Analysis of Signature Change Patterns

Sunghun Kim, E. James Whitehead, Jr., Jennifer Bevan
Dept. of Computer Science
Baskin Engineering
University of California, Santa Cruz
Santa Cruz, CA 95060 USA
{hunkim, ejw, jbevan}@cs.ucsc.edu

ABSTRACT

Software continually changes due to performance improvements, new requirements, bug fixes, and adaptation to a changing operational environment. Common changes include modifications to data definitions, control flow, method/function signatures, and class/file relationships. Signature changes are notable because they require changes at all sites calling the modified function, and hence as a class they have more impact than other change kinds.

We performed signature change analysis over software project histories to reveal multiple properties of signature changes, including their kind, frequency, and evolution patterns. These signature properties can be used to alleviate the impact of signature changes. In this paper we introduce a taxonomy of signature change kinds to categorize observed changes. We report multiple properties of signature changes based on an analysis of eight prominent open source projects including the Apache HTTP server, GCC, and Linux 2.5 kernel.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *Product metrics*, K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms

Measurement, Experimentation

Keywords

Software Evolution, Signature Change Patterns, Software Evolution Path

1. INTRODUCTION

Software continually changes due to performance improvements, new requirements, bug fixes, and adaptation to a changing operational environment [1]. Software changes include function body modification, local variable renaming, moving functions from one file to another, and function signature changes [2]. Among these changes, function signature changes have a significant impact on parts of the source code that use the changed functions. Most signature changes cause a signature mismatch problem. Understanding the character and evolution patterns of function signature changes is important to researchers concerned with alleviating the impact of signature changes.

MSR '05, May 17, 2005, Saint Louis, Missouri, USA

Others have observed code changes, though none have examined signature changes at the same level of detail. Kung et al. identified kinds of code changes [2] and Counsell et al. discussed the trends of changes in Java code [3]. Both of them identified large granularity change kinds, such as method body changes, method addition, method deletion and whether the signature changed. Their categorization of changes is useful for understanding software changes in overview. Our analysis of signature changes is motivated by the goal of eventually providing automated support for fixing signature mismatches, and for this we need a very fine-grain understanding and characterization of signature changes. Previous work did not examine signature changes at this level of detail, being concerned only with whether the signature did, or did not, change, but not what kind of change.

We focus on fine-grain changes in function signatures, categorizing them based on whether they increase, decrease, or do not modify the data flow between caller and callee. Within these broad categories, change kinds are further refined. We show the properties of function signature change patterns by answering the following research questions: How often do signatures change? What are the common signature change kinds? How often does each kind appear? Do they have a common evolution pattern?

The answers, along with analysis of the results, can be used to predict future signature changes, provide automatic change accommodation algorithms, develop glue code generators, or develop refactoring algorithms.

We analyzed eight prominent open source projects listed in Table 1. These eight open source projects are written in the C programming language. For our analysis, we used Kenyon, a data extraction, preprocessing, and storage backend designed to facilitate software evolution research [4]. Using Kenyon, we checked out all revisions or copied all releases of source code from each project, and extracted function signatures. We grouped signatures by function name, and observed the changes over revisions or releases to find properties of signature changes. We implemented an automatic signature change kind identification tool, but some change patterns are not automatically identifiable, such as concept splitting and merging. We also compared the number of signature changes over all functions to find the frequency of each signature change kind. Finally we looked for sequence patterns in the common evolution paths of function signature changes.

The remaining sections of the paper are as follows: In Section 2, we describe our analysis process with detailed information from the open source projects we analyzed. Sections 3, 4, 5, and 6 provide answers to our research questions. We discuss the limitations of our analysis in Section 7, and conclude in Section 8.

Table 1. Open source projects we analyzed. LOC indicates number of lines in .h and .c source files, including comments. The period shows the project history period for projects for which we directly accessed the SCM repository, otherwise we list release numbers. The number of revisions indicates the number of revisions we extracted or the number of releases we analyzed.

