

Feature Interaction Networks

Stefan Sobernig
Institute for Information Systems and New Media
Vienna University of Economics and Business (WU Vienna)
Augasse 2-6
A-1090 Vienna, Austria
stefan.sobernig@wu.ac.at

ABSTRACT

A quantitative approach for measuring and describing feature interactions in object-oriented software components based on source code inspection is presented. The methodical arsenal is borrowed from the field of network analysis. Based on data gathering through source code harvesting, network representations of feature implementations (i.e., feature interaction networks, FINs) are constructed. By applying established network statistics, various properties of interaction structures between features can be captured, e.g. the scatteredness, the degree of crosscutting (scattering), and the scattering concentration. This approach contrasts with related proposals based on frequency and dispersion measures.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics—*Complexity Measures*; D.1.5 [Programming Techniques]: Object-oriented Programming

General Terms

Feature interaction, network analysis, abstraction mismatch, feature-oriented software development, feature location

1. INTRODUCTION

A *feature* is considered a concrete and functional property characteristic for a family of object-oriented (OO) software systems, subsystems, or components, which is relevant for a particular, homogeneous group of stakeholders [6]. In an object-oriented GUI application framework, for instance, features could be widget kinds, widget lifecycle management, portability, and platform integration. Analysing and designing software in terms of features requires us to understand how these features are turned into object-oriented designs and implementations. Whilst more object-oriented programming environments become available, capable of expressing different types of features as first abstractions (e.g., mixin

classes [13]), selecting the appropriate abstraction is challenging. Adopting a specific language abstraction bears the risk of turning into an abstraction mismatch (e.g., classification, composition, and locality mismatches). As a result, feature implementations exert negative influence on functional (e.g., coupling, reconfigurability, need for invasive adaptations) and non-functional properties (e.g., testability, modular comprehensibility) of others. The results are unwanted feature interactions, for instance, the *optional feature problem* [12]. From a legacy and maintenance perspective, documentation on such implementation decisions is often missing, incomplete, or outdated. Hence, to understand feature implementations and the feature interactions they cause, we mine for feature implementations in the source code, using techniques of *feature location* [9]. At the source code level, these feature interactions appear as a patching code, that is, code replication [3], code interlacing (i.e., tangling, scattering), and code introductions [1]. A deficient feature implementation due to patching code is called *static crosscut*.

Techniques and tooling for feature location provide statistical measure instruments to characterise feature interactions in feature location data [18, 16, 15, 8, 7]. In [18], an analytical suite of three measure instruments is presented to indicate the *closeness* between functional, but higher-level concerns and source code artefacts. This includes the indicator measures for *disparity* between concerns and code units, *dedication* of a code unit to a given concern (concern cohesion), and the *concentration* of a concern in a given set of code units (concern coupling). In [8, 7], two refinements of the closeness measures [18] are presented. On the one hand, the authors suggest variance-based aggregates of concentration for a given concern as the *degree of scattering* (DoS). Similarly, the variance of dedications for all program components is discussed as the *degree of focus* (DoF; or its inversion, the *degree of tangling* DoT). The DoS is defined as the straightforward bias-corrected sample variance of the contributions (expressed in SLoC) by all code units to a given concern. The degree of focus (DoF) is calculated as the bias-corrected sample variance of the dedicated contributions (in SLoC) of a code unit over a given set of concerns. *Concern diffusion measures* [16] perform counts of code units required to implement a concern. Concern diffusion measures reflect the number of operations (i.e., the *concern diffusion over operations* CDO) and components (i.e., the *concern diffusion over components* CDC) required to implement a given concern. Comparatively higher (lower) concern diffusion counts are read as indicators for a low (high) cohesion.

These existing measure instruments exhibit a major short-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

Copyright 2010 ACM 978-1-60558-638-0/10/03 ...\$10.00.

coming. They only describe conditions of single feature implementations, e.g., the implementation of feature **X** is distributed over m disparate code units. I say that their emphasis is on the *absolute attributes* of feature implementations, such as the number of code units, source lines of code (SLoC), etc. This construction of the measure instruments does not reflect interdependency relations between feature implementations, that is, the feature interactions as such. With this, expressing, for instance, that feature **X** interacts with feature **Y** and **Z** due to their interleaving implementations is not possible.

