Using Structural and Textual Information to Capture Feature Coupling in Object-Oriented Software

Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk

The College of William and Mary
757-221-3476
757-221-1717
meghan@cs.wm.edu, mgethers@cs.wm.edu, denys@cs.wm.edu
http://www.cs.wm.edu/semeru/

Abstract Previous studies have demonstrated the relationship between coupling and external software quality attributes, such as fault-proneness, and the application of coupling to software maintenance tasks, such as impact analysis. These previous studies concentrate on class coupling. However, there is a growing focus on the study of features in software, and features are often implemented across multiple classes, meaning class-level coupling measures are not applicable. We ask the pertinent question, "Is measuring coupling at the feature-level also useful?" We define new feature coupling metrics based on structural and textual source code information and extend the unified framework for coupling measurement to include these new metrics. We also conduct three extensive case studies to evaluate these new metrics and answer this research question. The first study examines the relationship between feature coupling and fault-proneness, the second assesses feature coupling in the context of impact analysis, and the third study surveys developers to determine if the metrics align with what they consider to be coupled features. All three studies provide evidence that feature coupling metrics are indeed useful new measures that capture coupling at a higher level of abstraction than classes and can be useful for finding bugs, guiding testing effort, and assessing impact of changes.

Keywords *Feature coupling · Information Retrieval · Latent Semantic Indexing · program comprehension · open source software*

1 Introduction

Coupling is an important software relationship that has been used for numerous tasks related to software development and maintenance such as predicting software quality (Basili et al. 1996; Briand et al. 2000; Briand et al. 2002; Subramanyam and Krishnan 2003; Gyimóthy et al. 2005; Olague et al. 2007) and impact analysis (Briand et al. 1999; Wilkie and Kitchenham 2000; Poshyvanyk et al. 2009). Coupling is primarily measured at the class-level by determining the degree to which two classes in an object-oriented system depend on one another.

Features are functionalities described in functional requirement specifications[1] that have been actualized in a software system (Poshyvanyk et al. 2007). For example, consider an Internet

[1] We acknowledge that some features are not documented in requirements, but still can be analyzed in source code.

browser software system, such as *Mozilla Firefox*. Examples of the features for such a system are "bookmarking web pages" and "opening a new tab window". Often, features have implementations that span multiple methods or classes and cannot be modularized due to design decisions (Kiczales et al. 1997; Tarr et al. 1999). Features are important software entities that transcend the boundaries of classes. Note that the notion of concepts and concerns are more general than features capturing non-functional requirements in addition to functional requirements. Our work is also applicable when considering the more general abstractions. Currently, there are no metrics that explicitly capture the coupling between features, and the usefulness of such measures is not known.

In this paper, we argue that feature-level coupling metrics are needed and show that they are useful. Feature coupling can be used as a predictor of fault-proneness. Just as class coupling has been used in testing (Jin and Offutt 1996), if it is known that two features are tightly coupled, more testing effort can be applied to them to help eliminate bugs. Another example is software maintenance. Many software change tasks are framed in terms of a system's functionalities or features. Since a feature's implementation may be scattered throughout the source code of a software system, programmers may have difficulty determining which other features interact with it. Therefore, changes made to one feature may have unintended consequences for other, seemingly unrelated features, causing improper system behavior. To avoid such situations, feature-level impact analysis should be performed to discover other features that are tightly coupled to the feature undergoing modification. Thus feature coupling metrics are needed to measure the dependencies among features to support a variety of software development and maintenance tasks. Additionally, we speculate that our proposed feature-level coupling metrics may also be applicable to the problem of detecting feature-interactions (Griffeth and Lin 1993; Zave 1993; Aho and Griffeth 1995).

We introduce new feature coupling metrics because current coupling metrics are designed for classes, and features exist at a higher level of abstraction than classes. Features are defined by a portion of a specification and implemented in source code, meaning features are represented by both structured (*e.g.*, source code dependencies) and unstructured (*e.g.*, identifiers and comments in source code) information. Therefore, it is logical to measure feature coupling using both types of data: structured and unstructured. Structured information refers to source code and other related derivative artifacts such as program dependence graphs (PDGs) that are ordered in a particular way (*i.e.*, following programming language grammar rules). Unstructured information, on the other hand, refers to internal source code comments, identifier names, and external documentation that encode domain knowledge and design decisions. While comments and documentation can be structured in the form of sentences and organized into sections, they are more free form, unstructured, and do not follow specific rules.

We define feature coupling metrics based on these different sources of information. Structural Feature Coupling (*SFC*) captures the relationship between two features based on structured information, while Textual Feature Coupling (*TFC*) measures the coupling between features based on unstructured, textual information in source code using an information retrieval technique called **L**atent **S**emantic **I**ndexing (LSI) (Deerwester et al. 1990) (see Section 3 for more details). In addition, we conjecture that the structured and unstructured data are complimentary, as has been shown elsewhere (Hill et al. 2007; Poshyvanyk et al. 2007; Eaddy et al. 2008; Poshyvanyk et al. 2009), so we propose to combine *SFC* and *TFC* into a hybrid feature coupling metric called *HFC*. Hybrid feature coupling can be used when one source of information cannot be completely relied on, but programmers still want to incorporate it. For instance, in systems that are poorly structured, more weight can be given to textual information to compensate. Likewise, in software with little or no comments or poorly named identifiers, more weight can be placed on structural information.

This paper makes the following research contributions:

1. **Define feature coupling metrics**. We formally define coupling metrics for features using structural and textual information. Our metrics are novel and fill a void in the research area that currently lacks feature coupling metrics based on either type of information. We also theoretically validate our metrics and introduce a new dimension to the unified framework for coupling measurement (Briand et al. 1999).
2. **Demonstrate the relationship between feature coupling and fault-proneness**. To demonstrate both the usefulness and applicability of our new feature coupling metrics, we perform three separate case studies. In the first case study, we empirically investigate the relationship between our feature coupling metrics and fault-proneness. In this study, we

establish that there is a statistically significant correlation between feature coupling and defects. Our results build on previously published findings (Eaddy et al. 2008) that cross-cutting concerns may cause defects. In essence, the first case study extends prior results by showing that there is also a relationship between coupled features (concerns) and bugs.

3. **Evaluate the application of feature coupling to impact analysis**. We also demonstrate some implications of feature coupling measurement for feature-level impact analysis. Feature coupling is a good starting point for understanding how a change to one feature is likely to affect others. For example, during impact analysis, all features can be ranked by their strength of coupling to the feature being modified. If programmers know that feature *A* is more tightly coupled to feature *B* than to feature *C*, they can expect that a change to *A* is likely to impact *B* more than *C* and spend more time ensuring *B* was not adversely affected by the change to *A*. Also, analyzing related features using coupling metrics can help avoid introducing defects caused by intricate and potentially hidden dependencies (Yu and Rajlich 2001) among features. We show that feature coupling can be effectively used for impact analysis under certain configurations.
4. **Explore how feature coupling metrics align with developers' opinions**. The final way in which we evaluate our new feature coupling metrics is by investigating if they agree with developers' opinions of whether two features are coupled or not. We find that overall, there is agreement between the developers' ratings and our measures, meaning our feature coupling metrics do capture coupling among features as recognized by software developers.
5. **Create tool support for feature coupling**. We have developed an Eclipse plug-in for managing features. The tool has functionality to assign portions of code to features, perform structural and textual analyses, and the ability to compute and analyze feature coupling metrics on demand.

The three case studies provide evidence that feature coupling metrics are useful tools programmers can use while performing feature-level software maintenance tasks. Like class coupling measures, they can be used to predict fault-proneness and for impact analysis. These new metrics give programmers greater flexibility because they allow for analysis at a higher level of abstraction than classes.

The remaining sections of the paper are organized as follows. Section 2 presents related work, covering structural coupling measures, other static coupling measures, dynamic coupling measures, as well as applications of such measures. Subsequently, Section 3 discusses using structural and unstructured information for feature coupling. Our case studies conducted to evaluate our proposed feature coupling metrics are presented in Section 4 which is followed by the conclusions in Section 5.

2 Related Work

There are many existing coupling metrics that employ different types of information such as structural, dynamic, textual, or evolutionary. Most of these metrics determine coupling between classes. Our work is distinct from previous research in that it provides a formal way to capture and analyze the strength of coupling among features using various types of information, namely structural and textual. Furthermore, there are no existing metrics that combine information from two or more distinct sources (*e.g.*, structural and textual) to capture coupling. Table 1 summarizes the state-of-the-art in coupling measurement, and we offer a brief overview below.

2.1 Structural Coupling Measures

Most existing coupling metrics capture coupling between classes structurally. Coupling Between Objects (*CBO*) and Response for a Class (*RFC*) were introduced in Chidamber and Kemerer's suite of object-oriented metrics (Chidamber and Kemerer 1994). According to *CBO*, two classes are coupled if methods in one class use methods or fields in the other. *RFC* and RFC_α are counts of a class' methods plus methods that are directly or indirectly (Churcher and Shepperd 1995) invoked by those methods. Li and Henry (Li and Henry 1993) introduced several class coupling metrics that also utilize structural information. Message Passing Coupling (*MPC*) between classes *A* and *B* is based on the number static invocations of methods from class *B* in class *A*. Data Abstraction Coupling (*DAC*) is a count of the number of fields in class *A* that are of type *B*, while *DAC'* is a binary version of this metric. There are a wealth of other structural metrics based on class dependencies such as Efferent Coupling (C_e) and Afferent Coupling (C_a) (Martin 1994).

Briand et al. (Briand et al. 1997) developed several metrics for measuring the coupling between classes based on structural information from method invocations and the types of fields and parameters. These metrics, plus those by (Hitz and Montazeri 1995) and (Eder et al. 1994), were reviewed in (Briand et al. 1999) to build a unified framework for coupling measurement in object-oriented systems.

Information flow-based coupling (*ICP*) (Lee et al. 1995) is a structural measure that takes polymorphism into account. *ICP* counts the number of methods from a class *B* invoked in a class *A*, weighted by the number of parameters. Two alternative versions, *IH-ICP* and *NIH-ICP*, count invocations of inherited methods and classes not related through inheritance, respectively. Like *ICP*, some of the coupling measures defined in (Briand et al. 1999) take polymorphism into account. All of these existing coupling metrics are defined for classes, and therefore are at a lower level of abstraction than our feature coupling metrics.

2.2 Other Static Coupling Measures

Other static coupling measures exist along textual and evolutionary dimensions. Poshyvanyk and Marcus (Poshyvanyk and Marcus 2006) define a coupling metric for classes based on textual information extracted from source code identifiers and comments. Their conceptual coupling metric, *CoCC* (which stands for conceptual coupling of classes), captures a new dimension of coupling not addressed by structural or dynamic measures. More recently, Újházi et al. (Újházi et al. 2010) extended *CoCC*, defining a new conceptual metric namely Conceptual Coupling between Object Classes (*CCBO*). Another conceptual class coupling metric lately defined by Gethers and Poshyvanyk (Gethers and Poshyvanyk 2010) is coined as Relational Topic based Coupling (*RTC*). RTC utilizes topic-based analysis of source code to capture coupling among classes. *CoCC*, *CCBO,* and *RTC* are all defined for classes only, while the metrics we propose in this paper are for features. Interaction (Zou et al. 2007), logical (Gall 2003), and evolutionary (Zimmermann et al. 2005) coupling metrics utilize information from repositories to mine information from artifacts (including source code) that are frequently co-changed. Such evolutionary information has been used for impact analysis (Sherriff and Williams 2008), much like coupling metrics. Additionally, coupling metrics have been defined for other applications such as knowledge-based (Kramer and Kaindl 2004) and aspect-oriented (Zhao 2004) systems.

Table 1. State-of-the art in coupling measurement across two dimensions: level of coupling and type of information used to capture the strength of coupling. The metrics proposed in this paper are highlighted in bold.

Coupling dimension	Structural	Dynamic	Textual	Hybrid	Evolutionary
Class	*CBO*, *RFC*, MPC, *DAC*, C_e and C_a, Info coupling, class-attribute interaction, class-method interaction, method-method interaction	Dynamic import and export coupling	*CoCC, RTC*	Future work	Interaction coupling, evolutionary and logical coupling
Feature	***SFC***, ***SFC'***	*DIST*	***TFC***, $\boldsymbol{TFC_{max}}$	***HFC***	Future work

2.3 Dynamic Coupling Measures

Arisholm et al. (Arisholm et al. 2004) introduced dynamic import and export metrics to capture the coupling between classes at runtime. Dynamic analysis is often used to locate the code associated with features (Wilde and Scully 1995; Eisenbarth et al. 2003; Salah et al. 2005; Liu et al. 2007; Poshyvanyk et al. 2007; Eaddy et al. 2008) since a feature's behavior can be observed during execution. Currently the only existing feature-level coupling-like metric that we are aware of is based on dynamic information. Wong and Gokhale (Wong and Gokhale 2005) defined the distance (*DIST*) between two features using an execution slice-based technique. Similar feature metrics have been proposed to dynamically measure certain relationships or dependencies between features (Greevy and Ducasse 2005; Lienhard et al. 2007) other than coupling. Greevy et al. (Greevy et al. 2006) also created metrics for dynamically measuring the evolution of a feature. Similarly, Giroux and Robillard (Giroux and Robillard 2006) defined a measure for feature coupling across versions of a system using regression tests since tests typically align with features. The association graph matching similarity measure (*AGM*) introduced by Kothari et al. (Kothari et al. 2006) is a measure of pair-wise similarity between features based on dynamic call graphs. It has been used to find canonical feature sets (Kothari et al. 2006), feature version similarity (Kothari et al. 2008), and feature implementation overlap (Kothari et al. 2007). All of these feature metrics solely utilize dynamic information. However, dynamic information may not be sufficient to precisely capture coupling among features. The best way to collect dynamic information is to execute scenarios that exercise only one feature at a time, but developing such scenarios can be difficult, if possible at all (Wong and Gokhale 2005). Our metrics are the first to capture feature coupling using structural and textual information, thus avoiding the overhead of collecting execution traces.

