of a change we need to use *change measures*.

4.1 Change Measures

To illustrate the relation between internal and external system attributes Fenton and Neil [8] provides a table for classification of software metrics. They use s*oftware metrics* as a collective term for the wide range of measuring activities in software engineering. This table divides the attributes that any software metric is trying to predict into two categories: external and internal. In this way a distinction between the attributes we are interested in knowing or making predictions about (usually external) and the ones we can control and measure directly (internal) is made clear.

ENTITIES	ATTRIBUTES	
	Internal	***External***
Products		
Specifications	size, reuse, modularity, redundancy, functionality, syntactic correctness, ...	comprehensibility, maintainability, ...
Designs	size, reuse, modularity, coupling, cohesiveness, inheritance, functionality, ...	quality, complexity, maintainability, ...
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ...	reliability, usability, maintainability, reusability
Test data	size, coverage level, ...	quality, reusability, ...
...	...	...
Processes		
Constructing specification	time, effort, number of requirements changes, ...	quality, cost, stability, ...
Detailed design	time, effort, number of specification faults found, ...	cost, cost-effectiveness, ...
Testing	time, effort, number of coding faults found, ...	cost, cost-effectiveness, stability, ...
...	...	...
Resources		
Personnel	age, price, ...	productivity, experience, intelligence, ...
Teams	size, communication level, structuredness, ...	productivity, quality, ...
Organisations	size, ISO Certification, CMM level	Maturity, profitability, ...
Software	price, size, ...	usability, reliability, ...
Hardware	price, speed, memory size, ...	reliability, ...
Offices	size, temperature, light, ...	comfort, quality, ...
...	...	...

Figure 2: Software Metrics Classification Table, Fenton and Niel [8] p. 361

Fenton and Niel [8] points out that there are thousands of software metrics, but that the rationale for almost all these metrics have been motivated by two activities:

1. The desire to assess or predict effort/cost of development processes.
2. The desire to assess or predict quality of software products.

The assumption that leads us to be interested in the properties of *changes* is that measurements extracted about each and every change should give us a finer level of granularity than if we were to, for instance, compare metrics from system releases of a project. Our focus is therefore on identifying useful measures of software change.

German and Hindle[2] provides a classification of software change measures. They provide the following definition:

We define a change metric as a metric that can be used to measure how much a software system has been modified between two versions of it.
- German and Hindle [2], p. 1

By this definition most software metrics can be modified to be used as change metrics. One could for instance measure the Lines of Code (LOC), which is the first, simplest and most used software metric, of a system before and after a change. However, this does not make them *meaningful* as change metrics: A LOC count before and after a change could be the same simply because the programmer deleted a number of lines and then added the same number of lines of new code. Such a measurement would then give a false impression about the size of the change. Measures like these are what German and Hindle [2] identify as being *modification-unaware*.

According to German and Hindle [2] every type of change metric depends on a *delta function* (a function for calculating the quantity of change) which takes as parameter two or more versions of an entity, and uses that to compute a metric value. A change metric is modification unaware when the delta function is defined in terms of a metric designed to measure only a single version of a given entity, like in our LOC example. These metrics are computed by independently analyzing an entity's versions, never directly comparing one to another.

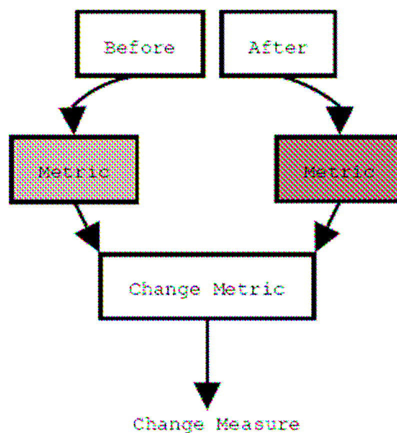


Figure 3: Modification unaware-metrics, German and Hindle [2] p. 4

In contrast to this kind of change metrics there are those that are *modification aware:* Here the delta function is aware of the versions of the entity to be measured, and the function is computed by inspecting and comparing these versions:

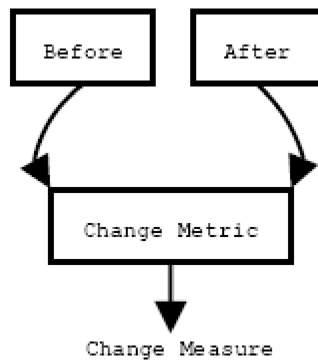


Figure 4: Modification aware-metrics, German and Hindle [2] p. 5

In other words, modification aware metrics measure the actual change. Lines added and removed as part of a modification to a file are examples of modification aware metrics. Needless to say, modification aware metrics are more useful than unaware when we want to extract the quantifiable properties of the changes themselves.

Note that the definition of change metric provided by German is somewhat narrow for some purposes, since it focuses primarily on size of change. It is possible to construct change-level measures that also capture aspects of affected code, developers performing the change, and the technological environment of the change.

4.2 Usefulness of Measurements

As mentioned in section 1.1, information about how a system evolves represented by the changes applied to it could provide valuable understanding and decision support for practitioners of software development. However, is should be noted that measures such as size and complexity of changes cannot be *directly* linked to predictions about attributes such as defects and quality. These kinds of measurements can be used as inputs to causal models which takes other factors into account as well. Ideally, the models should be constructed on the basis of historical data of effort and quality from the software organization in question.

4.3 Identifying the Measures

4.3.1 Definition of Change

We define change from a different level than a single modification to a file. We view a change as all such modifications that make up the implementation of a software change to the system as a whole. A change can be a bug fix, a new feature or other corrective actions and improvements. It may consist of everything from a single to many modifications on code and can span from one to several modules or subsystems. Formally, we define a software change to be a coherent and independent unit of work that improves a functional or non-functional quality of a software system.

4.3.2 The change measures

Size

As was mentioned in the introduction, size is the most common measure used to predict effort and quality [1, 8]. Lines of code (LOC) are the oldest and most used size measure. Comparing LOC before and after a change is not a very good change size metric, as mentioned in 4.2, since such metrics are modification unaware. Instead, we propose several different kinds of modification aware LOC metrics, where the delta functions takes as parameters the previous version of a file as well as the changed version. Our size measures are as follows:

Table 1: Size Measures

Measure	Target Attribute	Definition
Added LOC	Size	Added lines of code. Lines of code that are not present in the previous version but is introduced by the modifications part of a software change
Deleted LOC	Size	Deleted lines of code. Lines of code that are in the previous version but not in the version that is a result of a change.
Changed LOC	Size	Modified lines of code. These lines of code are in both the version that is the result of a change and the previous version but have been modified in some way.
Fresh LOC	Size	New lines of code. Lines of code that are part of a new file in its first initial version.

These measurements were adopted from the size measurements used by Mockus and Weiss [1], however, unlike Mockus and Weiss we differentiated between added lines of code and new lines of code to see if these measurements were differently related to effort. There were indications of this being the case in a correlation analysis performed by Ramil and Lehmann [9], their results indicated that there is a higher level of correlation to effort for the number of changed modules as predictors (the level of granularity of their measurements are much coarser than ours) than the number of new modules.

In addition, we explicitly separated out changed lines, instead of treating a change as one deletion and one addition. The advantage of using measures that are more specific is that by being able to weigh their importance individually, they are, in combination, likely to provide equal or better correlation with observed change effort or quality.

Structure

To be able to say something about the complexity of the change, and provide higher detail about the structure of the code modified to implement a software change, we propose the following modification aware change metrics on the structure of code affected by the change:

Table 2: Structural Measures

Measure (Added & Deleted)	Target Attribute	Definition
Private and public class variables	Complexity	These are variables that are owned by a class. We distinguish between those that are private to it and those that are externally accessible (also known as properties)
Private and public methods	Complexity	Methods declared in a class, differentiating between those that are private to the class itself or externally accessible.
Internal method calls	Complexity	A call to a method which are part of the class itself.
External method calls	Complexity	A call to or invocation of a method that does not belong to the class.
Local variables	Complexity	Variables that are not owned by the class itself but are declared and used inside the class where needed (as part of an algorithm or method).

We have not identified many attempts to measures change complexity beyond size in the research literature. However, in a recent paper Fluri and Gall [4] suggests an approach to better understand the impact of changes, based on analyzing changes to AST (Abstract Syntax Trees).

AST is a way to view code structure. The concrete syntax is the actual written code found in a file, while its abstract syntax is a representation of what the concrete syntax means. An AST is a representation of the abstract syntax mapped to a tree form. By traversing the tree representations of code before and after a change, we can pinpoint the exact structural changes by identifying which nodes of the tree have been modified and how. Fluri and Gall [4] points out that previous work has shown that such classification of source code changes is needed to increase change impact awareness.

The assumption is that this should give us a better indication of the complexity of a change than size measurements. These measurements can be considered as more experimental than the others as not much work were found about measurements on this level of detail except the work presented in the recently published paper by Fluri and Gall[4].

The correlation analysis as part of validation of measurements would provide more information about whether such measures seem to be useful for effort and quality assessment beyond that of size.

Architecture

In order to measure diffusion of a change, i.e. how widely spread the changes are in the system, and to identify modules and files that are often modified together, we have chosen to use several architectural measures.

These measures are not change metrics according to German and Hindle's [2] somewhat narrow definition, but we assume that they tell will us something about the properties of change that goes beyond size and structure of the change itself , and may therefore be useful.

Table 3: Architectural Measures

Measure	Target Attribute	Definition
Number of affected files	Diffusion	The number of files that have been affected by the implementation of a software change.
Name of module(s)	Diffusion	The name of the module(s) that have been affected by the change. This provides us with measures on a nominal scale, i.e. a list of names rather than numerical values [17].
Number of file types	Diffusion - Complexity	The number of different types of files that have been affected by the software change. The assumption is that this will correlate with the number of implementation technologies spanned by the change
List of file types	Diffusion – Complexity	Lists the file types spanned by the software change. The result is a list on a nominal scale.

The first two measures are based on Mockus & Weiss' [1] measures of diffusion and coupling of a change. However we have introduced measures of the different file types affected. The assumption is that this is indicative of the complexity of the change. Systems often use more than one technology (such as java, c++, sql scripts, xml and so on), and the more technologies the change span across, the more complex we assume it is.

Author

A factor that may influence change effort or quality is the experience of authors with respect to the software code affected by the change. Another human-related factor of interest is the number of authors involved in implementing the change, as this may be indicative of a difficult change, needing specialized knowledge to implement. Like the architectural measures, these go beyond German and Hindle's [2] definition of change metrics but are assumed to be useful as measures of factors that influence effort and quality of change.

Table 4: Author-based Measures

Measure	Target Attribute	Definition
Nr. of Authors	Diffusion	The number of developers involved in implementing the change.
Experience (EXP)	Experience	The experience of the authors: The average number of modifications on the file affected by the change request made by the authors implementing the change. This measure is supposed to be used in conjunction with the next metric which is; IEXP.
Inexperience (IEXP)	Experience	The "inexperience" of the authors: The average number of modifications on the file affected made by the authors *other* than the ones that are implementing the change.

Skill factors are known to largely influence developer performance [18]. The importance of the experience an author has with the part of the system he is modifying is a crucial factor in determining the success of a modification according to the analysis done by Gîrba et al. [16]. Therefore, we proposed the following measures: Experience and inexperience. Variations in

experience as quality predictor has also been used by Mockus and Weiss [1]. The number of authors associated with a change been used to measure diffusion. However, we have not seen other attempts at measuring the degree in which *others* have conducted work on the code, a concept we try to capture in the IEXP-measure. The assumption behind this measure is that there should be a difference between an author doing a number of modifications on a file he/she is the only one to have ever worked on, compared to doing the same number of modification to a file where *other* authors have done a large number of previous modifications.

5. The Change Model

5.1 Purpose and Taxonomy

In order to get clear overview of where and what information about change could be found in the ECMS framework and to be able to clearly define the measures, we chose to use the widely used and commonly known Unified Modelling Language (UML) [19], to create a visual representation of the change domain. The resulting model allowed us to gain better insights into how we trace the change resulting of a request for change in the bug-tracking tool, down to the modifications in files and code stored in the VCS. It also served as a basis on which we could provide exact definition of our measures as well as where and how they could be extracted. The model then served as a roadmap for the design and implementation of our measurement extraction tool, and thus ensured that the conceptually defined measures would be correctly designed and implemented in actual code.

As far as we know, an UML based approach to analysis, design and implementation of change measures at conceptual level, has not been used before, an approach that lets us make sure everything ties together correctly from definition to code.

To further enhance our understanding, and facilitate ease of discussion, we chose to categorise the measurements that can be extracted in different parts of our model into two different categories:

- *Analysis measures (AM)*:

Analysis measures represent the measures that will ultimately be used in an analysis context. They correspond to the measures described in section 4.3.2. They are usually constructed by using one or more Implementation Measurements (see below).

- *Implementation measures (IM)*:

These represent measures with which the analysis measures are implemented.
For example, the AM getting the number of added lines for a change relies on other measurements: For each file affected by the software change, we need to add together the results of measurements on how many lines were added to that specific file. These are implementation measures; they are measures implementing the analysis measures.

