Detecting Increases in Feature Coupling using Regression Tests

Olivier Giroux
Graphics Infrastructure Dept.
NVIDIA Corporation
Santa Clara, CA, USA
ogiroux@nvidia.com

Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
martin@cs.mcgill.ca

ABSTRACT

Repeated changes to a software system can introduce small weaknesses such as unplanned dependencies between different parts of the system. While such problems usually go undetected, their cumulative effect can result in a noticeable decrease in the quality of a system. We present an approach to warn developers about increased coupling between the (potentially scattered) implementation of different features. Our automated approach can detect sections of the source code contributing to the increased coupling as soon as software changes are tested. Developers can then inspect the results to assess whether the quality of their changes is adequate. We have implemented our approach for C++ and integrated it with the development process of a proprietary 3D graphics software. We report on our evaluation of the approach in the field, and on a study showing that, for files in the target system, causing increases in feature coupling is a significant predictor of future modifications due to bug fixes.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering*.

General Terms

Design, Reliability, Verification.

Keywords

Feature coupling, feature associations, feature implementation, feature location, regression testing, dynamic analysis.

1. INTRODUCTION

Successful software requires a maintenance investment that can dwarf that of its initial development. The long life and large install base that come with success typically combine to expose flaws and impose unforeseen requirements on a software system. In turn, such factors put pressure on software development organizations to keep up with customers' changing expectations, resulting in continual modifications to a software code base. As evidence of this situation, the issue tracking systems for large open-source software projects typically include thousands of completed modifications.

Many factors influence the quality of changes to a system, including developer experience, familiarity with the system, time constraints, and the quality of the system's design. In general, these practical considerations often lead to suboptimal changes that slightly deteriorate the quality of a code base [3, 4, 11], for example by increasing the overall amount of coupling. We refer to this phenomenon as *code decay* [4].

Code decay is problematic because it can make it harder to change a system in a way that is not easily observable. For example, a software modification may not cause any regression fault, but instead expose some subtle implementation details that were previously hidden. Later versions of the system may come to depend on the details, thus making the previously-encapsulated code difficult to change. While the effects of code decay will eventually become apparent, it may prove very expensive to remedy the situation at the later juncture. Code decay is a subtle phenomenon that is difficult to characterize [4]. However, it is nevertheless possible to detect potential symptoms, or risk factors, which can then be assessed by developers.

The intuition guiding the present research is that an *increase* in the amount of coupling between the implementation of different features (functional requirements) can be a symptom of code decay (i.e., if it is unplanned), and that such situations should be reported to developers for closer inspection. Unfortunately, the implementation of features is not always neatly encapsulated in a single module [7, 10], a situation which precludes the trivial use of standard coupling metrics to detect this symptom.

In this paper, we describe a new feature coupling detection technique. Our approach is based on a dynamic analysis of a software system as it undergoes regression testing. It can be completely automated and fully integrated in the software development process of an organization. With our technique, developers work as usual but when their changes are committed and tested, the execution of the test suite is monitored, analyzed, and compared with information obtained from the regression testing of a previous version of the code. When increased associations between the implementation of different features are detected, the parts of the code contributing to the evidence obtained are retrieved and reported to the developer.

We have implemented our technique and applied it to a real-world code base consisting of more than 100 000 lines of C++ source code exercised by thousands of tests. Our experience with this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM 1-59593-468-5/06/0011...$5.00.

technique showed that its overhead is low enough to integrate it in the build and test cycle of the organization and that it produces reports that are easy to understand and convenient to use by developers. A study of the target system using our technique also demonstrated that files contributing to increases in feature coupling were significantly more likely to be modified by future bug fixes, hence reinforcing the assumptions forming the basis for our technique. The contributions of this paper include a description of our automatic technique for the detection of increases in feature coupling and a detailed account of our experience with this technique in the field.

In the rest of this paper, we first provide the details of our technique for detecting increases in feature coupling (Section 2). We then describe our application of the technique (Section 3) and our initial experience with the technique along with the validation study (Section 4). Finally, we discuss related work in Section 5 and conclude in Section 6.

2. COUPLING DETECTION TECHNIQUE

Measures of coupling in software have traditionally been used to diagnose different conditions in software systems, such as the need for refactoring for more thorough validation activities [1]. In a similar perspective, we base our coupling detection technique on the following hypothesis: Given that a system implements a number of features, any *increase* in the association between the implementation of two features may indicate locations where unplanned dependencies have been introduced.

In this paper, we use the term "feature" to refer to a cohesive set of the observable properties of a software system (e.g., as would correspond to the functional requirements). For example, a word processing software would typically include features such as "spell checker", "auto save", and "undo". For a number of practical reasons, the implementation of features does not always align with module boundaries, and is instead scattered throughout the basic decomposition of the system [7, 10]. For example, the functionality to "undo" commands typically involves code that is scattered throughout the implementation of each undoable command in the system.

Although the idea of detecting increases in the coupling between features is conceptually simple, its practical realization must account for the numerous and complex ways in which different (and potentially scattered) sections of a software system can interact. For example, statically establishing data dependencies between sections of code requires complex, computationally expensive, and potentially imprecise calculations.

To investigate a technique that would apply to large, deployed software systems, we chose to estimate feature interactions using a probabilistic model based on test coverage information. Our technique associates features with tests, and tests with implementation components. By recording whether the overlap between components implementing different features increases as a regression test suite is applied to a new version of a system, we can determine which sections of the code cause the increases. We hypothesize that such sections may contribute to code decay and should be inspected by developers to ensure that the changes do not introduce undesirable weaknesses in the code. In the rest of this section, we present the details of our technique.

2.1 Basic Concepts

The following concepts are important to our analysis algorithm. The most basic concepts are that of a program version, a component, a feature, and a test.

DEFINITION 1 (PROGRAM VERSION). *A program version P=(C,F,T) is the combination of a set C of components, a set F of feature, and a set T of tests.*

DEFINITION 2 (COMPONENT). *Given a program version P=(C, F, T),* a *component $c \in C$ is an entity of the program represented by P whose execution can be detected as part of the execution of a test $t \in T$.*

Components can be defined to represent different constructs, such as lines of code, procedures, basic blocks, etc… Although practical considerations influence the selection of a component granularity, our approach is technically independent from the specific choice component types.