2.4 Applications of Coupling Metrics

There have been numerous studies showing that coupling is a good predictor of external quality attributes such as fault-proneness (Zimmerman and Nagappan 2008; Cataldo et al. 2009; Briand et al. 1997; Basili et al. 1996), maintainability (Li and Henry 1993), reengineering effort (Meyers and Binkley 2007), and change-proneness (Briand et al. 1999). Other studies have shown that coupling can be used for different tasks (Darcy and Kemerer 2005) such as impact analysis (Wilkie and Kitchenham 2000; Poshyvanyk et al. 2009; Kagdi et al. 2010), program comprehension (Zaidman et al. 2006), reengineering (Abreu et al. 2000), quality assessment (Bansiya and Davis 2002), reuse (Chidamber et al. 1998), change propagation (Geipel and Schweitzer 2009), and clone detection (Geiger et al. 2006). These studies focus on coupling at the class level, while our work examines feature coupling and investigates if it is also useful for predicting fault-proneness and performing impact analysis.

3 Using Structural and Textual Information for Feature Coupling

Our approach to measuring feature coupling is based on two main ideas: 1) features are entities that are coupled at a higher level of abstraction than methods and classes and 2) coupling can be measured in multiple ways by using structured and unstructured (textual) information. Features are domain concepts implemented in a system (Poshyvanyk et al. 2007), and their implementations are often scattered across a system's classes (Eaddy et al. 2008). Therefore, features exist at a level of abstraction outside of or above classes in object-oriented languages. Features, in the context of this research, closely relate the notion of features in the product line research community (Griss 2000; Kang et al. 2002). As described in Section 2, there exists an abundance of class coupling metrics that rely on structural dependencies and some that utilize textual information to measure class coupling. These metrics are useful and important because they capture essential forms of coupling. However, since features transcend class boundaries, we propose and define metrics that comprehensively capture and measure feature coupling using both structural and textual information.

Traditional coupling metrics focus on capturing the relationships among classes within a

software system. Therefore, those metrics are unable to properly quantify coupling among features. This, in part, is because of cross-cutting concerns. More specifically, within source code there potentially exist scattered implementations of features, crossing module or class boundaries. Since such features are not completely encapsulated within modules or classes the usefulness of existing coupling metrics is limited. With the introduction of our metrics we are able to bridge the gap between features and coupling measurement.

The structural feature coupling metric that we propose measures the coupling between two features structurally, drawing on information used by existing class-level coupling metrics. The textual feature coupling metric we introduce measures the conceptual or textual similarity between two features. Our approach is based on the premise that the unstructured information embedded in source code reflects, to a reasonable degree, the software's domain concepts since existing feature location techniques (Marcus et al. 2004; Liu et al. 2007; Poshyvanyk et al. 2007; Revelle et al. 2010) leverage such textual information to find code that implements features.

3.1 System Representation

To define structural and textual feature coupling metrics, we first define a representation of a software system.

Definition 1: (System, Classes, Methods)
A system S is an object-oriented software system. S has a set of classes $C = \{c_1, c_2, ..., c_n\}$. The number of classes in S is $n = |C|$. A class has a set of methods. For each class $c \in C$, let $M_c = \{m_1, m_2, ..., m_z\}$ be the set of methods implemented in c, where $z = |M_c|$ is the number of methods in c. The set of all methods in the system S is defined as M_S.

Definition 2: (Feature)
A feature f is a requirement, functionality, or behavior described in the specification of a system S. A system S has a set of features $F = \{f_1, f_2, ...f_p\}$ where $p = |F|$. A feature f is implemented by a set of methods $M_f \subseteq M_S$. The set of methods, M_f, represent the feature f, within the source code. The methods of M_f may belong to multiple classes. A method may belong to several features, and a feature may have methods that belong to other features as well. Note that the connection between requirements and methods implementing features is that requirements describe desired functionality, which will be realized upon development of a software system. We indirectly capture coupling between the requirements by analyzing the methods where their functionality is implemented.

We base this definition from Eaddy et al.'s (Eaddy et al. 2008) well-established model for representing cross-cutting concerns. The key differences include the level of granularity in which features are mapped to source code and the exclusion of the notion of hierarchy for both features and program elements. The prior model considered fields, methods and classes when mapping to source code whereas our model only maps features to methods. Additionally, we do not consider the hierarchical structure of either features or source code elements. Identification of methods associated with features is accomplished using the prune dependency rule (Eaddy et al. 2008). That is, we simulate the removal of each feature from the source code. All removed or altered methods are identified as contributors to the implementation of the given feature.

3.2 Structural Feature Coupling Using Structural Information

In this section we discuss how structural information is leveraged to capture coupling between features for a given software system.

3.2.1 Analyzing Structural Information in Source Code

Most software engineers are familiar with structural source code information that can be represented in various forms such as a **p**rogram **d**ependence **g**raph (PDG[2]). We use a PDG to add

[2] A PDG is a directed graph that represents the dependencies among objects in a software system.

additional information to our structural feature coupling metric. We obtain a method-level PDG using JRipples (Buckner et al. 2005; Petrenko and Rajlich 2009). It should be noted that class relationships, method invocations, and field references have all been used to compute class coupling (Briand et al. 1999). In our work, we focus on methods (method sets M_{f1} andM_{f2} for the feature pair *f1, f2*) as the main unit of structural information for several reasons. Working at method-level granularity is common with feature location. Most feature location techniques attempt to find methods associated with features (Wilde and Scully 1995; Eisenbarth et al. 2003; Liu et al. 2007; Poshyvanyk et al. 2007; Eaddy et al. 2008) because methods implement functionality in code. Also, several existing class coupling metrics, such as *CBO* and *RFC,* use methods only (Chidamber and Kemerer 1994; Arisholm et al. 2004; Poshyvanyk and Marcus 2006), ignoring fields. We provide more details on how we use this information to capture structural feature coupling in following section.

3.2.2 Structured Feature Coupling Metric

We define structural feature coupling metrics using our representation of a system, features, and methods.

Definition 3: (Structural Feature Coupling – SFC)

The structural feature coupling (*SFC*) between features f_a and f_b, implemented by the methods in sets M_a and M_b respectively, is defined as the ratio of the number of methods shared by the features to the total number of methods associated with the two features.

$$SFC(f_a, f_b) = \frac{|M_a \cap M_b|}{|M_a \cup M_b|} \quad (1)$$

We only consider features with non-empty methods sets to avoid a potential division by zero. *SFC* uses structured information to capture feature coupling by measuring the degree to which two features share code.

Instead of solely basing coupling on the methods that implement two features, an alternative is to consider the first order structural dependencies of those methods to also be associated with the features. Dependencies are taken into account in some existing coupling metrics (*e.g.* RFC_a), plus they are often traversed for maintenance, feature location, and program comprehension tasks. Therefore, we include the static callers and callees of a feature's methods in a variant *SFC*, which we coin *SFC'*.

Definition 4: (Structural Features Coupling Prime – SFC')

Let f_a and f_b be features implemented by the methods in sets M_a and M_b respectively. Let $M_a' \supseteq M_a$ and $M_b' \supseteq M_b$ be the set of methods that implement features f_a and f_b, respectively, plus the methods that are first order structural dependencies of the methods in M_a and M_b. So, let D_a be the set of methods which directly call or are called by a method in M_a. From this we have the following formal definition $M_a' = M_a \cup D_a$. That is, in addition to including the methods which implement the features, M_a' and M_b' include the methods that directly call or are directly called by the methods in M_a and M_b respectively. The structural feature coupling prime (*SFC*') is defined as the number of methods shared by two features over the total number of methods associated with both features.

$$SFC'(f_a, f_b) = \frac{|M_a' \cap M_b'|}{|M_a' \cup M_b'|} \quad (2)$$

Thus, *SFC*' incorporates additional structured information in the form of dependencies to measure feature coupling. Both *SFC* and *SFC*' are normalized, *i.e.*, they have values in the range [0, 1]. The closer the value is to one, the stronger the structural coupling between the features.

3.3 Textual Feature Coupling Using Unstructured Information

Although structural information captures key characteristics of a software system, the importance of textual information must not be overlooked. In this section we discuss the process of extracting and utilizing textual information to measure the degree of coupling between features in a software system. This is accomplished using Latent Semantic Indexing which identifies relationships between terms and concepts in unstructured text. LSI (Deerwester et al. 1990) has been successfully applied to a number of software engineering tasks, such as feature location (Poshyvanyk et al. 2007; Poshyvanyk and Marcus 2007; Cleary et al. 2009; Revelle and

Table 2. Mapping LSI concepts to source code.

LSI Model	Source Code Entities
word	Identifiers and comments extracted from source code comprise a vocabulary set. This set is refined to exclude programming language keywords, stop words, and punctuation. Finally, all compound identifiers are split based on the observed naming conventions. $V=\{w_1, w_2,..., w_v\}$.
document	A method is treated as a document, which can be expressed as n identifiers and comments from a vocabulary and appear in the implementation of a method $m_i=(w_1,w_2,...,w_n)$.
corpus	The software system S consists of a set of classes comprised of methods, $S = (C_1, ...,C_2)$ where the methods of the classes forms a corpus $D = (d_1, d_2, \ldots ,d_m)$.

Poshyvanyk 2009), traceability link recovery (Jiang et al. 2008; De Lucia et al. 2007; Hayes et al. 2006; Marcus et al. 2005; Lormans et al. 2008), software measurement (Marcus et al. 2008; Poshyvanyk et al. 2009), and detecting code clones (Marcus and Maletic 2001; Tairas and Gray 2009).

3.3.1 Analyzing Unstructured Information in Source Code with LSI

In LSI, a word is a basic unit of discrete data defined to be an item from a vocabulary $V=\{w_1, w_2,..., w_v\}$. A document is a sequence of n words denoted by $d = (w_1, w_2,..., w_n)$, where w_n is the n^{th} word in the sequence. A corpus is a collection of m documents, $D = (d_1, d_2,... ,d_m)$. Table 2Table 2 shows how these LSI concepts are mapped to source code. In general, a corpus consists of a set of documents and in our case the set of documents corresponds to methods in the source code. The documents consist of words extracted from the comments and identifiers of the corresponding methods in the source code.

The process of applying LSI to source code has three steps. First, the source code must be preprocessed to build a corpus. Second, the corpus is indexed. Third and finally, textual similarities between all pairs of documents (methods) are computed. If two methods use similar terminology and have a high textual similarity, they may implement related concepts and therefore be coupled. Each of these steps is explained in more detail in the following subsections.

3.3.2 Building the Corpus

A corpus represents all the words found in each document of a body of text. A document can be a sentence, a paragraph, a chapter, or in the case of source code, a method, a class, or a package. To build a corpus for the source code of a software system, a document granularity must first be chosen. In our work, we use methods as documents. Next, the text of each document must be preprocessed before being included in the corpus. There are several options for preprocessing, such as removing stop words and programming language keywords, splitting compound identifiers, including or excluding comments, and performing or not performing stemming. Stemming (Porter 1980) reduces words to their root form, such that stemming and stemmed would become stem. For every corpus created in this work, stop words (*e.g.*, *the, of*) and programming language keywords (*e.g.*, *public, for, try*) were removed and compound identifiers were split.

3.3.3 Indexing the Corpus

The central concept of LSI is that the information about the contexts in which a word appears or does not appear provides a set of mutual constraints that determines the similarity of meaning of sets of words (documents) to each other. LSI indexes a corpus and generates a real-valued vector description for each document based on the vector space model (VSM) (Salton and McGill 1983). LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with VSM approaches. Some words appear in the same contexts, and an important part of word usage patterns is blurred by accidental and inessential information. The method used by LSI to capture essential semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix (words by documents) decomposed using singular value decomposition (SVD) (Salton and McGill 1983). The word × document matrix holds term frequency-inverse document frequency (tf-idf) values

which assess how important a particular word is to a given document. SVD is a form of factor analysis and acts as a method for reducing the dimensionality of a feature space without serious loss of specificity. Typically, the word by document matrix is very large and quite sparse. SVD is applied to the word-by-document matrix to get rid of noise.

LSI requires, as input, a term-by-document co-occurrence matrix. Dimensionality reduction is performed using Singular Value Decomposition (SVD). The singular value decomposition of a co-occurrence matrix *X* produces three matrices. That is, we can represent our co-occurrence matrix as $X=T_0S_0D'$. The matrix T_0 essentially captures terms within domains, D' captures documents within domains and the matrix S is the singular values matrix. When S_0 is ordered by size we can reduce the dimensionality of the LSI subspace by considering only the top k largest dimensions, setting the remaining elements of the diagonal to zero. Appropriate selection of k allows for the subspace to discard unessential details and sampling error while capturing only the most significant aspects of the set of textual documents. The result of SVD is a subspace referred to as the LSI subspace. Within the LSI subspace each document is represented as a vector which is derived from the original co-occurrence matrix.

Using singular value decomposition LSI is able to capture relationships between terms in the co-occurrence matrix. The issues, associated with polysemy and synonymy, are alleviated through the use of the context in which terms are used and relationships between terms. Overall, latent semantic indexing provides a good solution to issues previously encountered in the information retrieval community and currently is used throughout the software engineering community.

3.3.4 Computing Textual Similarities

Once the corpus is indexed, the similarities between documents can be computed by taking the cosine between their corresponding vectors. The textual similarity between two documents (methods) m_i and m_j is defined as the cosine between vectors vm_i and vm_j, corresponding to m_i and m_j after dimensionality reduction is applied. Just as cosine values range from -1 to 1, so do textual similarities. The closer a value is to one, the more similar the texts of the documents/methods are. Note that textual similarities are symmetric, that is the similarity between m_i and m_j is the same as the similarity between m_j and m_i. In Section 3.8, an example of how to compute textual feature coupling using the textual similarities between two features is given.