5.2 The Model

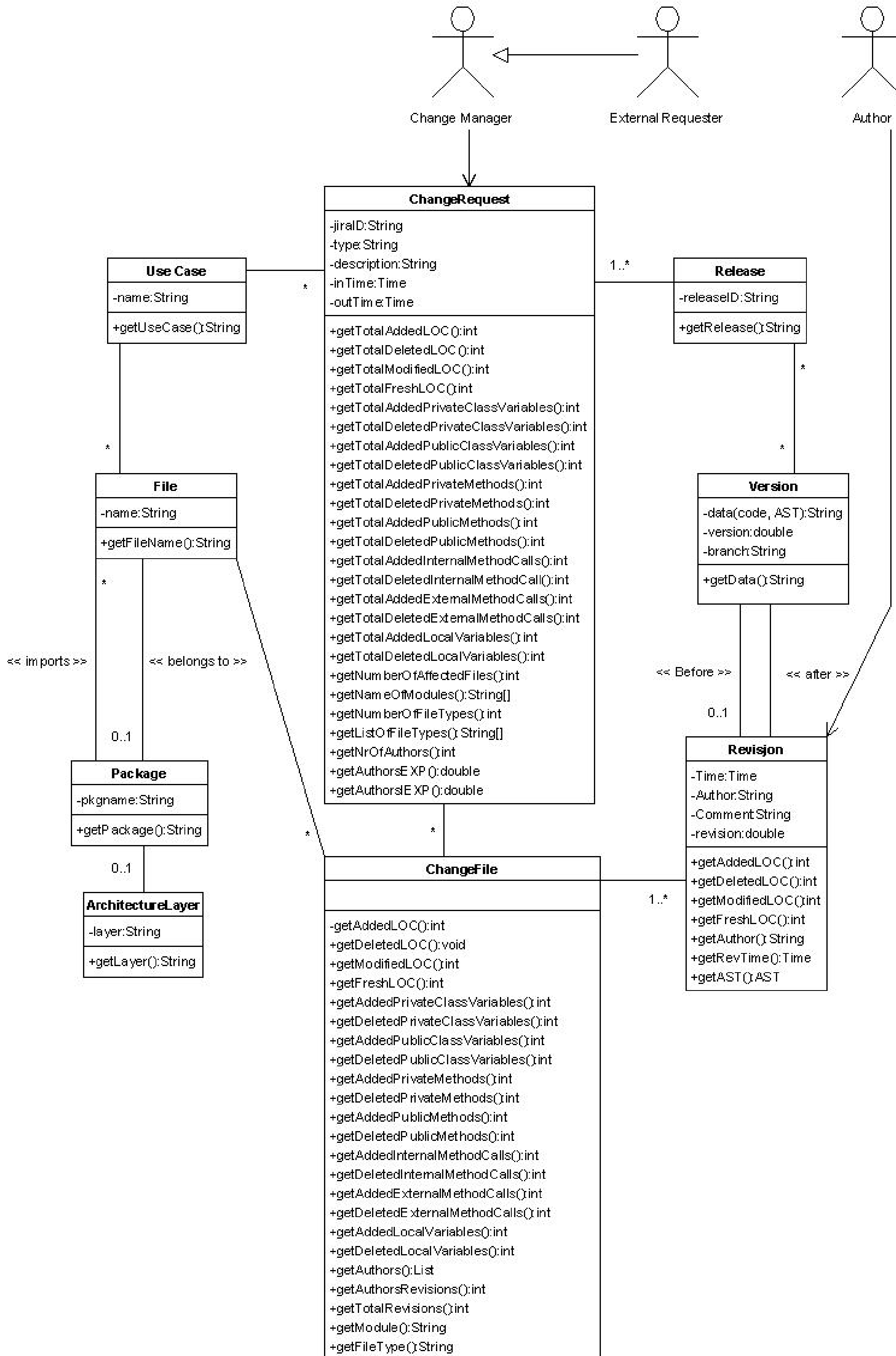


Figure 5: The Change Model

The software changes done to the system are the result of Change Requests (CR). This is comparable to the view presented by Mockus et al. [1] with the concept of Maintenance Requests (MRs). CRs are recorded in the Extended Change Management System (ECMS). In the case study from which we use empirical data in this thesis, a project management, bug and issue tracking system called JIRA [20] on top of an open-source VCS called CVS is used. The change requests submitted in JIRA are traced down to the files affected by it in CVS. In CVS, the changes to the files initiated by the CR can be examined. We were able to do this by having the developers working in the project include the change identifier in their check-in comments. This is a common technique for tracing changes in a management tool down to the modifications resulting from it in a VCS [10].

The measures are change request-scoped, which means that measures a drawn from modifications in the VCS resulting from a change requests in the bug/issue tracking management tool. This is equivalent to what German and Hindle [2] classify as m*odification* record –scoped, where MR is a term for the small increments in which a system is changed, added together to represent a change introduced into the ECMS.

Attributes in the UML model represent measurable data present in the specific part of the change management/versioning system, while operations are representations of measures based on the data available. It is the properties of software changes in the ECMS that we are ultimately interested in, the modifications implementing a ChangeRequest, hence most of the AM's are located in the ChangeRequest class. The IMs are mostly located in the Revisions caused by a ChangeRequest on a ChangeFile, which represents the connection between a change and each file it affects.

5.3 The Actors

The model includes actors that interact with the entities; these represent the different roles people in the domain can have and which entities they are responsible for.

An External Requester

This actor represents any stakeholder that requests a change to the system. This could be an end-user, a tester, or the developers themselves.

A Change Manager

This actor represents a person that is responsible for keeping track of and making sure changes are implemented.

An Author

Author is a representation of the person that implements a modification to a file as part of a request for a change.

5.4 The Entities

5.4.1 ChangeRequest

A ChangeRequest represents a change initiated by the Change Manager on the basis of an external request for change.

5.4.1.1 ChangeRequest attributes

The ChangeRequest contains information about the data that is stored with it in the ECMS. This includes:

- *ChangeID*
This attribute represents the unique identifier of a change request. It is usually assigned a value by the change management tool. This value is used by developers to identify which change request the code modifications implement.

- *Type*
This attribute indicates type of change. This is usually a categorical value, e.g., bug fix, new code, restructuring, refactoring, depending on the classification schema selected.

- *Description*
A textual description of the change

- *InTime*
This attribute represents the time of submittal into the ECMS.

-*OutTime*
This represents time of completion. It is registered in the ECMS when the developers have implemented the change.

5.4.1.2 Change request operations

The measures described in section 4.3.2 are represented as operations in the UML model. Since ChangeRequest is the entity we ultimately want to make measurements about, this is where all the AM's are located.

The AM's are implemented by aggregating the results of corresponding IM's on the changed files (ChangeFiles); most of those consist of measurements on the modifications made to it (Revisions) and the result of these (Versions). The available measurements, in correspondence to section 4.3.2, are:

Table 5: Measurements of ChangeRequest

Measurements	Type	Description
getTotalAddedLOC():int *getTotalDeletedLOC():int* *getTotalModifiedLOC():int* *getTotalFreshLOC():int*	AM	These Analysis Measurements (AMs) goes through every change to a file (ChangeFile) and uses the corresponding IM there for getting Added, Deleted, Modified or Fresh LOC the change has caused to it. This result is a natural number, i.e. an integer, for each of the measures. These measurements are indication of *size*
getTotalAddedPublicClassVariables():int *getTotalDeletedPublicClassVariables():int* *getTotalAddedPrivateClassVariables():int* *getTotalDeletedPrivateClassVariables():int* *getTotalAddedPublicMethod():int* *getTotalDeletedPublicMethod():int* *getTotalAddedPrivateMethod():int* *getTotalDeletedPrivateMethod():int* *getTotalAddedInternalMethodCalls():int* *getTotalDeletedInternalMethodCalls():int* *getTotalAddedExternalMethodCalls():int* *getTotalDeletedExternalMethodCalls():int* *getTotalAddedLocalVariables():int* *getTotalDeletedLocalVariables():int*	AM	These AMs also go through every ChangeFile caused by a CR and collects the number of added and deleted, public and private class variables and methods, internal and external method calls and local variables done to the files as part of Revisions initiated by the CR. When these are added together, they give us an integer for the total number of such modifications as result of the CR. These are measurements on *structure* (see 4.3).
getNumberOfAffectedFiles():int	AM	This AM simply counts the number of ChangeFiles the CR has initiated and returns the number in the form of an integer. This is also the first of the *architectural* measurements (see 4.3).
getNameOfModules():String[]	AM	This AM results in a list of modules in which its ChangeFiles belong, collected by summarizing the results of the ChangeFiles IMs getModule(). This gives a nominal list [17] of module names (A list with of strings of text).
getNumberOfFileTypes():int	AM	This architectural AM summarizes number of distinct (i.e. discounting duplicates) file types collected as a results of the getFileType() IM on the ChangeFiles belonging to the CR. The result is an integer.

Measurements cont.	Type	Description
getListOfFileTypes():String[]	AM	Using the same IM as the previous AM, this architectural AM collects the results of the owned ChangeFiles getFileType(), discounting duplicate results. Resulting in a nominal list of filetypes.
getNumberOfAuthors()	AM	This measurement rely on the results of IM on changes to files (ChangeFiles) as result of the change request to get a total number of authors implementing the CR.
getAuthorsEXP():double	AM	This experience based AM is based on the calculation of the experience the Authors implementing the CR has, on average, with the files they modified. The resulting number is a double precision number (double) which is the usual data type for numbers that contains decimals.
getAuthorsIEXP():double	AM	This is AM is based on the calculation of the *lack* of experience the Author implementing a CR has, on average, with the files they modified. The result is a double precision number.

The measurements getAuthorEXP and getAuthorIEXP are probably the most complicated of our model, and we will therefore illustrate how they work visually and by example:

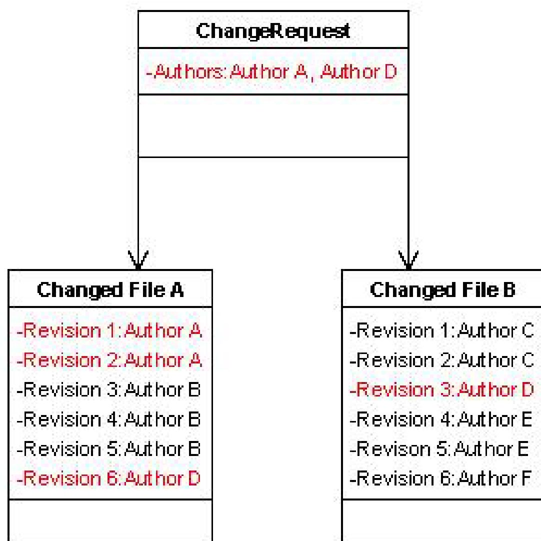


Figure 6: The Authors implementing a Change Request

Pseudo code for calculation:

```
File A:
Change Request's EXP= 3, IEXP=3 (3 Revision by Change Request's
Authors, 3 by others)

File B:
Change Request's EXP= 1, IEXP=5

The formula for Authors' EXP is:
File A EXP + File B EXP / NrOfAuthors
= Avg. EXP pr Author
Avg. EXP pr Author / ChangedFiles
= Avg. EXP of Authors pr ChangedFile

Applied to our example:
(3 + 1) / 2 = 2
2 / 2 = 1
AuthorsEXP = 1

The formula for Authors' IEXP is:
File A TotalRevs - File A EXP + File B TotalRevs - File B EXP /
NrOfAuthors
= Avg. IEXP pr Authors
Avg. IEXP pr Author / ChangedFiles
= Avg. IEXP of Authors pr ChangedFile

Applied to our example:
(6-3) + (6-1) / 2 = 4
4 / 2 = 2
AuthorsIEXP = 2
```

5.4.2 ChangeFile

Each ChangeFile represents one change in one file; each ChangeRequest owns several change-files. Each change to a file (ChangeFile) is implemented by one more Revisions.

5.4.2.1 ChangeFile attributes

This class primarily provides the link to the specific Files and the Revisions made to it as part of the CR, and contain no attributes by itself.

5.4.2.2 ChangeFile operations

Most of the operations here are implementation measures and are described indirectly when we described the analysis measures using then. We will therefore just give a short description of how they work on this level.

Table 6: Measurements of ChangeFile

Measurements	Type	Description
getAddedLOC():int *getDeletedLOC():int* *getModifiedLOC():int* *getFreshLOC(): int*	IM	These size measures are gathered from the Revisions initiated by a ChangeRequest.
getAddedPrivateClassVariables():int *getDeletedPrivateClassVariables():int* *getAddedPublicClassVariables():int* *getDeletedPublicClassVariables():int* *getAddedPrivateMethods():int* *getDeletedPrivateMethods():int* *getAddedPublicMethods():int* *getDeletedPublicMethods():int* *getAddedInternalMethodCalls():int* *getDeletedInternalMethodCalls():int* *getAddedExternalMethodCalls():int* *getDeletedExternalMethodCalls():int* *getAddedLocalVariables():int* *getDeletedLocalVarables():int*	IM	The structural measures are done by getting the structure (in the form of Abstract Syntax Trees – AST) of the Version a Revision has caused and comparing it to that of the previous Version. This is done for each Revision to a file
getAuthorsOfChange():String[]	IM	Returns a nominal list of Authors that have authored Revisions to the file as part of a CR.
getRevisionsByAuthor():int	IM	Returns how many Revisions a specific Author has made to the file in total (not just part of a CR).
getFileType():String	IM	A ChangeFile is the connection between the modifications to a file and the file itself. This IM retrieves the file type from its associated File
getModule():String	IM	This measurement gets the package of the associated File and returns this as a nominal value.

