DETECTING INCREASES IN FEATURE COUPLING USING REGRESSION TESTS

by

Olivier Giroux

School of Computer Science
McGill University, Montreal

February 2007

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Copyright © Olivier Giroux, 2007.

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 proprietary 3D graphics software. Our field study showed that, for files in the target system, causing increases in feature coupling is a significant predictor of future modifications due to bug fixes.

RÉSUMÉ

Chaque modification appliquée à un système logiciel peut y introduire de nouvelles failles telles que des dépendances structurelles entre ses éléments unitaires. Il peut être difficile de percevoir ce processus de dégradation de la qualité puisque qu'il n'implique pas nécessairement une dégradation fonctionnelle. Nous présentons ici une nouvelle technique permettant à l'ingénieur logiciel de comprendre l'impact de ses modifications sur les dépendances structurelles dans le contexte des fonctionnalités du système. Notre approche automatisée identifie les éléments logiciels ainsi potentiellement dégradés dès que le logiciel est soumis à sa procédure de vérification habituelle. L'ingénieur peut alors inspecter les résultats de notre analyse pour déterminer si la qualité de la modification appliquée est adéquate. Nous avons déployés notre système dans un environnement logiciel graphique 3D privé sous C++. Notre étude démontre que, pour ce système, l'addition de dépendances structurelles est un précurseur de modifications rectificatrices dans le futur.

ACKNOWLEDGEMENTS

I first wish to thank my supervisor Martin Robillard for his tremendous help, without whom I probably would have faltered and settled for less. As far as I am concerned Martin personifies excellence in research.

It is clear to me that this whole adventure would never have taken off without Gerald Ratzer's gift to me of opportunity and patience. Gerald had a compelling vision of what my path could be and he often nudged me in the right direction after removing barriers that had laid in the way. I very much enjoyed the years we worked together at the school, and I know we made a difference to the thousands of students we worked with.

On a personal note I am grateful for the patience of the people who need to put up with me on a daily basis. I know I imposed quite a bit of stress on Mirianne, my lovely wife. The support of my wife and family was a necessary ingredient.

Finally, I also wish to thank Harold Ossher and the anonymous reviewers of our published paper for their thorough and insightful comments.

This work was supported by an NSERC Discovery Grant and would not have been possible without the support of NVIDIA Corporation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 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. For example, the early success of Internet browsers in the late 1990's revealed how poorly their original design accounted for security and pushed the issue to the forefront of user concerns [7].

These 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. Conversely, the discontinuation of regular modifications to a software system is a sign of abandonment [26].

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 [5, 12, 26], a phenomenon referred to as code decay [12].

Software modifications that do not cause any regression faults, may 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 more difficult to change [25]. In other words, in a well-encapsulated system a wide variety of changes can be made to one module without affecting another, whereas in a system with poor encapsulation one must understand the impact of each change across all modules that may depend on components affected by the change.

This dependency relationship, called *coupling* [29], is a likely agent of code decay because it can make it harder to further modify the system.

1.1 Example of code decay via coupling

Modifications to a system exhibiting low coupling tend to be simple because the developer needs only consider the impact on a few modules to understand the scope of his modification (Figure 1, left). In this example, a change made to a module of the system may impact two more modules due to coupling, but existing encapsulation prevents more modules from being directly impacted.

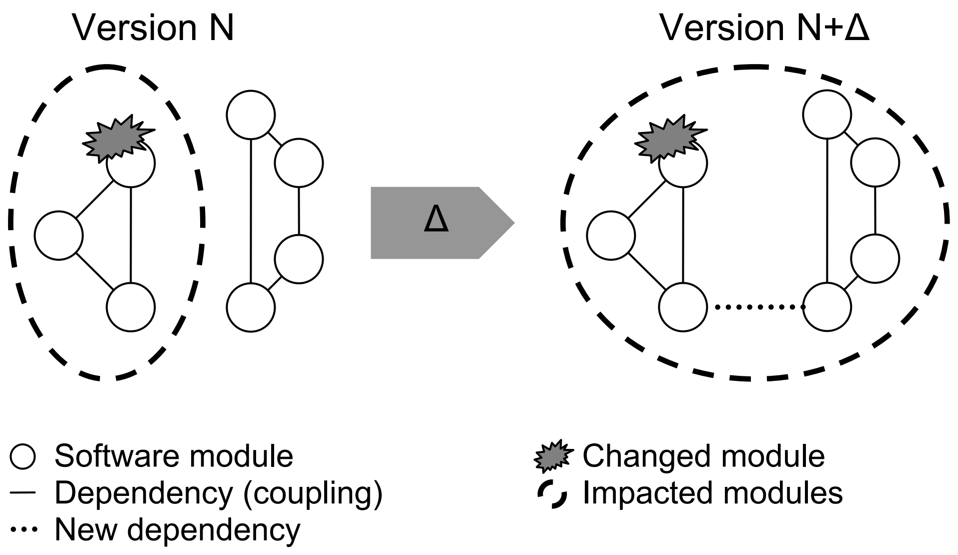

Figure 1: Coupling and Impact Sets

With the introduction of new coupling dependencies in the system, the number of modules potentially impacted by any change increases (Figure 1, right). In order to perform a correct modification to the new system, a developer would now need to consider the impact of the change on a larger set of related modules, resulting in both a higher change cost and higher risk of introducing functional regressions.

Any system must exhibit some amount of coupling, or the system's modules cannot communicate with one another. However, the introduction of *unplanned* coupling further increases the burden on developers in many ways:

1. The system as a whole becomes more difficult to understand because new unplanned behavior can emerge from inter-dependencies, including new defects.
2. Each module is more difficult to understand because other modules must also be understood together, resulting from the breakdown of encapsulation, while certain module compositions may become unworkable.
3. It is more difficult to verify and test the behavior of the new system, lowering the effectiveness of unit testing, and possibly relying heavily on more expensive integration testing to protect from defects.

Finally, not all developers may be aware when additional coupling is introduced in the system. Eventually, developers may each view the same system differently and will be more at risk of introducing inconsistencies leading to defects. Even the system's original architects may no longer understand the system as a whole.

1.2 Beyond functional verification

It is not easy to define code decay operationally because it is, at its core, a human problem. A simple formulation of code decay could follow a "black box" approach, combining both human and technological aspects of the problem by asking three high-level questions about the evolution of a software project:

1. Is the cost of making changes to the system increasing over time?
2. Is the time required to complete changes inflating?
3. Is the quality of the software deteriorating?

An affirmative answer to any of these questions could be evidence of code decay, but each may be difficult to assert by engineers and managers in the field. Belady and Lehman analyzed empirical data gathered during the development of OS/360 at IBM [5] with the goal to build a model of software development that accounted for these factors. The data they presented suggests that the three signs of decay above were present throughout the long evolution of OS/360.

Unfortunately, once the effects of code decay become readily apparent, it may prove more expensive to remedy the situation than to abandon the system and start anew. However it may be possible to detect potential symptoms, or risk factors of code decay. If assessed by developers early enough, the decay may be corrected at a lower cost.

The intuition that guided the research described in this thesis is that an increase in the amount of overlap in 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 automatically detected and reported to developers for closer inspection.

Unfortunately, the implementation of features is not always neatly encapsulated in a single module [17, 24], a situation which precludes the trivial use of standard automated coupling metrics to detect this symptom.