3.3.5 Textual Feature Coupling Metric

We define textual feature coupling metrics based on unstructured, textual information found in source code. In order to define a metric for the textual coupling between features, we first define the conceptual similarity between two methods as well as between a method and a feature. These measures are building blocks needed to define our textual feature coupling metric. It should be noted that, like structural feature coupling metrics, textual feature coupling metrics also infer relationships between features based on the methods in which they are implemented in.

Definition 5: (Conceptual Similarity between Methods – CSM)

As defined in (Poshyvanyk and Marcus 2006), the conceptual similarity, also known as the textual similarity, between methods $m_i \in M_S$ and $m_j \in M_S$ is $CSM(m_i, m_j)$ where

$$CSM(m_i, m_j) = \frac{vm_i^T vm_j}{|vm_i|_2 \times |vm_j|_2} \quad (3)$$

$CSM(m_i, m_j)$ is the cosine between vectors vm_i and vm_j, corresponding to m_i and m_j after indexing. As defined, the value of $CSM(m_i, m_j) \in [-1, 1]$. In order to comply with the non-negativity property of coupling (Briand et al. 1996), if $CSM(m_i, m_j) \leq 0$, we redefine $CSM(m_i, m_j) = 0$. *CSM* measures the textual similarity of two methods, but most features are composed of more than one method. Next, we define the conceptual similarity between a single method and a feature.

Definition 6: (Conceptual Similarity between a Method and a Feature – CSMF)

Let f_a and f_b be two distinct features in *S*. Each feature has a set of methods $M_a = \{m_{a1}, m_{a2,\ldots}, m_{ax}\}$, where $x = |M_a|$ and $M_b = \{m_{b1}, m_{b2,\ldots}, m_{by}\}$, where $y = |M_b|$. Between every pair of methods, there is a similarity measure $CSM(m_a, m_b)$. The textual similarity between a method m_a from f_a

and a feature f_b is:

$$CSMF(m_a, f_b) = \frac{\sum_{q=1}^{y} CSM(m_a, m_{bq})}{y} \quad (4)$$

which is the average of the textual similarities between a method m_a and all methods in feature f_b. Now that we have a measure of the textual similarity of one method to a feature, we can define the textual similarity among all the methods of two features, *i.e.* their textual coupling.

Definition 7: (Textual Feature Coupling – TFC)

Let f_a and f_b be two distinct features in S. The textual coupling between f_a and f_b is:

$$TFC(f_a, f_b) = \frac{\sum_{l=1}^{x} CSMF(m_{al}, f_b)}{x} \quad (5)$$

which is the average of the textual similarity measures between all unordered pairs of methods from feature f_a and f_b. $TFC(f_a, f_b)$ is a measure of the textual coupling between the two features. This definition guarantees that the coupling between two features is symmetric.

Definition 8: (Maximum Textual Feature Coupling – TFC_{max})

In (Poshyvanyk and Marcus 2006), a variant of the conceptual class coupling metric was used in which only the highest textual similarities between methods of a class are considered. Similarly, we define such an alternative measure for textual feature coupling. We refine *TFC* to only capture the strongest textual similarity between features. Under this definition, the maximum textual similarity between a pair of features f_a and f_b is:

$$TFC_{max}(f_a, f_b) = \max\{CSM(m_a, m_b) \mid m_a \in M_a, m_b \in M_b\} \quad (6)$$

3.4 Hybrid Feature Coupling

Definition 9: (Hybrid Feature Coupling – HFC)

Structural information aligns with a program's structured information (*e.g.*, source code) while unstructured, textual information aligns, to some degree, with domain concepts (*e.g.*, requirements). We combine structural and textual information into a single feature coupling metric to take advantage of this complementary relationship. Both SFC and TFC capture coupling utilizing relationships between sets of methods in which features are implemented; therefore HFC also measures coupling using identical underlying information. Thus, the hybrid coupling between features f_a and f_b is defined as:

$$HFC(f_a, f_b) = w_{SFC} * SFC(f_a, f_b) + \quad w_{TFC} * TFC(f_a, f_b) \quad (7)$$

A weight between zero and one is chosen for both the structural and textual feature coupling values such that the sum of the weights equals one. The higher the weight, the more preference is given to that metric. We chose this straightforward means of combining the two metrics because we were interested in investigating, in a controllable fashion, whether combining structural and textual information captures new facets of feature coupling.

3.5 Theoretical Evaluation

Our feature coupling metrics comply with the five mathematical measurement properties proposed by (Briand et al. 1996): non-negativity, null value, monotonicity, merging of modules, and disjoint module additivity. Both our structural and textual feature coupling measures assume non-negative values. *SFC* and *SFC*' are based on the cardinality of sets and therefore their minimum value is zero. By redefining *CSM* to always produce a value greater than or equal to zero, *TFC* and TFC_{max} comply with the non-negativity property. Since *HFC* is based on the combination of *SFC* and *TFC*, it also obeys the property. Additionally, when there is no relationship between two features, our metrics return a measurement of zero, meeting the null value property. To fulfill the monotonicity property, when a new method is added to a feature that is shared by another feature or had a strong textual similarity to methods in another feature, our coupling metrics increase instead of decreasing. Finally, the coupling obtained after merging two features is not greater than the sum of the coupling of the two original features. That is, if we assume the set of methods which implement two features are union together to create a new feature, the sum of the coupling metrics for the individual feature will not exceed that of the new feature. Thus, the final two properties are met.

Table 3. Types of connection, a dimension of the unified framework for coupling measurement.

#	Element 1	Element 2	Description	Measures
1	Attribute *a* of class *c*	Class *d*, $d \neq c$	Class *d* is of type *a*	*DAC*, *DAC'*, class-attribute
2	Method *m* of class *c*	Class *d*, $d \neq c$	Class *d* is the type of a parameter of *m* or *m*'s return type	class-method interaction
3	Method *m* of class *c*	Class *d*, $d \neq c$	Class *d* is the type of a local variable of *m*	
4	Method *m* of class *c*	Class *d*, $d \neq c$	Class *d* is the type of a parameter of a method invoked by *m*	
5	Method *m* of class *c*	Attribute *a* of class *d*, $d \neq c$	*m* references *a*	*CBO*, *CBO'*, *COF*
6	Method *m* of class *c*	Method *m'* of a class *d*, $d \neq c$	*m* invokes *m'*	*CBO*, *CBO'*, *RFC*, RFC_a, *MPC*, *COF*, *ICP*, *NIH-ICP*, *IH-ICP*, method-method interaction
7	Class *c*	Class *d*, $d \neq c$	High level relationships between classes	
8	**Method *m* of feature *f***	**Method *m'* of feature *g*, $g \neq f$**	***m* is the same as *m'***	***SFC*, *SFC'***
9	**Method *m* of feature *f***	**Method *m'* of feature *f***	***m* and *m'* are textually similar**	***TFC*, TFC_{max}**

Table 4. Mapping of Coupling Measure to Domain.

Domain	Measures
Attribute	
Method	*ICP*, *NIH-ICP*, *IH-ICP*
Class	*CBO*, *CBO'*, *RFC*, RFC_a, *MPC*, *COF*, class-attribute interaction, class-method interaction, method-method interaction, *CoCC, RTC, CCBO*
Set of Classes	*ICP*, *NIH-ICP*, *IH-ICP*
Feature	***SFC*, *SFC'*, *TFC*, TFC_{max}, *HFC***
System	*COF*

We have developed tool support for feature coupling measurement. FLAT[3] (Feature Location And Textual Tracing Tool), which is overviewed in Figure 1, is an Eclipse plug-in based on ConcernMapper[3] and ConcernTagger[4] that supports mapping features to source code and the computation of feature coupling metrics (Savage et al. 2010). Users can manually associate features with source code or use an embedded feature location technique based on our prior research. Alternatively, feature-method mappings can be imported from existing models (Marin et al. 2007; Robillard and Murphy 2007) or tools such as ConcernMapper or ConcernTagger. If the source code or mappings are changed in successive versions of a system, the data given to FLAT[3] must also be updated. Updating such mappings can be performed by importing up-to-date mappings, manually updating mappings or semi-automatically using an embedded feature location technique.

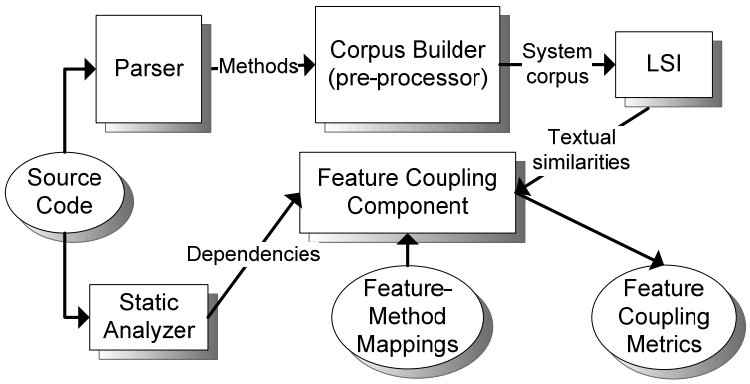


Figure 1. Architecture of the feature coupling component of FLAT[3].

Admittedly, the cost of mapping features to code can be expensive, but research areas such as feature location are focused on automatically recovering such mappings. For instance, Ratiu et al. (Ratiu and Deissenboeck 2006; Ratiu and Deissenboeck 2007) have developed a formal framework for mapping domain concepts to program elements. Also, some integrated development environments like IBM's Jazz[5] have embedded automatic traceability functionalities for requirements and bug fixes that could be leveraged. These techniques and tools can ease the burden creating feature-method mappings.

Based on the mappings of features to code, our feature coupling metrics can be computed. First, the source code of a system is parsed into methods. Then, the text of the methods is pre-processed to form the documents of the corpus. Pre-processing always removes stop words and programming language keywords and splits compound identifiers. Options include removing comments from the corpus and performing stemming. Then, LSI is used to create a word-by-document matrix that describes the distribution of terms in the methods of the corpus. Through the use of SVD, a semantic subspace is constructed in which each method from the corpus is represented as a vector. The cosine between two vectors (*i.e.*, *CSM*) is a measure of the textual similarity between two methods. Given the similarities between methods and the mappings of features to methods, FLAT[3] can compute *TFC*. To compute *SFC*, the tool simply requires feature-method maps as well as dependency information.

3.8 An Example of Measuring Feature Coupling

We provide an illustrative example of how *SFC* and *TFC* are calculated. The example is taken from our evaluation of the system *Rhino*, a Java implementation of JavaScript, and two of its features are type conversions ToString (f_{string}) and ToObject (f_{object}). Feature f_{string} is implemented by four methods ($M_{string} = \{m_{s1},\ldots, m_{s4}\}$), and feature f_{object} is implemented by eight methods ($M_{object} = \{m_{o1},\ldots, m_{o8}\}$). Note that m_{s2} is the same as m_{o8}.

The structural coupling between these two features is straightforward to compute. $SFC(f_{string}, f_{object}) = 1/11 = 0.09$ because the two feature have one method in common out of 11 total. Our metric captures the weak structural coupling between f_{string} and f_{object}. The two features are concerned with converting an argument, and the only method they share deals with determining the type of the argument before the conversion.

To compute textual coupling, the following formula is used: $TFC(f_{string}, f_{object}) = (CSMF(m_{s1}, f_{object}) + CSMF(m_{s2}, f_{object}) + CSMF(m_{s3}, f_{object}) + CSMF(m_{s4}, f_{object}))/4$. $CSMF(m_{s1}, f_{object})$ is the average of the textual similarities between method m_{s1} and all methods in f_{object} such that $CSMF(ms1, f_{object}) = (CSM(m_{s1}, m_{o1}) + CSM(m_{s1}, m_{o2}) + \ldots + CSM(m_{s1}, m_{o8}))/8$. The textual similarities between methods are shown in Table 5. These are the *CSM* values. Thus $CSMF(m_{s1}, f_{object}) = (0.60 + 0.24 + 0.54 + 0.68 + 0.36 + 0.23 + 0.19 + 0.24)/8 = 0.39$, $CSMF(m_{s2}, f_{object}) = 0.40$, $CSMF(m_{s3}, f_{object}) = 0.27$, and $CSMF(m_{s4}, f_{object}) = 0.09$. Finally, $TFC(f_{string}, f_{object}) = (0.39 + 0.40+ 0.27 + 0.09)/4 = 0.29$. The textual coupling between f_{string} and f_{object} is stronger than the structural coupling. The two features do use some common identifiers such as "Number," "Object," "ScriptRuntime," and "val," but otherwise, they have their own vocabularies.

Table 5. Textual similarities between methods of *Rhino*'s ToString and ToObject features.

$\mathbf{m_{s1}}$: ScriptRuntime.toString(Object); $\mathbf{m_{s2}} = \mathbf{m_{o8}}$: FunctionObject.convertArg(...);

$\mathbf{m_{s3}}$: Context.toString(Object); $\mathbf{m_{s4}}$: NativeRegExpCtor.setInstanceIdValue(...);

$\mathbf{m_{o1}} - \mathbf{m_{o5}}$: ScriptRuntime.toObject(*); $\mathbf{m_{o6}} - \mathbf{m_{o7}}$: Context.toObject(*)

	$\mathbf{m_{o1}}$	$\mathbf{m_{o2}}$	$\mathbf{m_{o3}}$	$\mathbf{m_{o4}}$	$\mathbf{m_{o5}}$	$\mathbf{m_{o6}}$	$\mathbf{m_{o7}}$	$\mathbf{m_{o8}}$
$\mathbf{m_{s1}}$	0.60	0.24	0.54	0.68	0.36	0.23	0.19	0.24
$\mathbf{m_{s2}}$	0.28	0.25	0.27	0.33	0.25	0.48	0.37	1.00
$\mathbf{m_{s3}}$	0.17	0.16	0.18	0.22	0.18	0.57	0.28	0.42
$\mathbf{m_{s4}}$	0.06	0.08	0.06	0.07	0.05	0.13	0.11	0.19

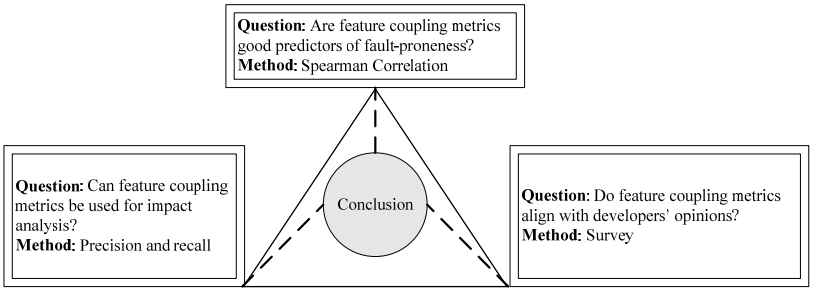


Figure 2. Our data triangulation evaluation approach.

