On the Relation between Internal and External Feature Interactions in Feature-Oriented Product Lines

A Case Study

Sergiy Kolesnikov, Judith Roth, Sven Apel
University of Passau, Germany

ABSTRACT

The feature-interaction problem has been explored for many years. Still, we lack sufficient knowledge about the interplay of different kinds of interactions in software product lines. Exploring the relations between different kinds of feature interactions will allow us to learn more about the nature of interactions and their causes. This knowledge can then be applied for improving existing approaches for detecting, managing, and resolving feature interactions. We present a framework for studying relations between different kinds of interactions. Furthermore, we report and discuss the results of a preliminary study in which we examined correlations between internal feature interactions (quantified by a set of software measures) and external feature interactions (represented by product-line-specific type errors). We performed the evaluation on a set of 15 feature-oriented, JAVA-based product lines. We observed moderate correlations between the interactions under discussion. This gives us confidence that we can apply our approach to studying other types of external feature interactions (e.g., performance interactions).

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: ManagementSoftware Configuration Management; D.2.13 [**Software Engineering**]: Reusable SoftwareDomain Engineering

General Terms

Measurement, Experimentation, Reliability

Keywords

Feature Interactions, Software Measures, Feature-Oriented Software Development

1. INTRODUCTION

Feature modularity is the holy grail of feature-oriented software development [3]. Ideally, one can deduce the behavior of a system composed from a set of features solely on the basis of the behavior of the features involved. But, feature interactions are still a major challenge and counteract feature modularity and compositional reasoning [11]. A *feature interaction* occurs when the behavior of one feature is influenced by the presence or absence of another feature (or a set of other features). Often, the interaction cannot be deduced easily from the behaviors of the individual features involved, which hinders compositional reasoning. A classic example is the inadvertent interaction between the fire-alarm and flood-control features of an alarm-and-emergency system [18]. In the case of fire, if both features are activated the system reaches an unsafe state: when the fire alarm feature activates the sprinkler system, the flood control feature cuts off water to sprinklers.

In our previous work, we proposed a classification of feature interactions along their order and visibility [5]. The *order* of a feature interaction is defined as the minimal number of features (minus one) that need to be activated to trigger the interaction; for example, an interaction between two features is of order one. The *visibility* of a feature interaction denotes the context in which a feature interaction appears. First, feature interactions may appear at the level of the externally observable behavior of a program, including functional behavior (e.g., segmentation faults and all kinds of other bugs) and non-functional behavior (e.g., performance anomalies and memory leaks). Second, feature interactions may manifest themselves internally in a system, at the level of code that gives rise to an interaction, or at the level of control and data flow of a system (e.g., dataflows that occur only when two or more features are present).

In previous work, we proposed to systematically examine the relations between internal and external interactions with the ultimate goal of learning more about the nature, causes, and interdependencies of different kinds of interactions [5]. In this paper, we report on a preliminary study that aims at identifying possible correlations between externally-visible product-line specific type errors and internally-visible structural and operational attributes of a product line. The motivation for considering type errors is that, while making the system flexible and its parts reusable, variability introduces a new kind of type errors that arise from the interaction of features in a product line. A simple example is a dependency of a mandatory feature on some program elements (e.g., methods or types) that are introduced by an optional feature. This dependency is unsatisfied in products without the optional features and results in type errors (dangling references). According to our classification of feature interaction [5], such type errors are external feature interactions.

Although, there are methods to efficiently and reliably type check all products of a product line, we are interested in finding reasons for these type errors and not just detecting them. There may be systematic correlations between externally-visible and internally-visible interactions [5], which is a major motivation for our endeavor to explore and understand the nature of feature interactions. Using correlations between external and internal feature interac-

FOSD '14, September 14 2014, Västerås, Sweden

http://dx.doi.org/10.1145/2660190.2660191.

tions, we can improve existing methods for feature-interaction detection and prediction. For example, we could detect hard to find external feature interactions based on the information about internal interactions and their relation to external ones. We can also improve existing approaches for predicting non-functional attributes of product lines that may strongly depend on external feature interactions [27].

In our study, we use 11 measures that capture information about internal feature interactions that may cause type errors. We examined the correlations between these internal interactions, as quantified by these measures, and product-line-specific type errors (i.e., a particular kind of external feature interactions). Despite that we found only moderate correlations between these measures and the occurrences of type errors, the main contribution of this conceptual study is the description of a general framework for studying relations between external and internal features interactions. Using this framework, we plan to analyze the data about internal and external feature interactions that we have been collecting from different real-world systems. This will allow us to gain more insights into the nature of feature interactions and their interplay in real-world systems [10, 17, 19, 23, 25, 26, 28].