5.4.3 Revisions and resulting Versions

The difference between Revisions and Versions is that Versions are the result of Revisions: Revisions are the actual modification as opposed to the resulting Version. This ties back to the concept of modification aware and unaware metrics, we want to measure the Revisions themselves, not compare Versions in isolation.

5.4.3.1 Revision attributes

- Time
When the revision took place.

- *Author*
The author of the modifications the Revision represents.

- *Comment*
Authors comments about the modifications.

- *Revision*
The revision number.

5.4.3.2 Revision operations

Table 7: Measurements of Revision

Measurements	Type	Description
getAddedLOC():int	IM	Returns the number of lines that were added by the Revision
getDeletedLOC():int	IM	Returns the number of lines deleted by the Revision.
getModifiedLOC():int	IM	Returns the number of lines that were not deleted or added, but altered in some way.
getFreshLOC():int	IM	Returns the number of lines of any *new* code, i.e. the number of lines of any new file.
getAuthor():int	IM	Returns the name/alias of the author that did the modification the Revision represents.
getAST():AST	IM	Gets the structure of the code in the form of the Abstract Syntax Tree (AST) of the code that was the result of this Revision.
getPrevAST():AST	IM	Gets the structure (AST) of the code as it was *before* the Revision. This allows for comparison to see what was added or deleted in the new AST.

5.4.3.3 Version attributes:

The attributes of Version represent the state of the file after modifications.

- *Data*
The actual content of this version of a file. Normally, this is software code written in a programming language. If so, one can extract the structure of the code by capturing its abstract syntax tree (AST).

- *Version*
The version of the file, represented as a number.

- *Branch*
If the version is part of a branch, this is the branch it belongs to. Branches are versions of files that are separate from the main system evolution (the trunk) to be e.g., experimented on without it having an impact on the rest of the system. This may then be merged back into the trunk as a new trunk version of a file at a later stage.

5.4.3.4 Version operations

Version contains only one IM, getData():data, which returns the contents of the version, from which the structure can be extracted.

5.4.4 Release

None of the measures in this thesis concerns themselves with releases of a system. However, this entity allows for the implementation of new measurements in the future. This entity is a representation of versions that are grouped together as part of a release of a system, and can be used to clearly define and specify measurements on system releases.

5.4.5 File

This entity represents a physical file in a system. Although a file can exist in many different versions (see 5.4.3) this entity represents the version independent attributes of a file, such as file name, location, which package it belongs to and what external system functionality, represented by use cases, its content is supposed to provide for in the system. This entity can be used to specify measures on how a change request affects architecture and external functionality of a system.

5.4.6 Use-Cases

We have included a representation of UseCases illustrating that a ChangeRequest can have an impact on the functionality of the systems. This allows for future specification of measurements of the UseCases a ChangeRequests affects.

5.4.7 Package and ArchitectureLayer

A File may belong to a package; this is usually the case if the file contains code and is a source file. Packages are usually tied to the architecture of the system, which is illustrated by the representation of an ArchitectureLayer class. For instance, a package *no.simula.program.ui.file* is tied to the presentation layer while *no.simula.program.storage.file* is located in the persistency layer. These entities can be used to specify measures on what parts of the architecture a change request spans, and how the architecture is affected by changes as part of a change request.

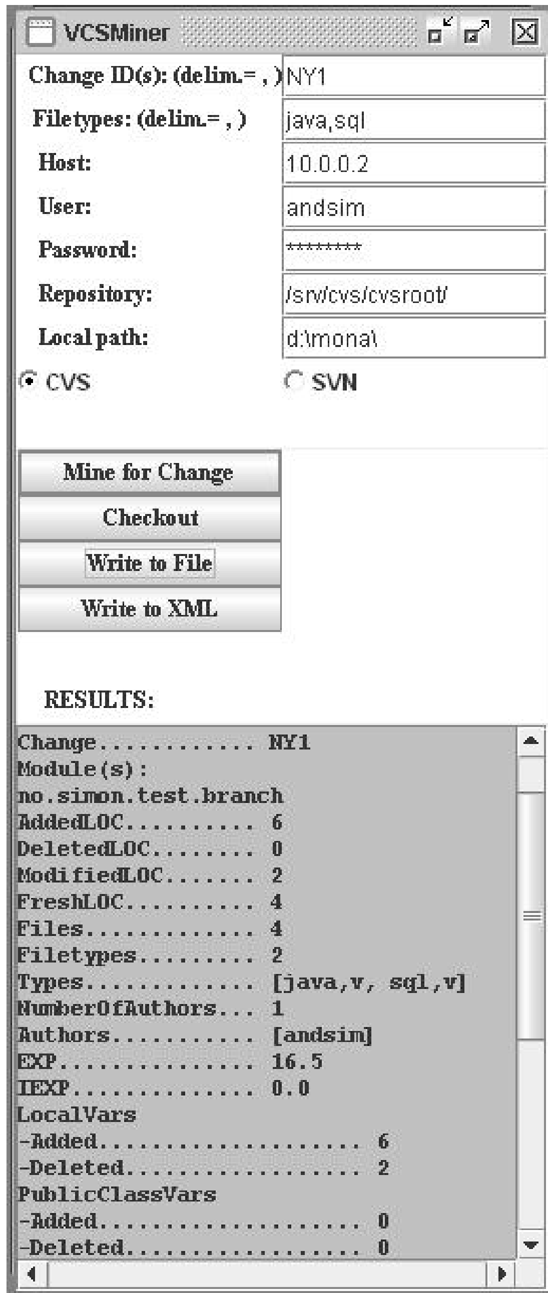
6 The Version Control System Miner

We used our model and measure definitions based on it to design a program that could be used for extracting such measurements.

It was required that the program would be flexible with respect to underlying operating system and VCS technology. It was also required that it should be easily extendable, in order to constitute an evolving measurement platform for the research group.

Based on these criteria we decided to develop our program using Java. One of the strengths of Java that it is, in principle, operating system (OS) independent. The only system requirement is

a Java Runtime Environment (JRE) on the OS in question and any Java program should be able to run on it. This is achieved in with techniques such as just-in-time (JIT) compilation and special java byte code, which we will not go into here. For more information regarding Java, [12] , [13]or [21] are appropriate sources. This approach also makes the programs both small and portable. When it comes to flexibility concerning VCS technology, this is something we needed to design for.




```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <Change id="NY1">
  - <Modules>
      <Module>no.simon.test.branch</Module>
    </Modules>
    <AddedLOC>6</AddedLOC>
    <DeletedLOC>0</DeletedLOC>
    <ModifiedLOC>2</ModifiedLOC>
    <FreshLOC>4</FreshLOC>
    <Files>4</Files>
    <Filetypes>2</Filetypes>
  - <Types>
      <Type>java,v</Type>
      <Type>sql,v</Type>
    </Types>
    <NumberOfAuthors>1</NumberOfAuthors>
  - <Authors>
      <Author>andsim</Author>
    </Authors>
    <EXP>16.5</EXP>
    <IEXP>0.0</IEXP>
  - <LocalVars>
      <Added>6</Added>
      <Deleted>2</Deleted>
    </LocalVars>
  - <PublicClassVars>
      <Added>0</Added>
      <Deleted>0</Deleted>
    </PublicClassVars>
  - <PrivateClassVars>
```


Figure 7: The VCSMiner and resulting XML file.

6.1 Assumptions

The design is based on the assumptions described in the next subsections.

6.1.1 Implementation of other VCS

We assume that any VCS to be supported by the program have a structure where modifications are stored and can be tied to a specific file and author, and that these modifications are possible to capture as output from either commands, or contents of a file or repository. The VCS must also allow programmers to annotate the modification with a comment, and that users of the VCS use this comment to identify which change request it is tied to.

6.1.2 Remote Connections

The program is designed for the use of remote connections to the VCS in question. We base our program's ability for remote connections on the assumption that, if the VCS repository is not local, the VCS is correctly set up to receive remote connections and that there is an available Java API for doing so. Alternatively, an appropriate command line client should be set up and configured for such communication on the same machine as the program

6.1.3 Runtime Environment

We assume that the program has been allowed execution, reading and writing rights on the machine it is running, allowing it to execute runtime commands and read their resulting output as well as write measurements to files. Finally, we assume that the machine where the system is to be run either has a Linux or a Windows OS with at least Java Runtime Environment (JRE) version 1.5 or higher installed. Mac OS is not supported by this version of the program.

6.2 Design

Our measurement tool in its first incarnation only accessed and performed measurements on code and modifications in a CVS repository. However, one of our goals was to create a measurement tool that was not too dependant on any specific VCS. Therefore the design of the system needed to be in such a way that it allowed for flexibility in letting new VCS technologies be supported. This is why we decided to let the system communicate with the classes dealing with VCS specific functionality through the use *Interfaces*. In this way the rest of the system does not need to know how the VCS specific access and functionality was handled, relying instead on the operations present in the interface. The VCS specific implementation of these interfaces then deals with how the functionality and implementation of the operations are carried out. In other words, it lets us abstract away and contains all technology specific functionality to parts of the system that are isolated from the rest. This allows for easier implementation of functionality needed for a new VCS, without having to redo large parts of the code. Another useful concept we took advantage of is *inheritance* and *polymorphism*. This allows classes that does not fit the needs of a specific VCS technology to

father children with all the same functionality, except those that need to be overridden and treated differently or added. This will be exemplified when we present the introduction of a new VCS, Subversion, in (6.4)

The Change Model presented in section 5 served as a basis for the design of the program. The classes have similar names and relationship to each other wherever possible, and the methods implementing measurements are, naturally, located the same places and have similar names as the measures in the Change Model. The Change Model was not intended to be a design model or an implementation model, in that respect it is more akin to a domain model [19]; a conceptual model to give us better understanding of what could be measured and where. There are for instance no classes dealing with VCS specific access or parsing of data from VCS operations in the Change Model.

The following class diagram shows the relations between classes, and the packages that contains them:

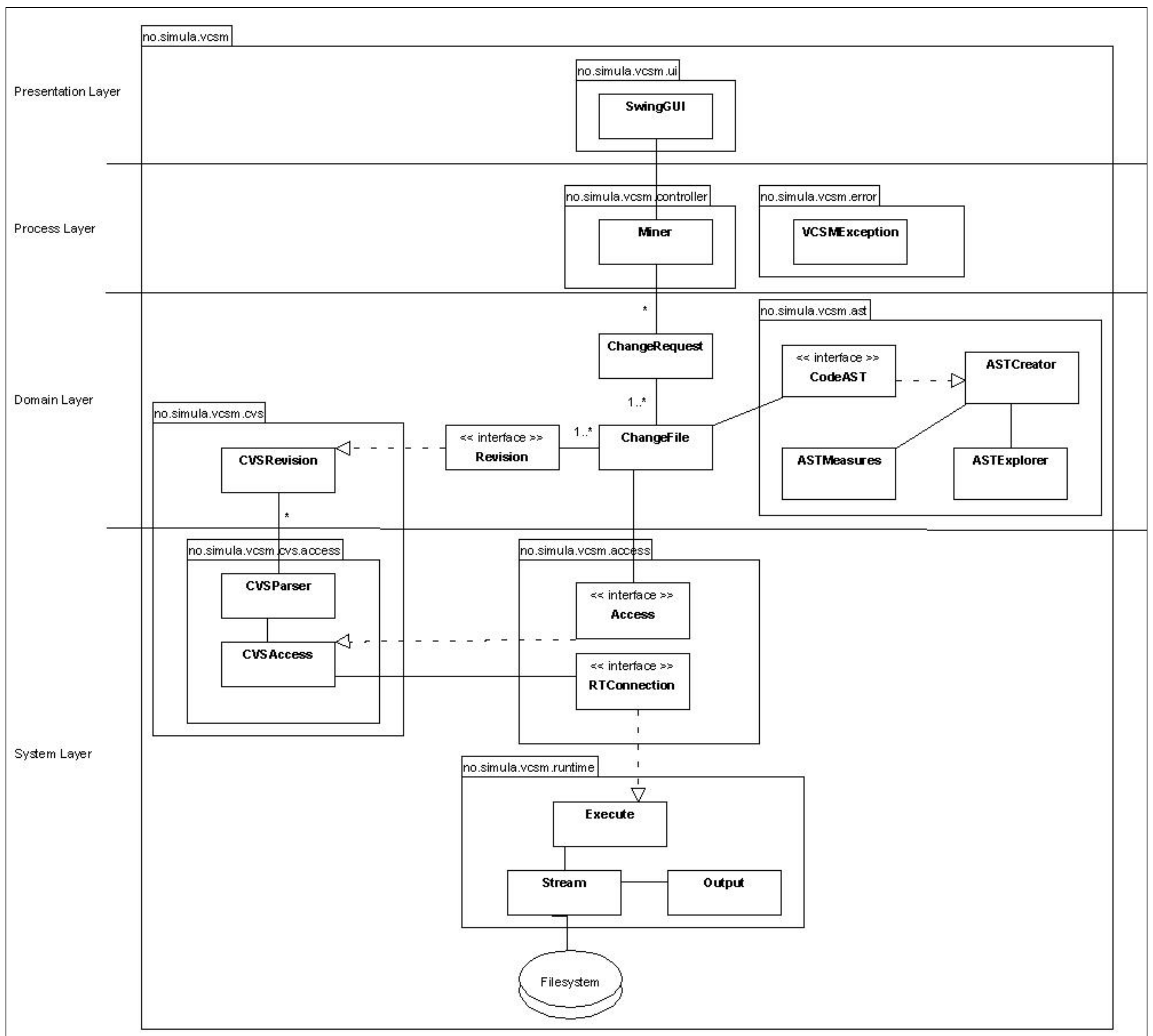


Figure 8: VCSM Class Diagram without attributes and methods showing

At the heart of the program, in the domain layer, is the concept of having a ChangeRequest owning a set of ChangeFiles which in turn owns a set of Revisions to the file. This is accordance with the Change Model (5.2). On top of this, there is a controller class, called Miner. This class creates ChangeRequest objects for each change that is measured. At the very top, in the presentation layer (package *no.simula.vcsm.ui*), we have a graphical user interface with which the user can interact and request information about changes.