To calculate the hybrid coupling between these two features, the weight given to each type of coupling needs to be established. If $w_{SFC} = 0.5$ and $w_{TFC} = 0.5$, then the hybrid feature coupling is computed as $HFC(f_{string}, f_{object}) = 0.5 * 0.09 + 0.5 * 0.29 = 0.19$.

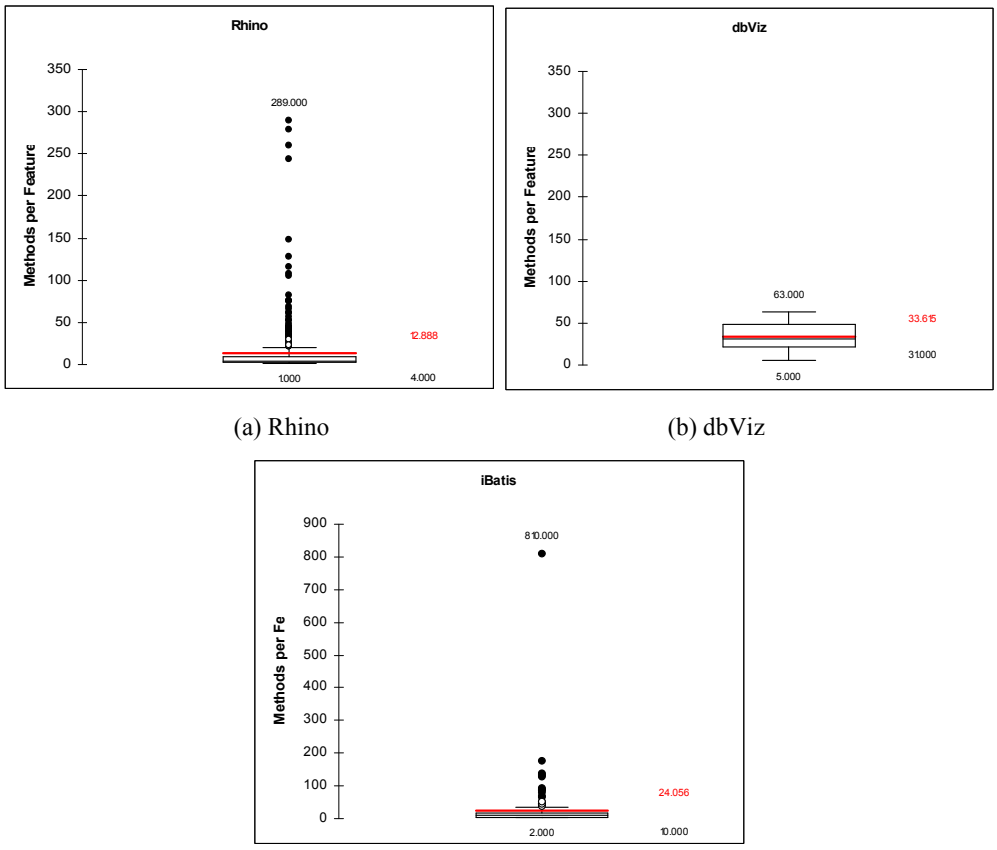
4 Case Studies

The purpose of our evaluation is to assess the usefulness of our new feature coupling metrics as well as to show that they have a practical application. We perform three assessments of the metrics, each targeting a different aspect of their utility or applicability. In the first case study, we explore the relationship between feature coupling and fault-proneness. To that end, we calculate the correlation between the metric values and bugs for all unique pairs of features in two software systems. If there is a high correlation between a feature coupling metric and defects, then that metric may serve as a useful predictor of fault-proneness among features. For our second case study, we examine the application of feature coupling metrics for impact analysis. If feature coupling metrics help determine other features likely to be affected by a change to a feature undergoing modification, then these new measures are helpful in the context of impact analysis. Finally, our third case study involves testing if the feature coupling metrics align with developers' opinions about which features are coupled or not. We carry out a survey in which 31 programmers rated the strength of coupling between 16 randomly selected pairs of features from three different software systems.

By considering the results of three evaluations, we can come to a stronger conclusion about the usefulness of feature coupling metrics than if we had used only one assessment. This idea of synthesizing data from multiple analyses is known as data triangulation (Yin 2003). The advantage of such an approach is that by corroborating multiple sources of evidence, any findings or conclusions are likely to be more valid. Figure 2 summarizes our data triangulation approach, and in the following sections we provide the details and results of each part of our evaluation.

4.1 Subject Systems and Data Sets

To be able to compute our feature coupling metrics, we required mappings of features to the methods that implement them in a given software system. Obtaining this information from a single developer is difficult, time-consuming, and biased (Robillard et al. 2007). These factors



(a) Rhino (b) dbViz

(c) iBatis

Figure 3. Box plots summarizing the distribution of methods per feature for (a) Rhino (b) dbViz and (c) iBatis.

led us to select several existing data sets made available by Eaddy et al. (Eaddy et al. 2008) in which multiple researchers compiled mappings of features to code. We used the information in these data sets to compute our feature coupling metrics, and we consider these data sets to be reliable since they have been previously used in other studies (Eaddy et al. 2008; Eaddy et al. 2008). Since we utilized previously published data, our study is reproducible; we invite other researchers to replicate our work. All of our data and results are provided in an online appendix[6].

The first data set we use is *dbViz*[7] version 0.5, an open-source database visualization tool written in Java. The system is comprised of 12,700 LOC (lines of code), 93 classes, and 554 methods. We also utilize the *Rhino* data set. Rhino[8] is a Java implementation of JavaScript consisting of approximately 32,000 LOC, 138 classes, and over 1,800 methods. The final data set we use is *iBatis*[9] version 2.3, an object-relational mapping tool written in Java that has 13,300 LOC, 212 classes, and over 1,800 methods.

The data sets include mappings of program elements to features. Eaddy et al. (Eaddy et al. 2008) identified 13 features from *dbViz*'s use cases, 411 features in *Rhino* from the ECMAScript specification[10] of JavaScript, and 132 features for *iBatis*. The distribution of the number of methods per feature for each system appears in Figure 3. For each feature in the data sets, the code associated with it was manually identified using the prune dependency rule: "A program element is relevant to a [feature] if it should be removed, or otherwise altered, when the [feature] is pruned" (Eaddy et al. 2008). In other words, to assign code (methods and fields) to the features they implement, the authors of (Eaddy et al. 2008) considered a scenario where a feature was to be removed from a system and attempted to remove as much relevant code as possible without affecting other features. While the data sets map some fields to features, we excluded field mappings from our evaluation because our model does not currently support them.

If the features identified in the data sets are well encapsulated by classes, then measuring feature-level coupling is without merit. To check if the features in the three data sets are implemented in multiple classes, we calculated the average and median number of classes per feature (this data is also available in (Eaddy 2008)). In *dbViz*, features are located in nine classes, on average, with two being the minimum, 21 the maximum, and 9 the median. The average

Table 6. Corpus configurations used in the case study.

Name	Description
c-ns	Comments included but without stemming
c-s	Comments included and stemming performed
nc-s	Without comments but with stemming
nc-ns	Comments excluded and no stemming

number of classes per feature in *Rhino* is four, with a minimum of one, a maximum of 67, and a median of 2. Finally, *iBatis*' features are implemented in six classes on average, with a minimum of one, a maximum of 128, and a median of 3. Since most of the features from the three data sets are implemented in multiple classes, traditional class-level coupling metrics are not able to capture the dependencies between features. Therefore, metrics at a higher level of abstraction, such as feature coupling metrics, are needed.

The data sets also include defect information. We use this data on bugs and where they occur in our first two case studies. In *dbViz*, 47 bugs are mapped directly to features. Each feature has at least two bugs associated with it, and on average, a feature has 4.7 bugs. In *Rhino*, 149 bugs are mapped to program elements. Of the 411 features, 344 have bugs, and each feature has 6.4 bugs on average. The publically available data sets did not include defect data for *iBatis*. If a method was modified to fix a bug, that method is associated with that bug. Transitively, if a feature is associated with a method, and that method was changed to fix a bug, then that bug is mapped to that feature. See (Eaddy et al. 2008) for the complete details on how the mappings were obtained.

In this work we perform case studies to (1) study the relationship between feature coupling and faults, (2) support feature level impact analysis using textual and structural coupling, and (3) investigate the relationship between the metrics and software developers. The first two case studies utilize *dbViz* and *Rhino* datasets, while the third study makes use of *dbViz*, *iBatis*, and *Rhino*. The difference in the datasets used for the studies is attributed to the lack of defect data for *iBatis* (*i.e.*, we could not answer research questions in study (1) and (2) since we did not have bug data available for *iBatis*).

4.2 Case Study Settings

In Section 3.3.2, we explained the process of building a corpus in order to obtain textual similarities between methods. There are several options for building a corpus; comments can be included or excluded and text can be stemmed or not. Comments are associated with a method if they appear within the method body or directly above a method definition. More details (with accompanying examples) on how we associated the comments with methods can be found in (Fluri et al. 2009). Comments embed additional domain knowledge within the source code of a system. Their inclusion, or exclusion, from a corpus can have an impact on the textual similarities between methods (Marcus and Poshyvanyk 2005). Stemming reduces words to their root, thus potentially increasing the textual similarity of two documents. Both of these options have implications for textual feature coupling. One of the secondary goals of our evaluation is to discover the optimal configuration for measuring textual feature coupling. We generated different versions of the corpus of a software system in order to explore the effect of corpus creation on feature coupling. The four corpus versions we created appear in Table 6. These corpora represent all possible combinations of the preprocessing options for comments and stemming. We consider the *c-ns* corpus to be the default. For one system in which external documentation was available (*Rhino*), we made a fifth corpus (*c-ns+d*). This corpus included source code text including comments, the external documentation's text, and words were not stemmed. The documentation is simply added to the corpus as more text; it is not mapped to source code. This augmented corpus was then used by LSI to compute similarities. The idea behind including documentation is that it encodes additional domain knowledge which may bolster the textual information in source code.

For each system, we computed five feature coupling metrics for each pair of features: *SFC*, *SFC*', *TFC*, $TFC_{\max}$, and *HFC*. *TFC* and TFC_{max} are based on the default corpus, and for HFC, we placed equal weight on structural and textual information. We also refer to this instance of *HFC* as $S_{0.5}T_{0.5}$, indicating a structural weight of 0.5 and a textual weight of 0.5. For each of these feature coupling metrics, we investigated their relationship with faults. Table 7 summarizes the descriptive statistics for the feature coupling measures using the default corpora

Table 7. Descriptive statistics of the feature coupling metrics.

System	Metric	*Max*	*75%*	*Med.*	*25%*	*Min*	*μ*	*σ*
dbViz	***SFC***	0.85	0.08	0.04	0.01	0	0.08	0.15
	SFC'	0.92	0.40	0.32	0.25	0	0.33	0.19
	TFC	0.22	0.09	0.08	0.06	0.02	0.08	0.04
	TFC_{max}	1	1	1	1	0.21	0.92	0.19
	HFC	0.53	0.09	0.06	0.04	0.01	0.08	0.09
Rhino		*Max*	*75%*	*Med.*	*25%*	*Min*	*μ*	*σ*
	SFC	1	0	0	0	0	0.02	0.11
	SFC'	1	0.05	0.01	0	0	0.06	0.16
	TFC	1	0.22	0.13	0.09	0	0.19	0.17
	TFC_{max}	0	0.27	0.50	0.86	1	0.55	0.31
	HFC	1	0.12	0.07	0.04	0	0.11	0.12
iBatis		*Max*	*75%*	*Med.*	*25%*	*Min*	*μ*	*σ*
	SFC	1	0	0	0	0	0.01	0.05
	SFC'	1	0.02	0	0	0	0.03	0.09
	TFC	0.99	0.13	0.09	0.06	0	0.11	0.10
	TFC_{max}	0	0.19	0.35	0.55	1	0.40	0.29
	HFC	0.91	0.07	0.04	0.03	0	0.06	0.07

configurations. We list the maximum (max), minimum (min), inter-quartiles (75%, median, 25%), mean (μ), and standard deviation (σ). The values are based on the 78 unique pairs of features in *dbViz*, 84,255 unique feature pairs in *Rhino*, and 13,041 pairs in *iBatis*.

4.3 The Relationship between Feature Coupling and Faults

To investigate the relationship between feature coupling and fault-proneness, we performed an empirical study. We conjecture that since features can be implemented in classes and methods dispersed throughout a system, the impact of changes to features can be difficult to determine, possibly leading to faults or system failures. Therefore, we hypothesize that the more coupled two features are, the more likely they are to share a bug. More formally, we seek to evaluate the following hypotheses.

$H_{o,1}$ The null hypothesis is that there is no significant correlation between the strength of coupling of two features and the number of bugs they have in common.

$H_{a,1}$ The alternative hypothesis is that there is a statistically significant correlation between the strength of coupling of two features and the number of bugs they share.

If we are able to reject $H_{0,1}$ with high confidence, it means that, the correlations we obtained are not likely to occur by coincidence. To test our hypotheses, we computed feature coupling metrics between all pairs of features in *dbViz* and *Rhino*. Additionally, we counted the number of bugs shared by two features for all feature pairs in each system. Then, we computed the Spearman rank order correlation between the metrics and defects.