1.3 Our focus

This dissertation shows that the execution of test suites may be used to detect signs of increased overlap between the implementation of features as a sign of code decay. To compute the overlap we look for the implementation of features in code using *Feature Location* techniques. Feature location is a form of reverse engineering whereby high-level concerns (features) are mapped to low-level components (code) through either an interactive or variably automated process, based on mining or instrumentation data. Existing automated feature location techniques (see Chapter 2) serve as the foundation of our work.

We believe that if feature location tools help developers understand aspects of software architecture in practice, the evidence used to document the architecture should be a relative invariant of the system. That is to say that software architecture is expected to be highly inertial in a software system undergoing maintenance. Insofar as the abstractions and inter-dependency of feature implementations are elements of the software architecture, they too should be relatively invariant in that phase of the software lifecycle.

By automatically determining feature locations across changes to the system as they are applied on the source repository, we can inform developers of violations to the invariance. Our technique computes *Feature Associations*, the degree of codependence between the implementations of separate features, and reports on potentially harmful variations between versions of the system. 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 present the novel concept of feature coupling, as well as a 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.

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 technique showed that its computational 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. Our contributions 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.

1.4 Structure of this dissertation

In Chapter 2, we describe foundational work. In Chapter 3, we provide the details of our technique for detecting increases in feature coupling. We then describe our application of the technique in Chapter 4 and our initial experience with the technique along with a validation study in Chapter 5. We present a qualitative analysis in Chapter 6 and finally conclude in Chapter 7.

Chapter 2
Background

The seminal work that motivated this research is the investigation of code decay in a large-scale phone switching system conducted by Eick et al. [12]. 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.

When the architecture of a system can be stated explicitly, the effect of code decay on software architectures can also be construed as the introduction of differences between an intended and an actual architectural design. A number of approaches have been proposed to detect inconsistencies between intended and actual designs. For example, Murphy et al. proposed software reflexion models, a technique allowing developers to easily model the architecture of a system and to automatically verify the conformance of the actual system to the posited architecture based on a static analysis of the system [23]. Sefika et al. proposed to establish the conformance of a system to design-level rules (e.g., an implementation of the Mediator design pattern [15]) using a combination of static analysis and dynamic analysis [27]. More recent developments in this area include ArchJava [1], an extension to Java allowing developers to specify the architecture of a system directly in the code (and to automatically verify the conformance of the code to the architecture), and the IntensiVE environment [20], which allows developers to document regularities (patterns) in the structure of a system, check whether the patterns hold as the system evolves, and report discrepancies between documented and observed patterns to developers. Although the motivation behind our approach (to mitigate code decay) is the same as the one pursued with the work describe above, our strategy was different. While the conformance verification approaches rely on a precise, coarse-grained, and explicitly-specified architectural model, our present approach relies on an approximate, fine-grained model that can be (partly or completely) inferred automatically from the test suite.

2.1 Coupling

The coining of the term "coupling" is attributed to Stevens [29], who defines it as "inter-relations between modules" that make the system more difficult to understand, change and correct, increasing the complexity of the whole. Stevens' definition was focused on the dominant paradigm of his time, but was later ported to Object-Oriented Programming by Coad and Yourdon [10], who added class inheritance and friendship as evidence of coupling.

Chidamber and Kemerer further extended the object-oriented definition of coupling with the popular Coupling Between Objects (CBO) metrics [8, 9], which influenced many recent works on coupling. CBOs form the basis for work on static [4] and dynamic [22] coupling. The CBOs have also been independently related to change-proneness in many publications [4, 6, 11, 32].

2.2 Analysis Techniques

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 [3]) to the very specific (e.g., impact analysis [18]). 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.

2.2.1 Coupling detection techniques

Arisholm et al. investigated how dynamic coupling measures can help assess various properties of a software system [2]. 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 (as reported in Chapter 5), in that "dynamic export coupling measures were shown to be significantly related to change proneness" [2, p. 505].

Mitchell and Power later contrasted the predictions of static coupling metrics such as CBOs with Arisholm's dynamic inter-relation metrics between objects instances [21]. They found that dynamic metrics reveal a different picture than static metrics. As a result they propose that the results are best interpreted in the context of coverage information.

2.2.2 Feature location techniques

Although the main focus of this research is not specifically the location of features in source code, the technical foundations for this work have benefited from a number of dynamic analysis-based feature location techniques. We conclude this survey 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 [31, 32]. 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 [31]. 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 [14]. 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.

Eisenbarth et al. [13] 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.

The approach developed by Licata et al. [19] finds “Feature Signatures” by comparing execution traces of tests over consecutive versions of a program. A feature signature is a group of tests, assumed to correlate well with features, that exercised a given portion of the changed code. Feature signatures account for the features affected by the changes spanning two program versions, and serve both as a feature-location and a change-characterization mechanism. By summarizing feature signatures into impact size histograms, Licata et al. show that feature signatures are able to distinguish between localized and infrastructure changes. By clustering changed code blocks with similar signatures, they also show that feature signatures can locate cross-cutting feature code over the entire span of the modified program.

A final use of feature signatures draws more analogy to our work. Feature signatures may be used to investigate the structure of tests: they can describe the relationship between tests as a concept lattice [28]. Licata et al. assume that tests map directly to features, hence their lattice simultaneously expresses feature relationships and resembles our concept of feature associations. Although both locate features in the source code by matching code blocks exercised by similar features through dynamic analysis, the essential distinction is that feature signatures rely on code differences between program versions to reveal features, whereas feature associations rely on the differences between tests.

Chapter 3
Feature 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 [2]. 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. This technique was first described in a paper presented at the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering [16].

In this work, 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 [17, 24]. 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.

3.1 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 features, 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 F_t and covers a set of components C_t, where $F_t \subseteq F$ and $C_t \subseteq C$. 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).

3.2 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, … a_n) and b = (b_1, b_2, …, b_n), the implementation product $a \otimes b$ is defined as*

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

Figure 2: Implementation Product

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[a_{ik}] be an m $\times$ n matrix, and let B = B[b_{kj}] be an n $\times$ s matrix. The matrix implementation product A $\otimes$ B is the m $\times$ s matrix C = C[c_{ij}],where c_{ij} is the implementation product of the i^th row vector of A and the j^th 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 exercises. We define a feature implementation FI as FI = exercises^T $\otimes$ covers.*

3.2.1 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.

	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	

Table 1: Covers matrix for the example program

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.

	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	

Table 2: Exercises^T matrix for the example program

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

	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

Table 3: Feature implementation for the example program

For example, taking the implementation product of row F_1 in exercisesT 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.

3.3 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 a space of components (Figure 3) where the components are the dimensions.

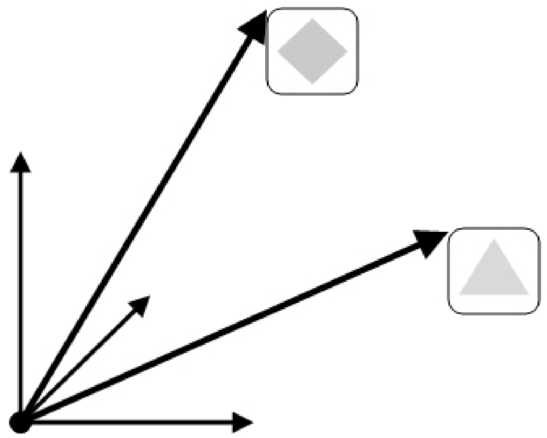

Figure 3: Conceptual alignment of features in a feature space

The higher the association 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.