DEFINITION 3 (FEATURE). *Given a program version P=(C, F, T), a feature $f \in F$ is a functionality of the program expressed such that it is possible to unambiguously determine whether a test $t \in T$ exercises f.*

DEFINITION 4 (TEST). *Given a program version P=(C, F, T), a test $t \in T$ is an execution of a subset of the program represented by P that exercises a set of features Ft and covers a set of components Ct, where $Ft \subset F$ and $Ct \subset C$. We have exercises(t,f) if t exercises f, and covers(t,c) if c is executed as part of t.*

It follows from the last two definitions that the association between features and tests is many-to-many. In other words, it is not necessary for a feature to be uniquely associated with a test.

In practice, the binary relation *exercises* can be obtained in a number of ways, including through manual inspection, feature location techniques, or others. In the context of our approach we assume that this relation exists and that the information is available as part of a software project. Section 3.2 describes one way to automatically generate the *exercises* relation. As for the *covers* relation, the components covered by individual tests can be determined from the execution of a test using straightforward instrumentation techniques (see Section 3.1).

2.2 Feature Implementation

We estimate the association between different features in two steps. First, we estimate how strongly each component is associated with the implementation of a feature. We call this estimate the *feature implementation*. Second, and based on the feature implementation, we estimate the strength of the association between the implementation of different features. We call this last estimate the *feature association*.

The calculations of the feature implementations and associations are based on linear algebra. Given a program version P = (C,F,T), we model the *exercises* relation as a matrix of size $|T| \times |F|$ where the row/column tuple (t,f) is 1 if t exercises f and 0 otherwise. Similarly, we model the *covers* relation as a matrix of size $|T| \times |C|$ where the row/column tuple (t,c) is 1 if t covers c and 0 otherwise.

The intuitions behind our definition of a feature implementation are that *a)* a component implements a feature if it is covered by all

tests exercising the feature, and *b)* the strength of the implementation relation is determined by the ratio of tests covering the component that are associated with the feature over the ratio of all tests covering the component. For example, if a component c_1 is covered by 20 tests, and all 5 tests for feature f_1 cover c_1, then we will say that that c_1 implements f_1 with a degree of 0.25. At the other end of the spectrum, if c_1 is covered by 20 tests, and all 20 tests for feature f_1 cover c_1, then we will say that c_1 implement f_1 with a degree of 1.0. In order to operationalize these intuitions, we define a vector operation we call the *implementation product*. The implementation product is similar to a standard dot product but makes provisions for intuitions *a)* and *b)* above.

DEFINITION 5 (IMPLEMENTATION PRODUCT). *Given two vectors of size n, $a = (a_1, a_2, \ldots a_n)$ and $b = (b_1, b_2, \ldots, b_n)$, the implementation product $a \otimes b$ is defined as*

$$a \otimes b \equiv \begin{cases} \dfrac{\sum_{i=1}^{n} a_i b_i}{\sum_{i=1}^{n} b_i}, \text{ if } \forall a_i \neq 0, b_i \neq 0 \\ 0, \text{ otherwise} \end{cases}$$

With our definition of the implementation product, we can define a matrix implementation product that works just like the standard matrix multiplication except that the implementation product is used instead of the dot product to multiply component vectors.

DEFINITION 6 (MATRIX IMPLEMENTATION PRODUCT). *Let $A = [a_{ik}]$ be an $m \times n$ matrix, and let $B = [b_{kj}]$ be an $n \times s$ matrix. The matrix implementation product $A \otimes B$ is the $m \times s$ matrix $C[c_{ij}]$,where c_{ij} is the implementation product of the ith row vector of A and the jth column vector of B.*

With the above definitions, we can now define a feature implementation.

DEFINITION 7 (FEATURE IMPLEMENTATION). *Let exercises and covers be the matrices corresponding to the exercises and covers relations for a program version, respectively. Let $exercises^T$ be the transpose of exercise. We define a feature implementation FI as $FI = exercises^T \otimes covers$.*

Example

We illustrate the calculation of a feature implementation with a small example. Consider a simple program comprising four tests and seven components. Table 1 shows the *covers* matrix for a program version (for clarity we do not show the 0 values). We can assume that this information is obtained by running test programs with execution instrumentation.

Table 1: *Covers* matrix for the example program

	C_1	C_2	C_3	C_4	C_5	C_6	C_7
T_1	1	1		1		1	
T_2	1	1	1				1
T_3		1	1	1	1		
T_4		1		1		1	

Additionally, individual tests exercise only a subset of the features of the program. Table 2 shows the transpose of the *exercises* matrix. This information can be provided along with the test suite, for example.

Table 2: $Exercises^T$ matrix for the example program

	T_1	T_2	T_3	T_4
F_1	1			
F_2	1	1		
F_3		1		
F_4	1			1
F_5			1	

Taking the implementation product of $exercises^T$ and *covers* produces the FI matrix, as shown in Table 3.

Table 3: Feature implementation for the example program

	C_1	C_2	C_3	C_4	C_5	C_6	C_7
F_1	0.5	0.25	0	0.33	0	0.5	0
F_2	1	0.5	0	0	0	0	0
F_3	0.5	0.25	0.5	0	0	0	1
F_4	0	0.5	0	0.67	0	1	0
F_5	0	0.25	0.5	0.33	1	0	0

For example, taking the implementation product of row F_1 in $exercises^T$ and column C_1 in *covers* produces the value $(F_1, C_1) = 1 \times 1/(1+1) = 0.5$ in FI. This value estimates that C_1 implements F_1 with a degree of 0.5 since one other test not associated with F_1 covers C_1.

2.3 Feature Association

A feature association is a square matrix representing the degree of association between the implementation of different features.

DEFINITION 8 (FEATURE ASSOCIATION). *Given a program version P = (C,F,T) and its corresponding feature implementation FI, a feature association FA is the square matrix of size $|F| \times |F|$ defined as the (true) matrix product $FA = FI \bullet FI^T$.*

The dot product between two feature implementation vectors represents the cosine of the angle between them (multiplied by the magnitude of each vector). Hence, the feature association matrix models how strongly any two features "align" in the space of components. The higher the value for a pair of features, the larger the number of components they share in their implementation or the more important the shared components are to both features. In our approach, we do not take into account the absolute value of feature associations. Instead, we simply detect whether such values increase as a system evolves.