In addition to computing coupling metrics for each pair of features, we also determined the number of defects shared by any two features. We considered a bug to be associated with a pair of features if any methods mapped to the features are also associated with the bug. Consider the example in Figure 4. Bug_1 is mapped to methods m_1, m_2, and m_3, while Bug_2 is associated with m_4 and m_5. Feature f_a is implemented by methods m_1 and m_3, while f_b is mapped to m_3, m_4, and m_5. f_a is associated with Bug_1 because its two methods are associated with the defect. Likewise, f_b is associated with Bug_1 and Bug_2. Therefore, f_a and f_b are both associated with Bug_1, so these features share that bug in common. This provides an example of one scenario where bugs are associated with a pair of features. Additionally, we associate a bug with a pair of features if it is associated with a method from each feature and also if the bug is associated with one of the two features in the pair.

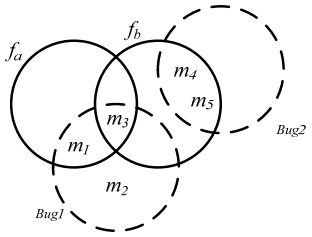


Figure 4. An example showing how shared bugs between features f_a and f_b were determined.

Table 8. Spearman correlation coefficients for *dbViz* and *Rhino*. All values are statistically significant at the one percent level (two-tailed). The sample size (number of feature pairs) is 78 for *dbViz* and 84,255 for *Rhino*.

c-ns: comments, no stemming, **c-s**: comments, stemming; ***nc-s***: no comments, stemming, **nc-ns**: no comments, no stemming; ***c-ns+d***: comments, no stemming, external documentation

	dbViz				*Rhino*				
Metric	***c-ns***	***c-s***	***nc-s***	***nc-ns***	***c-ns***	***c-s***	***nc-s***	***nc-ns***	***c-ns+d***
SFC	0.38	0.38	0.38	0.38	0.62	0.62	0.62	0.62	0.62
SFC'	0.35	0.35	0.35	0.35	0.58	0.58	0.58	0.58	0.58
TFC	0.52	0.13	0.15	0.15	0.38	0.35	0.35	0.37	0.38
TFC_{max}	0.16	0.16	0.16	0.16	0.63	0.63	0.63	0.62	0.63
HFC	0.49	0.47	0.47	0.47	0.44	0.42	0.41	0.43	0.44

Using our feature coupling metrics and the defect data, we calculated the Spearman rank order correlation coefficient (Spearman 1904) to determine the relationship between the feature coupling measures and fault-proneness. Table 8 lists the Spearman correlation coefficients for *dbViz* and *Rhino* for all the versions of the corpora. Note that the results based on structural information are constant across versions of corpora as processing techniques used do not impact structural properties of the corpora. Correlation coefficients can take values in the range of -1.0 to 1.0. A perfect negative correlation is denoted by -1.0, a perfect positive correlation is designated by a value of 1.0, and zero means no correlation. All of the Spearman correlations in Table 8 are statistically significant at the one percent confidence level, meaning there is only a 1% probability that the relationship is caused by chance.

The results for *dbViz* and *Rhino* indicate that there is a moderate to strong[11] correlation between the feature coupling metrics and defects. Under the default configuration (comments, no stemming) in *dbViz*, textual coupling had the strongest correlation (0.52) with bugs, while in Rhino structural coupling was the strongest (0.62). *HFC* is also moderately correlated with bugs in both systems. From these results, we can reject $H_{0,1}$, the null hypothesis. In other words, based on our study, the correlations obtained between our feature coupling metric and defects is statistically significant, therefore *feature coupling is correlated with defects*.

Under the different versions of the corpus, *SFC* is unchanged since corpus building does not impact structural information. However, textual coupling does change, and with it its correlation with bugs. In *dbViz*, *TFC*'s correlation with defects is significantly impacted by the exclusion of comments and the use of stemming since *dbViz* is a relatively small system. *TFC*'s correlation with bugs in *Rhino* does not suffer from the lack of comments or use of stemming as greatly as in *dbViz*, but there is still a slight weakening of the correlation. From this, we conclude that *the best configuration under which to build a corpus to measure textual feature coupling is to include comments but not to use stemming*. Stemming may indeed be useful in other contexts (De Lucia et al. 2007), but we did not observe it have an impact on these results.

Using this top-performing configuration, we created one additional corpus for *Rhino* that included the ECMAScript specification, an external document for *Rhino*. By including this documentation in the corpus, we are adding additional domain information. The last column of Table 8 (*c-ns+d*) lists the correlation values between the metrics and bugs for this version of the corpus. The numbers in the table are rounded so it is not obvious, but for all the metrics except TFC_{max}, the version of the corpus with the strongest correlation with bugs is *c-ns+d*. Consequently, *if programmers are seeking to use feature coupling to evaluate the fault-proneness*

Table 9. Spearman correlation coefficients for *HFC* in *dbViz* and *Rhino* using the default configuration of the corpora (***c-ns***). All values are statistically significant at the one percent level (two-tailed). The sample size (number of feature pairs) is 78 for *dbViz* and 84,255 for *Rhino*.

Metric	*dbViz*	*Rhino*	Metric	*dbViz*	*Rhino*	Metric	*dbViz*	*Rhino*	Metric	*dbViz*	*Rhino*
$S_{0.05}T_{0.95}$	0.52	0.38	$S_{0.3}T_{0.7}$	0.51	0.41	$S_{0.55}T_{0.45}$	0.47	0.44	$S_{0.8}T_{0.2}$	0.42	0.48
$S_{0.1}T_{0.9}$	0.53	0.39	$S_{0.35}T_{0.65}$	0.51	0.42	$S_{0.6}T_{0.4}$	0.46	0.45	$S_{0.85}T_{0.15}$	0.41	0.50
$S_{0.15}T_{0.85}$	0.53	0.39	$S_{0.4}T_{0.6}$	0.51	0.42	$S_{0.65}T_{0.35}$	0.45	0.46	$S_{0.9}T_{0.1}$	0.40	0.51
$S_{0.2}T_{0.8}$	0.52	0.40	$S_{0.45}T_{0.55}$	0.50	0.43	$S_{0.7}T_{0.3}$	0.44	0.47	$S_{0.95}T_{0.05}$	0.39	0.53
$S_{0.25}T_{0.75}$	0.52	0.41	$S_{0.5}T_{0.5}$	0.49	0.44	$S_{0.75}T_{0.25}$	0.43	0.48			

of features and have documentation available, it should be included in the corpus for improved results. This finding supports other results in the literature that state that the inclusion of documents besides source code improves information retrieval results (Ye and Fischer 2005).

4.3.1 Hybrid Feature Coupling

In addition to investigating the five feature coupling metrics above, we also explored the effect of varying the weights assigned to our hybrid feature coupling metric, *HFC*. By varying the weights, preference is given to one type of information over the other, which may be useful in cases when one source of information is more reliable than the other. For instance, if a system is poorly structured but has good identifier names, more weight can be placed on textual coupling. Table 9 lists the Spearman correlation coefficients for all possible *HFC* combinations with a step size of 0.05 for the default corpus. All the correlations are statistically significant at the one percent confidence level. In *dbViz*, textual coupling is more strongly correlated with bugs than structural coupling (0.52 vs. 0.38), so increasing the textual weight improves *HFC*'s correlation. The opposite is true in *Rhino* where structural coupling has a stronger correlation with bugs than textual coupling (0.62 vs. 0.38). Therefore, increasing the structural weight strengthens *HFC*'s correlation with defects. *Rhino* may have a stronger structural coupling than *dbViz* since it is an order of magnitude larger in size. Overall, the *HFC* variants have moderate correlations with defects, and programmers using *HFC* should select weights based on their assessment of the system and type of coupling they want to emphasize. However, when the quality of the structured or unstructured information is unknown, using the default weight of 0.5 provides good results.

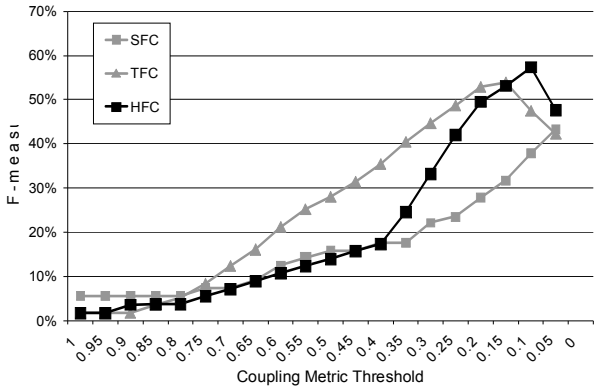
4.3.2 Comparison with an Existing Metric

The distance between features metric (*DIST*) introduced by Wong and Gokhale (Wong and Gokhale 2005) is a feature metric that is very similar to coupling because it measures the distance (or similarity) between features. *DIST* is computed based on information collected by dynamically executing a system. Since *DIST* is the state of the art in feature measurement, we compared our metrics to it. *DIST* was originally defined on basic blocks, but we redefine it here at the method level to be able to directly compare it with our metrics. Let M_a and M_b be the sets of methods executed by inputs that invoke features f_a and f_b respectively. Therefore, the distance between features f_a and f_b is

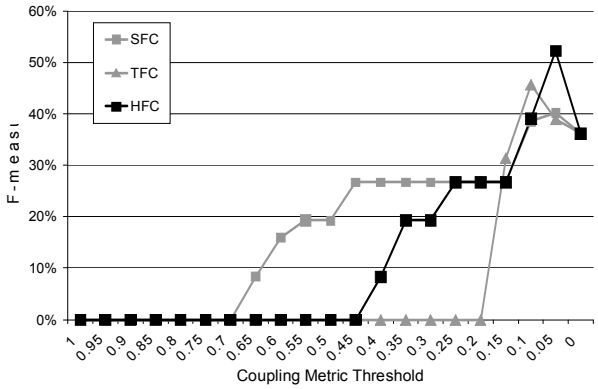
$$DIST(f_a, f_b) = \frac{|M_a \oplus M_b|}{|M_a \cup M_b|} \tag{8}$$

where $\oplus$ is the exclusive OR operator.

We collected one execution trace for each of *dbViz*'s 13 features and 51 of *Rhino*'s. The *dbViz* traces were based on the developers' use cases, while the *Rhino* traces were based on available test cases, and not all features had a test case. We computed *DIST* between all pairs of features and calculated the Spearman correlation to determine the relationship between *DIST* and fault-proneness. Bugs were associated with features as described in Section 4.1. For *dbViz*, The Spearman correlation coefficient for *DIST* and bugs is 0.02, and for *Rhino*, it is 0.05. Both values are not statistically significant. *DIST*'s correlation with defects is very close to zero, meaning that there is almost no correlation between the metric values and bugs. *In comparison, all of our metrics have positive moderate to strong statistically significant correlations with bugs*. *DIST* is



(a) Rhino



(b) dbViz

Figure 5. Average F-measure of coupled features in (a) *Rhino* and (b) *dbViz* for various thresholds.

expensive to compute because of the overhead of collecting traces. It is not a good predictor of faults, likely due to the imprecise nature of dynamic analysis. In contrast, our metrics are less expensive, and all of them are good predictors of fault-proneness.

Besides being the only feature coupling metric with no statistically significant correlation to bugs, an example of *DIST* highlights the problems associated with using dynamic information. Consider *dbViz*'s features to start *dbViz* and to exit the system. The dynamic coupling between these two features is 1 because despite what other features are invoked, the system must always be started and exited. This example shows the difficulty inherent in using dynamic information for feature coupling because some features cannot be invoked separately. On the other hand, *SFC* between these features is 0 and *TFC* is 0.08, reflecting the true lack of coupling between them, as is also supported by the fact that they do not share a bug.

4.4 Using Structural and Textual Coupling to Support Feature-Level Impact Analysis

Our second case study investigates the application of feature coupling metrics for impact analysis. Given a starting point, such as a change to some module, impact analysis involves detecting other modules within a system that may be affected by a change (Orso et al. 2004; Ren et al. 2004). Both class-level coupling and information retrieval have been used for impact analysis (Briand et al. 1999; Poshyvanyk et al. 2009; Gethers and Poshyvanyk 2010; Kagdi et al. 2010). Generally, to select candidate modules to investigate for impact analysis, a threshold is set on the coupling or textual similarity values. Previous research on using coupling or information retrieval for impact analysis has focused on identifying methods and classes (Poshyvanyk et al. 2009), not features.

Table 10. Precision and recall values for impact analysis of different metric thresholds in *Rhino*. The first value in a cell is precision, and the second is recall.

Threshold	*SFC*	*TFC*	*HFC*	Threshold	*SFC*	*TFC*	*HFC*
1	40%, 3%	5%, 1%	5%, 1%	**0.5**	66%, 9%	39%, 22%	57%, 8%
0.95	41%, 3%	5%, 1%	8%, 1%	**0.45**	67%, 9%	42%, 25%	61%, 9%
0.9	43%, 3%	8%, 1%	18%, 2%	**0.4**	71%, 10%	48%, 28%	66%, 10%
0.85	43%, 3%	17%, 2%	22%, 2%	**0.35**	72%, 10%	52%, 33%	69%, 15%
0.8	45%, 3%	18%, 3%	25%, 2%	**0.3**	75%, 13%	54%, 38%	68%, 22%
0.75	48%, 4%	24%, 5%	31%, 3%	**0.25**	75%, 14%	53%, 45%	70%, 30%
0.7	50%, 4%	27%, 8%	35%, 4%	**0.2**	77%, 17%	**52%, 54%**	68%, 39%
0.65	53%, 5%	30%, 11%	42%, 5%	**0.15**	77%, 20%	46%, 65%	63%, 46%
0.6	58%, 7%	36%, 15%	48%, 6%	**0.1**	**78%, 25%**	34%, 79%	**55%, 60%**
0.55	60%, 8%	37%, 19%	52%, 7%	**0.05**	**78,% 30%**	**28%, 86%**	34%, 80%

Therefore, we explore if feature coupling metrics can be used to find other features that are likely to be affected by a change to a feature undergoing modification by using defects identified in these features as an oracle.