Returning to the ChangeFile class, we see that it is connected to a set of interfaces. The Revision interface hides any VCS specific way of handling information about revisions to a file. The ChangeFile does not need to know anything about this as it only invokes methods through the interface; instead, it is up to the class implementing the interface to provide the results the ChangFile expects. Likewise, the interface Access provides the system with access to any specific VCS while the RTConnection (RunTime Connection) interface lets the implementation of Access interact with and execute commands against the operating system the program runs on. The CodeAST interface allows for measurements of the AST of code without having to worry about how AST extraction and measurements are handled for different kinds of programming languages.

6.3 Implementation of measures

6.3.1 Extracting Data from CVS

To find out what kind of data that can be extracted from the CVS repository the documentation for CVS was examined in detail [22] & [23], a selection of existing studies basing data extraction from CVS [6], [24]& [15] was reviewed and experimentation into the manipulation of extracted data was performed.

Mierle et al.[24] provides a simple and concise description of the CVS storage structure. Each file in the CVS repository is associated with a corresponding Revision Control System (RCS) file that contains data about the revisions performed on it. In addition to this, there are the administrative files that are located in a /CVSROOT/ folder. Records of activity are therefore split into two occasionally disjointed records. File history is stored in the RCS files (modifications, additions, deletions) and a history file in the CVSROOT directory tracks most of the interactions between users and the repository, including those in the RCS files in addition to actions such as checkouts and updates.

We have explored the use of a Java based API for CVS (jCVS) as part of the experimentation but found it too bulky, complex and undocumented to use for our purposes. Another API for Java exists as part of the Netbeans package but this did not prove easily accessible either. Fortunately, the command line-based *cvs toolset* presents itself as useful for extracting data about change in code, providing access to data such as what was added, when, and by whom in addition to the ability to compare different versions of code (i.e. current or selected version with a previous version of choice). We therefore chose to use the cvs toolset as the basis for the measurements.

Another approach would have been to parse the RCS files and the administrative files directly, however doing so require handling the complexity of disjointed and split records as mentioned earlier, without taking advantage of the functionality already made available through the cvs toolset. It would also mean that direct file access to the repository would be needed for the system to work.

6.3.1.1 Data provided by CVS log:

The output from a cvs log is as follows:

Command:

```
cvs –d :ssh;username=theuser;password=thepassword;hostname=host:/path/to/repository log
```

The command is performed over SSH, as the repository is located on a server secure server, only accessible through this protocol. Alternatively, "ssh" can be replaced with "ext" for ordinary password server protocol. If the command is run on the same machine as the repository only the path to repository is required.

Result:

```
cvs log: Logging src/test

RCS file: /home/andsim/.cvsroot/VCSMiner/src/test/Tester.java,v
Working file: src/test/Tester.java
head: 1.3
branch:
locks: strict
access list:
symbolic names:
	initial: 1.1.1.1
	VCSMiner: 1.1.1
keyword substitution: kv
total revisions: 4;	selected revisions: 4
description:
----------------------------
revision 1.3
date: 2006-08-29 18:04:02 +0000;  author: andsim;  state: Exp;  lines: +2 -2
MT013: slettet kommentar avmerking, ny kode
----------------------------
```

- *Example output from a cvs log command.*

For each file, logged information is provided on

- The name and path to the RCS (Revision Control System) file in the repository and the local work file.
- Which version is the current (identified by "head"), branched versions, locks, access list, symbolic names, keyword substitution, total and selected revisions.
- A description for each revision consisting of the revision number, date, author, state, lines added or deleted and a comment is also provided.

6.3.1.2 Data provided by diff:

The algorithm behind the file comparison tool "diff" is an old and tried one, first presented in the papers [25, 26] and later [27]. It has since been improved upon and is used as part of the toolset for large VCS such as CVS [23] and Subversion (SVN) [28]. It takes as parameters two versions of a file and compares them.

The resulting output of the diff tool varies according to the parameter specified when executing it:

Standard output: diff
A simple diff between two versions or files (specified by the –r[revision nr]) lets you extract if a line exists in the first version but not in the other or vice versa by the preceding symbol ">" and "<" (The two first columns of output from a diff command is reserved for special codes). This lets you get a measure of how many lines has been added and how many deleted. However, it does not let us know whether a line has been modified. The output also identifies in which lines of code the affected section is located (note: not the actual line number of the modified line(s)!).

Context output: diff –c
By use of the option –c after the diff command the result is an output where the change is seen in the context of the preceding version. Based on the resulting output one can discern which lines has been added, modified or deleted; the affected lines are preceded by the symbol "!" where there has been a modification but the number of lines has remained the same, "+" where there has been an addition of a new line, and "-" where a line has been deleted.

Side by side output: diff –y
This functions much the same way as diff –c but displays the two different versions side by side for easier comparison. The symbols denoting modifications is the same as in an ordinary diff ("<" and">") with the addition of "|" to specify that the line exists in both versions but has been a modified. The symbols are listed between the compared versions in a dedicated column (Column 64).

Unified output: diff –u
Compares the versions on a line-by-line basis, a modified line is first displayed in its original form and then in its modified form. The lines are preceded by the symbol "-"for the line that has been modified and "+" for the new version where the lines have been modified, and just "+" or "-"where lines has been added or removed.

Since we want to collect information about not only added and deleted lines of code but also modified lines, only the context or the side-by-side output is useful.

The diff tool provides us with the measurements of size: The added, deleted and modified lines of code, for the Revision objects. The diff delta function takes as parameter two versions of a file and compares them. This makes the measurements on the number of added, deleted and modified files extracted from the output modification aware [2].

6.3.1.4 Parsing and capturing the data

The data provided by the cvs toolset are encapsulated as objects of the classes in the system. To do this, the class CVSParser reads the output of the *cvs log* command and use that data to create both ChangeFile objects and owned Revision objects. This approach was also used with success by Gîrba et al. [16]. The output resulting by any commands run on the runtime environment is executed and captured by classes belonging to the package *no.simula.vcsm.runtime* . The command is executed by the Execute class, the output are then captured by a Stream thread, either as an error or a normal output, and encapsulated as an Output object. The Output object gives the parser the results of the cvs log command as a String.

Since we know the format and chronology of entries resulting of a cvs command, parsing these for use in the system should be easy. Unfortunately, the output described as result of a cvs log in section 6.3.1.1 is not always consistently correct. Sometimes the text “cvs log: Logging x/x …” is inserted in various unexpected places. However, thankfully, the rest of the output format is always the same. We are therefore able to parse the data and insert them into a ChangeFile or a Revision object on a line-by-line basis, although complicated by the fact that there was a need to filter out the text mentioned above.

6.3.3 AST representations of code

For each version of a file measured, and the previous version of the same file, we create an AST from the code through the interface CodeAST. We then do measurements on these, which are returned as objects of the class ASTMeasures. For CVS this is done by fetching the version of the file from the VCS (using a command updating a specified local file to a specified version in the repository), capture the code, and creating the AST of this code. This is done every time we want to do measurements on a single ChangeFile owned by a ChangeRequest.

The program uses Eclipses JDT library to create ASTs from captured code, Eclipse’s projects are non-profit and its licensing (the Eclipse Public Licence) is approved by the Open Source Initiative [29]. This gives us the freedom to incorporate the use of their libraries into the system without having to worry about legal or financial ramifications. This library lets us create AST for code written in Java only. Since the measurements provided by the ASTs are considered a *proof of concept*, and are more experimental than the others, it is acceptable to restrict such measurements to the Java language.

6.3.4 Size comparison (difference)

A ChangeFile’s size attributes are extracted by a method called doDiffs(). This method uses its Access object which in turn uses the *no.simula.vcsm.runtime* package to run cvs diff commands for generating side by side comparison between the different versions resulting of a change and the previous version (as described in section 6.3.1.2). Diff options for ignoring added, deleted or modified blank lines or white spaces are also used:

Command:

```
cvs –d :ssh;username=theuser;password=thepassword;hostname=thehost:/path/to/repository/
diff –y –t –B –w –r[revnr] –r[revnr2] path\to\file\
```

The option “–y” indicates side by side format as explained earlier, “-t” indicates expansion of tabs to spaces in the output, this is done to preserve the alignment of tabs in the input files The option “-B“ indicates that it ignores changes in blank lines and “-w” that it ignores white spaces when comparing lines.

Result:

The output is then captured as objects of the Output type by the Stream (one for normal execution and one for errors) threads. The Output object returns its content to the measurement algorithm in ChangeFile’s method doDiffs() as a list of strings, each string representing a line of output. Each string is examined one by one, and the character at position 64 (which corresponds to the column with information about added, deleted or changed symbols) is collected. Every occurrence of the character “<” the algorithm updates the ChangeFile’s dloc attribute (deleted lines) by adding one, in a similar way aloc (added lines) is added by one for each “>” and mloc (modified lines) with each “|”.

Methods implementing the ChangeFile’s IMs for size measurements in the Change Model then return the values of the aloc, dloc and mloc variables.

Fresh lines of code are collected by the method implementing the ChangeFile’s IM getFreshLOC() in the Change Model (bearing the same name) itself. If a revision to a file has no previous revisions (i.e. its new to the VCS), the Access object is used, which in turn uses the runtime package, to get a simple line count of the file discounting blank lines. The ChangeFile’s attribute floc (Fresh lines of code) is then set to be the number returned by the line count.

6.3.4 SSD: Getting Measurements

A System Sequence Diagram (SSD) is a UML diagram that is often used to visualise how and in what sequence, different objects and classes in a system interact to perform some function. To illustrate how the measurements are done in the system we have chosen to use such a diagram:

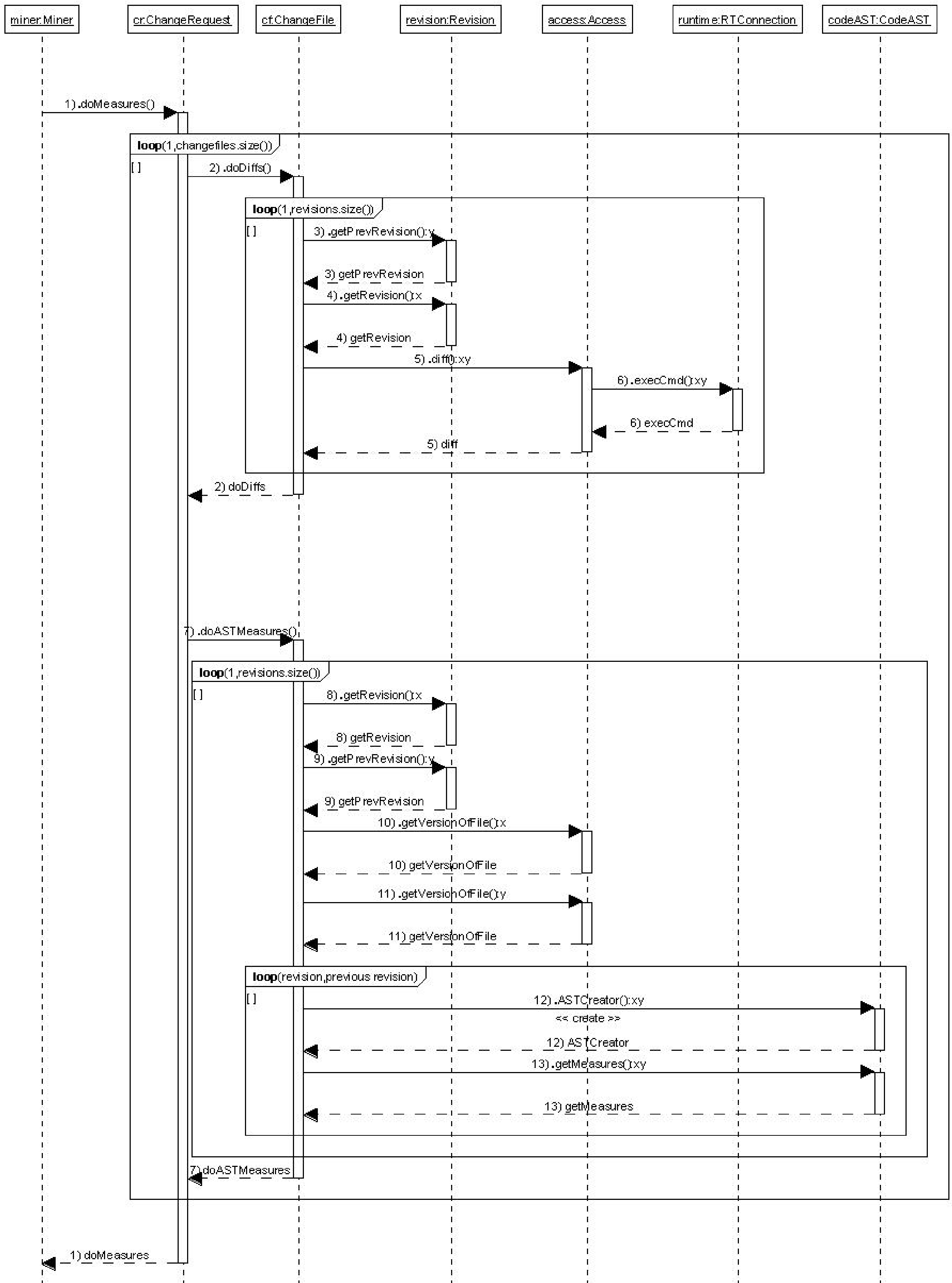


Figure 9: System Sequence Diagram: doMeasures()

Explanation:

The boxes at the top represents instances of a class, where the name is first and then the class it is a representation of separated by a colon. Each instance of a class has a "life-line" representing the lifecycle of that object. Arrows represent Method calls between the objects from the object calling the method pointing to the object on which the methods are located. The sequence of operations is from top to bottom. Repetition of method calls are represented by a box with the text loop (start condition, end condition) in the upper left corner. This symbolizes that the operations inside the box are repeated from start condition to the end condition.