The contribution of this paper is the outline of a network-statistical approach [4, 14] which allows to express feature interactions caused by OO abstraction mismatches directly by a set of three network measures. While graph- or network-centric indicators on feature location data have been explored (see, e.g., [15]), these proposals show either limitations (e.g., due to a dyadic analysis of interactions) or have not been applied for evaluating OO abstraction mismatches. Following a motivating example in Section 2, the reader is presented the construction and the analysis of feature interaction networks (FINs) in Section 3. In Section 4, this presentation is completed by characterising FINs by three network measures. I conclude by reflecting briefly on applying FINs in Section 5.

For the scope of this paper, *code units* are processable language constructs. Processability is guaranteed by providing in-memory representations (e.g., through syntax parsers) and language-level introspection. We consider three different levels of UML structural model elements, i.e., components, classes, and operations, with each capable of representing different language constructs. Language-specific examples and listings refer to the object-oriented scripting language XOTcl [13].

2. A MOTIVATING EXAMPLE

Consider a family of stack-like dispenser components. The features and the possible feature compositions are presented as feature diagram [6] in Figure 1. Apart from the mandatory features **push**, **pop**, **top**, and **empty**, there are three optional ones: **counter**, **logging**, and **sync**.

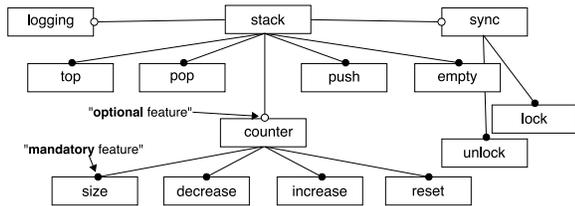


Figure 1: A feature diagram representing a stack family [6]

To track and introspect on the size of a stack, the optional feature **counter** is described by four mandatory sub-features: **increase** and **decrease** record **pop** and **push** operations, **reset** realises the invalidation of the internal counter. The **size** sub-feature denotes a storage for keeping the transformation count and for requesting the transformation count at a given time. To log core stack operations, the second optional feature **logging** handles adding trace statements as well as configuring output formats, adjusting logging levels, and choosing output channels for stack components. Under concurrency, basic floor control mechanisms for mutating

operations (i.e., **pop** and **push**) are required. The optional **sync** feature achieves this by providing means to **lock** and **unlock** access to mutator operations.

This stack family is, therefore, described by seven different stack compositions. Apart from bare stack components, there can be counter-, logging-, or synchronisation-enabled stack dispensers; and stacks having any inclusive-or combination of the three optional features.

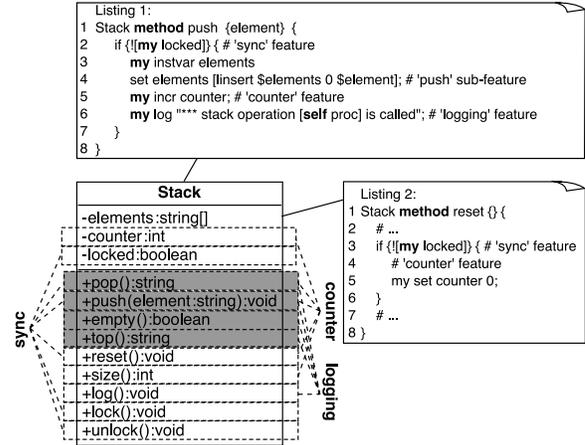


Figure 2: A first, tangled stack implementation

A first implementation is centred around a single object-class **Stack** (see Figure 2). The **stack** feature and its core sub-features map directly to this object-class and its operations. They operate on an internal vector representation of the stack. A close inspection of this implementation artefact, however, shows that important composition rules are violated. This is because the implementation of the optional features **counter**, **logging**, and **sync** appear interwoven among each other *and* with the basic **stack** feature.