To evaluate feature coupling in the context of impact analysis, we use available bug data from two systems to compute the precision, recall, and f-measure of the relevant coupled features recommended by our metrics. The process can be described as follows. For a bug b, we create a set $F_b = \{f_1, f_2, ..., f_n\}$ of features that all share the bug. That is, every feature in the set is associated with bug b. For each feature f_i in F_b, we determine which other features from all of the system's features are coupled to f_i by setting a threshold. For example, if the threshold is 0.5, then every feature that is coupled to f_i with a metric value equal to or above 0.5 is included in a new set T. Then, precision and recall are computed with T being the retrieved set and F_b (excluding f_i) as the relevant set. Precision is the ratio of the number of relevant features retrieved over the total number of features retrieved, while recall is computed as the number of relevant features retrieved divided by the total number of relevant features. The f-measure is the harmonic mean of precision and recall. For each bug b, we get precision, recall, and f-measure values. To get an overall measure of all bugs in the system, we summarize these precision, recall, and f-measure values using a macro-evaluation averaging technique as in (Zimmermann et al. 2005). Macro-evaluation means an average is taken of the values for all f_i in F_b and then for all bugs in the system. These values were computed for all threshold values with a step size of 0.05. We only present the results for *Rhino* here; the *dbViz* results, which are similar (as depicted in Figure 5 for both systems following similar properties with respect to HFC and thresholds), can be found in the online appendix.

Figure 5 shows the average f-measure and Table 10 shows the average precision and recall values for *SFC*, *TFC*, and one version of *HFC* ($S_{0.5}T_{0.5}$) at various coupling thresholds with a step size of 0.05. These results are for the default corpora of *Rhino* and *dbViz*. For *Rhino* the best precision for structural coupling is 78.4% with a recall of 24.8% at a threshold of 0.1, while the best recall is 30.2% with a precision of 77.9% at the 0.05 threshold, meaning at best slightly over three quarters of the candidate features are relevant, but only 25 to 30% of the relevant features are found. Textual coupling's best performance in terms of precision is 54.4% with a recall of 38.1% at the 0.3 threshold, while its best recall of 86% with 28.1% precision is at a threshold of 0.05. The precision of *SFC* seems to increase and then level out as the threshold decreases. The precisions of both *TFC* and S*FC* increase until a certain point, then both decline, likely due to the fact that the threshold is low enough that too many features are deemed textually coupled when they are not. We speculate that the increase in accuracy acquired by reducing the threshold is, in part, a result of relaxing the strength of the association indicated by the metrics, which indicates a pair of features is coupled. *SFC* had the best precision overall but the worst recall. The precision for *HFC* generally fell below that of *SFC* but above *TFC*, and its recall is above *TFC* and below *SFC*. Therefore, using hybrid feature coupling is a good compromise between the two other metrics. For example, at threshold 0.1, *HFC*'s precision is 55% and its recall is 60%.

The results in Figure 5 show similar results that are also obtained for *dbViz*. One obvious difference is that *SFC,* as opposed to *TFC,* yields highest accuracy for *dbViz*. This finding is perhaps related to the variation in characteristics of the software systems. But in light of this difference we continue to observe that the performance *HFC* falls in between the two techniques. This occurs until we reach a threshold (approximately 0.1 in both cases) at which point we observe

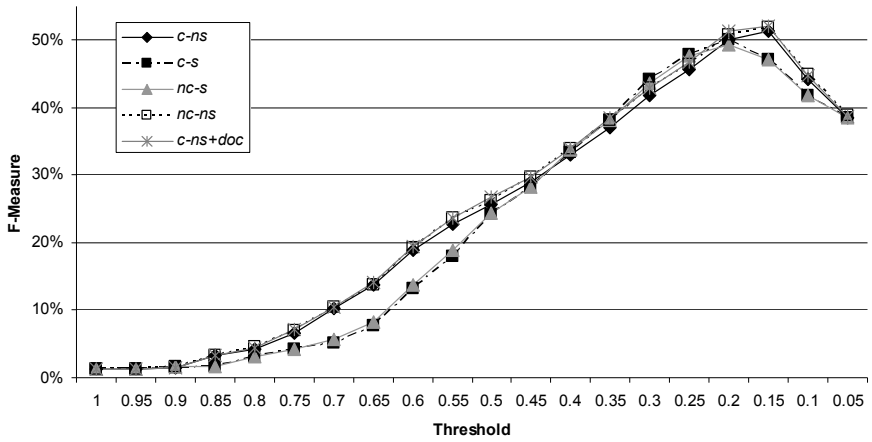


Figure 6. Impact analysis F-measure values of *TFC* for different *Rhino* corpora.

that the performance of *HFC* exceeds that of both *SFC* and *TFC*. It is possible that *HFC* performs better at lower thresholds because in those scenarios the metric is capable of identifying coupled features where the values returned by the two individual metrics were in disagreement. Consider the case where one metric returns a value of zero while the other returns a low coupling value. It is only at low thresholds where *HFC* is capable of overcoming this disagreement and identifying the coupling which exists between the two features. One other trend, which holds across systems, is the fact that there is a decline in F-Measure once a certain threshold is reached.

While feature coupling may not provide the best solution to the impact analysis problem, these results suggest that the metrics can still be useful. More research is needed to provide more practical techniques. However, these initial results are promising and comparable to some existing techniques on impact analysis based on structural and textual information (Briand et al. 1999; Poshyvanyk et al. 2009).

The precision and recall results also add weight to our claim that structural and textual feature coupling are complementary since their curves are different. We also executed the Kruskal-Wallis statistical test, a nonparametric alternative to the analysis of variance test, to assess if *SFC* and *TFC* are significantly different. At a significance level of 0.01, the test for both *dbViz* and *Rhino* show that *SFC*'s and *TFC*'s precision and recall values are indeed significantly different.

The impact analysis results presented thus far have been for the default version of the corpus used to obtain textual information. We also investigate the use of feature coupling metrics for impact analysis using different corpora configurations (see Section 4.2) for the *Rhino* system. Figure 6 shows the average f-measure of *TFC* for the various versions of the corpus. Recall that only textual information is affected by the corpus' configuration, so *SFC* remains the same across corpora. The graph indicates that the way in which a corpus is built does little to influence precision and recall for impact analysis, no matter the threshold. However, the corpus with comments and stemming typically has the highest precision and recall. Just as was observed with the Spearman correlation coefficients, the inclusion of comments yields better results. However, textual feature coupling still works well in cases where comments are missing.

Finally, we study the effects of *HFC*'s weights on precision, recall, and f-measure during feature level impact analysis. We provide only the results for the default corpora since the results for the other versions were similar. We select two metric thresholds that performed well for *SFC* and *TFC* (0.2 and 0.15) and calculate precision and recall of *HFC* for all weights with a step size of 0.05. Figure 7 shows the f-measure curves for *HFC* at a threshold of 0.2 (black lines) and 0.15 (gray lines). The x-axis denotes the structural weight. The corresponding textual weight is simply one minus the structural weight.

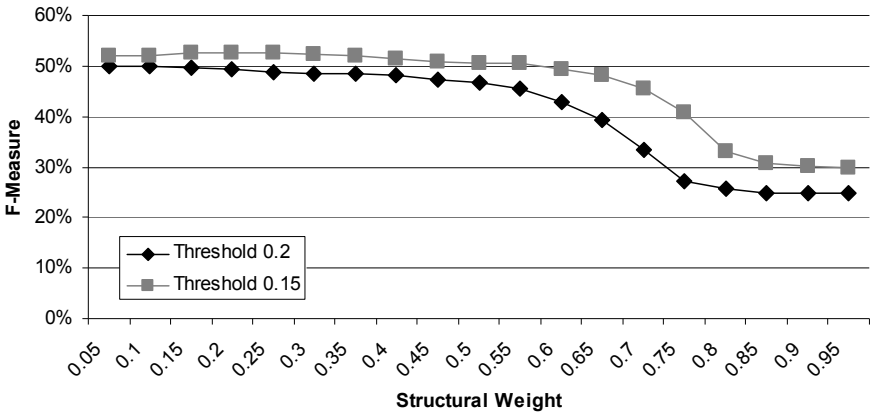


Figure 7. F-measure of *HFC* in *Rhino* at selected thresholds.

The graphs illustrate the effects of relying on one type of information over the other. Depending more heavily on structural information yields good precision at the cost of poor recall. Overall, using *HFC* produces better results than the standalone *SFC* and *TFC* metrics. Consider *HFC* with a structural weight of 0.2 and a textual weight of 0.8 at a threshold of 0.2. The precision is 51% and the recall is 55%. At the same threshold, *SFC*'s precision is 77%, but its recall is only 17%, so *HFC* is a better overall performer in this situation because its recall is much higher without sacrificing too much precision. Therefore, *HFC* helps alleviate those cases where the quality of either structural or textual information is low.

4.5 Developer Study

In the final part of our evaluation, we investigate if our feature coupling metrics align with developers' opinions of feature coupling. If the metrics indicate that two features are coupled and so do the majority of developers surveyed, then we can be confident in the utility of the measures. More formally, we formulate two null and two alternative hypotheses:

$H_{0,2}$: There is no agreement between developer's responses of how they rate coupling between pairs of features.
$H_{A,2}$: There is agreement between developer's responses of how they rate coupling between pairs of features.

$H_{0,3}$: There is no correlation between the developer's responses and the metrics about whether or not two features are coupled.
$H_{A,3}$: There exist correlation between the developer's responses and the values returned by the feature coupling metrics.

To test our hypotheses, we conduct a survey in which developers were asked to rate the strength of coupling among pairs of features. Below, we offer general details about the participants and the task they performed, as well as exploring the results of the survey. All the details on this developer study are available in the online appendix[12].

4.5.1 Participants

The respondents to our survey were 31 volunteer programmers from several different institutions. Twenty-three of the programmers were graduate students, one was an undergraduate, and seven were industry professionals. On average, they had 7.2 years of programming experience, 3.8 with Java, and 2.6 with Eclipse. Each volunteer was given a link to the survey's instructions and could complete it on their own time. The survey took 97 minutes to complete, on average.

4.5.2 Task Description

The programmers downloaded an Eclipse installation that was preloaded with our FLAT^3 plug-in and all the necessary source code. FLAT^3 included mappings of features to code for selected features from Eaddy et al.'s (Eaddy et al. 2008) data sets. The programmers could click on a feature's name to see the methods associated with it, and double clicking on a method to show its source code in the editor. The programmers were asked to consider the code of two features and rate whether the features were coupled. The responses varied according to the five-level Likert scale: "Strong No," "Weak No," "Unknown," "Weak Yes," or "Strong Yes." If a developer could not decide on a rating, they could respond "Unknown." The pairs of features included five from *dbViz*, six from *Rhino*, and five from *iBatis*. The set of feature pairs were randomly selected from the set of all feature pairs for each system prior to the evaluation. The exact instructions and pairs of features given to the participants as well as detailed descriptions of the features can be found in the online appendix.

4.5.3 Agreement among Participants and with the Metrics

The survey is a rating of n subjects (the 16 randomly selected feature pairs) by k raters (the 31 programmers). We tested if there was a sufficient amount of agreement among the developers' responses to be able to draw conclusions about the feature coupling metrics. To determine the amount of agreement among the raters, we designed our analysis in a fashion similar to (Mende et al. 2009) by using the intra-class correlation coefficient (*ICC*) (McGraw and Wong 1996). We used *ICC(A,1)*, which calculates the agreement of all the raters, where each person rates each subject (feature pair). The *A* means it is an absolute agreement, and the one indicates the ratings are not an average. With the ratings stored in a matrix with feature pairs as the rows and raters as the columns, *ICC(A,1)* is calculated as follows:

$$ICC(A,1) = \frac{MS_r - MS_c}{MS_r + (k-1)MS_e + k/n(MS_c - MS_e)} \quad (10)$$

where k is the number of raters, MS_r is the mean square for rows, MS_c is the mean square for columns, and MS_e is the mean square error. *ICC(A, 1)* relates the variance of the ratings of each feature pair to the overall variance.

The ratings given by the developers were ordinal, but numeric data is required to compute *ICC*. Therefore, we transformed the ratings of "Strong No," "Weak No," "Weak Yes," and "Strong Yes" to the values 1, 2, 3, and 4 respectively coding to interval level scores for analysis purposes (Sirkin 2005). Eight programmers gave a rating of "Unknown" for at least one feature pair. We omit all responses of those subjects to avoid issues pertaining to responses of "Unknown" and the calculation of *ICC*. The *ICC(A,1)* of the programmers in our survey is 0.443 (values can range from -1 to 1), meaning there is a moderate amount of agreement in their ratings of the pairs of features. Additionally, the result obtained was statistically significant allowing us to reject $H_{0,2}$. We believe that there is enough concordance to be able to draw conclusions. While we had to remove some responses from the computation of *ICC* (replies of "Unknown"), the rest of our analyses are based on the responses of remaining 23 developers.

4.5.4 Correlation between Participants Responses and the Metrics

In order to test our hypothesis ($H_{0,3}$) we make use of Spearman rank-order correlation. The ordinal data collected during our user study makes Spearman rank-order correlation ideal for determining the correlation between developer's responses and coupling metric results. We compute the Spearman rank-order correlation and obtain a correlation between developer responses and HFC metric of 0.404 with a p-value <0.0001. Based on these finding we are able to reject our null hypothesis $H_{0,3}$, thus suggesting that, based on our study, there exist sufficient evidence to suggest that there is a correlation between developer's responses and the HFC metric.

Table 11 (columns 5-8) summarizes the number of developers that gave each rating for *dbViz, iBatis,* and *Rhino*'s feature pairs, respectively. The columns contain the frequency in which a particular response was given by a developer for the corresponding feature pair. Frequencies presented in each column can be compared to the metric values in the third and the fourth columns. Such a comparison provides insight into the relationship between developer responses

Table 11. Feature coupling values for the *dbViz, Rhino,* and *iBatis* feature pairs in the developer study.