The controller class (Miner) invokes the method doMeasures() on one of its ChangeRequest objects. The ChangeRequest object then goes through all the ChangeFiles it owns and asks them to perform diff measurements (doDiff()) and AST measurements (doASTMeasures()). The method doDiff() goes through each Revision the ChangeFile owns and gets its Revision number (i.e. the version number of the resulting file) and the previous Revision's number. These numbers are used to perform a diff command (as explained in 6.3.4). The doASTMeasures() method also goes through the ChangeFile's Revisions and gets its and the previous ones version number. This is used to get first one version of the file then the other, capture the AST (6.3.3), and comparing these.

After doing these operations the ChangeFiles all hold the result of their measurements, and the ChangeRequest just adds these together using getter methods (i.e. methods like <u>get</u>FreshLOC()) for each ChangeFile and makes them available through its own getter methods (i.e. get<u>Total</u>FreshLOC()).

6.4 Implementing support for SVN

Implementing support for the version control system Subversion (SVN) was relatively easy due to the program being design with such VCS expansions in mind. Since a good java API existed for communication with SVN, SVNKit (formerly known as JavaSVN) [30] we did not have to create a own parser for capturing output from a command line client, further simplifying the process. However, SVN differs from CVS in one very distinct way. The concept of versions is *global* for the system. In other words, if you modify one file, the version of all the files in the entire system changes. Luckily, comments are allowed for each individual modification to a file, and are not global like the version numbers. Hence, we are still able to trace a change request in the ECMS (JIRA) down to the modifications on files.

We implemented a SVN specific Access class, SVNAccess, which used the javaSVN API [28] to collect data about each file in the repository and the modifications made to it. Because of the global versions of files, we needed to handle data extraction differently. When we captured data to create the ChangeFile objects and the Revisions they owned we needed to first get a version, then get all the files in that version, create a Revision, and, if no ChangeFile object existed, create a ChangeFile for each file and add the Revision to it. If a ChangeFile for a specific file existed, we needed to get that ChangeFile from the collection and add the Revision to this already existing object. In other words, this is a reverse way of doing it compared to CVS where we from "cvs log" got the logged file and then its privately owned revisions!

Another complication because of the global version numbering is that a ChangeFile's first revision, unlike CVS, does not start at version number one. We tackled this by creating SVNChangeFile, inheriting everything from ChangeFile but modified (using polymorphism) to know which version it first appeared in, its starting version, and using that version as a baseline instead of the number one.

All in all, implementation of SVN proved to be easy, but it is worth noticing how complications arose because of the difference in concept of change than what is modelled in the change model i.e. a global version owning the files, instead of files having different versions.

7. Verification and validation

7.1 Verification of Miner

7.1.1 Correctness

To test the correctness of the measurements we measured 63 different changes, comparing the measurements to that of semi-manual scripts that had been developed for measuring the same properties of changes, except AST based measurements. This approach is known as the cleanroom approach[6]. We found that the results were strongly correlated, although not identical due to slightly different definitions of the measures. This can also be seen by how similarly they relate to effort in 7.2.1.1.

7.1.2 OS Independence

The program successfully executed all the operations required of it on both Linux (Ubuntu) and Windows XP.

7.1.3 VCS Independence

As described 6.1, this was tested by implementing support for Subversion (SVN). The requirement for Subversion support was not known at the start of this project. However, the tool is not completely VCS independent, as it still requires new VCSs to be implemented by creating the appropriate Access classes. The more the structure of the data stored about change differs from how it is modelled in the Change Model, the more work we would have to do to implement it.

7.2 Validity of Measures

7.2.1 Correlation analysis

In order to discern whether the candidate measurements could be used to explain variations in effort, on real project data, a correlation analysis was performed. The goal of this analysis was to establish if there were evidence of correlation between the measurements done by the VCSMiner tool and the actual effort data. The actual effort data was available as part of a industrial case study examining 63 change requests recorded in the JIRA bug and issue tracking tool. The VCSMiner was then used for extracting data and doing measurements on these 63 changes. The analysis investigated the level of correlation between these measurements and the effort data.

A low correlation (numbers below 0.3) would indicate that there were little correlation between effort and the measurements, and thus that such measurements would be of little use when trying to explain variations in effort for the investigated data.

In order to find out if certain measurements were highly correlated with others, and therefore potentially redundant, we analysed the internal correlations between the measurements done by the VCSMiner. A high correlation (numbers above 0.8) would indicate such redundancy and only one of the correlated measures would be useful in further analysis of the data.

For the actual analysis we chose to use Spearman's rank-order correlations. The alternative, Pearssons correlations, require that the distribution of variables are close to normal:

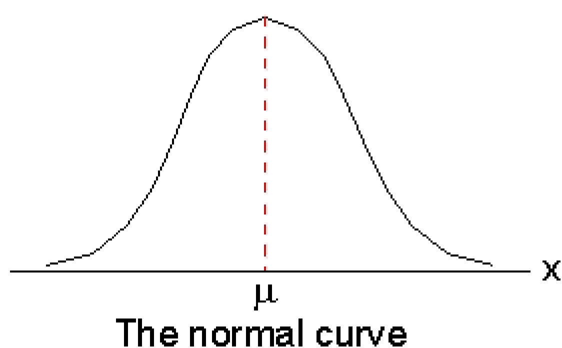


Figure 10: Curve resulting of normal distribution of variables.

However, it is clear from plotting effort and added LOC (figures 11 and 12) that the data is typically not normal. Using Spearman's rank-order correlation analysis means we will not use the actual values but the value's position in an ordinal ranking scale [17].

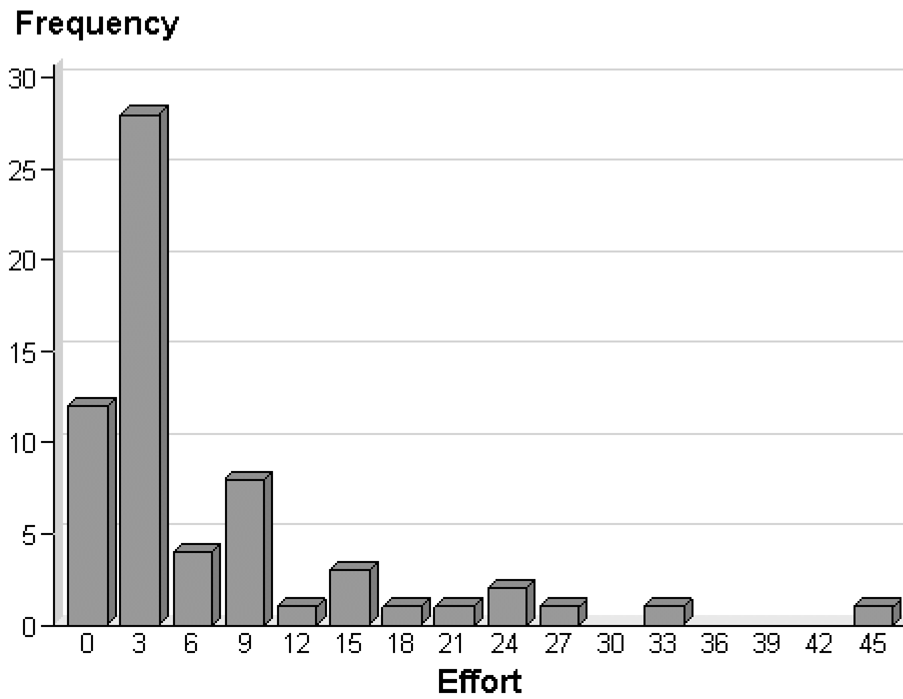


Figure 11: Distribution of data for Effort

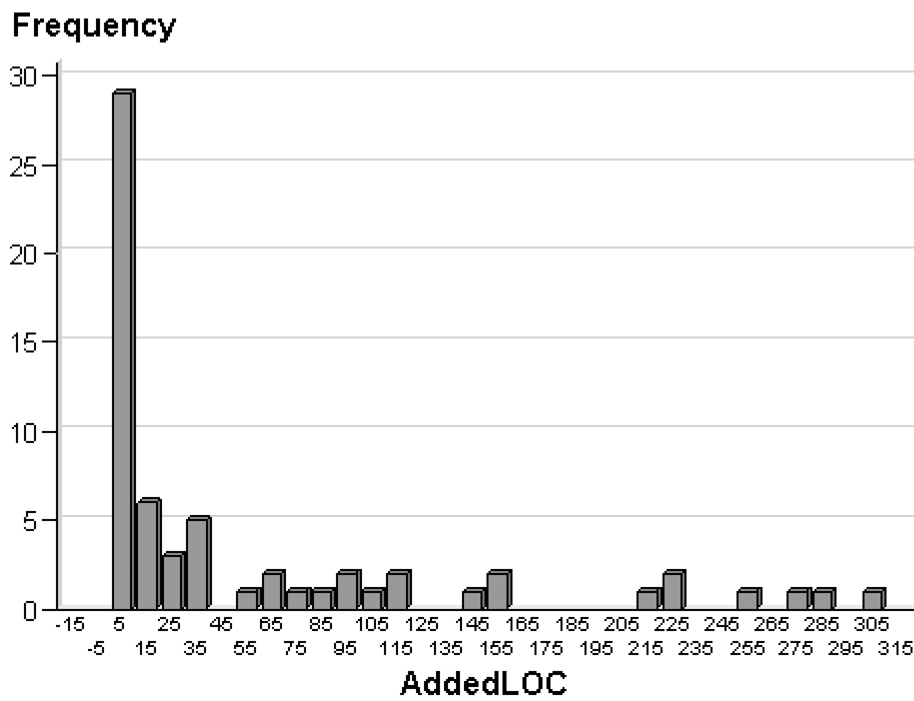


Figure 12: Distribution of data for Added LOC

7.2.1.1 Correlation to Effort

We use the symbol **rho** to stand for the correlation. The null hypothesis for this correlation analysis was that there is no relationship between the measurements and effort. The alternative hypothesis is that there is a relationship between the different measurements and effort:

Null Hypothesis:	**rho** = 0
Alternative Hypothesis:	**rho** <> 0

The proper interpretation of p value is complicated topic but, simplifying, we choose to say that a p value higher than 0.05 indicate that we might not have enough basis to suggest that a correlation exists. A value less than 0.05 allow is the usual level chosen to reject the notion that the correlation is zero. Thus if the p value is less than 0.05 but the rho is slightly under the critical value of 0.3 we can still say there is a correlation.

Results:

Table 8: Results of Correlation Analysis

	VCSMiner		Manual/Script	
	rho	P value	rho	P value
AddedLOC	0,46814	0,0001	0,43268	0,0004
DeletedLOC	0,41408	0,0007	0,36235	0,0035
ModifiedLOC	0,46622	0,0001	0,49266	<.0001
FreshLOC	0,2641	0,0365	0,13261	0,3002
AffectedFiles	0,52482	<.0001	0,56242	<.0001
NROfTypes	0,25986	0,0397		
Authors	0,17322	0,1746		
EXP	-0,2853	0,0234	-0,3486	0,0051
IEXP	-0,1934	0,1288	-0,2181	0,086
AddedLVars	0,30671	0,0145		
DeletedLVars	0,37363	0,0026		
AddedPubClassVars	0,29535	0,0188		
DeletedPubClassVars	0,27953	0,0265		
AddedPrivClassVars	0,18054	0,1568		
DeletedPrivClassVars	0,23617	0,0624		
AddedPrivMethodDec	0,30993	0,0134		
DeletedPrivMethodDec	0,09985	0,4362		
AddedPubMethodDec	0,25192	0,0464		
DeletedPubMethodDec	0,13936	0,276		
AddedExtMethodCall	0,38137	0,002		
DeletedExtMethodCall	0,29359	0,0195		
AddedIntMethodCall	0,38403	0,0019		
DeletedIntMethodCall	0,16621	0,1929		

Observations

Using 0.3 as the critical value for deciding level of correlation, we can see that:

- Measurements on size show a substantial correlation with effort, except perhaps for fresh LOC.
- The architectural measurements for diffusion of a change, AffectedFiles, have a relatively high relation with effort. Surprisingly, the span of how many different filetypes are involved has just some correlation, and the number of authors has very little correlation.
- Structural measurements show some to low correlation with effort, depending on the specific measurement and whether it is a deletion or an addition.