Example

To complete our example, Table 4 shows the final feature association for our example.

Table 4: Feature association for the example program

	F_1	F_2	F_3	F_4	F_5
F_1	0.67	0.63	0.31	0.85	0.17
F_2	0.63	1.25	0.63	0.25	0.13
F_3	0.31	0.63	1.56	0.13	0.31
F_4	0.85	0.25	0.13	1.70	0.35
F_5	0.17	0.13	0.31	0.35	1.42

From Table 4 we see that, for example, feature f_1 is more strongly associated with feature f_2 than with feature f_5. There are two things to note from this table. First, a feature association matrix is in fact a triangular matrix as the association relation is symmetrical. Second, the values representing the association of a feature with itself vary between features. This is simply a consequence of the fact that, for simplicity, we have not normalized the feature implementation vectors (the row vectors of the feature implementation matrix). If we normalize the feature implementation vectors in Table 3, the diagonal of the feature association matrix will contain only values of 1.

2.4 Coupling-Increasing Components (CIC)

Coupling-Increasing Components (CIC) are the components that contribute to an increase in the level of association between two features. We obtain the set of CICs by comparing the feature implementations and feature associations of two different program versions.

To identify CICs, we first locate feature pairs whose association has increased between two versions. We define an association to have increased if the association between two features in a (more recent) program version is greater than the association between the same features in a previous program version by a certain multiplicative factor α. The α factor is a parameter of our approach that can take values in the interval $[1..\infty)$.

DEFINITION 9 (COUPLING-INCREASING FEATURE PAIRS). *Given two program versions P = (C,F,T) and P*=(C*,F*,T*), and their corresponding feature association FA[fa_{ij}] and FA*[fa_{ij}*], the coupling-increasing feature pairs CIF[cif_{ij}] is a matrix of the same size as FA* where:*

$$cif_{ij} = \begin{cases} 1, \text{if } fa_{ij}^* > \alpha\ fa_{ij} \\ 0, \text{otherwise} \end{cases}$$

DEFINITION 10 (COUPLING-INCREASING COMPONENTS). *Given two feature implementations FI and FI* and a matrix of coupling-increasing features CIF, we define the set of coupling-increasing components of a modified program P*=(C*, F*, T*) as the set of components contributing to values in CIF. The set of CIC can be calculated with the following algorithm:*

```
1:  param: P*=(C*, F*, T*): Modified Program
2:  param: FI[f_ij] and FI*[f*_ij]: Feature Implementations
3:  param: CIF[cif_ij]: Coupling-Increasing Features
4:  var: CIC={}: Coupling-Increasing Components
5:  for i = 1..|f_i| (where f_i is a row vector of FI)
6:      for j = 1..|f_i|, i ≠ j
7:          if cif_ij = 1
8:              for k = 1..|f_i|
9:                  if f*_ik • f*_jk > f_ik • f_jk
10:                     CIC ← CIC ∪ c | c is the component
                                corresponding to column k in FI
11:                 end if
12:             end for
13:         end if
14:     end for
15: end for
16: return CIC
```

Once the analysis is complete, we present the CIC set to the developers, who will determine if the components are in fact contributing to code decay.

2.5 Discussion

The quality of the results produced by our algorithm is dependent on the stability of feature associations in the absence of code decay. For example, if changes that do not cause code decay in practice introduce variations in associations, then our algorithm could produce false positives. In general, the role of the parameter α is to stabilize the algorithm, by making it more resilient to small variations in feature associations. However, if α is set too high then important symptoms of code decay could go unnoticed, and so the effective range of α is also limited.

Essentially, variations in feature association are a factor of two main phenomena: *a)* relevant variations due to an increase in feature coupling (and potentially indicative of code decay), and *b)* irrelevant variations due to imprecision in the computation of feature implementations. The primary source of imprecision in the computation of feature implementations is an insufficient number of tests exercising certain features to obtain reasonable estimates of the components that implement them. The importance of this imprecision will typically diminish as the number of tests increases and the focus of tests narrows to fewer features.

Finally, the inclusion of components in the CIC set implies the existence of a mapping of components between system versions ($c \in C \rightarrow c^* \in C^*$). In other words, given two feature implementation matrices representing two different program versions, it is assumed that a column in the matrix for one version represents the same component as the corresponding column in the matrix for the other version. In practice, this assumption requires special treatment when components are added or removed between versions. Additionally, if using lines of code as components (commonly identified by file/line information), even unchanged components may require remapping because of the addition and removal of other components above them in the same file. This bidirectional mapping between components of different program versions is assumed to exist in the CIC algorithm, but the details are left to the implementation (see Section 3.4.2).

3. CASE STUDY

To investigate the feasibility and usefulness of our approach, we implemented our technique and applied it to a proprietary 3D graphics program developed at NVidia Corporation.[1] The target system consists of more than 100 000 lines of C++ code exercised by thousands of tests, and each change is tested for regression before it is submitted to the source repository. Although many parts of the implementation built for this case study are generic enough to apply to a wide range of software systems, practical considerations required us to tailor the overall implementation to the environment of our target system.

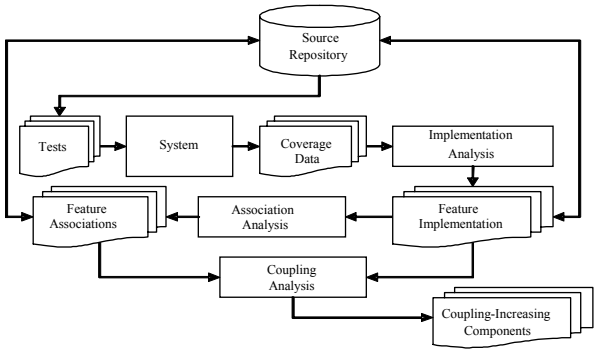

Figure 1 : Implementation Diagram

Our current implementation (depicted in Figure 1) is designed to be applied to all new changes made to our target system before they are submitted to the source repository. To this end, our implementation extends existing proprietary regression-testing infrastructure and practices without interfering with the normal activities of software developers. For our analyses, we defined components as the lines of code of the system, as an approximation for C++ statements. However, for practical reasons we aggregate the results by source files for the final presentation to developers.