3.3.1 Example

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

	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

Table 4: Feature association for the example program

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. This operation is not a requirement for our technique however, and we recommend that it be avoided in production environments where performance is a concern. Although normalization would not become the bottleneck of the analysis, its cost remains significant and should be avoided whenever possible.

3.4 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 $[1..\infty)$ (see Chapter 5).

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:*

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

Figure 4: Coupling-Increasing Condition

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:*

```
param: $P^*=(C^*, F^*, T^*)$: Modified Program
param: $FI[f_{ij}]$ & $FI^*[f_{ij}^*]$: Feature
                                Implementations
param: $CIF[cif_{ij}]$: Coupling-Increasing Features
var: CIC={}: Coupling-Increasing Components
for i = 1..$|f_i|$ (where $f_i$ is a row of FI)
  for j = 1..$|f_i|$, $i \neq j$
    if $cif_{ij} = 1$
      for k = 1..$|f_i|$
        if $f_{ik}^* \bullet f_{jk}^* > f_{ik} \bullet f_{jk}$
          CIC $\leftarrow$ CIC $\cup$ c | c is the component
               corresponding to column k in FI
        end if
      end for
    end if
  end for
end for
return CIC
```

Figure 5: Coupling-Increasing Component Algorithm

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.

3.5 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 4.4.2).

Chapter 4
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. 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.

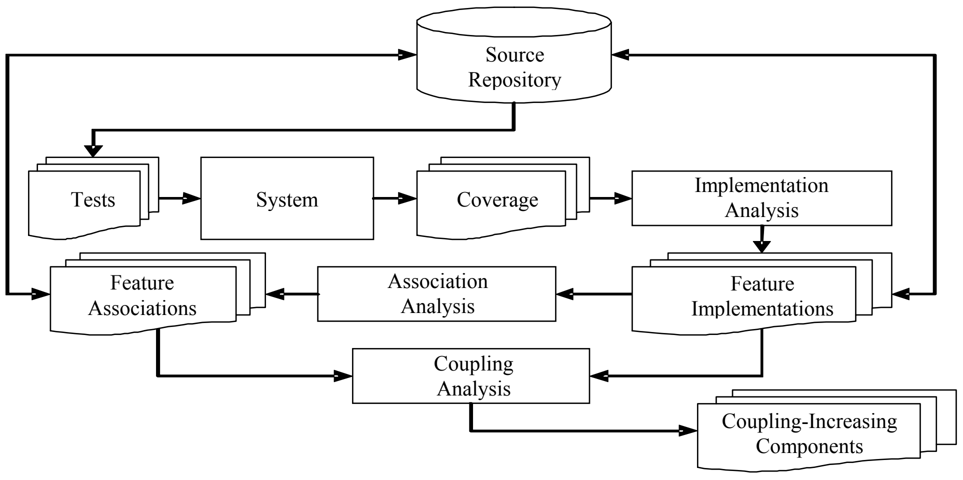

Figure 6 : Implementation Diagram

Our current implementation (depicted in Figure 6) 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 4.1). 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 4.2).

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 4.3). 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 4.4), 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 3.4. 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.

4.1 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 [30], 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 5.3 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. The core implementation of this technique is described in Section 4.3, with C++ source provided in Appendix A (the “Aggregate” class).

4.2 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 (Figure 7: Excerpt from a system log). 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 in 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…).

```
PRIMITIVE_TYPE = TRIANGLE_STRIP
BLEND_MODE = WRITE_SRC_ONLY
USING_FMAD = TRUE
USING_TEX2 = TRUE
ENABLE_L1_WRITE_THROUGH = FALSE
...
```

Figure 7: Excerpt from a system log

Since the exact details of the logging feature are proprietary, for the purpose of this thesis 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 produces a very fine-grained definition for features, 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 4.5).

Although the number of features detected remains much less than the number of lines of code in the system, our feature identifier tokens are much larger than a simple integer and their analysis produces physical data sets of similar size when stored uncompressed. Like the covers matrices, exercises matrices are also naturally sparse and their cost can be made manageable using the same grouping strategy (see Section 4.3).

4.3 Sparse Matrix Strategy

We implemented our sparse compression strategy in an abstracted module in order to reuse it for both *covers* and *exercises* matrices. The "Aggregate" class implementation is provided in Appendix A and deserves an expanded commentary because it is essential to our implementation. Two concerns influenced its design: efficiency of final storage in space (exploiting sparseness) and run-time performance of the compression algorithm. In order to share a core implementation we abstracted the matrix’s major index (tests), dubbed “key” in the sources, and the minor index (line of code, or feature), dubbed “name”.

The output of the compression operation is a pair of data structures, a *set of sets of keys* and a *map of names to [pointers to] sets of keys*. The first is the set of unique groups of keys (tests) to which names (lines of code, or features) are associated, hence unique compressed (sparse) columns, while the second is the transpose of the association (*covers*, *exercises*) table.

Two additional indirections were later added to improve run-time performance significantly: space-inefficient temporary working sets and unique name/key tables.

Firstly, in order to accelerate operations on the data structure, an additional *map of names to sets of keys* is used to accumulate many small operations into larger transactions. When a change is applied to a named entry, this map is first populated with a copy of its associated key set, and subsequent changes are applied there rather than in the final data structure. This allows changes localized around a name to be performed on a smaller, faster structure. At the end of the process, the Aggregate (Aggregate::compressKeySets) folds all the copies in this map back into the main data structure by searching for and assigning matching sets, creating new sets or destroying unused sets.

Secondly, a pair of string tables alters the *actual* data structures to use pointers to names (dubbed name handles) and pointers to keys (key handles) instead of names and keys proper (both strings). This change is functionally transparent for operations that only require names and keys to be equality-comparable, which dominate the compression aspects of the technique. Not surprisingly, we observed the difference in performance between using string comparisons and pointer comparisons to be very significant (around two orders of magnitude). The reduction in storage size also achieved with this optimization was not as significant.

4.4 Feature Implementations

The implementation of the computational support for feature implementations as described in Section 4.2 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[1] 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

[1] The groups are seen in the sparse matrix, as mentioned in 4.1.

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. Manual inspection of the most important features of the target system showed that implementation vectors typically had a clear signal contained in the first 50-100 components (see Section 5.1). This measurement is likely a property of our target system and should be evaluated again for each different 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 4.5).

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.

4.5 Feature Coupling Analysis

Our implementation of feature coupling analysis is faithful to the algorithm described in Section 3.4. 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.

4.5.1 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. 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 3.4.

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 5.2).

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 3.4, 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):

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

Figure 8: Sensitivity Threshold

Implementation vectors that show clear features locations in the source code share a characteristic shape where components vary significantly in implementation values (see Section 5.1). The sensitivity threshold uses the standard deviation to select only implementation vectors that contain significant differences in implementation values.

4.5.2 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.

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.

4.5.3 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 4.2) 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.

4.6 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. 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.

A final sensitive choice is the granularity in time at which our technique is applied. The comparison of very distant versions of the target system can produce large differences, while the comparison of very close versions, very few. In order to pinpoint specific changes as causes of decay, our preference is for the comparison of close versions. In turn this increases the risk of failing to recognize very slow increases in coupling. A simple attempt to mitigate this effect could involve multiple comparisons against variably-distant versions in the repository – this is a viable option for a future implementation.

Chapter 5
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:

1. Feature implementation vectors meaningfully associate components with features;
2. The CIC sets produced are usable by developers;
3. The computational cost of the approach is acceptable;
4. 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.