Features	Pair	*SFC*	*TFC*	Strong No	Weak No	Weak Yes	Strong Yes	Bugs
Connect to database & Exit *dbViz*	*dbViz #1*	0	0.03	17	4	2	0	0
Autoarrange Diagram & Undo/Redo	*dbViz #2*	0.07	0.06	2	5	8	8	0
Import from database & Import from SQL file	*dbViz #3*	0.61	0.15	0	1	3	19	1
Add table & Remove table	*dbViz #4*	0.85	0.22	0	2	0	21	0
Save/Load diagram & Load saved diagram	*dbViz #5*	0.45	0.13	0	1	4	18	2
Unary + operator & Addition operator	*Rhino #1*	0.33	0.27	3	3	3	14	15
Addition operator & Subtraction operator	*Rhino #2*	0.71	0.28	4	1	2	16	17
Date.prototype.toString & Date.prototype.valueOf	*Rhino #3*	0.75	0.74	3	5	0	15	2
Unicode format control chars & ToPrimitive	*Rhino #4*	0	0.08	16	3	3	1	0
parseInt & parseFloat	*Rhino #5*	0.40	0.46	8	2	3	10	2
SQRT2 & Date.prototype.getTimezoneOffset	*Rhino #6*	0	0.85	14	6	3	0	1
Data sources & JTA	*iBatis #1*	0.08	0.42	1	2	9	11	-
JDBC & JTA	*iBatis #2*	0	0.44	10	4	7	2	-
Query & Max Results	*iBatis #3*	0.15	0.47	0	1	10	12	-
Update & Autogenerated keys	*iBatis #4*	0	0.17	9	4	7	3	-
SELECT & SQL Scripts	*iBatis #5*	0	0.06	11	6	5	1	-

and metric values. For instance, the row in Table 11 for the pair *dbViz* #1, "Connect to database" and "Exit dbViz" shows both, metric values and user responses, are in agreement.

When both *SFC* and *TFC* are low, the overwhelming majority of responses are "Strong No," as can be seen by the first pair of features in *dbViz*. These features are to connect to a database and exit the program and have little in common, so low structural and textual coupling values are valid, as supported by the developers' ratings. Additionally, these features do not share any common bugs, which is further evidence that they are not coupled.

Another overall trend is that when *SFC* is high, most raters responded "Strong Yes." As an example of high structural coupling, consider the feature pair *dbViz* #3, "Import from database" and "Import from SQL file." These features are very similar in function and share a number of methods, so a high *SFC* value (0.61) makes sense, and the programmers' responses also support *SFC* being high. Furthermore, the two import features have a common bug, which also supports higher coupling between them. However, *TFC* between these two features is rather low (0.15) because the methods that are distinct to each feature have their own vocabulary.

One interesting case is the *Rhino* #6 feature pair. The two features are "SQRT2," the number value of the square root of two, and "Date.prototype.getTimezoneOffset" that gets the local time and UTC in minutes. There is no structural coupling between the features, but rather high textual coupling. The majority of responses for this feature pair were "Strong No" despite these two features having a high *TFC* value. Two options are possible: the features are not actually coupled and the high textual coupling is a coincidence, or the programmers did not pick up on the similarity in the two features' vocabularies because textual coupling is not as well known a concept as regular, structural coupling. The two features do have a shared bug, but after reviewing the two features' source code, the textual coupling seems to be artificial. "SQRT2" has two methods, and both of those methods' names happen to be the same as two of "Date.prototype.getTimezoneOffset" three methods. These methods perform similar parsing

functionalities and use a lot of the same variable names, so high textual coupling in this case seems to be accidental.

Another interesting case is the *Rhino* #5 feature pair: “parseInt” and “parseFloat.” *SFC* and *TFC* have approximately equal values which are both substantially greater than the average for each metric in *Rhino*. The developers are almost evenly split in their opinions of whether these two features are coupled, with a slight majority thinking they are coupled. The feature’s coupling is also supported by the fact that they have two bugs in common. The developers’ mixed ratings suggest that perhaps there is a coupling threshold, but that threshold varies from person to person.

The results for *iBatis* include an interesting case, the *iBatis* #4 feature pair, where the features "Update" and "Autogenerated keys" exhibit low coupling with respects to both *SFC* and *TFC* metrics. The "Update" feature relates to modifying data previously entered into a table in the database while the feature "Autogenerated keys" automatically generates keys for new entries into a database table. Although the majority of participants either responded "Strong No" or "Weak No" there were a noticeable number which thought the two were coupled to some degree. Clearly there exists a logical relationship between the two which may explain why a few participants indicated that the two were coupled.

Overall, the ratings given by the programmers seem to support our feature coupling metrics. This implies that the measures do capture the coupling between features. Generally, the respondents’ opinions support *SFC* more than *TFC*, but that may be due to the fact that feature textual coupling is a newer concept.

4.6 Threats to Validity

In this section, we discuss the main threats to the validity of our case studies and provide details on how we minimized these threats.

4.6.1 Internal Threats to Validity

Internal validity refers to the degree to which statements about cause and effect are valid. Since we use previously published data sets, we inherit all of the threats to validity associated with them. One internal threat of the data sets is the subjective manner in which methods were assigned to features. These facts limit the consistency of our results because different mappings would produce different results. However, since the data sets have been used and verified by other researchers (Eaddy et al. 2008; Eaddy et al. 2008), these threats are minimized. Additionally, Spearman rank-order correlation can mitigate unreliable measurements as long as their relative order is correct (Kan 2003). Also, the *Rhino* data set has a large sample size (84,252 feature pairs). The moderate and strong correlations observed are unlikely if the data is unreliable. Another threat we inherit from the data sets pertains to the assignment of bugs. As with any approach to mining software repositories, defects can potentially be mapped to wrong or missing methods if methods undergo a change in signature. Similarly, automated repository mining does not always provide a complete picture of a bug’s history. It may lack social, technological, and organizational knowledge (Aranda and Venolia 2009) or may be biased and only record a fractions of bug fixes (Bird et al. 2009).

Another threat related to the data sets is their granularity. Full methods are associated with features. However, only a small portion of the code in a method may actually pertain to a feature (Koschke and Quante 2005; Revelle et al. 2005). Therefore, a finer level of granularity such as statements or basic blocks would be more accurate. Since we are not experts in any of the systems we studied, we made no attempts to refine the granularity of the data sets.

Another threat to validity associated with our data set pertains to our approach for associating bugs with pairs of features. In this work we increment the bug count per feature pair for any bug associated with a method from either feature in the pair. In doing so we overlook the distinct impact scenarios where (1) a defect is associated with a method shared by the two features and (2) a defect is associated with methods uncommon to both features. So our results provide more general insight to the association between features and bugs and do not explore these scenarios individually in which bugs can be associated with pairs of features.

In our case studies, we observed a high correlation between feature coupling and defects, which may imply that feature coupling can serve as a predictor for faults. However, correlation values only measure goodness of fit, not predictive power. To better assess predictive power, we would

need to perform some form of data splitting, such as ten-fold cross-fold validation, which is part of our future work.

As with other similar approaches to impact analysis, we currently do not suggest any technique for selecting an optimal threshold. So, it is possible that the set of thresholds considered in this paper excludes the threshold which provides optimal performance. Additionally, the best threshold will depend on the specific software system. To overcome this threat, we would need to implement an approach, which is capable of identifying the best threshold for a given data set, which is out of the scope for this paper. On the other hand, we minimize this threat by providing the results using a number of threshold values.

Our approach to combining multiple sources of information to measure coupling among pairs of features requires weights be assigned to both structural and textual coupling metrics. Currently, we do not provide any heuristic to determine such weights. Instead we explore various possibilities and highlight scenarios, which lead to the best performance. It is possible that the weights selected for HFC in each case will not provide the best results on different software systems.

In the context of our survey, there are a number of threats to validity. First, the programmers' proficiency with Java and Eclipse is a threat because we did not select participants based on their familiarity with either technology. Some of the programmers had no experience with Java or Eclipse. By including programmers with little or no Java experience in our survey, there is a danger that they made poor choices due to their unfamiliarity with the programming language. Another threat related to the programmers is their motivation. All the developers who participated were volunteers and received no compensation for their time or effort, so there was no motivation for them to perform well. On the other hand, we did not give them a time constraint, so there was no time pressure to complete the survey quickly.

Two final threats to the validity of our survey pertain to the task the programmers were asked to complete. The participants were instructed to consider if two features were coupled in the context of performing a change task to either. No specific change task was given, leaving the task rather open-ended and general. However, it may be difficult to gauge the relationship between two features without a specific context. There could be changes made to a feature that affect the other one, but other changes made to the same feature may not affect the other feature. To avoid making a judgment call about a specific change tasks, we kept the task general. Additionally, the mappings of features to code the participants were given was the same as was used to compute *SFC*. This introduces a bias that could not be avoided.

Lastly, there exists a threat related to the results obtained for the ICC coefficient, which captures agreement amongst developers in the study. ICC is a parametric test statistic, which assumes the data set is sampled from a normal distribution. Based on our data set normality, tests indicate that we are unable to make such an assumption. It is possible that not abiding to the assumption influences our results for ICC to some degree.

4.6.2 External Threats to Validity

External threats to validity limit the degree to which generalizations can be drawn from our results. We studied only three systems, one small and the other two medium in size. In future work, our feature coupling metrics will be validated on larger systems. However, the number of features studied in *Rhino* was large (411), and the feature coupling metrics of both systems had statistically significant correlations with bugs. While both systems are open-source, their development shares many characteristics in common with industrial systems such as the use of specifications, use cases, and change management systems. Therefore, it is reasonable to expect that our results would hold for industrial software of similar sizes. All the systems we studies are written in Java. To see if our results are not language-specific, we are planning to study systems written in other programming languages in the future.

Concerning the survey we conducted, there are also threats to external validity. The majority of the programmers were graduate students, so the participants are not necessarily representative of all developers. However, some of our participants were industrial programmers, and in general, their responses aligned with those of the graduate students.

5 Conclusions

We have introduced novel metrics that capture feature-level coupling by using structural and textual information, filling a critical gap in the area of empirical software measurement. We have theoretically validated our metrics and extended the unified framework for coupling measurement (Briand et al. 1999) with important new dimensions. Through our three-pronged evaluation, we have shown that these metrics are useful since they are good predictors of fault-proneness. Additionally, they have an application in feature-level impact analysis to determine if a change made to one feature may have undesirable effects on other features. Finally, based on the results of a survey of 31 developers asked to rate the strength of coupling between pairs of features, our metrics align with those ratings. Altogether, these results point to a solid conclusion that structural and textual feature coupling metrics are valid and useful tools for developers performing feature-level software maintenance tasks.

A secondary goal of this work was to discover the optimal way in which to obtain our metrics so developers can use them most efficiently. Both *TFC* and *HFC* can be computed under different configurations. Textual information can be mined based on several options (*i.e.*, include comments, perform stemming). When available, external documentation should be included in the corpus to boost textual similarities by adding more domain terminology and concepts. In the absence of external documentation, comments should be preserved. When combining structural and textual information for *HFC*, more weight should be placed on the stronger of the two sources to be able to better predict faults or perform impact analysis tasks. Although accuracy of our metrics for the task of impact analysis exceeds results, which appear in the literature, we still must improve our technique in order to reach a level of accuracy which will allow our metrics to be used in practice.

We make all of the source code, data, and results of the case studies available and invite other researchers to replicate our work.

Acknowledgements

We are grateful to the anonymous reviewers for their relevant and useful comments and suggestions, which helped us in significantly improving the initial version of this paper. We thank Trevor Savage for developing FLAT^3 tool, which incorporates source code from the open source projects ConcernMapper and ConcernTagger. FLAT^3 also uses visualization ideas from AspectBrowser. We also acknowledge Maksym Petrenko for his help with JRipples, Bogdan Dit for collecting execution traces for *Rhino*, and thank many academic researchers and industrial practitioners who responded to our survey. This work was supported in part by NSF CCF–1016868, NSF CCF–0916260, and AFOSR FA9550-07-1-0030 grants and the start-up package provided by the College of William and Mary in Virginia. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

Abreu F., Pereira G. and Sousa P. (2000) A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems. Conference on Software Maintenance and Reengineering (CSMR'00), Zurich, Switzerland, IEEE Computer Society, 13-22.

Aho A. V. and Griffeth N. D. (1995) Feature interactions in the global information infrastructure. 3rd ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT'95), Washington, D.C., United States, 2-4.

Aranda J. and Venolia G. (2009) The secret life of bugs: Going past the errors and omissions in software repositories. 31st IEEE/ACM International Conference on Software Engineering (ICSE'09), Vancouver, British Columbia, Canada, 298-308.

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

Bansiya J. and Davis C. G. (2002) A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering **28**(1): 4-17.

Basili V. R., Briand L. C. and Melo W. L. (1996) A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering **22**(10): 751-761.

Bird C., Bachmann A., Aune E., Duffy J., Bernstein A., Filkov V. and Devanbu P. (2009) Fair and Balanced? Bias in Bug-Fix Datasets. 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09), 121-130.

Briand L., Melo W. and Wust J. (2002) Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. IEEE Transactions on Software Engineering **28**(7): 706-720.

Briand L., Wust J. and Louinis H. (1999) Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. IEEE International Conference on Software Maintenance (ICSM'99), IEEE Computer Society Press, 475-482.

Briand L. C., Daly J. and Wüst J. (1999) A Unified Framework for Coupling Measurement in Object Oriented Systems. IEEE Transactions on Software Engineering **25**(1): 91-121.

Briand L. C., Devanbu P. and Melo W. L. (1997) An investigation into coupling measures for C++. International Conference on Software engineering (ICSE'97), Boston, MA, ACM Press, 412 - 421.

Briand L. C., Morasca S. and Basili V. R. (1996) Property-Based Software Engineering Measurements. IEEE Transactions on Software Engineering **22**(1): 68-85.

Briand L. C., Wüst J., Daly J. W. and Porter V. D. (2000) Exploring the relationship between design measures and software quality in object-oriented systems. Journal of System and Software **51**(3): 245-273.