7.2.1.2 Correlation between Measurements

The null hypothesis for this analysis is that there is no correlation between the measures; the first hypothesis is that there is. The second hypothesis is that the correlations are lower than 0.8 and are therefore of value beyond the other measures.

Null Hypothesis:	**rho** = 0
1st Hypothesis	**rho** <>0
2nd Hypothesis:	**rho** < 0.8

Results:

We refer to attachment nr 2 for detailed results.

Observations:

All measures have some correlation with each other however miniscule, therefore we can reject hypothesis one and two.

When we consider our second hypothesis we can see that most of the measures are not highly correlated enough to reject our second hypothesis, except Added LOC with regards to Deleted LOC and Affected Files and Added External Method Calls with regards to Deleted External Method Calls.

Apart from this, we observe that measures of size are highly correlated to each other. It is interesting to note that measurements of size are not significantly correlated to structural changes. In addition, measures of experience have a low correlation to the other measures. It would therefore seem that both measures of experience and structure could be useful in explaining change effort beyond that of size. However, the low correlation between experience and size shown by the analysis, have a relatively high P value so the validity of this result is somewhat uncertain.

The measures on architecture also highly correlated to size, although well below the 0.8 mark. AffectedFiles are also highly correlated to some of the measures of structure.

7.3 Usefulness

To get feedback from practitioners of software engineering about how they perceived the usefulness of these kinds of measurements we arranged a meeting with the developers involved in the industrial case study. The developers in the case study used a methodology called Agile Retrospectives[5]. The activities of the Agile Retrospective methodology are illustrated in the following way:

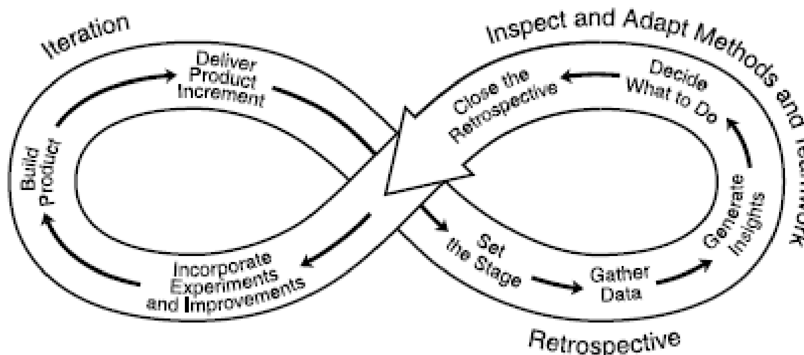


Figure 13: Retrospective steps as part of an iterative cycle. Larsen and Derby [5] p. 6

The hope was that empirical data about the cost (in effort) and quality of changes made to their system would be of use to them and possibly included in their “Gather Data” activity as a quantitative supplement to their retrospective sessions. This could enable them to generate insights into their work that could be used to support decisions and find areas of improvement of both project processes and product.

As the results and analysis of our measures were presented to them, the developers were asked to discuss their use and comment on any factors that might have influenced the results.

Feedback

The developers had found that commenting their modifications with the JIRA identifier had been unproblematic, in fact, they considered it a useful practice in and of itself as it had allowed them to trace changes made to the system more clearly and with greater ease

They expressed that they did not believe that size alone would be a good indicator of effort, since simple refactorings in development tools such as Eclipse would have widespread changes, but require little actual work. They also commented that changes as part of refactoring and restructuring of code to reduce complexity need to be handled differently than normal changes because such changes might require some effort, but would result in the reduction of the cost of changes in the future.

When presented with the concept of measurements on experience a developer has with the files he modifies, they pointed out that having been the only one to modify a part of the system not only makes changes easier to perform, but can be a disadvantage since one can become blind to ones own errors, resulting in lower quality. They felt that ownership and specialization by authors to particular parts of the system could therefore be considered a vulnerability. They also explained that there are some large files they are all familiar with and they all change often without much effort, something that might have an impact on such measurements.

They believed in the assumption that there is a relationship between structure and complexity but that this only one of many factors that decides complexity. Furthermore, the relationship between objects and modules also affects complexity.

Incorporation into methodology

The developers seemed interested in using measurements to complement their qualitative assessments; however, they expressed concerns that introducing measurements would perhaps be too disruptive and remove the focus from the other tasks of their retrospective sessions. Instead, they proposed that such measurements could be used and discussed in a separate session. When asked if they felt that measurements such as these could lead to predictive frameworks, they had no comments one way or the other.

8. Discussion

8.1 The change measures

As Fenton and Niel [8] points out the problem is not to find measures to use, but rather which of the plethora of measures that exists that fits ones need. In this thesis we have explored and used just a few measures, concentrating on those that represent change size and complexity, and are modification aware [2], in addition to measures of skill and architecture. Some of these have been based on our own experience with effort to implement changes, such as number of file types, but most have been based on a compilation of select metrics from various authors working on similar topics of research. A few of these are new and innovative, such as author's inexperience, and structure of code measured by AST. Others have been used in many studies such as LOC counts. The correlation analysis showed us that some measures were indeed related to change effort. These can then potentially be used in estimation models, or in casual models trying to identify cost drivers of software change.

New measured may later, as part of other studies, be implemented into the change model, extracted by the VCSMiner tool and tested for their validity. For instance, the AST measurements seemed to say something about effort without being related to size. Measurements on AST of an even finer level of granularity, for instance differentiating between modification of entities in contrast to deleting or adding them, might be interesting to explore. Future work may also explore other measures of author experience such as measures of how much experience the authors have had *recently* with the files they modify.

Based on the discussion with the developers we gained some insights into the data measured which could help explain some of the results in our analysis: The fact that they often use the developer tool to do automatic refactorings could for instance help explain why there is such a high correlation between the added and deleted external method calls. It is likely that this correlation is influenced by the fact that the developer tool automatically modifies the callers of a method to correspond with changes to the method called, i.e. if the signature of a method is changed, it is also changed in all the calling classes to reflect this change. Another interesting factor that might have influenced our analysis results is the fact that they have some large files that are easy to modify, and are frequently modified by all developers. This can have influenced

how the measured experience correlates to effort .Knowledge about such system and project specific factors is an essential complement to the quantitative measures.

8.2 The VCSMiner

In this section we discuss aspects of the design and implementation of the VCSMiner

8.2.1 AST-based measures

A per measurement basis of creating ASTs causes more overhead for each measurement and one could argue that it would have been better to create ASTs of all the files in the VCS in all their versions in one go, and then use the accumulated ASTs when necessary. On the other hand, the drawbacks of this approach would be that there might (and probably would) be a lot of versions of a file we do not need to extract the AST from, they might simply not be connected to the changes we want to measure. In addition to that, the size of memory needed to hold the AST of all the versions of each and every single file in the VCS could be large, depending on the amount of data in the VCS, and the wait and overhead caused by the initial capture would be significant. Instead of waiting slightly longer for a measurement to complete, you would have to wait a very long time before even starting to measure anything at all. There are pros and cons to both approaches.

8.2.2 CVSParser

The parser extracts data about every file in the VCS and the revisions done to them and creates a set of change-files and owned revisions representing all the files and revisions in the VCS. This may deviate from the concept of a ChangeRequest owning a set of ChangeFiles as representation of changes to files *initiated by it*. Strictly speaking, we should not collect information about *all* the files, only the information about the files affected by and the revisions implementing a ChangeRequest in the ECMS. We chose to deviate from the conceptual model in this way because following it would have caused a lot of extra overhead in execution time, process power and network use, complicated by congestion, packet loss and other issues. It would force the tool to parse cvs log data every time we wanted to collect information about a new ChangeRequest. Instead we collected all the information available in one go, and let the controller class (the *no.simula.vcsm.controller.Miner* class) sort through the pooled information on changes to files and assign the correct ChangeFile objects (and its Revisions) to a ChangeRequest for each change we want to measure.

8.2.3 Measurements on Multiple Changes

We wanted to be able to do measurements on multiple changes in one go. However, in practice, it required too much memory to hold every ChangeRequest object, its ASTs and Revision for a large amount of changes. Instead we decided to capture only the results of the measurements on each CR and store these for each CR we wanted to measure. Unfortunately, this makes it impossible to let the ChangeRequests write measurements to files themselves, and we had to let

the controller class to do this for them, taking the output and writing it to a file directly. This broke the architectural layering of the model, but proved necessary.

9. Conclusions and Further Work

The tool that was developed proved to be a useful alternative to manual measurements and scripts, it requires much less work and allows for more flexibility in the execution of the measurements, allowing extraction of multiple measurements and the specification of which file types to include. Implementing support for a new versioning system was relatively easy. The creation of a user interface was not given much priority in this work as we wanted to concentrate on its functional usefulness for verification and validation of measures and tool. This is an area for future improvement.

Using a UML model to visualize the change domain in an ECMS framework proved very beneficial. Its use can be recommended to gain understanding and overview, as it helps with making sure that measures are correctly translated from the conceptual level to their implementation in tools.

Some of the measures of structure were significantly correlated to effort without being related to the size measurements. Hence, using such measures in addition to size measures may help in constructing better estimation models and causal models. Investigation into even finer grained measures of structure could be beneficial. By implementing techniques such as the ones used by Fluri and Gall [4] to recreate the modifications done to the structure, could provide more accurate measurements, and let us identify not only added or deleted structural entities but also those that has been modified.

Author experience showed a low correlation to effort in the analysis; however, the p value is far below 0.05 so we can still say there is a correlation. There is also a lot of evidence in other studies, such as Gîrba et al. [16] and Curtis [18], that author experience is important. These measures is therefore recommended for use in the future, and perhaps even expand upon with measures such an authors recent experience with the files affected by a change. Number of authors has, according to the analysis low to no correlation to effort, as have some of the structural measures such as added private class variables, deleted public method declarations, deleted internal method calls, deleted private method declarations and experience measure inexperience. However, the p-value of all these are distinctly above 0.05 and therefore the low correlation could just be a coincidence. Therefore it is recommended that more measurements and on other projects need to be done to fully evaluate their usefulness. The analysis is based on the data from a single software project, so although we have measured the modifications of several change requests, other factors specific to this project might have an impact on the results.

Both miner and model have been useful in extracting and defining measures and the analysis proved that there are good indications that the measures found is worth further investigation. The conclusion is therefore that model, measures and tool should be expanded upon and implemented in a larger framework that can provide fully automatic measurements and analysis. The analysis of the measurements should be expanded to other projects as well in order to establish their effort assessing capabilities on a broader scale. Fresh LOC and all the

other measures that are just under 0.3 are borderline cases and can benefit from additional analysis to better validate their usefulness.

References

1. Mockus, A.W., David M., *Predicting Risk of Software Changes.* Bell Laboratories Technical Journal, 2000. **5**: p. 169-180.
2. German, D.M.H., A., *Measuring fine-grained change in software: towards modification-aware change metrics.* Software Metrics, 2005. 11th IEEE International Symposium, 2005.
3. Lehman, M.M.R., J.F., *Metrics of Software Evolution as Effort Predictors - A Case Study.* Software Maintenance, 2000. Proceedings. International Conference on, 2000: p. 163-172.
4. Fluri, B.G., Harald C., *Classifying Change Types for Qualifying Change Couplings.* 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006: p. 35-45.
5. Derby, E.L., Diana, *Agile Retrospectives: Making Good Teams Great*. The Pragmatic Programmers. 2006: Pragmatic Bookshelf.
6. German, D.M., A., *Automating the Measurement of Open Source Projects.* 2004.
7. Atlassian. *JIRA - Because you've got issues.* 2006 [cited 2006 December 4th]; A bug tracking, issue tracking and project management application]. Available from: http://www.atlassian.com/software/jira/.
8. Fenton, N.E.N., Martin. *Software metrics: roadmap*. in *International Conference on Software Engineering*. 2000. Limerick, Ireland ACM.
9. Ramil, J.F., Lehman, M.M. *Cost Estimation and Evolvability Monitoring for Software Evolution Processes*. in *WESS 2000 Workshop on Empirical Studies of Software Maintenance*. 2000. San Jose, CA.
10. Canfora, G.C., L., *Impact Analysis by Mining Software and Change Request Repositories.* IEEE International Software Metrics Symposium (METRICS 2005), 2005.
11. Gentleware. *Poseidon for UML*. 2006 [cited 2006 December 7th]; A powerful UML modelling tool dor analysis design and documentation in the software development process.]. Available from: http://www.gentleware.com/.
12. Boger, M., *Java in Distributed Systems*. 2001: John Wiley & Sons, Ltd.
13. Deitel, H.M., Deitel, P.J, Santry, S., *Advanced Java2 Platform: How to Program*. 1st ed. 2001: Prentice-Hall, Inc.
14. German, D.M., *An empirical study of fine-grained software modifications.* Empirical Software Engineering, 2006. **11**: p. 369-393.
15. Keir Mierle, K.L., Sam Roweis & Greg Wilson, *Mining Student CVS Repositories for Performance Indicators.* MSR, 2005: p. 5.
16. Gîrba, T., Kuhn, A., Seeberger, M., Ducasse S., *How Developers Drive Software Evolution.* Proceedings of the 2005 Eight International Workshop on Principles of Software Evolution, 2005: p. 113-122.
17. Fenton, N.E.P., Shari L., *Software Metircs: A Rigorous & Practical Approach*. 2nd ed. 1997: PWS Publishing Company.
18. Curtis, B., *Fifteen years of psychology in software engineering: Individual differences and cognitive science.* IEEE Press, 1984: p. 97-106.
19. Larman, C., *Applying UML and Patterns*. 2nd ed. 2002: Prentice-Hall, Inc. 627.
20. Systems, A.S. *JIRA - Because you've got issues.* 2006 [cited 2006 December 4th]; A bug tracking, issue trancking and project management application]. Available from: http://www.atlassian.com/software/jira/.