5.1 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 4.5.1).

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 4.5.1 holds and significant otherwise. Figure 9 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.

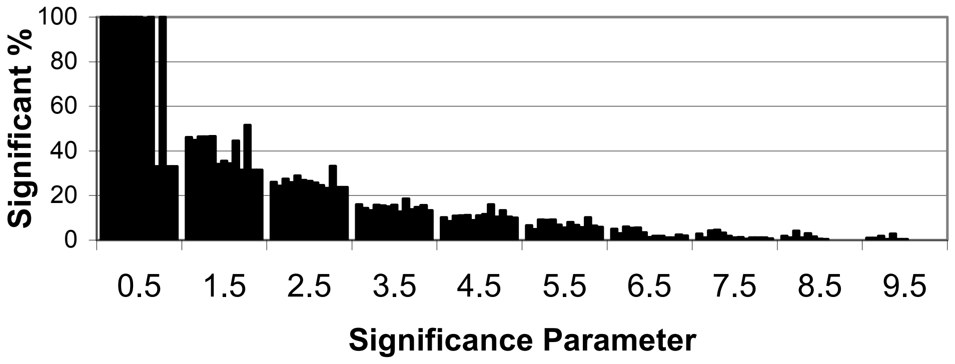

Figure 9: Effect of the β parameter on noise detection

The results of significance testing vary smoothly as β changes. However, groups of features that exhibit similar implementation vector distributions cause local discontinuities as they pass or fail the significance test together. This grouping effect can be caused equally by similarity at the source level or by using poorly differentiated tests.

We selected β=1.5 for our system because we felt it provided adequate protection from noise without eliminating weaker 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 10 shows the value distribution of both a significant (solid line) and noisy feature implementation (dotted line), sorted by decreasing degree values.

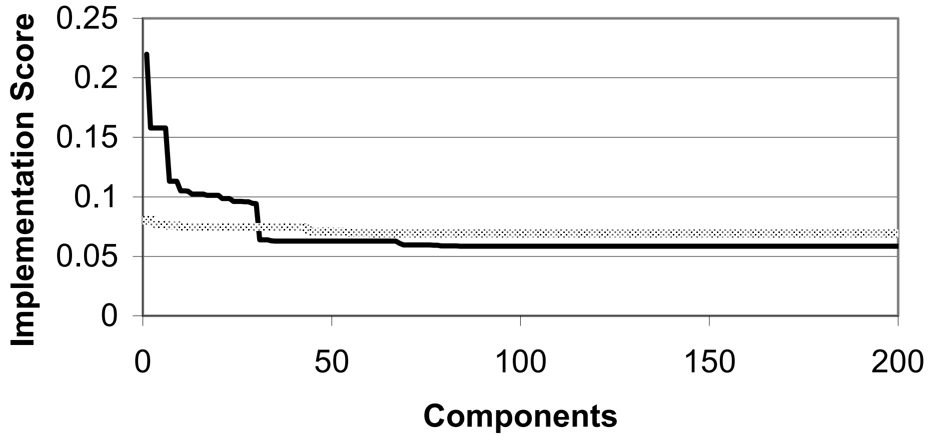

Figure 10: Sample feature implementations

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.

5.2 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.

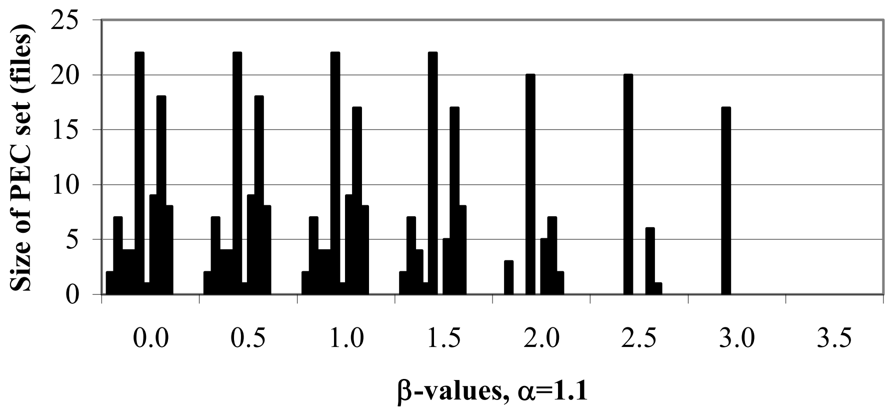

Figure 11: Effect of β on the size of CIC sets

Figure 11 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.

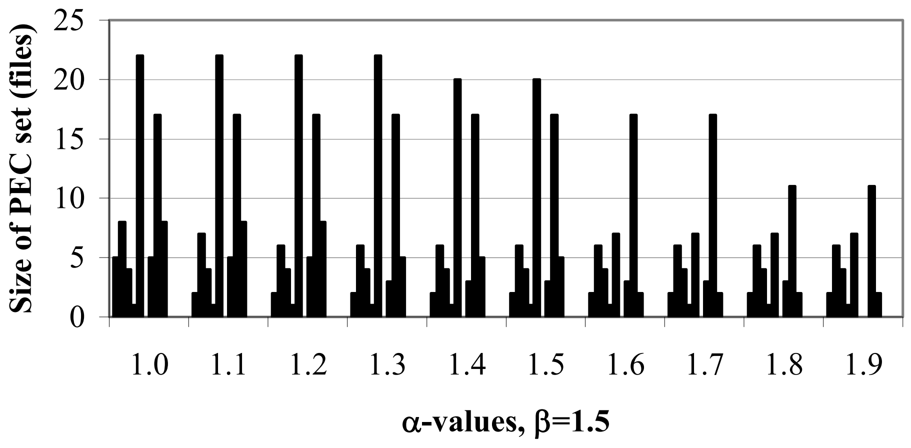

Figure 12: Effect of α on the size of CIC sets

Figure 12 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.

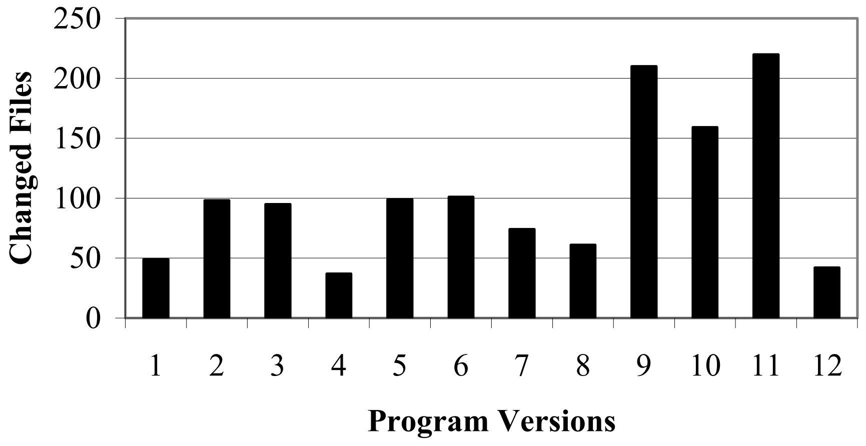

Figure 13: Number of file changes between revisions

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 13 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.

5.3 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 4, 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.

5.3.1 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[2] as it is written.

5.3.2 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

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

from the file system where it is written during the execution of the regression test suite. The computational 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.

5.3.3 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.

5.3.4 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.