Project	Software type	LOC	SCM	Period/Releases	# of revisions/releases
Apache Portable Runtime (APR)	Portable C library	72,630	Subversion	Jan 1999 ~ Jan 2005	5832 revisions
Apache HTTP 1.3 (Apache 1.3)	HTTP server	116,393	Subversion	Jan 1996 ~ Jan 2005	7508 revisions
Apache HTTP 2.0 (Apache 2)	HTTP server modules	104,417	CVS	Jul 1999 ~ Aug 2003	3877 revisions
Subversion	SCM software	183,740	Subversion	Aug 2001 ~ Feb 2005	5886 revisions
CVS	SCM software	62,415	CVS	Dec 1994 ~ Sep 2003	2873 revisions
Linux Kernel 2.5 (Linux)	Linux OS	5,140,625	N/A	2.5.1 ~ 2.5.75	75 releases
GCC	C/C++ compiler	506,931	N/A	1.35, 1.36, …, 2.7.2	15 releases
Sendmail	SMTP server	127,733	N/A	8.7.6, 8.8.3, …, 8.13.3	37 releases

2. ANALYSIS PROCESSES

We analyzed eight open source projects, listed in Table 1, using the Kenyon system. Kenyon checks out all revisions from a SCM repository and invokes a fact extractor we implemented to extract function signatures. The extracted signatures are grouped by function names. The grouped signatures are ordered by revisions and stored in a signature change history file.

For the projects we analyzed, the revision history was stored using either the CVS or Subversion SCM system. An important issue in software evolution research is the extraction of logical transactions from the SCM repository. Since Subversion assigns a revision number per commit, there is no need to recover transactions for Subversion-managed projects [5]. CVS does not keep the original transaction information, usually requiring a process of transaction recovery [6]. Kenyon provides CVS transaction recovery using the Sliding Time Windows algorithm [4, 6]. Recently, the Apache Software Foundation (ASF) changed its SCM repository to Subversion from CVS using the cvs2svn converting tool. We analyzed some ASF projects, including Apache 1.3 and APR, whose repositories were converted. Since the cvs2svn tool uses the fixed time window algorithm [6] to convert CVS data for Subversion, using the converted data won't affect our analysis results.

We manually observed the signature change history file to identify common signature change kinds. After analyzing the signature change history files from various open source projects, we found the common change kinds shown in Table 3. While most of the change patterns can be automatically identified by a static software analysis, some change kinds, such as concept merging/splitting changes are not automatically identifiable, requiring project knowledge concerning the project and parameter concepts. We implemented an automatic signature change kind identifier that reads a signature change history file, and annotates the file based on the identified kinds. After the signature change history file annotation, we calculate the frequency of each change kind. We also examine the sequence of signature change kinds of a given function to see if there was a common pattern in the signature evolution. The results of our analysis are presented in following sections.

3. SIGNATURE CHANGE KINDS

Before presenting our results, we describe our fine-grain taxonomy of signature change kinds. First we define the basic elements of a function signature: *parameter*, *argument*, *return parameter*, and the *signature*.

The *modifier* indicates a data type modifier such as const, register, and static. A *type* is the data type of a parameter, and name indicates the parameter name. The *array/pointer* is the count of * or [] when a parameter is an array or pointer type. This represents both the array/pointer type and its dimension. Using these basic definitions, we now identify and define signature change kinds. In the remainder of the definitions, we use the subscript $_{new}$ to indicate a later revision and $_{old}$ a previous revision. If we omit the equality of elements, assume the other elements are the same. For example, in Definition 2 we define *N* if the $name_{old}$ and $name_{new}$ are different. We assume all other elements such as *type* and *modifier* are the same.

Definition 1 (Parameter, Argument, Return parameter, Signature)

Parameter **Param** ≡ {modifier, type, name, array/pointer, order}
Argument **Arg** ≡ a set of zero or more **Param**
Return parameter **R** ≡ {modifier, type, array/pointer}
Signature **S** ≡ {**R**, function name, **Arg**}

Definition 2 (Name change)

Function name change FN ≡ function $name_{new}$ ≠ function $name_{old}$
Parameter name change N ≡ $name_{new}$ ≠ $name_{old}$

The name change category has two kinds: function name change and parameter name change. Table 2 shows an example of parameter name changes. A parameter name change does not introduce a signature mismatch problem since the parameter name is used internal to the function. However, parameter name changes may cause semantic errors. For example, as shown in Table 2, if the change of parameter from '*service_name*' to '*display_name*' indicates a change in parameter meaning, call sites will compile without error, but the software may not work as expected due to the change in meaning.

Table 2. A parameter name change in Apache 1.3, os/win32/service.c file, ValidService function. The old version is on top, the new version is on bottom. Changes between versions are shown in bold.