21. Microsystems, S. *About Java Technology*. [website] 2006 24. November, 2006 [cited 2006 24.11]; Sun's "About Java Technology" page]. Available from: http://www.sun.com/java/about/.
22. Karl Fogel, M.B., *Open Source Development with CVS, 3rd Edition*. 2003: Paraglyph Press.
23. Per Cederqvist, e.a., *The CVS manual --- Version Management with CVS*. 2003: Network Theory Limited.
24. Keir Mierle, K.L., Sam Roweis & Greg Wilson, *CVS Data Extraction and Analysis: A Case Study*. 2004.
25. Heckel, *A technique for isolating differences between files.* Communications of the ACM, 1978. **21**(4): p. 264-268.
26. J. W. Hunt, M.D.M., *An Algorithm for Differential File Comparison.* Bell Laboratories Computing Science Technical Report, 1976. **41**.
27. Meyers, E.W., *An O(ND) Difference Algorithm and Its Variations.* Algorithmica, 1986. **1**(2): p. 251-266.
28. Collins-Sussman, B., Fizpatrick, B.W., Pilato, M.C., *Version Control with Subversion: For Subversion 1.3*. 2006: TBA.
29. Eclipse. *Legal Resources*. [webpage] 2006 [cited 2006 28.11]; Legal resource page for the Eclipse Foundation]. Available from: http://www.eclipse.org/legal/.
30. SVNKit. *SVNKit The only pure Java Subversion library in the world!* 2006 30 November [cited 2006 30.11]; Available from: http://svnkit.com/index.html.

Appendix

Abbreviations, Terms and Acronyms

ECMS: Extended Change Management System. In this thesis represented by a bug/issue tracking tool where requests for changes are recorded and a version control system where the files and modifications themselves are stored.

VCS: Version Control System. A system that keeps track of changes made to files and the modifications that has been done to them, when and by whom.

JIRA: A project management, bug and issue-tracking tool. Where the information about requests for changes are stored.

Bugzilla: A bug-tracking tool featured a lot in related work.

CR: ChangeRequest, entity representing a record in a bug or issue-tracking tool.

RCS: Revision Control System, another name for VCS.

UML: Unified Modelling Language, a widely used modelling language often used in analysis and design.

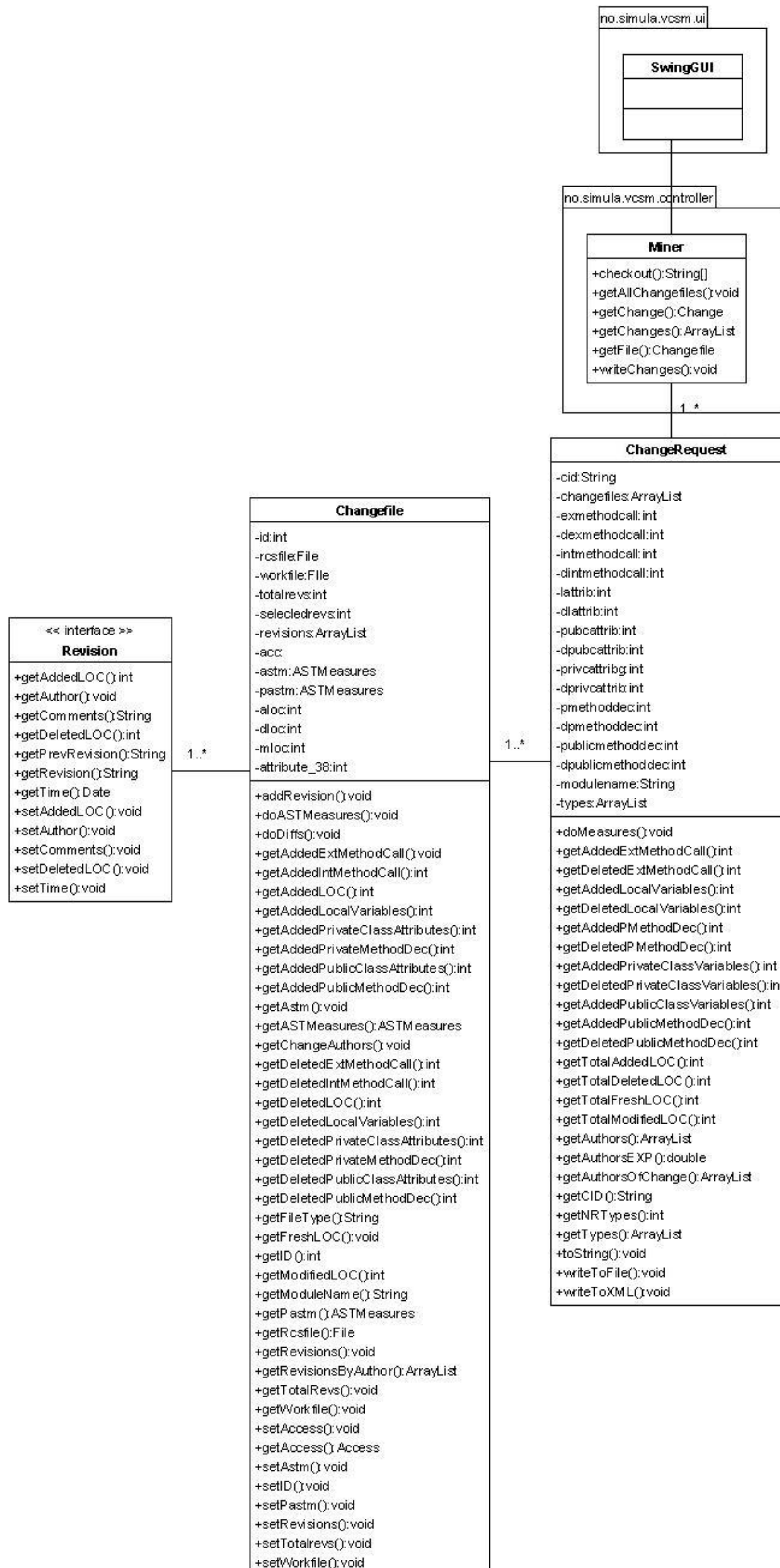
AST: Abstract Syntax Tree. A representation of the abstract syntax of code mapped to a tree structure.

SSH: Secure Shell. A protocol that allows for a secure channel between a local and a remote computer.

AM: Analysis Measures. Measures that will be used in an analysis context.

IM: Implementation Measures. Measures with which the Analysis Measures are implemented.

Attachment 1: Class diagram of domain layer with attributes and methods.


no.simula.vcsm.ui
SwingGUI
no.simula.vcsm.controller
Miner
+checkout():String[]
+getAllChangefiles():void
+getChange():Change
+getChanges():ArrayList
+getFile():Changefile
+writeChanges():void
1..*
ChangeRequest
-cid:String
-changefiles:ArrayList
-exmethodcall:int
-dexmethodcall:int
-intmethodcall:int
-dintmethodcall:int
-lattrib:int
-dlattrib:int
-pubcattrib:int
-dpubcattrib:int
-privcattribg:int
-dprivcattrib:int
-pmethoddec:int
-dpmethoddec:int
-publicmethoddec:int
-dpublicmethoddec:int
-modulename:String
-types:ArrayList
+doMeasures():void
+getAddedExtMethodCall():int
+getDeletedExtMethodCall():int
+getAddedLocalVariables():int
+getDeletedLocalVariables():int
+getAddedPMethodDec():int
+getDeletedPMethodDec():int
+getAddedPrivateClassVariables():int
+getDeletedPrivateClassVariables():int
+getAddedPublicClassVariables():int
+getAddedPublicMethodDec():int
+getDeletedPublicMethodDec():int
+getTotalAddedLOC():int
+getTotalDeletedLOC():int
+getTotalFreshLOC():int
+getTotalModifiedLOC():int
+getAuthors():ArrayList
+getAuthorsEXP():double
+getAuthorsOfChange():ArrayList
+getCID():String
+getNRTypes():int
+getTypes():ArrayList
+toString():void
+writeToFile():void
+writeToXML():void
1..*
Changefile
-id:int
-rcsfile:File
-workfile:File
-totalrevs:int
-selecledrevs:int
-revisions:ArrayList
-acc:
-astm:ASTMeasures
-pastm:ASTMeasures
-aloc:int
-dloc:int
-mloc:int
-attribute_38:int
+addRevision():void
+doASTMeasures():void
+doDiffs():void
+getAddedExtMethodCall():void
+getAddedIntMethodCall():int
+getAddedLOC():int
+getAddedLocalVariables():int
+getAddedPrivateClassAttributes():int
+getAddedPrivateMethodDec():int
+getAddedPublicClassAttributes():int
+getAddedPublicMethodDec():int
+getAstm():void
+getASTMeasures():ASTMeasures
+getChangeAuthors():void
+getDeletedExtMethodCall():int
+getDeletedIntMethodCall():int
+getDeletedLOC():int
+getDeletedLocalVariables():int
+getDeletedPrivateClassAttributes():int
+getDeletedPrivateMethodDec():int
+getDeletedPublicClassAttributes():int
+getDeletedPublicMethodDec():int
+getFileType():String
+getFreshLOC():void
+getID():int
+getModifiedLOC():int
+getModuleName():String
+getPastm():ASTMeasures
+getRcsfile():File
+getRevisions():void
+getRevisionsByAuthor():ArrayList
+getTotalRevs():void
+getWorkfile():void
+setAccess():void
+getAccess():Access
+setAstm():void
+setID():void
+setPastm():void
+setRevisions():void
+setTotalrevs():void
+setWorkfile():void
1..*
<< interface >>
Revision
+getAddedLOC():int
+getAuthor():void
+getComments():String
+getDeletedLOC():int
+getPrevRevision():String
+getRevision():String
+getTime():Date
+setAddedLOC():void
+setAuthor():void
+setComments():void
+setDeletedLOC():void
+setTime():void

Attachment 2: Correlation between measurements

Measurement	AddedLOC	DeletedLOC	ModifiedLOC	FreshLOC	AffectedFiles	NROfTypes	Authors	EXP
AddedLOC		0,86637	0,65007	0,22906	0,81545	0,46583	0,38268	0,08623
AddedLOC	1	<.0001	<.0001	0,071	<.0001	0,0001	0,002	0,5016
DeletedLOC	0,86637		0,63275	0,31274	0,71407	0,40857	0,38168	0,12294
DeletedLOC	<.0001	1	<.0001	0,0126	<.0001	0,0009	0,002	0,3371
ModifiedLOC	0,65007	0,63275		0,26353	0,7989	0,54663	0,40041	0,13777
ModifiedLOC	<.0001	<.0001	1	0,0369	<.0001	<.0001	0,0011	0,2816
FreshLOC	0,22906	0,31274	0,26353		0,27438	0,2734	0,13144	-0,135
FreshLOC	0,071	0,0126	0,0369	1	0,0295	0,0301	0,3045	0,2915
AffectedFiles	0,81545	0,71407	0,7989	0,27438		0,68963	0,41368	0,07312
AffectedFiles	<.0001	<.0001	<.0001	0,0295	1	<.0001	0,0008	0,569
NROfTypes	0,46583	0,40857	0,54663	0,2734	0,68963		0,58468	0,17557
NROfTypes	0,0001	0,0009	<.0001	0,0301	<.0001	1	<.0001	0,1687
Authors	0,38268	0,38168	0,40041	0,13144	0,41368	0,58468		0,32072
Authors	0,002	0,002	0,0011	0,3045	0,0008	<.0001	1	0,0104
EXP	0,08623	0,12294	0,13777	-0,135	0,07312	0,17557	0,32072	
EXP	0,5016	0,3371	0,2816	0,2915	0,569	0,1687	0,0104	1
IEXP	-0,12145	-0,14787	-0,10322	-0,2036	-0,1846	-0,16076	-0,02302	0,30425
IEXP	0,343	0,2474	0,4208	0,1095	0,1475	0,2082	0,8578	0,0153
AddedLVars	0,63684	0,60175	0,5762	0,10883	0,62998	0,2679	0,07451	0,08448
AddedLVars	<.0001	<.0001	<.0001	0,3958	<.0001	0,0338	0,5616	0,5103
DeletedLVars	0,54905	0,56517	0,59847	0,10568	0,5913	0,23296	0,18497	-0,01209
DeletedLVars	<.0001	<.0001	<.0001	0,4097	<.0001	0,0661	0,1467	0,9251
AddedPubClassVars	0,38905	0,33476	0,30417	0,25963	0,35892	0,09601	-0,10968	-0,1869
AddedPubClassVars	0,0016	0,0073	0,0154	0,0399	0,0039	0,4541	0,3922	0,1425
DeletedPubClassVars	0,37828	0,38848	0,39432	0,15488	0,39482	0,25513	-0,11289	-0,10338
DeletedPubClassVars	0,0022	0,0017	0,0014	0,2255	0,0014	0,0436	0,3784	0,4201
AddedPrivClassVars	0,27164	0,2728	0,31718	0,43953	0,3587	0,20118	0,26691	-0,1248
AddedPrivClassVars	0,0313	0,0305	0,0113	0,0003	0,0039	0,1139	0,0345	0,3298
DeletedPrivClassVars	0,2787	0,33526	0,45546	0,1544	0,42288	0,26141	0,26099	-0,07111
DeletedPrivClassVars	0,027	0,0072	0,0002	0,227	0,0006	0,0385	0,0388	0,5797
AddedPrivMethodDec	0,38976	0,44235	0,26217	0,09643	0,32605	0,02899	0,143	0,00443
AddedPrivMethodDec	0,0016	0,0003	0,0379	0,4521	0,0091	0,8216	0,2636	0,9725
DeletedPrivMethodDec	0,21196	0,28381	0,32831	-0,09503	0,22409	-0,08285	0,02553	-0,05174
DeletedPrivMethodDec	0,0954	0,0242	0,0086	0,4588	0,0775	0,5186	0,8425	0,6871
AddedPubMethodDec	0,36724	0,38738	0,22038	0,29445	0,30369	0,09014	0,13231	0,01642
AddedPubMethodDec	0,0031	0,0017	0,0826	0,0192	0,0155	0,4823	0,3013	0,8984
DeletedPubMethodDec	0,37479	0,5136	0,38497	0,22673	0,38638	0,09235	0,10515	0,0161
DeletedPubMethodDec	0,0025	<.0001	0,0018	0,0739	0,0018	0,4716	0,4121	0,9003
AddedExtMethodCall	0,71621	0,61813	0,648	0,10632	0,69083	0,25949	0,29788	0,11151
AddedExtMethodCall	<.0001	<.0001	<.0001	0,4069	<.0001	0,04	0,0177	0,3843
DeletedExtMethodCall	0,53919	0,53186	0,62234	0,09923	0,53263	0,09429	0,06435	0,05163
DeletedExtMethodCall	<.0001	<.0001	<.0001	0,4391	<.0001	0,4623	0,6163	0,6878
AddedIntMethodCall	0,68186	0,71583	0,60564	0,23035	0,68351	0,41863	0,27938	0,1008
AddedIntMethodCall	<.0001	<.0001	<.0001	0,0693	<.0001	0,0006	0,0266	0,4318
DeletedIntMethodCall	0,47453	0,61373	0,50609	0,25641	0,49338	0,29498	0,22035	0,14386
DeletedIntMethodCall	<.0001	<.0001	<.0001	0,0425	<.0001	0,0189	0,0827	0,2607