Firstly, the signature interface and the implementation of the **Stack** object-class contains operations (e.g., **log()** for the **logging** and **size()** for the **counter** feature) as well as attributes specific to all three optional features (e.g., **@locked** for the **sync** and **@counter** for the **counter** feature). Secondly, we find problematic code interleaving in the core stack operations. The **push()** operation shows that the **@counter** attribute is effectively increased after pushing an element onto the stack (see Listing 1 in Figure 2, line 5). Also, a logging-specific call is declared (see the Listing 1 in Figure 2, line 6). Finally, the **sync** implementation is not only interfering in the implementation of the **push()** operation (see Listing 1 in Figure 2, line 2), but also in **reset()** as part of the **counter** feature implementation (see Listing 2 in Figure 2, line 3). It turns out that the implementations of the three optional features and the **stack** feature are closely tied together. They appear de facto mandatory. In other words, we cannot use this implementation variant to create stacks according to all seven feature compositions specified. In this, we encounter an example of the optional feature problem.

3. A RELATIONAL VIEW OF FEATURE INTERACTIONS

In the stack implementation shown in Figure 2, measure instruments over absolute attributes [18, 16, 8, 7] allow to

describe conditions of each feature implementation (e.g., low cohesion due to a high number of disparate operations). Yet, we are not capable of turning these findings into a statement on the unwanted relations between the `stack` and its three optional features. This is because features are treated as nonstructural, relationally independent units of analysis. While this is not necessarily by intention, it is yet inherent in the construction of measure instruments proposed (e.g., concern diffusion, degrees of scattering and tangling; see the introductory section). This paper makes a different proposal. Structures of feature interactions are considered as a network of features, formally represented by a graph [4, 14]. This approach permits us to capture feature interactions as a structural phenomenon, based on an exact mathematical representation of features and their interactions.

Feature Interactions as Co-Link Relations. Locating a feature [9] refers to annotating code units according to their participation in implementing a single or several features. This is either achieved by manual inspection (as in our motivating example) or by appropriate tool support (see, e.g., [11]). Once a feature is located in a set of source code units, this finding is preserved as a set of links. A *link* is defined as a pair type (*CodeUnit, Feature*). Code units are said to be assigned to features. We capture interactions between features based on common links connecting code units and features, so-called *co-linkages*. The co-linkage of features denotes that two (or more) features have been linked to one and the same code unit at a given level of code unit granularity (e.g., per-component, per-class, or per-operation).

Let us return to the stack implementation. We observed the occurrence of a code tangling between the feature implementations of the `stack` and its optional features `counter`, `logging`, and `sync`. Consider the interaction stated for the `stack` and `counter` features. Their interaction is due to the code tangling found in the `pop()` and `push()` operations of the `Stack` object-class as well as code introductions, e.g., the `reset()` and `size()` operations. As for operations, the tangling translates into two links for each operation to the `stack` and `counter` feature, respectively. As a consequence, we can establish a co-linkage relation between the two features. This is justified because of the two features sharing links to `pop()` and `push()`.

Feature Interaction Networks. A feature interaction network (FIN) is formally represented by a graph, having the following characteristics: A FIN-graph G^{FIN} is an abstract representation of a FIN that comprises a non-empty set of vertices and a possibly empty set of edges. Vertices represent features while edges represent interactions between these features. Consequently, FINs are one-mode networks containing a single kind of entity, i.e., features. The FIN-graph is based solely on undirected edges. Feature interactions *by co-linkage* are considered *symmetric* (i.e., non-directional) relations. Recall that co-linkage is stated when two links emanating from the same code unit point to two distinct features (or vice versa). A so-established relation does not permit us to introduce a direction of the relation without further qualifying either features or code units.

Link data yielded by locating features in a base of source code units form a *feature assignment matrix* (FAM; see also Figure 3). The rows of this feature assignment matrix represent the kind of code unit considered; i.e., UML components, classes, or operations. The matrix columns represent features. Feature assignment matrices are not directly suit-

able to represent a FIN. Firstly, since there are two kinds of entities (code units and features), a feature assignment matrix would describe a two-mode network. Secondly, a feature assignment matrix is non-dyadic because links between code units and features (or vice versa) relate sub-sets of arbitrary size.