BOOL ← char *service_name
BOOL ← **char *display_name**

Definition 3 (Ordering change)

Order ≡ the position of an argument
Ordering change O ≡ $order_{new}$ ≠ $order_{old}$
Only ordering change o ≡ O and $|\mathbf{Arg}_{new}| = |\mathbf{Arg}_{old}|$
Ordering change by addition OA ≡ O and $|\mathbf{Arg}_{new}| > |\mathbf{Arg}_{old}|$
Ordering change by deletion OD ≡ O and $|\mathbf{Arg}_{new}| < |\mathbf{Arg}_{old}|$

The parameter ordering changes occur when the order of two or more parameters has been changed. The typical motivation behind these changes is parameter order consistency with other function

signatures. Sometimes adding or deleting parameters causes signature ordering changes.

Definition 4 (Parameter modifier change)

Parameter modifier change $M \equiv modifier_{new} \neq modifier_{old}$

Modifier changes happen when developers alter a modifier without changing the data type. We mostly observed the addition or removal of the 'const' modifier in the C programs of our data set.

Table 3. A taxonomy of signature change kinds. The * item indicates that the item is manually identifiable, and hence the frequency is not reported in this paper.

Data flow invariant	*Function name change (MN) Parameter only ordering change (o) Parameter name change (N) Parameter modifier change (M) *Concept merge/splitting change (CM/CS) Array/Pointer operation change (P) *Return type change (R) Primitive type change (T) Complex type name change (CN)
Data flow increasing	Parameter addition (A) Ordering change by addition (OA) *Return type addition (RA) *Complex type inner variable addition (CA)
Data flow decreasing	Parameter deletion (D) Ordering change by deletion (OD) *Return type deletion (RD) *Complex type inner variable deletion (CD)

Definition 5 (Parameter array/pointer change)

Parameter array/pointer change $P \equiv array/pointer_{new} \neq array/pointer_{old}$

Array/pointer dimension changes occur when developers add or delete dimensions of pointer or array parameters. An example of this change is shown in Table 4.

Table 4. A pointer change example in APR, threadproc/unix/procsup.c file, ap_detach function.

ap_status_t ← ap_proc_t ****new**, ap_pool_t *cont
ap_status_t ← ap_proc_t ***new, ap_pool_t *cont

Definition 6 (Parameter addition/deletion)

Parameter addition $A \equiv p \in \mathbf{Arg}_{new}$ and $p \notin \mathbf{Arg}_{old}$
Parameter deletion $D \equiv p \notin \mathbf{Arg}_{new}$ and $p \in \mathbf{Arg}_{old}$

The parameter addition and deletion changes are common change kinds; an example is shown in Table 5.

Table 5. Parameter addition changes in the Linux kernel, kernel/sched.c file, try_to_wake_up function. First *sync* was added, then later the variable *state* was added.

static int ← task_t * p
static int ← task_t * p, int sync
static int ← task_t * p, unsigned int state, int sync

One of the most interesting change kinds is the concept splitting/merging change defined in Definition 7. Usually concept splitting/merging changes look like parameter addition or deletion changes. But if we observe the changes carefully, the new parameters can be derived from existing or deleted parameters. For example, suppose a function takes '*first name*' and '*last name*' as its arguments. In the next version, the function takes only '*name*'. It seems the '*first name*' and the '*last name*' parameters are deleted while the new '*name*' parameter is added. In fact, the new parameter, '*name*', is a combination of the deleted parameters, '*first name*' and '*last name*'. In this case, a derivation function F exists.

Definition 7 (Concept merging/splitting change)

$\mathbf{A}_{sub} \subseteq \mathbf{Arg}_{old}$
Concept merging CM ≡ A and ∃ a derivation function *F*,
such that $p_{added} = F(\mathbf{A}_{sub})$ and $|\mathbf{A}_{sub}|>1$
Concept splitting CS ≡ A and ∃ a derivation function *F*,
such that $p_{added} = F(\mathbf{A}_{sub})$ and $|\mathbf{A}_{sub}|=1$

The '*name*' parameter can be derived using a derivation function F: '*name*' = F ('*first name*', '*last name*'). We define this kind of changes as a concept merging change. If the evolution goes in the opposite direction, we define it as a concept splitting change.