Our implementation works as follows. First, we obtain the test suite from the source repository and compile the locally-modified program with code instrumentation to produce statement coverage information when executed. The test suite is then executed as usual, producing the *covers* relation matrix that relates tests with components (see Section 3.1). Executing the test suite on our target system also produces the *exercises* relation matrix that relates tests with features thanks to a different type of instrumentation that forms an integral part of our specific target system (see Section 3.2).

As described in the previous section, the *covers* and *exercises* matrices serve as input to the computation of feature implementations and association analyses (see Section 3.3). Feature implementations and associations are then marked for storage in the source repository together with the current changes so that they can be versioned along with the software and used in future analyses. To perform feature coupling analysis (see Section 3.4), we recover the version of the feature implementations and associations that match the previous version of the program. The old and new associations are then compared for increased associations and the CIC set is constructed from the lines of code that caused the differences, as described in Section 2.4. Finally, the lines of code are aggregated by files and the set of coupling-increasing files is presented to the developer.

In the rest of this section, we discuss key implementation issues specific to each step of our approach.

3.1 The *Covers* Relation

We obtain the *covers* relation by instrumenting the program code to automatically detect each line of code covered by each test. Inspired by the work of Tikir and Hollingsworth [12], we designed our instrumentation such that it removes itself once triggered, leaving the original subroutines. This strategy greatly reduces the cost of instrumentation, especially for code containing loops. This characteristic of our implementation is in fact critical given the size and heavy computational nature of the target system. We observed, as also noted by Tikir and Hollingsworth, that the performance impact of this type of instrumentation is low, increasing the run time by only 5~10% (see Section 4.3 for the details of the performance evaluation).

The *covers* matrix produced by our coverage instrumentation can be very large. Thousands of tests executing over hundreds of thousands of lines of code will produce hundreds of millions of entries in this matrix. Fortunately, *covers* matrices are naturally sparse and contain some simple patterns, such as groups of components that are always covered together. We reduced the effective size of the stored data by indexing, storing, and analyzing these groups of components as a single entity.

3.2 The *Exercises* Relation

Ideally, the features exercised by individual tests in the test suite would be documented alongside and versioned with the test suite. In practice, we found that this information was not consistently available. In our target system, each test is relatively complex and exercises many features, often leaving only vague and informal references to the dominant feature to be encoded in the test name. In some cases, even the names were misleading, due to the test's ultimate purpose changing over time.

To recover the *exercises* relations, we relied on execution logs produced by our target system as it executes. These execution logs form an integral part of the target system and are different from our instrumentation system. The primary purpose of the execution logs is to assist in the analysis of inputs given to the system, both manually by developers and through automated tools. Built by the developers alongside the system's functionality, the logs provide extensive details about the execution of the system, including a fine-grained description of the functionalities exercised by the program during its execution. For example, the logs produced by our target system are analogous to a trace of user interactions that could be generated by a word processor, logging the commands invoked by the users through menus and buttons (e.g., spellchecking, justification, etc…).

Since the exact details of our logging feature are proprietary, for the purpose of this paper we abstract the logging feature as a module that produces a list of the commands called on the graphics software. We collected these logs for each test and matched the functionality they referenced to features, hence reconstructing the *exercises* relations between tests and features. The main consequence of this strategy is that it makes our

[1] This software is used internally and is not released to the public.

definition of feature to be fine-grained, yielding more than ten thousand features for our target system. However, this strategy supports a completely automatic recovery of the *exercises* matrix, which is a critical element of the feasibility of our approach. This strategy for mapping features to tests is a parameter of our approach that may not be directly realizable for all target systems (see Section 3.5).

Although the number of features detected remains much less than the number of lines of code in the system, our feature identifiers are much larger than an integer and their analysis produces physical data sets of similar size. Like the *covers* matrices, *exercises* matrices are also naturally sparse and their cost can be made manageable using a similar grouping strategy.

3.3 Feature Implementations

The implementation of the computational support for feature implementations as described in Section 2.2 gives rise to a matrix product of staggering size if the sparseness is not exploited. To compute the implementation of a feature, the associated *exercises* test group[2] for the feature is used as the *reference test set*. All test groups from the *covers* relations are then compared to the *reference test set*. If all tests from the *reference test set* are found in the *covers* test group, then all components associated to it are added to the implementation of the feature. The implementation value of each of these components is then calculated as the size of the *reference test set* over the size of the test group (see Definition 5). This process produces as output a set of tuples of components and implementation values, representing the non-zero values of the feature implementation vectors.

Even in their compact form, the feature implementation vectors remain large and dominated by components with very low implementation scores (e.g., components that are covered by all tests). To increase the performance of our feature coupling analysis, we limit the size of feature implementation vectors to 200 components, and truncate the less significant components. The components truncated in this manner vary from feature to feature, leaving a selection of the 200 highest-degree components for each individual feature, and resulting in a sparser (but not smaller) feature implementation matrix. The choice of 200 as the length of implementation vectors is based on experience with applying feature location techniques on our target system.

The tradeoff of this optimization strategy is that the components removed in this manner will also vary from program version to program version. As a result, features insufficiently exercised by the test suite will appear to make significant feature implementation losses and gains between versions. Although in principle the low implementation values of the truncated components means that they should not affect the end result (the computation of CICs), in practice we have found that this process introduces noise that warrants additional filtering during coupling analysis (see Section 3.4).

Finally, even though it is not required by our algorithm, we normalize our implementation vectors after truncation. As a result the implementation products are themselves normalized and provide useful meaning to associations when debugging the implementation of coupling analysis.

[2] The groups are seen in the sparse matrix, as mentioned in 3.1.

3.4 Feature Coupling Analysis

Our implementation of feature coupling analysis is faithful to the algorithm described in Section 2.4. However, use of the technique in the field required the development of an additional noise filtering support, and support for the mapping of components and features across program versions.