5.3.5 Summary

All analyses were performed on an IBM T42 Thinkpad laptop computer with a 1.86 GHz Pentium-M processor and 2 GB of physical memory. The following table summarizes the resource requirements of applying our technique:

	Overhead Time	Overhead Memory	Overhead Storage
Testing	5~10%	-	300Kb/Test
Aggregation	1.5s/Test	2Kb/Component	~20Mb
Location	0.02s/Feature	2Kb/Feature	-
Association	10s	2Kb/Feature	-

Table 5: Resource Requirements

In our environment, the total time required to run the test suite itself dominated the cost of applying our approach thereafter.

5.4 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 [6]. 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 [2]. 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.

Versio n	Coupling Buggy	Coupling Not Buggy	Not Coupling Buggy	Not Coupling 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

Table 6: Feature coupling increase as a Predictor of Bugs

Table 6 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.

To ascertain whether there is some truth in our hypothesis, that feature coupling is a predictor of bugs in a software system, we performed a c*hi-quare test of validity*. This test is used to reject a null-hypothesis, in this case that feature coupling has nothing to do with bugs in software systems, by computing the probability ("p-value") that the results would be purely due to chance.

Because of low values in the first two columns[3], we could only perform a chi-square test of independence for versions 5 and 8.

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

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 ($p \leq 0.001$, or 0.1%). In other words, our feature coupling increase metric is possibly 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.

5.5 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 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.

Chapter 6
Qualitative Analysis

We present a qualitative analysis using the results of the empirical study described in Chapter 5. We manually inspected the report of our tool for one of the studied revisions, and comment on our findings based on our 4-year experience with the system.

Through this analysis we sought to qualify the strengths and weaknesses of our approach, and determine if the approach is helpful at finding recently-modified code that needs the attention of programmers. Also, in keeping with our initial motivation, we looked for concrete signs of code decay in the coupling-increasing components identified by our tool. These signs would consist primarily of new or modified components that make the system more complex to understand, hence making further changes more difficult, based on our experience with the system.

6.1 Initial observations from the case study

Over the three-month period analyzed, four out of five result sets returned by our tool pointed at the same module, the implementation of one of the key execution kernels that make up the system. This kernel decodes and executes instructions that are part of the input to the system, and thus directs the overall system's operation. The results showed that certain instructions and independent features of the kernel itself became more tightly coupled than before. Figure 14 shows the complete report[4] resulting from the application of our analysis between versions #4 and #5, separated by 6 days and 99 file modifications.

The report includes the names of coupled feature pairs whose association scores have increased due to a change in the system, and the association score prior to and after the change. The report ends with the names of the components (source files of the target system's code base) that form the body of evidence for the increased association scores. These are components whose contributions to the association score have themselves increased for at least one of the feature pairs mentioned in the report.

```
C2R & RAM      association INCREASED:  10.1% -> 22.8%
C2R & SAM      association INCREASED:  10.1% -> 22.8%
R2C & RAM      association INCREASED:  10.1% -> 22.8%
R2C & SAM      association INCREASED:  10.1% -> 22.8%
SM & ATE       association INCREASED:  43.2% -> 92.9%
SM & BCA       association INCREASED:  36.7% -> 85.2%
SM & BCB       association INCREASED:  36.7% -> 85.2%
SM & BCG       association INCREASED:  36.7% -> 85.2%
SM & BCR       association INCREASED:  36.7% -> 85.2%
SM & R2A       association INCREASED:  35.8% -> 77.4%
TEX & RAM      association INCREASED:   9.3% -> 20.2%
TEX & SAM      association INCREASED:   9.3% -> 20.2%
VSIII & ATE    association INCREASED:  43.5% -> 93.2%
VSIII & SM     association INCREASED:  38.0% -> 96.2%
VSIII & GE     association INCREASED:  38.5% -> 81.6%
VSIII & GMOVC  association INCREASED:  38.5% -> 81.6%
VSIII & GORC00 association INCREASED:  38.5% -> 81.6%
VSIII & GORC01 association INCREASED:  38.5% -> 81.6%
```

[4] Note that the names of features and components (files) presented in this chapter have been changed (shortened) to bear resemblance to their real names without revealing them.

```
VSIII & GORC02 association INCREASED:  38.5% -> 81.6%
VSIII & GORC03 association INCREASED:  38.5% -> 81.6%
VSIII & GOT    association INCREASED:  38.5% -> 81.6%
VSIII & BCA    association INCREASED:  36.7% -> 85.5%
VSIII & BCB    association INCREASED:  36.7% -> 85.5%
VSIII & BCG    association INCREASED:  36.7% -> 85.5%
VSIII & BCR    association INCREASED:  36.7% -> 85.5%
VSIII & RAM    association INCREASED:   9.0% -> 21.2%
VSIII & SAM    association INCREASED:   9.0% -> 21.2%
VSF & ATE      association INCREASED:  43.5% -> 93.2%
VSF & SM       association INCREASED:  38.0% -> 96.2%
VSF & GE       association INCREASED:  38.5% -> 81.6%
VSF & GMOVC    association INCREASED:  38.5% -> 81.6%
VSF & GORC00   association INCREASED:  38.5% -> 81.6%
VSF & GORC01   association INCREASED:  38.5% -> 81.6%
VSF & GORC02   association INCREASED:  38.5% -> 81.6%
VSF & GORC03   association INCREASED:  38.5% -> 81.6%
VSF & GOT      association INCREASED:  38.5% -> 81.6%
VSF & BCA      association INCREASED:  36.7% -> 85.5%
VSF & BCB      association INCREASED:  36.7% -> 85.5%
VSF & BCG      association INCREASED:  36.7% -> 85.5%
VSF & BCR      association INCREASED:  36.7% -> 85.5%
VSF & RAM      association INCREASED:   9.0% -> 21.2%
VSF & SAM      association INCREASED:   9.0% -> 21.2%
   vcf.cpp     v.cpp       c.h         rec.cpp     rec.h
   cbr.h       ls.h        ct.cpp      ct.h        cs.cpp
   cp.cpp      ccf.cpp     tfco.cpp    cbr.cpp     b2o.cpp
   cs2o.cpp    ps.h        tvt.cpp     tvst.cpp    tvsh.cpp
   tt.cpp      tarf.h
```

Figure 14: Complete tool report showing coupled features

The features VSF, VSIII, BCA, BCB, BCG, BCR, GORC00-03, GE, GMOVC, GOT, SM and ATE are features of the execution kernel itself. They control its behavior at the beginning and end of the program execution, and affect how input data is processed. The features TEX, RAM, SAM, C2R, R2C and R2A are instructions that the execution kernel interprets as commands when found in the inputs to the system. The analysis report thus reveals the addition of coupling between various features of the kernel and its instruction set.

6.2 Evaluation of the evidence

The report contained some surprising results, both in terms of the feature associations, and the components presented.

From experience, we could discount a number of components because they were not meaningfully related to the features listed in the report.

Component	Useful	Related Features in Set
vcf.cpp	No	None
v.cpp		
rec.cpp		
c.h		
rec.h	Yes	R2C, C2R
cba.h	No	None
ls.h		
ct.cpp	Yes	R2C, C2R
ct.h		
tarf.h	Yes	TEX, R2A, RAM, SAM
cs.cpp	Yes	R2C, C2R
cbr.cpp	No	None
cp.cpp	Yes	R2C, C2R
ccf.cpp		
tfco.cpp	Yes	TEX
b2o.cpp		
cs2o.cpp		
ps.h	No	None
tvt.cpp	Yes	VSF, VSIII, GE, GMOVC, GOT, GORC00-03
tvst.cpp		
tvsh.cpp		
tt.cpp	Yes	VSF, VSIII, GE, GMOVC, GOT, GORC00-03, R2C, C2R, R2A, TEX, RAM, SAM

Table 7: Usefulness of reported components

Table 7 lists the components from the report of Figure 14, and denotes for each component whether we believe (from experience) it was useful to reason about the reported coupling, and which features are related to it.