Definition 8 (Primitive types and Complex types)

Primitive type set PTS ≡ {char, int, long, float, double}
Is primitive type PT(t) ≡ true iff t ∈ PTS, else false
Is complex type CT(t) ≡ true iff t ∉ PTS, else false

Definition 9 (Primitive type change)

Primitive type change ≡ $type_{new} \neq type_{old}$ and
$PT(type_{new})$ and $PT(type_{old})$

Definition 10 (Complex type change)

Type variable set **TVS** ≡ a set of variables used in a complex type
Complex type name change CN ≡ $type_{new} \neq type_{old}$
and $(CT(type_{new})$ or $CT(type_{old}))$
Complex type inner variable addition
CA ≡ $CT(type_{new})$ and $CT(type_{old})$
and $type_{new} = type_{old}$ and $|TVS_{new}| > |TVS_{old}|$
Complex type inner variable deletion
CD ≡ $CT(type_{new})$ and $CT(type_{old})$
and $type_{new} = type_{old}$ and $|TVS_{new}| < |TVS_{old}|$

Definition 11 (Return parameter change)

Return type change R ≡ $modifier_{new} \neq modifier_{old}$ or
$type_{new} \neq type_{old}$ or $array/pointer_{new} \neq array/rpointer_{old}$
and $type_{new} \neq void$ and $type_{old} \neq void$
Return type addition RA ≡ $type_{new} \neq type_{old}$ and $type_{old} = void$
Return type deletion RD ≡ $type_{new} \neq type_{old}$ and $type_{new} = void$

We define primitive type and complex types in Definition 8, and based on this definition we define primitive type and complex type changes.

The primitive type change indicates one of the parameter types has been changed while the parameter name remains unchanged. For example, if a parameter, '*int age*' is changed to '*long age*', it is a primitive type change. If the primitive type and the parameter name of an argument change together, it is a parameter addition/deletion change.

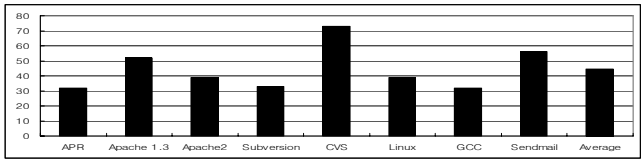

Figure 1. The percentage of the primitive data types used in function signatures of each project.

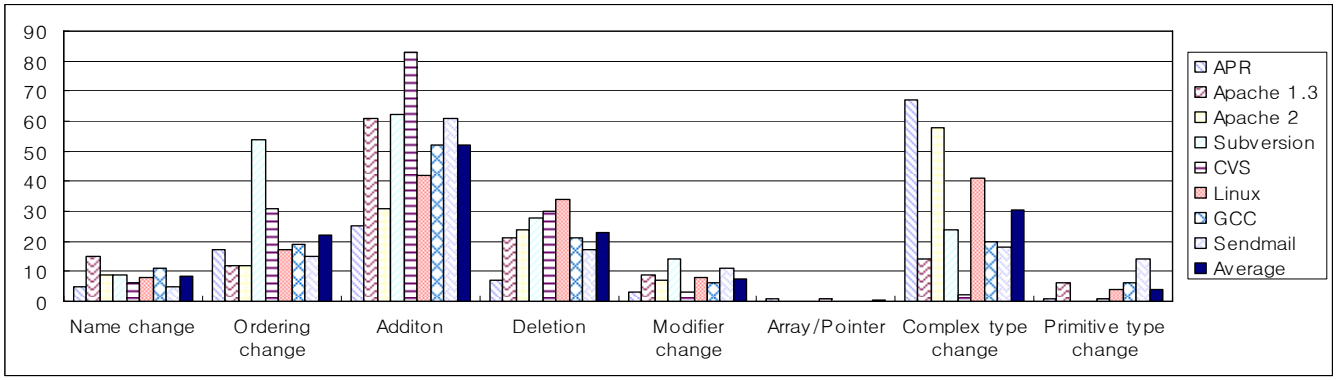

Figure 2. The percentages of each change kind frequency of the eight open source projects and average.

In the open source projects we observed, on average 55% of data types in signatures are complex data types (class, typedef, struct or union); see Figure 1. If one of the complex data types is changed, we define this change as a complex type change. These changes are different from parameter addition or deletion changes in that the old and new data types are related. Usually, when there are major changes in a class or structure, developers change the class/structure name. If there are only minor changes to the structure or class, such as adding a member variable, the structure/class name will not be changed. Since we are analyzing only signatures, we cannot automatically identify changes inside of structures or classes. To identify these changes, we need to monitor the structure/class body for changes in each revision. We may observe this in future work.