3.4.1 Eliminating Noise

The set of tests used to validate changes made to our target system varied greatly depending on the scope of the changes performed. Current practices for our target system call for executing a "sanity" test suite instead of the much larger "full" test suite when changes are deemed at low risk of causing functional regressions.[3] As a result, we encountered many cases where some features were insufficiently exercised to reliably identify the components implementing them (in other words, resulted in significant noise in the feature implementation matrix). We solved this problem by adding a filtering pass to the algorithm described in Section 2.4.

We employ two different filtering methods to reduce the effect of noise at the feature coupling analysis phase. First, the algorithm's sensitivity threshold α eliminates insignificant variations in associations. For our target system, values as small as α=1.1 provided an appropriate baseline for noise reduction. We determined this value heuristically by estimating how much a feature association should increase before being considered significant. This initial estimate was assessed empirically and found to be adequate for our initial investigation of the approach (see Section 4.2).

Second, we defined an analysis on individual feature implementations to discard variations resulting from noisy feature implementation vectors that do not appear to reliably associate a feature to its implementation. Specifically, we define a noisy implementation vector as one whose components are all more or less equally relevant, such that no component is significantly more important than any other. As in Section 2.4, we parameterized the significance detected with a sensitivity threshold β, such that a feature implementation vector (of components) $[c_i]$ is noisy if the following predicate holds (the overbar denotes the mean and σ the standard deviation):

$$c_i - \bar{c} < \beta\sigma(c)$$

3.4.2 Mapping Components with Program Versions

We identify our components (lines of source code) with unique indices in the *covers* and feature implementation matrices. The indices are derived from file names (indexed in a file name table) and line numbers. This choice is convenient when gathering *covers* relations, but problematic during feature coupling analysis because changes to the source code cause source lines to move (potentially including unchanged source lines). To allow the comparison of feature implementation matrices during feature coupling analysis, we build a (line number→line number) map for each file of the system between program versions, by applying the UNIX *diff* utility to the different versions of the files and accumulating the additions and subtractions of lines to find the mapping of old line numbers to new line numbers.

[3] This is ultimately left at the developer's discretion.

Our implementation uses this mapping to link components in the new version to those of the old version, ignoring removed components and assuming that new components previously held implementation scores of zero (i.e., that they were never covered). This assumption is reasonable, since it shows new components with nonzero feature implementation values as implementation gains, and allows them to contribute correctly to feature coupling analysis.

3.4.3 Mapping Features with Program Versions

Features can also vary between program versions, though they are far more stable than components. In all cases where algorithms manipulate features we refer to them by an index in a table of feature names, for instance when referring to features in the *exercises* or feature implementation matrices. Because features change over time, the table of features that we build for our analysis (see Section 3.2) also changes over time and indices in the *exercises* and feature implementation matrices of different program versions are incompatible. To enable the comparison of features of different program versions, we search for the names of features from one program version's feature table in the other program version's feature table. We note the pair of indices in a one-way mapping from new to old indices and use the mapping during feature coupling analysis whenever we compare new features with old features.

3.5 Discussion

The most sensitive aspects of the implementation of our approach revolve around the definition of components and features. Selecting components as functions instead of source code lines, and coarse- rather than fine-grained features, would simplify the feature coupling analysis significantly. With fewer, larger features, the noise elimination process may not be necessary, since each feature is more likely to have been sufficiently exercised by the test suite. Using functions as components would simplify the mapping of components between versions. However, for our application, our choice of definitions for components and features was influenced mostly by the concern that the implementation of features may be scattered across different functions.

Lines of code were a natural fit for comparison and integration of the results of coupling analysis with other tools of the existing infrastructure surrounding the target system. The data we collect subsumes the data function-level instrumentation produces: we have the flexibility to recover function coverage from our data through very simple analysis of the source code to support functions as components in the coupling analysis.

The granularity of features was also dictated by the existing infrastructure, through the level of detail of the existing execution logs. For our definition of features, alternatives consisted mostly of the manual mapping of tests to features, a choice that was simply not practical, requiring too much human intervention to scale up to the size of the test suite. In practice, execution logs are not uncommon in the field, and we expect that our approach can be replicated for systems with logging features, although the quality of the results will necessarily vary depending on the details of the logging data produced. In the cases where it is not feasible to instrument the program in this manner, then the mapping of tests to features must be provided by some other means, such as formal documentation or as an integral part of the test suite.[4] However, for some software systems that are under active development it may be reasonable to install instrumentation that produces execution logs detailing the features in use.

4. EMPIRICAL RESULTS

The applicability of our feature coupling detection technique is based on a number of assumptions that can only be validated empirically. Specifically, we rely on the fact that, in practice:

- Feature implementation vectors meaningfully associate components with features;
- The CIC sets produced are usable by developers;
- The computational cost of the approach is acceptable;
- The symptoms detected by the approach have value.

To help determine whether these assumptions held in the case of our target system, we applied our approach to 13 different versions of our target system distributed over a three-month period, to simulate the analysis of weekly development releases. Because of practical constraints on the computational resources available for this research project, we limited the number of tests executed on the 13 versions of the system to the "sanity" subset of the tests. This subset was previously selected using the execution logs to identify the smallest subset of tests from the "full" test suite that exercised 95% of the same features.

4.1 Feature Implementation Vectors

To be able to determine coupling-increasing components, we need to be able to reliably associate components with features. In our approach, the association between a feature and its components is modeled with a feature implementation vector (a row in the feature implementation matrix). For the purpose of our approach, we consider that a feature implementation vector is useful if it clearly identifies certain components as associated with a feature. In our approach the parameter β determines if a feature implementation vector is "good enough" to be used in the computation of CICs (see Section 3.4.1).

As an initial investigation we measured the relative number of significant versus noisy implementation vectors in our feature implementation matrix, given different values of β. We consider an implementation vector to be noisy if the predicate of Section 3.4.1 holds and significant otherwise. Figure 2 shows the relative number of significant vectors in the matrix for different values of β. For each value of β, each bar represents the value for one of the 13 versions of the program we analyzed.