To compute the relations between features based on overlapping links (co-links) to code units, we apply a straightforward transformation to the FAM to obtain the needed co-linkage (or feature-feature) matrix (CLM). Figure 3 shows that the *co-linkage matrix* (CLM) is yielded from the post-multiplication of the feature assignment matrix (FAM) with its transposition (FAM'): $CLM = FAM' * FAM$. The co-linkage matrix has the required properties to represent a feature interaction network (FIN), i.e., it is a symmetric matrix representing a one-mode network. The matrix elements along the main diagonal of the co-linkage matrix report the total number of links established for a feature (i.e., representing the concern diffusion measures in [16]). The matrix elements either in the upper or lower matrix segment represent the number of co-links between pairs of two features (i.e., dyads). Thus, non-zero values denote the existence of a feature interaction *by co-linkage*. The resulting FAM and the CLM for our stack example are shown in the Tables 1 and 2, respectively.

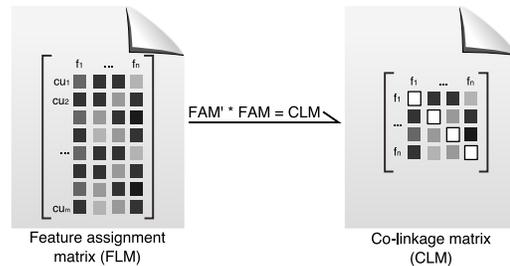


Figure 3: Matrix transformation

4. DESCRIBING FEATURE INTERACTION NETWORKS

The transformation of feature location data (i.e., links) into a feature interaction network (FIN) permits us to use certain network measures for describing characteristics of the identified feature interactions. In this paper, I propose three measures for FINs: the density-based **scatteredness**, the degree-based **scattering**, and the degree-based **scattering concentration**. Among others, they allow to address questions such as: (1) How deficient is the implementation of a set of features? (2) How interdependent are features due to their implementation? (3) Are there features whose implementations cause a predominant share in total feature interactions occurring? (4) Are there features whose implementations are self-contained?

To illustrate the interpretative capacity of the proposed network measure instruments, we establish an alternative implementation variant for the stack example (see Section 2). This variant assumes that there are advanced, object-oriented abstractions available for expressing the feature variability specified. We consider the use of mixins to realise the optional features `counter`, `logging`, and `sync` as free-standing extensions. XOTcl mixin classes [13] realise this

	stack	counter	logging	sync
Stack.pop	1	1	1	1
Stack.push	1	1	1	1
Stack.empty	1	0	1	0
Stack.top	1	0	1	0
Stack.@elements	1	0	0	0
Stack.reset	0	1	0	1
Stack.size	0	1	0	0
Stack.@counter	0	1	0	0
Stack.@locked	0	0	0	1
Stack.lock	0	0	0	1
Stack.unlock	0	0	0	1
Stack.log	0	0	1	0

Table 1: A feature assignment matrix (FAM)

	stack	counter	logging	sync
stack	5	2	4	2
counter	2	5	2	3
logging	4	2	5	2
sync	2	3	2	6

Table 2: A co-linkage matrix (CLM)

concept, to name a concrete language example. The UML

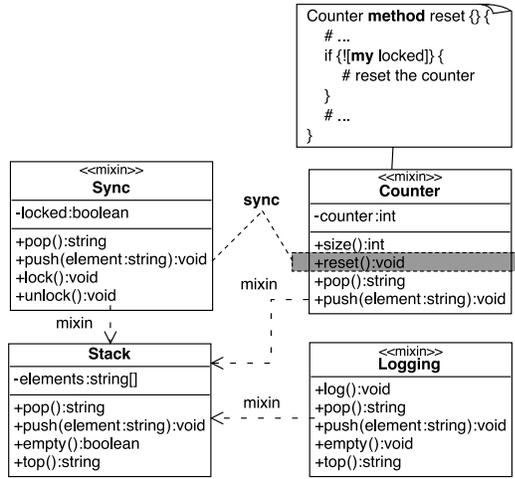


Figure 4: Stack implementation using mixins

class model in Figure 4 sketches out this implementation variant. The implementation consists of three mixin classes **Logging**, **Counter**, and **Sync**. Each of the mixin classes owns the feature-specific operations and describes required operations (e.g., `pop()` and `push()`). These required operations wrap the corresponding **Stack** operations and add the feature-specific behaviour. Note that the `counter` and `sync` implementations are still intermingled (due to the way `reset()` is implemented). We expect the FIN analysis to confirm that (1) the `logging` and `stack` feature implementations do not cause any interactions with other features and that (2) `counter` and `sync` features are still tied.