To define the major categories of our taxonomy, we use a data flow model between a function and a client. A client calls a function by passing arguments (**Arg**) and expecting returns (**R**) as shown in Figure 3. The total data flow is the union of **Arg** and **R**, defined in Definition 12. Broadly, when parameters or return values are added, there is an increase in the amount of data flowing between caller and callee, while parameter deletion or removal of return values results in reduction of data flow. Modifier changes or parameter name changes have no impact on the data flow.

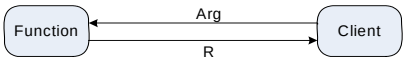

Figure 3. Data flow model.

Definition 12 (Data Flow)

DF ≡ **Arg** ∪ **R**
Data flow invariant ≡ $|\mathbf{DF}_{old}| = |\mathbf{DF}_{new}|$
Data flow increasing ≡ $|\mathbf{DF}_{old}| < |\mathbf{DF}_{new}|$
Data flow decreasing ≡ $|\mathbf{DF}_{old}| > |\mathbf{DF}_{new}|$

4. FREQUENCY OF CHANGE KINDS

After identifying signature change kinds, we computed the frequencies of each kind. Figure 2 shows the signature change kind frequency percentages of each project. To simplify the graph we aggregated ordering changes (Ordering change = o+OA+OD). Figure 2 shows percentages for each change kind; the percentage is calculated by taking the number of observations of a particular change kind, and dividing it by the total number of signature changes observed for that system. For example, in Apache 1.3, we observed 202 parameter additions, and 327 total signature changes, resulting in a frequency percentage of 61%.

Note that one signature change can include more than one change kind. For example a signature change can include parameter addition, parameter deletion, and ordering changes. As a result, the summation of each percentage is greater than 100%. For example, the sum of all the CVS project change kinds is 157 %. It means that whenever a function has a signature change in the CVS project, the signature change includes 1.57 different kinds of change, on average. If there is more than one instance of a particular change kind in a signature change, we count the kind only once. For example, if a signature change includes a parameter addition change three times, we count only one parameter addition change.

Figure 2 shows that the most common change kinds are parameter addition (average 52.13%), complex type changes (average 30.5%), and parameter deletion (average 22.75%). The array/pointer and primitive type change are relatively uncommon change kinds.

5. RATIO OF SIGNATURE CHANGES

To show the distribution of signature changes across functions, we counted the number of functions having *n* signature changes, with *n* varying from 0 to 16 signature changes (see Figure 4 for the signature change distribution for Subversion). Figure 4 shows that 5466 functions (77%) never changed their signature and 95% of the functions had fewer than three signature changes.

Another interesting ratio of signature changes can be obtained by comparing the number of signature changes and number of function body changes. We may examine this in future work.

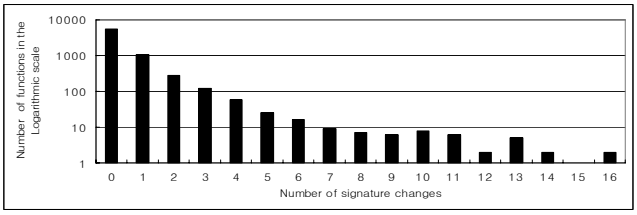

Figure 4. Count of signature changes of functions in the Subversion project. The x-axis indicates the number of signature changes, and the y-axis indicates the number of functions (log scale).

6. SIGNATURE EVOLUTION PATH

We wondered whether common signature evolution paths could be used to predict future software changes. For example, we

might detect that the most common signature changes occurred in this order: parameter addition (A), parameter deletion (D), ordering change (O), return type change (R), and parameter addition (A). In the future, when a known signature change evolution sequence occurred, such as A, D, O and R, we could predict the next signature change is likely to be a parameter addition (A).

To determine whether or not such common evolutionary paths exist, we noted all signature change evolution sequences. For example, when the signature of a function changes in this order: A, D, O, R, and A (See Table 3 for the change pattern abbreviations), we generate a change sequence, '*ADORA*'. We examined all signature change sequences whose length is larger than three. We assumed that change sequences with fewer than four changes are rarely associated with common evolution paths.

After having an array of the sequences, we looked for the most common sequence (MCS) patterns using a modified longest common sequence (LCS) search algorithm [7]. Table 6 shows the top five common sequences of the Subversion project and overall eight projects. The occurrence shows how many times we found the change sequence patterns over all patterns, and percentage shows how common each occurrence is as a fraction of all observed occurrences (1,428 for the Subversion project and 2,025 for overall). We need to determine the conditional probabilities of each change kind to see if it depends on previous changes, and that the dependency rate is high enough to predict future change kinds. We weren't able to find predictable evolution paths from common sequences.