Similarly the ATE, BCA, BCB, BCG and BCR features are not related the components listed in the report. Overall, 8 out of 22 components and 5 out of 20 features in the report appear uncorrelated to the rest of the evidence presented. We believe both types of errors are due to testing with a test suite of insufficient size (the “sanity” set described in Section 4.5.1).

In Sections 3.4, 4.3, 4.5.1 and 5.1 we discussed the effects of insufficient testing of features, and described mechanisms we put in place to minimize the effect of noisy data. The sensitive portion of our approach centers on our application of a modern feature location technique, which produces relatively inaccurate results when applied on indiscriminate inputs. Together, the α (Section 3.4) and β factors (Section 4.5.1) form a tunable heuristic that diminishes the impact of noise in the feature-location data, but cannot truly eliminate it. The best way to control noise is to better condition the inputs to feature location, such using larger suites of tests that are more closely focused on features.

We attribute two types of errors in the report to noisy feature-location data:

1. Incorrect components were attributed some of these features (which ones is not specified in the report) because other features that are implemented by these components are also present in

the tests that were run. This is also why VSF and VSIII report the same coupling gains with the same features. These two features must be used together to function properly, so the feature location technique assigns the same components to both (there is no exclusivity between features, only between tests).

2. Incorrect coupling is attributed to the features ATE, BCA, BCB, BCG and BCR because they are always exercised in the "sanity" tests where SM, VSF and VSIII are also exercised, preventing our feature location technique from properly separating them. The association score is not 100% because the reverse is not true (ATE, BCA, BCB, BCG and BCR are also tested without SM, VSF and VSIII). The change in association which ultimately caused the report to include this data was due to a change in the test suite, not the system, causing them to be tested together.

Both of these types of errors call for using the full test suite with our approach, rather than the "sanity" subset used in our empirical study. However, our experience with the target system allowed us to quickly identify these artifacts and ignore them.

6.3 Coupling detected in the target system

The report's findings on TEX, RAM, SAM, R2C and C2R were correct. Upon closer inspection of the system code, the TEX, RAM, SAM, R2C and C2R features form a group of associated features that share some of their implementations: the TEX feature requires special information to operate which the RAM, SAM, R2C and C2R features provide in the input to the target system. This association was always present (formerly 9~10%), but changes made between versions #4 and #5 increased the proportion of shared code (~20%).

A survey of changes made to the target system's architectural specification over the same period of time showed that the design of these features (TEX, RAM, SAM, R2C, C2R) was altered to meet constraints external to the target system, coming from other groups in the enterprise. These changes increased the co-dependence of these features by design, but our approach nonetheless recovered this information and hence validates the changed implementation of the features. This shows that a tool reporting on feature coupling can be used to validate the architecture via both positive and negative results.

6.4 Discussion

The technique we have presented here cannot discern between features coupled at the implementation level and features coupled by the tests that exercise them. This is to say that if two features are always exercised in the same tests, our feature location step will produce the same set of implementation components for both. If one feature must always be tested with another, but the opposite is not required, one of the sets will be a subset of the other and the association score will not be 100%.

While this can be the result of insufficiently-discriminated testing of the features of the system, it can also result from inter-dependencies at a higher-level than features: *use-cases*. While two features could be used separately, it may simply make no sense to do so: a resource acquisition (e.g. opening a file) should always be followed by a release of the resource (e.g. closing the file).

This type of tight inter-dependency between features that may not share much code would be very difficult to find without a dynamic test-driven analysis such as ours. Architects of the system would still be able to predict the coupling, but only when factoring-in extensive experience with the nature of the inputs of the system in the field. These were found to naturally emerge in the results of our study (surrounding VSF/VSIII), providing additional insight into the system and its use-cases.

Chapter 7
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 proportion of features that can be located with enough accuracy to be analyzed for coupling increases.

Our experiments helped confirm that source files identified with our approach may be in need of preventative maintenance. A small experiment showed that files identified by our approach were significantly more likely to be affected by change requests in the future. Based on our experience with the target system, we were able to find significant results in coupling reports.

We will soon be deploying a new implementation of this approach in a larger production environment. With this new system we hope to achieve a higher level of confidence in the reports by leveraging the full test suite and applying the analysis at a finer change granularity (daily). This new implementation added little extra cost to an ongoing effort to implement test-case selection techniques by sharing much of the same input data.

Intermediate results generated by our approach are also valuable on their own merits, and will deserve further attention. For example, the feature location data generated by our approach could be used to accelerate the progress of programmers new to the group. Navigation aids centered on the source code could allow them to quickly learn where features are implemented, and which tests exercise them. The feature information collected on the test suite can be also used help the system's architects reason about functional coverage of tests. As an added benefit, the regular application of our technique will automatically refresh this information, preventing it from ever becoming stale.

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.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187-197, 2002.

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

[3] 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*, pages 216-234, 1999.

[4] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. In *IEEE Transactions on Software Engineering*, 22(10), pages 751-761, 1996.

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

[6] L.C. Briand. Empirical investigations of quality factors in object-oriented software. In *Empirical Studies of Software Engineering: An International Journal*, 3(1), pages 65-117, 1999.

[7] L. F. Cranor. Internet Privacy. In *Communications of the ACM*, 42(2), pages 28-38, 1999.

[8] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. In *IEEE Transactions on Software Engineering*, 20(6), pages 476-493, 1994.

[9] S.R. Chidamber and C.F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications*. pages 197-211, 1991.

[10] P. Coad and E. Yourdon. Object Oriented Analysis. Yourdon Press, Upper Saddle River, NJ, USA, 1991. 233 pages.

[11] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. Technical Report 2/93, Department of Information Systems, University of Linz, Linz, Austria, 1993.

[12] 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. In *IEEE Transactions on Software Engineering*, 27(1), pages 1-12, 2001.

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

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

[15] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Inc., Reading, MA, 1995.

[16] O. Giroux and M. P. Robillard. Detecting Increases in Feature Coupling using Regression Tests. In *Proceedings of the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 163-174, 2006.

[17] 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*, pages 220-242, 1997.

[18] J. Law and G. Rothermel, Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308, 2003.

[19] 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*, pages 281-285, 2003.

[20] K. Mens and A. Kellens. Towards a framework for testing structural source-code regularities. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 679-682, 2005.

[21] A. Mitchell and J.F. Power. A study of the influence of coverage on the relationship between static and dynamic coupling metrics. In *Science of Computer Programming*, 59(1-2), pages 4-25, 2006.

[22] A. Mitchell and J.F. Power. Using Object-Level RunTime Metrics to Study Coupling Between Objects. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1456-1462, 2005.

[23] G.C. Murphy, D. Notkin, and K. J. Sullivan. “Software Reflexion Models: Bridging the Gap Between Design and Implementation.” In *IEEE Transactions on Software Engineering*, 27(4), pages 364-380, 2001.

[24] 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*, pages 107-119, 1999.