Buckner J., Buchta J., Petrenko M. and Rajlich V. (2005) JRipples: A Tool for Program Comprehension during Incremental Change. 13th IEEE International Workshop on Program Comprehension (IWPC'05), 149-152.

Chidamber S., Darcy D. and Kemerer C. (1998) Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. IEEE Transactions on Software Engineering **24**(8): 629-639.

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

Churcher N. and Shepperd M. (1995) Towards a conceptual framework for object oriented software metrics. SIGSOFT Softw. Eng. Notes **20**(2): 69-75.

Cleary B., Exton C., Buckley J. and English M. (2009) An empirical analysis of information retrieval based concept location techniques in software comprehension. Empirical Software Engineering **14**(1): 93-130.

Cohen J. (1988) Statistical Power Analysis for the Behavioral Sciences (2nd Edition), Routledge Academic.

Darcy D. and Kemerer C. (2005) OO Metrics in Practice. IEEE Software **22**(6): 17-19.

De Lucia A., Fasano F., Oliveto R. and Tortora G. (2007) Recovering Traceability Links in Software Artefact Management Systems. ACM Transactions on Software Engineering and Methodology **16**(4).

Deerwester S., Dumais S. T., Furnas G. W., Landauer T. K. and Harshman R. (1990) Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science **41**: 391-407.

Eaddy M. (2008). An Empirical Assessment of the Crosscutting Concern Problem, Columbia University. **PhD:** 196.

Eaddy M., Aho A. V., Antoniol G. and Guéhéneuc Y. G. (2008) CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. 17th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands, 53-62.

Eaddy M., Zimmermann T., Sherwood K., Garg V., Murphy G., Nagappan N. and Aho A. V. (2008) Do Crosscutting Concerns Cause Defects? IEEE Transaction on Software Engineering **34**(4): 497-515.

Eder J., Kappel G. and Schreft M. (1994). Coupling and Cohesion in Object-Oriented Systems, University of Klagenfurt.

Eisenbarth T., Koschke R. and Simon D. (2003) Locating Features in Source Code. IEEE Transactions on Software Engineering **29**(3): 210 - 224.

Fluri B., Würsch M., Giger E. and Gall H. (2009) Analyzing the co-evolution of comments and source code. Software Quality Journal **17**(4): 367-394.

Gall H., Jazayeri, M., Krajewski, J. (2003) CVS Release History Data for Detecting Logical Couplings. Sixth International Workshop on Principles of Software Evolution (IWPSE'03): 13 - 23.

Geiger R., Fluri B., Gall H. and Pinzger M. (2006) Relation of Code Clones and Change Couplings. 9th International Conference of Fundamental Approaches to Software Engineering (FASE'06).

Geipel M. M. and Schweitzer F. (2009) Software Change Dynamics: Which Dependencies do matter? Empirical Evidence from 35 Java Projects. 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09), 269-272.

Gethers M. and Poshyvanyk D. (2010) Using Relational Topic Models to Capture Coupling among Classes in Object-Oriented Software Systems. 26th IEEE International Conference on Software Maintenance (ICSM'10), Timişoara, Romania.

Giroux O. and Robillard M. P. (2006) Detecting increases in feature coupling using regression tests. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Portland, Oregon, USA, 163-174.

Greevy O. and Ducasse S. (2005). Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach. Proceedings of the Ninth European Conference on Software Maintenance and Reengineering**:** 314-323.

Greevy O., Ducasse S. and Girba T. (2006) Analyzing software evolution through feature views. Journal of Software Maintenance and Evolution: Research and Practice **18**(6): 425 - 456.

Griffeth N. D. and Lin Y.-J. (1993) Extending Telecommunications Systems: The Feature-Interaction Problem. Computer **26**(8): 14-18.

Griss M. L. (2000) Implementing product-line features by composing aspects. 1st Conference on Software product lines : experience and research directions: experience and research directions, Denver, Colorado, USA, 271-288.

Gyimóthy T., Ferenc R. and Siket I. (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering **31**(10): 897-910.

Hill E., Pollock L. and Vijay-Shanker K. (2007) Exploring the Neighborhood with Dora to Expedite Software Maintenance. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), 14-23.

Hitz M. and Montazeri B. (1995) Measuring Coupling and Cohesion in Object-Oriented Systems. International Symposium on Applied Corporate Computing, Monterrey, Mexico.

Jin Z. and Offutt A. J. (1996) Coupling-based Integration Testing. IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96), Montreal, Quebec, Canada, 10-17.

Kagdi H., Gethers M., Poshyvanyk D. and Collard M. (2010) Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. 17th IEEE Working Conference on Reverse Engineering (WCRE'10), Boston, USA, 119-128.

Kan S. H. (2003) Metrics and Models in Software Quality Engineering. Boston, Addison-Wesley.

Kang K. C., Lee J. and Donohoe P. (2002) Feature-Oriented Product Line Engineering. IEEE Software **19**(4): 58-65.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Videira Lopes C., Loingtier J.-M. and Irwin J. (1997) Aspect-Oriented Programming. European Conference on Object-Oriented Programming, Jyvaskyla, Finland, Springer-Verlag, 220-242.

Koschke R. and Quante J. (2005) On dynamic feature location. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), Long Beach, CA, USA, 86-95.

Kothari J., Bespalov D., Mancoridis S. and Shokoufandeh A. (2008) On evaluating the efficiency of software feature development using algebraic manifolds. IEEE International Conference on Software Maintenance (ICSM'08), 7-16.

Kothari J., Denton T., Mancoridis S. and Shokoufandeh A. (2006). On Computing the Canonical Features of Software Systems. 13th IEEE Working Conference on Reverse Engineering (WCRE'06). Benevento, Italy.

Kothari J., Denton T., Mancoridis S. and Shokoufandeh A. (2007) Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence. IEEE International Conference on Program Comprehension (ICPC), Banff, Canada.

Kramer S. and Kaindl H. (2004) Coupling and cohesion metrics for knowledge-based systems using frames and rules. ACM Transactions on Software Engineering and Methodology **13**(3): 332-358.

Lee Y. S., Liang B. S., Wu S. F. and Wang F. J. (1995) Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow. International Conference on Software Quality, Maribor, Slovenia.

Li W. and Henry S. (1993) Object-oriented metrics that predict maintainability. Journal of Systems and Software **23**(2): 111-122.

Lienhard A., Greevy O. and Nierstrasz O. (2007) Tracking Objects to Detect Feature Dependencies. 15th IEEE International Conference on Program Comprehension (ICPC '07), Banff, Canada, 59-68.

Liu D., Marcus A., Poshyvanyk D. and Rajlich V. (2007) Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), Atlanta, Georgia, 234-243.

Lormans, M., van Deursen, A., and Grob, H, G.. (2008) An industrial case study in reconstructing requirements views. Empirical Softw. Eng. **13**(6): 727-760.

Marcus A. and Maletic J. I. (2001) Identification of High-Level Concept Clones in Source Code. Automated Software Engineering (ASE'01), San Diego, CA, 107-114.

Marcus A. and Poshyvanyk D. (2005) The Conceptual Cohesion of Classes. 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 133-142.

Marcus A., Poshyvanyk D. and Ferenc R. (2008) Using the Conceptual Cohesion of Classes for Fault Prediction in Object Oriented Systems. IEEE Transactions on Software Engineering **34**(2): 287-300.

Marcus A., Sergeyev A., Rajlich V. and Maletic J. (2004) An Information Retrieval Approach to Concept Location in Source Code. 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, 214-223.

Marin M., Moonen L. and van Deursen A. (2007) Documenting Typical Crosscutting Concerns. Working Conference on Reverse Engineering, 31-40.

Martin R. (1994) OO Design Quality Metrics-An Analysis of Dependencies. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94.

McGraw K. O. and Wong S. P. (1996) Forming inferences about some intraclass correlation coefficients. Psychological Methods **1**(1): 30-46.

Mende T., Koschke R. and Beckwermert F. (2009) An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. Journal of Software Maintenance and Evolution: Research and Practice **21**(2): 143-169.

Meyers T. M. and Binkley D. (2007) An Empirical Study of Slice-based Cohesion and Coupling Metrics. ACM Transactions on Software Engineering and Methodology (TOSEM) **17**(1).

Olague H., Etzkorn L., Gholston S. and Quattlebaum S. (2007) Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. IEEE Transactions on Software Engineering **33**(6): 402-419.

Orso A., Apiwattanapong T., Law J., Rothermel G. and Harrold M. J. (2004) An empirical comparison of dynamic impact analysis algorithms. IEEE/ACM International Conference on Software Engineering (ICSE'04), 776-786.

Petrenko M. and Rajlich V. (2009) Variable Granularity for Improving Precision of Impact Analysis. International Conference on Program Comprehension (ICPC'09), 10-19.

Porter M. (1980) An Algorithm for Suffix Stripping. Program **14**(3): 130-137.

Poshyvanyk D., Guéhéneuc Y. G., Marcus A., Antoniol G. and Rajlich V. (2007) Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval. IEEE Transactions on Software Engineering **33**(6): 420-432.

Poshyvanyk D. and Marcus A. (2006) The Conceptual Coupling Metrics for Object-Oriented Systems. 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, PA, 469 - 478.

Poshyvanyk D., Marcus A., Ferenc R. and Gyimóthy T. (2009) Using Information Retrieval based Coupling Measures for Impact Analysis. Empirical Software Engineering **14**(1): 5-32.

Poshyvanyk D. and Marcus D. (2007) Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, 37-48.

Ratiu D. and Deissenboeck F. (2006) How Programs Represent Reality (and How They Don't). 13th Working Conference on Reverse Engineering (WCRE'06), 83 - 92.

Ratiu D. and Deissenboeck F. (2007) From Reality to Programs and (Not Quite) Back Again. 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Canada, 91-102.

Ren X., Shah F., Tip F., Ryder B. G. and Chesley O. (2004) Chianti: a Tool for Change Impact Analysis of Java Programs. 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04), Vancouver, BC, Canada, 432-448.

Revelle M., Broadbent T. and Coppit D. (2005) Understanding Concerns in Software: Insights Gained from Two Case Studies. 13th IEEE International Workshop on Program Comprehension (IWPC'05), St. Louis, Missouri, 23-32.

Revelle M., Dit B. and Poshyvanyk D. (2010) Using Data Fusion and Web Mining to Support Feature Location in Software. 18th IEEE International Conference on Program Comprehension (ICPC'10), Braga, Portugal, 14-23.

Revelle M. and Poshyvanyk D. (2009) An Exploratory Study on Assessing Feature Location Techniques. 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, British Columbia, Canada, 218-222.

Robillard M. P. and Murphy G. C. (2007) Representing concerns in source code. ACM Transactions on Software Engineering and Methodology (TOSEM) **16**(1).

Robillard M. P., Shepherd D., Hill E., Vijay-Shanker K. and Pollock L. (2007). An Empirical Study of the Concept Assignment Problem, McGill University.

Salah M., Mancoridis S., Antoniol G. and Di Penta M. (2005) Towards Employing Use-Cases and Dynamic Analysis to Comprehend Mozilla. 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 639-642.

Salton G. and McGill M. (1983) Introduction to Modern Information Retrieval, McGraw-Hill.

Savage T., Revelle M. and Poshyvanyk D. (2010) FLAT^3: Feature Location and Textual Tracing Tool. 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, 255-258.

Sherriff M. and Williams L. (2008) Empirical Software Change Impact Analysis using Singular Value Decomposition. International Conference on Software Testing, Verification, and Validation (ICST'08), 268-277.

Sirkin R. M. (2005) Statistics for the Social Sciences. CA, Sage Publications.

Spearman C. (1904) The proof and measurement of association between two things. The American Journal of Psychology **15**(1): 72-101.

Subramanyam R. and Krishnan M. S. (2003) Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering **29**(4): 297-310.

Tairas R. and Gray J. (2009) An Information Retrieval Process to Aid in the Analysis of Code Clones. Empirical Software Engineering **14**(1): 33-56.

Tarr P., Ossher H., Harrison W. and Sutton Jr. S. M. (1999). N degrees of separation: multi-dimensional separation of concerns. Proceedings of the 21st international conference on Software engineering. Los Angeles, California, United States, ACM**:** 107-119.

Újházi B., Ferenc R., Poshyvanyk D. and Gyimóthy T. (2010) New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems. 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10), Timişoara, Romania, 33-42.

Wilde N. and Scully M. (1995) Software Reconnaissance: Mapping Program Features to Code. Software Maintenance: Research and Practice **7**: 49-62.

Wilkie F. G. and Kitchenham B. A. (2000) Coupling measures and change ripples in C++ application software. The Journal of Systems and Software **52**: 157-164.

Wong W. E. and Gokhale S. (2005) Static and dynamic distance metrics for feature-based code analysis. Journal of Systems and Software **74**(3): 283-295.

Ye Y. and Fischer G. (2005) Reuse-Conducive Development Environments. Journal of Automated Software Engineering **12**(2): 199-235.

Yin R. K. (2003) Applications of Case Study Research. CA, USA, Sage Publications, Inc.

Yu Z. and Rajlich V. (2001) Hidden Dependencies in Program Comprehension and Change Propagation. 9th IEEE International Workshop on Program Comprehension (IWPC'01), Toronto, Canada, 293-299.

Zaidman A., Du Bois B. and Demeyer S. (2006). How Webmining and Coupling Metrics Improve Early Program Comprehension. Proceedings of the 14th IEEE International Conference on Program Comprehension**:** 74-78.

Zave P. (1993) Feature Interactions and Formal Specifications in Telecommunications. Computer **26**(8): 20-29.

Zhao J. (2004) Measuring Coupling in Aspect-Oriented Systems. 10th IEEE International Software Metrics Symposium (METRICS'04), Chicago, USA.

Zimmermann T., Zeller A., Weissgerber P. and Diehl S. (2005) Mining Version Histories to Guide Software Changes. IEEE Trans. on Software Engineering **31**(6): 429-445.

Zou L., Godfrey M. W. and Hassan A. E. (2007) Detecting Interaction Coupling from Task Interaction Histories. 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, 135-144