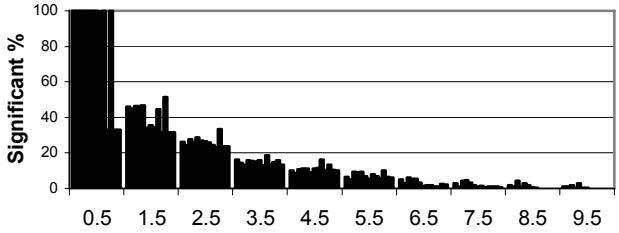

Figure 2: Effect of the β parameter on noise detection

We selected β=1.5 for our system because we felt it provided adequate protection from noise without eliminating weaker

[4] This is a common assumption, made in [6,10,17,18]

evidence in feature implementation vectors. For this value of β, we observed that (on average) 56% of feature implementation vectors were rejected when executing the "sanity" test suite. Executing the "full" test suite reduces this number to 25%, strengthening our intuition that more thorough testing of features reduces noise in the feature implementation matrix.

In the process of selecting a value for β, we manually looked at the value distributions in feature implementation vectors. To illustrate this phenomenon, Figure 3 shows the value distribution of both a significant (solid line) and noisy feature implementation (dotted line), sorted by decreasing degree values.

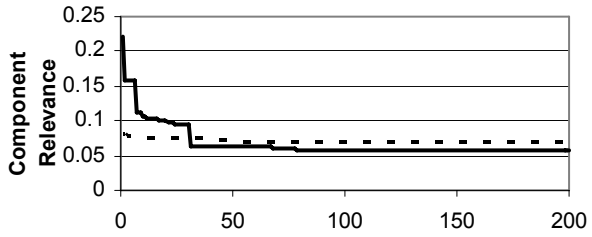

Figure 3: Sample feature implementations

For the significant feature implementation, the figure shows a few very relevant components that stand out from a long tail of less relevant components. For the noisy feature implementation vector, we see instead an almost straight line, with no component being more or less associated with a feature than others. In general, we find that noisy vectors usually correspond to features that are insufficiently exercised by tests.

4.2 CIC Sets

The characteristics of CIC sets matter in our approach since this is the information directly reported to developers. If CIC sets contain large numbers of source locations scattered throughout the system, the developers will be overwhelmed with information. The size of CIC sets is affected by the parameters α and β, which determine whether association changes constitute valid symptoms to be reported, and the usefulness of feature implementation vectors, respectively. To assess their sensitivity to α and β for our system, we measured the CIC sets produced from 13 target revisions of the system (yielding 12 CIC sets). Since our approach automatically aggregates CIC sets by source file, we present our results at this level of granularity.

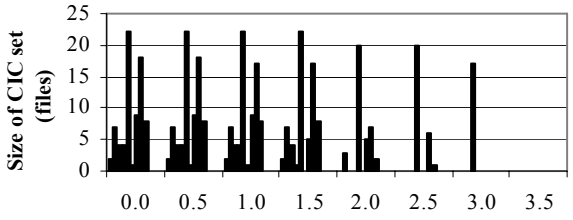

Figure 4: Effect of β on the size of CIC sets

Figure 4 shows the impact of the β parameter on the size of CICs (number of files) for a fixed value of α. Note that in general increasing the value of β decreases the number of CICs.

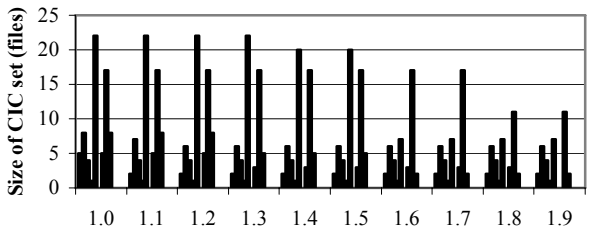

Figure 5: Effect of α on the size of CIC sets

Figure 5 shows the effect of α on the size of CIC sets for a fixed value of β=1.5. We observed that the number of coupling-increasing files produced remains largely stable for changing values of α. We surmise that the spikes in the graph represent versions exhibiting significant increases in some feature associations.

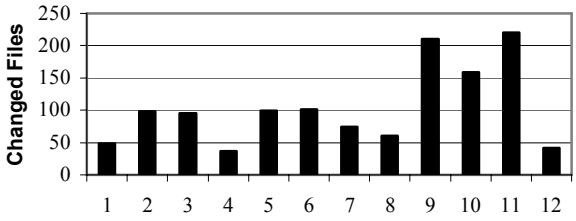

Figure 6: Number of file changes between revisions

Except for two versions of the system, we find that the number of files reported as coupling-increasing to be manageable (often under five files). This observation makes it reasonable to expect that a developer could inspect the complete list of files reported to evaluate whether the last changes to each file could have been suboptimal. To provide a better context for this interpretation, Figure 6 shows the number of files changed between each version considered. As can be seen from this last figure, feature coupling analysis can help narrow the focus of the developer to a number of files about ten times lower than the overall number of changed files.

4.3 Performance

Our approach is only feasible if it can be applied without incurring overhead that would severely disrupt the normal activities of developers. In general, thanks to the various optimizations described in Section 3, we found that our implementation of the approach exhibited acceptable performance characteristics for its intended use. In the rest of this section, we discuss the performance characteristics and tradeoffs corresponding to the different steps of our approach. Unless otherwise noted, the experimental machine for our performance assessments was an IBM T42 Thinkpad laptop computer with a 1.86 GHz Pentium-M processor and 2 GB of physical memory. The analysis implementation was written in C++, compiled using Visual Studio 2005 (with optimizations enabled) and executed on Windows XP SP2.

For our target system, using the "sanity" test suite comprising 70 tests, the entire analysis process requires about 2 minutes. For larger test suites, comprising several thousand tests, the process completes in less than 2 hours.

4.3.1 Executing the Test Suite

In our environment, tests execute on dedicated computer nodes that exploit parallelism between tests and reduce testing latency by sharing nodes between all developers. This system allows developers to test their changes for regression within minutes or hours, depending on the size of the test suite used.

The only part of our approach that affects the testing phase is the line coverage instrumentation, which increases the execution time by 5~10% and requires additional storage requirements to store line coverage information. Roughly 300KB of disk space per test is required, with the data compressed with zlib[5] as it is written.