[25] D. L. Parnas, On the criteria to be used in decomposing systems into modules. In *Communications of the ACM*, 15(12) pages 1053-1058, 1972.

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

[27] M. Sefika, A. Sane, R.H. Campbell. Monitoring compliance of a software system with its high-level design models. *In Proceedings of the 18th International Conference on Software Engineering*, pages 387-396, 1996.

[28] G. Snelting. Concept analysis—a new framework for program understanding. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 1-10, 1998.

[29] W.P. Stevens, G.J. Meyers and L.L. Constantine. Structured Design. In *IBM Systems Journal*, 13(2), pages 115-139, 1974.

[30] 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*, pages 86-96, 2002.

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

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

[33] F.G. Wilkie and B.A. Kitchenham. Coupling measures and change ripples in C++ application software. In *The Journal of Systems and Software*, 52(2), pages 157-164, 2000.

Appendix A
Exercises and Covers Aggregation implementation

A.1 Dependencies not included here

This code depends on one support library developed within NVIDIA, and many libraries that are either standardized (ISO+IEC-14882/STL) or are available in the public domain (www.boost.org)(www.zlib.net).

Of those libraries that are freely available, the following are required:

- STL <iostream>
- STL <fstream>
- STL <algorithm>
- STL <numeric>
- STL <memory>
- STL <string>
- STL <map>
- STL <set>
- Boost File System
- Zlib

Finally, for the sake of understanding the implementation code, the NVIDIA library can be assumed to be a standard iostream-compatible implementation of binary streams exposing the following prototypes:

```
namespace stdext {

    class binary_ostream;
    class binary_ofstream; //: public binary_ostream
```

```
        class binary_istream;
        class binary_ifstream; //: public binary_istream

        template < class Type >
        inline binary_ostream & operator<<( binary_ostream & s , Type const & b );

        template < class Type >
        inline binary_istream & operator>>( binary_istream & s , Type & b );
    }
```

A.2 Aggregate.hpp

```
#ifndef NVTESTHUB_AGGREGATE_HPP_INCLUDED
#define NVTESTHUB_AGGREGATE_HPP_INCLUDED

namespace NvTestHub {

    typedef bool Value;

    typedef std::string             Name;
    typedef std::set< std::string > Names;

    struct NameHandle {
        Name const& operator*( ) const {
            return *m_name;
        }
        Name const* operator->( ) const {
            return m_name;
        }
        bool operator<( NameHandle const& other ) const {
            return m_name <  other.m_name;
        }
        NameHandle & operator=( NameHandle const& other ) {
            m_name = other.m_name;
            return *this;
        }
        NameHandle( ) : m_name( NULL ) {
        }
    protected :
        NameHandle( Name const* name ) : m_name( name ) {
        }
        friend struct Aggregate;
    private :
        Name const* m_name;
        friend inline stdext::binary_ostream & output( stdext::binary_ostream & s ,
NameHandle const& b ) {
            return output( s , b.m_name );
        }
        friend inline stdext::binary_istream & input( stdext::binary_istream & s , NameHandle
& b ) {
            return input( s , b.m_name );
        }
    };

    typedef std::pair< NameHandle , Value > Atom;
    typedef std::deque< Atom >              AtomGroup;

    typedef std::string             Key;
    typedef std::set< std::string > Keys;

    struct KeyHandle {
        Key const& operator*( ) const {
            return *m_key;
        }
        Key const* operator->( ) const {
            return m_key;
        }
        bool operator<( KeyHandle const& other ) const {
            return m_key <  other.m_key;
        }
        KeyHandle & operator=( KeyHandle const& other ) {
            m_key = other.m_key;
            return *this;
        }
        KeyHandle( ) : m_key( NULL ) {
        }
```

```
protected :
    KeyHandle( Key const* key ) : m_key( key ) {
    }
    friend struct Aggregate;
private :
    Key const* m_key;
    friend inline stdext::binary_ostream & output( stdext::binary_ostream & s ,
                                                   KeyHandle const& b ) {
        return output( s , b.m_key );
    }
    friend inline stdext::binary_istream & input( stdext::binary_istream & s ,
                                                  KeyHandle & b ) {
        return input( s , b.m_key );
    }
};

typedef std::set< KeyHandle > KeySet;
typedef std::set< KeySet >    KeySets;

struct KeySetHandle {
    KeySet const& operator*( ) const {
        return *m_keySet;
    }
    KeySet const* operator->( ) const {
        return m_keySet;
    }
    bool operator<( KeySetHandle const& other ) const {
        return m_keySet <  other.m_keySet;
    }
    KeySetHandle & operator=( KeySetHandle const& other ) {
        m_keySet = other.m_keySet;
        return *this;
    }
    KeySetHandle( ) : m_keySet( NULL ) {
    }
protected :
    KeySetHandle( KeySet const* keySet ) : m_keySet( keySet ) {
    }
    friend struct Aggregate;
```

```
private :
    KeySet const* m_keySet;
    friend inline stdext::binary_ostream & output( stdext::binary_ostream & s ,
                                                   KeySetHandle const& b ){
        return output( s , b.m_keySet );
    }
    friend inline stdext::binary_istream & input( stdext::binary_istream & s ,
                                                  KeySetHandle & b ) {
        return input( s , b.m_keySet );
    }
};

typedef std::map< NameHandle , KeySetHandle > Coverage;

struct Aggregate {

    typedef ::boost::shared_ptr< Aggregate > Pointer;

    typedef std::map< ::boost::filesystem::path , Pointer > Map;

    static Map const& aggregates( );
    static Aggregate::Pointer const& load( ::boost::filesystem::path const& );

    KeyHandle exchangeKey( Key const& );

    NameHandle exchangeName( Name const& );

    bool getValue( KeyHandle , NameHandle ) const;
    void setValue( KeyHandle , NameHandle , bool );

    void merge( KeyHandle key , AtomGroup const& group );
    AtomGroup extractKey( KeyHandle , bool setNames = true ) const;
    void erase( KeyHandle );

    KeySet const& extractName( NameHandle ) const;

    Keys const& keys( ) const {
        return m_keys;
    }
```

```
        Names const& names( ) const {
            return m_names;
        }
        KeySets const& keySets( ) const {
            return m_keySets;
        }

        void compressKeySets( );

        void commit( ) const;

    private :
        static Map & aggregates_internal( );

        void commitKeySets( );

        Aggregate( ::boost::filesystem::path const& );

        KeySetHandle exchangeKeySet( KeySet const& );

        ::boost::filesystem::path       m_path;

        bool                            m_changed;

        Keys                            m_keys;
        Names                           m_names;
        KeySets                         m_keySets;

        std::map< NameHandle , KeySet > m_temporaryKeySets;
        Coverage                        m_coverage;
    };

}

#endif //NVTESTHUB_AGGREGATE_HPP_INCLUDED
```