Scatteredness. We define the scatteredness in a given feature set as the *density* $\rho(G^{FIN})$ [4] of the feature interaction network (FIN). The density is computed as the share of identified feature interactions in the number of maximum possible feature interactions. Density describes the degree of completeness of a FIN. A density close to one or equating one (i.e., in a perfectly complete FIN) indicates that there are feature interactions between the majority or all features. Inversely, a density close to zero or equating zero (i.e., in a perfectly incomplete FIN) denotes that the implemented features show few or no interactions at all.

In the stack example, scatteredness reflects the decreasing number of feature interactions due to the mixin-based refactoring. The FIN for the initial, tangled implementation variant (see also Figure 5) exhibits a density of one (i.e., six out of six possible interactions). The second FIN on the

mixin-based stack implementation has only one out of six possible interactions realised, i.e., a density of 0.17 (see Figure 6).

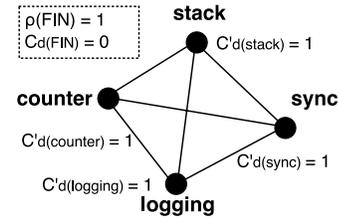


Figure 5: Feature interaction network before refactoring

Scattering. The density-based scatteredness provides an indication of the overall state of overlap between feature implementations. The question arises whether particular feature implementations contribute over-proportionally to this state of overlap, i.e., whether they qualify as *central* crosscuts. We propose the *degree centrality* $C'_d(feature)$ [4] of features as the indicator. We turn a feature's degree into a centrality index by normalising it based on the network size, i.e., the total number of features considered in the FIN. A degree centrality of zero and one reflect perfectly non-scattered (zero) or perfectly scattered (one) feature implementations. Zero refers to a situation where a single feature and its underlying implementation are organised in a self-contained and non-overlapping manner. In contrast, a centrality of one describes the situation that a given feature is tied up to any other feature in the FIN.

The FIN resulting from the first implementation scenario yields a degree centrality of 1 for all four features (see Figure 5). There is no single central crosscutting feature. All features participate in the maximum possible feature interactions. Reorganising the optional feature implementations into mixin classes introduces features with a degree centrality of zero, i.e., `stack` and `logging` features (see Figure 6).

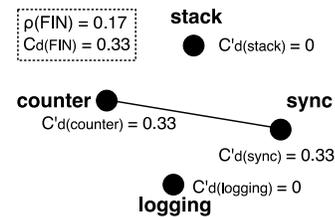


Figure 6: Feature interaction network after refactoring

Scattering concentration. Degree centrality is a local, feature-specific indicator. However, degree centrality cannot indicate the importance of a feature implementation for the total implementation overlap, i.e., the scatteredness. We devise the *degree centralisation* $C_{d(G^{FIN})}$ [4, 5] of a FIN as a global centrality measure to indicate the *scattering concentration* of the entire FIN. It is an index of the degree distribution in a FIN. The degree distribution score is the sum of residua between each feature’s degree and the maximum feature degree found within the FIN [10]. Through normalisation, scattering concentration is restricted to the interval from zero (for a complete FIN) to one. A scattering concentration $C_{d(G^{FIN})} \simeq 0$ describes FINs whose scatteredness is *not* attributable to any particular crosscutting feature. Conversely, a scattering concentration $C_{d(G^{FIN})} \simeq 1$ points to a centralised configuration of the FIN.

When comparing the two implementation variants, we discover that there are low levels of scattering concentration reported (i.e., 0 and 0.33; see Figures 5 and 6). These findings second the scattering measurement. There is no particularly central feature responsible for the implementation overlap alone.