4.3.2 Recovering the Exhibits & Covers Relations

We merged the recovery of the *exhibits* and *covers* matrices into a single process, centered on the recovery of test-related information from the file system where it is written during the execution of the regression test suite. The computational (and memory) cost of this operation grows linearly with the number of tests, components and features.

On the experimental machine, this phase represents about 1.5 second of computation per test, which is mostly due to file system management (seeking and opening files), I/O (reading), decompression (zlib), decoding the file format, and memory management. This process is the most time-consuming because it is performed serially. This entire process completed after less than 3 minutes for all versions of the program, using the "sanity" test suite, but typically took more than one hour on larger test suites.

When this process has completed, the output is written to a single file, roughly 20MB in size for our target system, containing both the *exhibits* and *covers* matrices in their compressed form.

4.3.3 Computing Feature Implementations

The time required to compute feature implementations is solely bounded by the processor speed. The computational cost of this operation in our implementation grows linearly with the number of features, components and tests. Although the algorithm does not take tests into account, our implementation compresses the *covers* and *exhibits* matrices using test groups. The computational cost introduced by test groups grows linearly with the number of tests in the worst case. However, the practical compression of data (and data processing) we get from working with test groups more than makes up for any added performance cost.

For our target system this processing step executes at a rate of about 50 features per second. The output is an in-memory feature implementation matrix that requires about 2KB per feature of memory.

4.3.4 Feature Associations and Coupling Analysis

The computational time required for calculating feature associations and to perform feature coupling analysis grows quadratically with the number of features, but is positively impacted by the truncation of implementation vectors to constant lengths. This decouples both operations from the specific number of components, resulting in a constant-time operation. Furthermore, the small size of the vectors means that the processor can process almost 100 000 of our implementation products every second. The entire coupling analysis phase takes just 10 seconds on the experimental machine.

4.4 Validation Study

For our initial assessment of our approach, the final question we wanted to answer was whether the files identified with our approach were actually responsible for code decay. This question is a difficult one given that code decay is an abstract concept that is difficult to operationalize [4]. As a starting point, we decided to work with the weaker hypothesis that files identified with our approach correlate with files touched by future bug fixes. To determine whether this hypothesis held in our case, we built contingency tables recording, for each file in our target system and each version of the system, whether the file was flagged as coupling-increasing or not, and whether the file was touched by bug fixes afterwards or not. This strategy is similar to previous studies of dynamic coupling, which have also used future changes as the dependent variable for empirical evaluation [1]. With this data, a standard statistical procedure (the chi-square test of independence) can determine whether increased feature coupling is a predictor of future bug fixes.

For this experiment, we considered all the source files of the system for the 12 revisions used in the rest of our investigation. A file was considered to be "coupling increasing" at a given version of the program if it appeared in the CIC set produced by the application of our technique to that version, using α=1.1 and β=1.5. To determine whether a file was associated with future bug fixes or not, we searched the issue tracking database. A file was considered "buggy in the future" for a version of the program if it was involved in at least one bug fix in the following 4 months.

Table 5 shows our aggregated contingency tables. Each row corresponds to one versions of the system. Columns 2 to 5 present, for each version, the number of files with the characteristics listed in the header. For example, version 5 of the system comprised 14 files identified as both coupling-increasing and buggy in the future.

Table 5: Feature coupling increase as a Predictor of Bugs

Version	CI Buggy	CI Not Buggy	Not CI Buggy	Not CI Not Buggy
1	2	0	302	837
2	7	0	299	835
3	4	0	292	845
4	1	0	295	845
5	14	8	274	845
6	0	0	257	884
7	2	3	240	896
8	11	6	227	897
9	5	3	227	906
10	0	0	235	906
11	0	0	234	907
12	0	0	234	907

[5] http://www.zlib.net/

Because of low values in the first two columns, we could only perform a chi-square test of independence for versions 5 and 8.[6] However, for both versions 5 and 8 the chi-square test indicates a statistically significant relation between the "coupling increasing" and "buggy in the future" variables ($p \leq 0.001$). In other words, our feature coupling increase metric is a good predictor that a file will be touched by a bug fix in the future.

Manual inspection of the files identified as coupling-increasing showed that these files did correspond to code units judged by the developers of the system to be in need of preventative maintenance. Although not surprising, these initial results can already serve to confirm informal observations about the perceived deteriorated state of the coupling-increasing files. Additional research should help improve the precision with which our technique can identify problematic code locations.

4.5 Discussion

Our experience with the current implementation of our feature coupling increase detection technique has allowed us to answer many practical questions regarding the assumptions stated at the beginning of this section.

First, we were able to determine that our approach could clearly identify feature implementation vectors that strongly associate features with components. Empirical evidence (e.g., Figure 3) shows a "natural" distinction between significant and noisy feature implementations. By being able to select and use only "good" feature implementations, we can increase the overall quality of the results produced. However, due to the filtering of noisy feature implementation vectors, some significant feature coupling increases might go undetected simply because the test suite is not able to accurately factor out a feature. When combined with a test selection strategy, it might be advisable to favor or simply add tests that improve feature coverage.

Second, our experience showed that, when aggregated into files, the size of CIC sets constitutes a manageable amount of information for developers. Although we found the size of CIC sets to vary depending on the values of the α and β parameters, the main factor determining the size of CIC sets is the nature of the actual program versions analyzed.

Third, our implementation of the proposed approach demonstrated that it can be used at a reasonable cost (10% slowdown for the execution of the test suite plus a few minutes of additional computation). As such, the total cost will vary greatly based on the size of the test suite executed. However, as in the case of testing, the quality of the results will increase with the number of tests. More experience should help determine in which situations the benefits of the approach are worth the cost.

Finally, we were able to obtain evidence that files identified as coupling-increasing with our approach are more likely to be touched by bug fixes than randomly-selected files. Although we construe this initial result as confirming evidence of the assumptions underlying our approach, our interpretation is subject to the usual threats to validity that must be considered for quantitative studies of this type. In our case, an important consideration is that the phenomenon of code decay might not be adequately measured by the single occurrence of bugs in a file. More detailed, qualitative investigation should help us strengthen the link between CICs and actual code decay.

[6] The chi-square test is generally considered invalid (but not necessarily failed) if a cell value is lower than 5.