A.2 Aggregate.cpp

```
#include "stdafx.hpp"
#include "Aggregate.hpp"

namespace NvTestHub {

    Aggregate::Map const& Aggregate::aggregates( ) {
        return aggregates_internal( );
    }

    Aggregate::Map & Aggregate::aggregates_internal( ) {
        static Map aggregates;
        return aggregates;
    }

    Aggregate::Pointer const& Aggregate::load( ::boost::filesystem::path const& path ) {

        Pointer & aggregate = aggregates_internal( )[ path ];
        if( aggregate == NULL )
            aggregate = Pointer( new Aggregate( path ) );

        return aggregate;
    }

    typedef std::map< KeyHandle , Key > RemapKeys;
    typedef std::map< NameHandle , Name > RemapNames;
    typedef std::map< KeySetHandle , KeySet > RemapKeySets;

    Aggregate::Aggregate( ::boost::filesystem::path const& path ) : m_path( path ) ,
                                                                    m_changed( false ) {

        if( !::boost::filesystem::exists( m_path ) ) {
            std::cerr << "File not found [" << m_path.string( )
                      << "], will create a new database.\n";
            return;
    }

    RemapKeys remapKeys;
    RemapNames remapNames;
    RemapKeySets remapKeySets;

    stdext::binary_ifstream f( m_path.string( ).c_str( ) , 9 );
    f >> remapKeys;
    f >> remapNames;
    f >> remapKeySets;
    f >> m_coverage;

    std::map< KeySetHandle , KeySet > remappedRemapKeySets;
    for( RemapKeySets::const_iterator it = remapKeySets.begin( );
                                      it != remapKeySets.end( ); ++it ) {
        KeySet keySet;
        for( KeySet::const_iterator kit = it->second.begin( );
                                    kit != it->second.end( ); ++kit )
            keySet.insert( exchangeKey( remapKeys[ *kit ] ) );
        remappedRemapKeySets[ it->first ] = keySet;
    }

    Coverage remappedCoverage;
    for( Coverage::const_iterator it = m_coverage.begin( );
                                  it != m_coverage.end( ); ++it )
        remappedCoverage.insert( std::make_pair(
                                 exchangeName( remapNames[ it->first ] ) ,
                                 exchangeKeySet( remappedRemapKeySets[ it->second ] )));

    swap( remappedCoverage , m_coverage );
}

void Aggregate::commit( ) const {

    if( !m_changed )
        return;

    RemapKeys remapKeys;
    RemapNames remapNames;
    RemapKeySets remapKeySets;
    {
        size_t i = 0;
        for( Keys::const_iterator it = m_keys.begin( ); it != m_keys.end( ); ++it , ++i )
            remapKeys[ &( *it ) ] = *it;
    }
    {
        size_t i = 0;
        for( Names::const_iterator it = m_names.begin( );
                                   it != m_names.end( ); ++it , ++i )
            remapNames[ &( *it ) ] = *it;
    }
    {
        size_t i = 0;
        for( KeySets::const_iterator it = m_keySets.begin( );
                                     it != m_keySets.end( ); ++it , ++i )
            remapKeySets[ &( *it ) ] = *it;
    }

    stdext::binary_ofstream f( m_path.string( ).c_str( ) , 9 );
    f << remapKeys;
    f << remapNames;
    f << remapKeySets;
    f << m_coverage;
}

KeyHandle Aggregate::exchangeKey( Key const& key ) {
    Keys::iterator it = m_keys.find( key );
    if( it == m_keys.end( ) )
        it = m_keys.insert( it , key );
    return &( *it );
}

NameHandle Aggregate::exchangeName( Name const& name ) {
    Names::iterator it = m_names.find( name );
    if( it == m_names.end( ) )
        it = m_names.insert( it , name );
    return &( *it );
}
```

```
KeySetHandle Aggregate::exchangeKeySet( KeySet const& keySet ) {
    KeySets::iterator it = m_keySets.find( keySet );
    if( it == m_keySets.end( ) )
        it = m_keySets.insert( it , keySet );
    return &( *it );
}

bool Aggregate::getValue( KeyHandle key , NameHandle name ) const {

    //Get the name in the coverage database and its associated keyset
    Coverage::const_iterator it = m_coverage.find( name );
    if( it == m_coverage.end( ) )
        return false;

    //If name exists then coverage value is simply the existence of the key in its set
    return it->second->find( key ) != it->second->end( );
}

void Aggregate::setValue( KeyHandle key , NameHandle name , bool value ) {

    //Get the working set for this name
    std::map< NameHandle,KeySet >::iterator temporary = m_temporaryKeySets.find( name );
    if( temporary == m_temporaryKeySets.end( ) ) {

        //Get the name in the coverage database and its associated keyset
        Coverage::iterator it = m_coverage.find( name );
        if( it == m_coverage.end( ) )
            it = m_coverage.insert( it , std::make_pair( name ,
                                                         exchangeKeySet( KeySet( ) ) ) );

        //Create a new working set, copied from the original set
        temporary = m_temporaryKeySets.insert( temporary , std::make_pair( it->first ,
                                                                            *it->second ) );

        //Assign the working set as this name's set
        it->second = &( temporary->second );
    }
    //Modify the working set
    if( value )
        temporary->second.insert( key );
    else
        temporary->second.erase( key );

    m_changed = true;
}

void Aggregate::merge( KeyHandle key , AtomGroup const& group ) {

    //Simply merge every entry into the database
    for( size_t i = 0; i < group.size( ); ++i )
        setValue( key , group[ i ].first , group[ i ].second );
}

AtomGroup Aggregate::extractKey( KeyHandle key , bool setNames ) const {

    AtomGroup g;
    for( Names::const_iterator it = m_names.begin( ); it != m_names.end( ); ++it ) {
        Coverage::const_iterator cit = m_coverage.find( &( *it ) );
        if( cit == m_coverage.end( ) )
            g.push_back( std::make_pair( setNames ? NameHandle( &( *it ) ) :
                                                    NameHandle( ) , false ) );
        else
            g.push_back( std::make_pair( setNames ? NameHandle( &( *it ) ) :
                                                    NameHandle( ) ,
                                  cit->second->find( key ) != cit->second->end( ) ) );
    }
    return g;
}

void Aggregate::erase( KeyHandle key ) {

    //First commit pending keyset changes
    commitKeySets( );

    //Erase the key from all keysets that reference it
    while( 1 ) {
            bool erased = false;
           for( KeySets::iterator it = m_keySets.begin( ); it != m_keySets.end( ); ++it ) {

               if( it->find( key ) == it->end( ) )
                 continue;

               KeySet k = *it;
               k.erase( key );
               m_keySets.erase( it );
               m_keySets.insert( k );

               erased = true;
               break;
           }
             if( !erased )
                 break;
         }

     //Erase the key from the key index
     m_keys.erase( *key );

     m_changed = true;
 }

 KeySet const& Aggregate::extractName( NameHandle name ) const {
     static KeySet ks;
     Coverage::const_iterator cit = m_coverage.find( name );
     if( cit == m_coverage.end( ) )
         return ks;
     return *cit->second;
 }

 void Aggregate::commitKeySets( ) {

     //Roll the temporary key sets into the main pool
     for( std::map< NameHandle,KeySet >::const_iterator it = m_temporaryKeySets.begin( );
                                                   it != m_temporaryKeySets.end( ); ++it )
         m_coverage[ it->first ] = exchangeKeySet( it->second );
```

```
        m_temporaryKeySets.clear( );

         m_changed = true;
    }

    void Aggregate::compressKeySets( ) {

        //First commit pending keyset changes
        commitKeySets( );

        KeySets keySets = m_keySets;
        for( Names::const_iterator it = m_names.begin( ); it != m_names.end( ); ++it ) {
            Coverage::const_iterator cit = m_coverage.find( &( *it ) );
            if( cit == m_coverage.end( ) )
                continue;
            keySets.erase( *cit->second );
        }
        for( KeySets::const_iterator it = keySets.begin( ); it != keySets.end( ); ++it )
            m_keySets.erase( *it );

        m_changed = true;
    }

}
```