Notice: First row for each measurement is show the rho, the second (light grey) the P value.

Measurement	IEXP	AddedLVars	DeletedLVars	AddedPubClassVars	DeletedPubClassVars	AddedPrivClassVars	DeletedPrivClassVars	AddedPrivMethodDec
AddedLOC	-0,12145	0,63684	0,54905	0,38905	0,37828	0,27164	0,2787	0,38976
AddedLOC	0,343	<.0001	<.0001	0,0016	0,0022	0,0313	0,027	0,0016
DeletedLOC	-0,14787	0,60175	0,56517	0,33476	0,38848	0,2728	0,33526	0,44235
DeletedLOC)	0,2474	<.0001	<.0001	0,0073	0,0017	0,0305	0,0072	0,0003
ModifiedLOC	-0,10322	0,5762	0,59847	0,30417	0,39432	0,31718	0,45546	0,26217
ModifiedLOC	0,4208	<.0001	<.0001	0,0154	0,0014	0,0113	0,0002	0,0379
FreshLOC	-0,2036	0,10883	0,10568	0,25963	0,15488	0,43953	0,1544	0,09643
FreshLOC	0,1095	0,3958	0,4097	0,0399	0,2255	0,0003	0,227	0,4521
AffectedFiles	-0,1846	0,62998	0,5913	0,35892	0,39482	0,3587	0,42288	0,32605
AffectedFiles	0,1475	<.0001	<.0001	0,0039	0,0014	0,0039	0,0006	0,0091
NROfTypes	-0,16076	0,2679	0,23296	0,09601	0,25513	0,20118	0,26141	0,02899
NROfTypes	0,2082	0,0338	0,0661	0,4541	0,0436	0,1139	0,0385	0,8216
Authors	-0,02302	0,07451	0,18497	-0,10968	-0,11289	0,26691	0,26099	0,143
Authors	0,8578	0,5616	0,1467	0,3922	0,3784	0,0345	0,0388	0,2636
EXP	0,30425	0,08448	-0,01209	-0,1869	-0,10338	-0,1248	-0,07111	0,00443
EXP	0,0153	0,5103	0,9251	0,1425	0,4201	0,3298	0,5797	0,9725
IEXP	1	-0,07796	-0,23192	-0,12545	0,00227	-0,17414	-0,10793	-0,06802
IEXP		0,5436	0,0674	0,3273	0,9859	0,1723	0,3998	0,5963
AddedLVars	-0,07796	1	0,61174	0,3744	0,35955	0,28603	0,37287	0,44525
AddedLVars	0,5436		<.0001	0,0025	0,0038	0,0231	0,0026	0,0003
DeletedLVars	-0,23192	0,61174	1	0,35526	0,35639	0,31633	0,34603	0,45896
DeletedLVars	0,0674	<.0001		0,0043	0,0041	0,0115	0,0055	0,0002
AddedPubClassVars	-0,12545	0,3744	0,35526	1	0,59233	0,30131	0,27198	0,25735
AddedPubClassVars	0,3273	0,0025	0,0043		<.0001	0,0164	0,0311	0,0417
DeletedPubClassVars	0,00227	0,35955	0,35639	0,59233	1	-0,00286	0,25785	-0,08431
DeletedPubClassVars	0,9859	0,0038	0,0041	<.0001		0,9823	0,0413	0,5112
AddedPrivClassVars	-0,17414	0,28603	0,31633	0,30131	-0,00286	1	0,52165	0,34906
AddedPrivClassVars	0,1723	0,0231	0,0115	0,0164	0,9823		<.0001	0,005
DeletedPrivClassVars	-0,10793	0,37287	0,34603	0,27198	0,25785	0,52165	1	0,29524
DeletedPrivClassVars	0,3998	0,0026	0,0055	0,0311	0,0413	<.0001		0,0188
AddedPrivMethodDec	-0,06802	0,44525	0,45896	0,25735	-0,08431	0,34906	0,29524	1
AddedPrivMethodDec	0,5963	0,0003	0,0002	0,0417	0,5112	0,005	0,0188	
DeletedPrivMethodDec	-0,05686	0,21893	0,27273	0,16384	0,39306	-0,16243	0,22127	-0,09505
DeletedPrivMethodDec	0,6581	0,0847	0,0306	0,1995	0,0014	0,2034	0,0814	0,4587
AddedPubMethodDec	-0,30755	0,39	0,32785	0,06213	-0,12559	0,1353	0,03091	0,42247
AddedPubMethodDec	0,0142	0,0016	0,0087	0,6286	0,3267	0,2904	0,81	0,0006
DeletedPubMethodDec	-0,24775	0,31075	0,44044	0,22335	0,31773	0,1878	0,56603	0,2413
DeletedPubMethodDec	0,0503	0,0132	0,0003	0,0785	0,0112	0,1405	<.0001	0,0568
AddedExtMethodCall	-0,14955	0,76018	0,679	0,42429	0,35394	0,32947	0,35221	0,41907
AddedExtMethodCall	0,2421	<.0001	<.0001	0,0005	0,0044	0,0084	0,0046	0,0006
DeletedExtMethodCall	-0,14885	0,63757	0,79028	0,37189	0,30241	0,26567	0,32982	0,38716
DeletedExtMethodCall	0,2443	<.0001	<.0001	0,0027	0,016	0,0353	0,0083	0,0017
AddedIntMethodCall	-0,13015	0,66546	0,49161	0,42519	0,43964	0,21218	0,40192	0,51961
AddedIntMethodCall	0,3093	<.0001	<.0001	0,0005	0,0003	0,095	0,0011	<.0001
DeletedIntMethodCall	-0,14095	0,44725	0,515	0,48671	0,49643	0,31389	0,30524	0,30185
DeletedIntMethodCall	0,2705	0,0002	<.0001	<.0001	<.0001	0,0122	0,015	0,0162

Measurement	DeletedPrivMethodDec	AddedPubMethodDec	DeletedPubMethodDec	AddedExtMethodCall	DeletedExtMethodCall	AddedIntMethodCall	DeletedIntMethodCall
AddedLOC	0,21196	0,36724	0,37479	0,71621	0,53919	0,68186	0,47453
AddedLOC	0,0954	0,0031	0,0025	<.0001	<.0001	<.0001	<.0001
DeletedLOC	0,28381	0,38738	0,5136	0,61813	0,53186	0,71583	0,61373
DeletedLOC	0,0242	0,0017	<.0001	<.0001	<.0001	<.0001	<.0001
ModifiedLOC	0,32831	0,22038	0,38497	0,648	0,62234	0,60564	0,50609
ModifiedLOC	0,0086	0,0826	0,0018	<.0001	<.0001	<.0001	<.0001
FreshLOC	-0,09503	0,29445	0,22673	0,10632	0,09923	0,23035	0,25641
FreshLOC	0,4588	0,0192	0,0739	0,4069	0,4391	0,0693	0,0425
AffectedFiles	0,22409	0,30369	0,38638	0,69083	0,53263	0,68351	0,49338
AffectedFiles	0,0775	0,0155	0,0018	<.0001	<.0001	<.0001	<.0001
NROfTypes	-0,08285	0,09014	0,09235	0,25949	0,09429	0,41863	0,29498
NROfTypes	0,5186	0,4823	0,4716	0,04	0,4623	0,0006	0,0189
Authors	0,02553	0,13231	0,10515	0,29788	0,06435	0,27938	0,22035
Authors	0,8425	0,3013	0,4121	0,0177	0,6163	0,0266	0,0827
EXP	-0,05174	0,01642	0,0161	0,11151	0,05163	0,1008	0,14386
EXP	0,6871	0,8984	0,9003	0,3843	0,6878	0,4318	0,2607
IEXP	-0,05686	-0,30755	-0,24775	-0,14955	-0,14885	-0,13015	-0,14095
IEXP	0,6581	0,0142	0,0503	0,2421	0,2443	0,3093	0,2705
AddedLVars	0,21893	0,39	0,31075	0,76018	0,63757	0,66546	0,44725
AddedLVars	0,0847	0,0016	0,0132	<.0001	<.0001	<.0001	0,0002
DeletedLVars	0,27273	0,32785	0,44044	0,679	0,79028	0,49161	0,515
DeletedLVars	0,0306	0,0087	0,0003	<.0001	<.0001	<.0001	<.0001
AddedPubClassVars	0,16384	0,06213	0,22335	0,42429	0,37189	0,42519	0,48671
AddedPubClassVars	0,1995	0,6286	0,0785	0,0005	0,0027	0,0005	<.0001
DeletedPubClassVars	0,39306	-0,12559	0,31773	0,35394	0,30241	0,43964	0,49643
DeletedPubClassVars	0,0014	0,3267	0,0112	0,0044	0,016	0,0003	<.0001
AddedPrivClassVars	-0,16243	0,1353	0,1878	0,32947	0,26567	0,21218	0,31389
AddedPrivClassVars	0,2034	0,2904	0,1405	0,0084	0,0353	0,095	0,0122
DeletedPrivClassVars	0,22127	0,03091	0,56603	0,35221	0,32982	0,40192	0,30524
DeletedPrivClassVars	0,0814	0,81	<.0001	0,0046	0,0083	0,0011	0,015
AddedPrivMethodDec	-0,09505	0,42247	0,2413	0,41907	0,38716	0,51961	0,30185
AddedPrivMethodDec	0,4587	0,0006	0,0568	0,0006	0,0017	<.0001	0,0162
DeletedPrivMethodDec		0,13907	0,48509	0,26287	0,25711	0,20559	0,31353
DeletedPrivMethodDec	**1**	0,277	<.0001	0,0374	0,0419	0,106	0,0123
AddedPubMethodDec	0,13907		0,21861	0,33634	0,25179	0,52791	0,1679
AddedPubMethodDec	0,277	**1**	0,0852	0,007	0,0465	<.0001	0,1884
DeletedPubMethodDec	0,48509	0,21861		0,37682	0,43263	0,42706	0,66126
DeletedPubMethodDec	<.0001	0,0852	**1**	0,0023	0,0004	0,0005	<.0001
AddedExtMethodCall	0,26287	0,33634	0,37682		0,81586	0,58903	0,48095
AddedExtMethodCall	0,0374	0,007	0,0023	**1**	<.0001	<.0001	<.0001
DeletedExtMethodCall	0,25711	0,25179	0,43263	0,81586		0,44704	0,44142
DeletedExtMethodCall	0,0419	0,0465	0,0004	<.0001	**1**	0,0002	0,0003
AddedIntMethodCall	0,20559	0,52791	0,42706	0,58903	0,44704		0,57765
AddedIntMethodCall	0,106	<.0001	0,0005	<.0001	0,0002	**1**	<.0001
DeletedIntMethodCall	0,31353	0,1679	0,66126	0,48095	0,44142	0,57765	
DeletedIntMethodCall	0,0123	0,1884	<.0001	<.0001	0,0003	<.0001	**1**