5. RELATED WORK

The seminal work motivating our research is the investigation of code decay in a large-scale phone switching system conducted by Eick et al. [4]. In their study of the 15-year history of the system, Eick et al. analyzed a number of decay indices such as the span of changes (number of files touched), which is shown to increase as the software evolves. Although this study motivated our research by providing evidence of code decay, our decay assessment strategy differs from Eick et al.'s code decay indices in that we do not analyze the history of the code, but rather immediate differences between versions. This difference in strategy is mainly due to different research goals. While Eick et al. sought to provide evidence of long term decay, we were interested in preventing such decay by providing an early warning system.

A large number of approaches have been proposed that involve the analysis of a running program for purposes that range from the broad (e.g., program understanding [2]) to the very specific (e.g., impact analysis [8]). In this space, a few approaches relate more closely to our work through either their relationship to coupling analysis or their reliance on the concept of feature.

Arisholm et al. investigated how dynamic coupling measures can help assess various properties of a software system [1]. The dynamic measures studied by Arisholm et al. include characterizations such as the number of messages sent by each object, the number of distinct methods invoked by each method, etc. This work does not take into account the notion of feature as a separate entity that can span multiple modules. Nevertheless, the results of this study are consistent with ours, in that "dynamic export coupling measures were shown to be significantly related to change proneness" [1, p. 505].

The approach developed by Licata et al. [9] produces “feature signatures” by taking the textual difference between two versions of a software system and analyzing the number of distinct test suites that execute each code block that differs between the two versions. The main assumption behind the concept of feature signatures is that a test suite corresponds to a feature. Feature signatures are represented as histograms plotting the number of “difference blocks” executed by the number of distinct test suites. Licata et al. propose to use feature signatures to recover the rationale associated with a change, to understand relationships between test suites, and to identify scattered code associated with a feature. There are a number of important differences between the approach of Licata et al. and our feature coupling detection approach. First, we do not assume that test suites map one-to-one with features, but recover feature associations using a separate process. Second, we rely primarily on differences in the execution traces, as opposed to differences in the source code text. Finally, our approach produces a specific metric (increase in feature coupling), as opposed to a general representation of the impact of the change on test suites.

Although the main focus of this research is not specifically the location of features in source code, the technical foundations for this work has benefited from a number of dynamic analysis-based feature location techniques. We conclude this discussion of related work with a description of feature location techniques that have inspired the design and implementation of our approach.

The Software Reconnaissance technique developed by Wilde et al. identifies features in source code based on an analysis of the execution of a program [13, 14]. Software Reconnaissance determines the code implementing a feature by comparing a trace of the execution of a program in which a certain feature was activated to one where the feature was not activated. Wilde at al. also proposed a second formulation of Software Reconnaissance where components are attributed implementation scores based on the frequency of their occurrence in a test suite, and the frequency of their occurrence together with the feature to locate [13]. This definition is the basis for our feature association calculations.

Eisenberg and De Volder extended Software Reconnaissance by devising more sophisticated heuristics for determining component implementation scores [6]. They combine both of Software Reconnaissance's formulations by requiring the user to provide sets of exhibiting and non-exhibiting tests, and then performing multiple probabilistic analyses on them. They combine the result of the analyses into a final implementation score which is used to assign components to a feature.

Finally, Eisenbarth et al. [5] proposed a different extension to the ideas of Wilde et al., by producing the mapping between components and test cases using mathematical concept analysis. Their approach, however, requires more human intervention than would be practical for our application.

6. CONCLUSION

One important challenge for organizations involved in software maintenance is to ensure that the repeated modifications applied to a software system do not result in a gradual decay of the system's code base. Unfortunately, symptoms of code decay can be difficult to detect in the short term, and clear evidence may only appear once it is too late to easily remedy the situation.

In an attempt to mitigate this problem, we proposed to analyze a system for symptoms of potential decay with every execution of a regression test suite. Our technique is based on the assumption that an increase in the level of association between the implementation of two features may indicate the introduction of unplanned dependencies, and constitutes a symptom of potential code decay. By analyzing the execution of regression tests, we automatically determine the degree of coupling between features based on the sections of code they execute in common. With this information, we can then identify any section of code that contributes to an increase in feature coupling between two different versions of a system.

We assessed the feasibility of our approach by implementing it and integrating it with the development environment of a proprietary 3D graphics software comprising over 100 000 lines of C++ source code. This experience provided us with valuable insights about the engineering tradeoffs required to integrate feature coupling increase detection with regression testing in practice. For example, we were able to measure the tradeoff between the size of the test suite used (which impacts execution time) and the number of features that can be located with enough accuracy to be analyzed for coupling increases.

Our experience has also helped confirm that source files identified with our approach may be in need of preventative maintenance. A small experiment confirmed that files identified by our approach were significantly more likely to be affected by change requests in the future. Although we expect that additional experimentation will help us better understand the link between increased feature associations and code decay, we conclude that detecting increases in feature coupling as part of regression testing is a feasible and promising approach for maintaining the quality of software systems.

7. ACKNOWLEDGEMENTS

The authors are thankful to Harold Ossher and to the anonymous reviewers for their thorough and insightful comments on this paper. This work was supported by an NSERC Discovery Grant and by NVIDIA Corporation.

8. REFERENCES

[1] E. Arisholm, L.C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30 (8), pp. 491-506, 2004.

[2] T. Ball, The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of Lecture Notes in Computer Science, Springer-Verlag, pp. 216-234, 1999.

[3] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3), pp. 225-252, 1976.

[4] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering,* 27(1), pp. 1-12, 2001.

[5] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3), pp. 210-224, 2003.

[6] A. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 337-346, 2005.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pp. 220-242, 1997

[8] J. Law and G. Rothermel, Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering*, pp. 308, 2003.

[9] D. R. Licata, C. D. Harris and S. Krishnamurthi. The feature signatures of evolving programs. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 281-285, 2003

[10] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 107-119, 1999

[11] D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pp. 279-287, 1994

[12] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pp. 86-96, 2002

[13] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the Conference on Software Maintenance*, pp. 200–205, 1992.

[14] N. Wilde and M.C. Scully. Software Reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1), pp. 49-62, 1995.