Table 6. The top five common function signature change pattern sequences of the Subversion project and across all projects. # means the count of occurrences of the pattern, and % means the percentage of times this sequence occurs.

Subversion Project			Overall projects		
Common Sequence	#	%	Common Sequence	#	%
ACDA	186	13%	AADA	198	9%
AADA	183	12%	ACDA	186	9%
AACD	159	11%	ADDD	171	8%
ADDD	152	10%	AACD	159	7%
ACAA	133	9%	ADAD	141	6%

7. THREATS TO VALIDITY

The results presented in this paper are based on selected eight open source projects. It includes major open source projects, but other open source or commercial software projects may not have the same properties we presented here. We analyzed only projects written in the C programming language; software written in other programming languages may have different signature change patterns. Some open source projects have revisions that are not compilable and contain syntactically invalid source code. In that case, we had to guess at the signatures or skip the invalid parts of the code. We ignored '*#ifdef*' statements because we cannot determine the real definition value; ignoring '*#ifdef*' caused us to add some extra signatures which will not be compiled in the real program.

8. CONCLUSIONS AND FUTURE WORK

We have introduced a fine-grain taxonomy of signature change kinds. Among change kinds, the common change kinds are parameter addition (52.13%), complex type change (30.5%) and parameter deletion (22.75%). In future work we hope to this result can be used to alleviate signature change impact. If we can provide an ontological framework that includes a conceptual meaning for each parameter with its data type, it is possible to accommodate ordering changes and parameter deletion changes by generating glue code that resolves the signature mismatch problem. We found that about 77% of functions never change their signature and another 23% of functions change their signature once or twice.

We used a function name as an identifier to keep track of signature changes. Unfortunately, this means that if a function name changes, we loose its previous history of signature changes. The C++ and Java programming languages allow method overloading – more than one method with the same name but different parameters. When groups of overloaded methods evolve, sometimes ambiguity prevented us from determining which old method changed to which new method. Tu et al. introduced an origin analysis algorithm to find the origins of new procedures or files [8]. Origin analysis helps to find evolution paths when function names are changed or methods are overloaded. However, origin analysis requires heavy computation for entity analysis and dependency analysis. Providing more accurate results using origin analysis remains future work.

About 55% of parameters are complex data types such as structures, unions, or classes. Even though the signature remains unchanged, when a complex data type has changed internally, such as the addition of a member variable, it should be regarded as a signature change. Monitoring changes to each complex data type used in a signature to observe this kind of change remains future work.

Finally, further study is needed to explore the correlations between signature evolution and whole system evolution.

9. ACKNOWLEDGMENTS

Thank you to Mark Slater, and the anonymous reviewers for their valuable feedback on this paper. Work on this project is supported by Samsung Electronics, NSF Grant CCR-01234603, and a Cooperative Agreement with NASA Ames Research Center.

10. REFERENCES

[1] M. M. Lehman, "Rules and Tools for Software Evolution Planning and Management," *Proc. Int'l Workshop on Feedback and Evolution in Software and Business Processes (FEAST 2000)*, Imperial College, London, July 10-12, 2000.

[2] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance," *Proc. the Int'l Conf. on Software Maintenance*, Victoria, Canada, 1994, pp. 202-211.

[3] S. Counsell, et al., "Trends in Java code changes: the key to identification of refactorings?" *Proc. 2nd Int'l Conf. on Principles and Practice of Programming in Java*, Kilkenny City, Ireland, 2003, pp. 45 - 48.

[4] J. Bevan, "Kenyon Project Homepage," 2005 http://kenyon.dforge.cse.ucsc.edu

[5] B. Behlendorf et al., "Subversion Project Homepage," 2005 http://subversion.tigris.org/

[6] T. Zimmermann and P. Weißgerber, "Preprocessing CVS Data for Fine-Grained Analysis," *Proc. MSR 2004*, Edinburgh, Scotland, 2004, pp. 2-6.

[7] D. S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *Journal of the ACM (JACM)*, vol. 24, no. 4, pp. 664 - 675, 1977.

[8] Q. Tu and M. W. Godfrey, "An Integrated Approach for Studying Architectural Evolution," *Proc. Intl. Workshop on Program Comprehension (IWPC 2002)*, Paris, June, 2002, pp. 127.