5. CONCLUDING REMARKS

By applying first principles from network analysis, we generate useful insights on the notion of feature interaction. Precise rules for forming feature interaction networks (FINs) are presented: Feature interactions derive from stating *co-links* between features and shared code units (e.g. components, classes, or operations). In addition, applying network-specific measure instruments such as density, degree centrality, and degree concentration permits us to characterise crosscutting feature implementations in terms of *scatteredness*, *scattering*, and *scattering concentration*. Also, the process of constructing feature interaction networks produces as by-products related measure instruments, in particular concern diffusion metrics [16].

For performing steps of feature location and network analysis, I employed an auxiliary toolkit. For annotating source code bases with features, I adopted the existing concern location tool ConcernTagger [11]. The actual further-processing into feature interaction networks (FINs) and their evaluation are performed by the statistical computing environment GNU/R [17]. GNU/R provides a versatile set of extension packages for conducting network analyses. This toolkit was successfully employed in larger-scale case studies on object-oriented middleware frameworks (> 200,000 SLoC each). This allowed for characterising crosscutting features in these framework designs (e.g., support for different invocation styles).

As for follow-up work, I plan to explore the use of *weighted* feature interaction networks (FINs) and corresponding measure instruments (e.g., *strength* centrality [2]). Weightings incorporate absolute attributes of feature implementations (e.g., number of code unit links or SLoC scores) as indicators for relational intensity. Relational intensity further qualifies feature interactions, for instance, as being predominantly *dynamic* (e.g., caused by excessive conditional branching in the control flow).

6. REFERENCES

- [1] S. Apel, C. Lengauer, D. S. Batory, B. Möller, and C. Kästner. An Algebra for Feature-Oriented Software

Development. Technical Report MIP-0706, University of Passau, 2007.

- [2] A. Barrat, M. Barthelemy, R. Pastor-Satorras, and A. Vespignani. The Architecture of Complex Weighted Networks. *PNAS*, 101(11):3747–3752, March 2004.
- [3] L. Bergmans, B. Tekinerdogan, M. Glandrup, and M. Aksit. On Composing Separated Concerns, Composability and Composition Anomalies. In *Proceedings of the WASC’00*, 2000.
- [4] U. Brandes and T. Erlebach, editors. *Network Analysis: Methodological Foundations*, volume 3418 of *Lecture Notes in Computer Science*. Springer, 2005.
- [5] C. T. Butts. Exact Bounds for Degree Centralization. *Social Networks*, 28(4):283–296, 2006.
- [6] K. Czarnecki and U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley Longman Publishing Co., Inc., 6th edition, 2000.
- [7] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. *Proceedings of the ICPC 2008*, pages 53–62, 2008.
- [8] M. Eaddy, A. V. Aho, and G. C. Murphy. Identifying, Assigning, and Quantifying Crosscutting Concerns. In *Proceedings of the ACoM’07*, 2007.
- [9] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *TSE*, 29(3):210–224, 2003.
- [10] L. C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3):215–239, 1979.
- [11] V. Garg and M. Eaddy. ConcernTagger 2.0.1. <http://www1.cs.columbia.edu/~eaddy/concerntagger/>, last accessed: October 14, 2008.
- [12] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the SPLC’09*, 2009.
- [13] G. Neumann and U. Zdun. Enhancing Object-Based System Composition through Per-Object Mixins. In *Proceedings of the APSEC’99*, pages 522–529, 1999.
- [14] M. E. J. Newman. The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256, 2003.
- [15] M. P. Robillard and G. C. Murphy. Representing Concerns in Source Code. *TOSEM*, 16(1):3–38, 2007.
- [16] C. Sant’Anna, A. Gracia, C. Chavez, C. Lucena, and A. von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings of the BSSE’03*, 2003.
- [17] The R Foundation for Statistical Computing. GNU/R 2.8.1. <http://www.r-project.org/>, last accessed: December 13, 2008.
- [18] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the Closeness between Program Components and Features. *JSS*, 54(2):87–98, 2000.