The contributions of this paper are:

- A discussion about relations between external and internal feature interactions, based on the concrete example of product-line-specific type errors.
- A set of measures that quantify variability, and operational and structural properties of feature-oriented product lines.
- Results of a studying correlations between our measures and external feature interactions (represented by product-line-specific type errors).
- A general framework for studying relations between different types of external and internal feature interactions, and a report of our experiences in applying this framework.

2. BACKGROUND

Figure 1 presents a very simple product line of list data structures, illustrating a *product-line specific type error.* The mandatory feature BASE implements a singly linked list. Based on this implementation, an optional feature BATCH implements a customized version of a list for batch jobs. Feature BATCH requires feature BASE because of the reference in Line 9. This requirement is satisfied in all valid products, because BASE is mandatory and it is present in every valid product. Another reference goes from BASE to BATCH (Line 5). This type reference to BatchList is a dangling reference in those products that do not contain the optional feature BATCH, leading to a type error in these products. Formally, if the presence condition BASE $\wedge$ $\neg$BATCH is satisfied by a product configuration, then we will get this product-line-specific type error.

2.1 Type Errors as Feature Interactions

We define a *feature interaction* as a violation of a specification under a certain feature presence condition [5]. The presence condition must have at least two features (i.e. a first-order interaction).

A type error violates a common implicit specification: There must be no type errors in the system. By definition, a product-line-specific type error always involves, at least, two features. Thus, the corresponding presence condition always contains, at least, two features, but possibly more. Consequently, a product-line-specific type error satisfies our definition of a feature interaction.

2.2 Visibility of Feature Interactions

Different levels of visibility of feature interactions have been discussed in the literature [6,8]. Feature interactions may appear at the

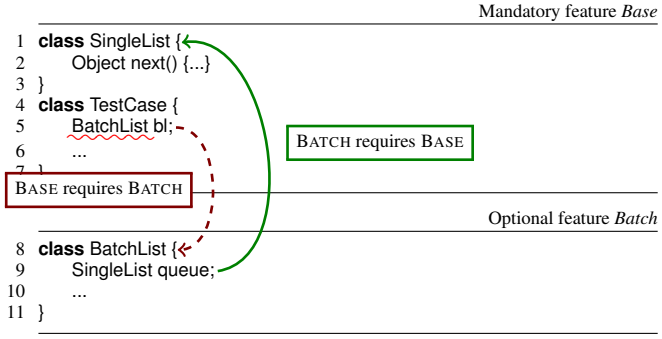

Figure 1: Example of a simple product line with a product-line-specific type error: a basic list implementations (in the mandatory feature BASE) and an extension for batch jobs (in the optional feature BATCH); the type error is underlined; arrows denote references; the dashed arrow denotes a possibly dangling reference.

level of the externally-visible behavior, which we call henceforth *external feature interactions,* for short, as well as at the level of the internal properties of a system, which we call henceforth *internal feature interactions,* for short.

External Feature Interactions. We call external interactions that violate the *functional specification* of a composed system—which also includes type errors—*functional feature interactions.*

We call interactions that influence *non-functional properties* of a composed system—including performance, memory consumption, energy consumption—*non-functional feature interactions.* [1]

Internal Feature Interactions. Beside the behavior-centric view of external feature interactions, there is an implementation-centric view, which aims at the internals of a system [6, 12, 21].

To let features interact, we need corresponding *coordination code*. For example, if we want to coordinate the fire-alarm and flood-control features of the alarm-and-emergency system example, we have to add additional code for this task (e.g., to deactivate flood control in the case of fire). This coordination code gives rise to a *structural feature interaction.* Features are considered to interact structurally if some coordination code is present for their collaborative working.

Apart from just analyzing the code base and searching for coordination code that gives rise to structural interactions, one can collect more detailed information on internal feature interactions by analyzing the execution or operation of a system. Which features refer to which other features? Which features pass control to which other features? Which features pass data to which other features? This information on *operational interactions* cannot be easily extracted from just looking syntactically at the source code, but requires more sophisticated (static or dynamic) analyses of the control and data flow. Features are considered to interact operationally, if the occurrence of specific control and data flows diverges from the combination of the flows of the individual features involved. For example, the reference between Lines 5 and 8 in Figure 1 is actually present only if both features are present.

This operational interaction plus the unconditional optionality of feature BATCH are the cause for the type error (the cause for the external feature interaction). This is exactly the kind of relation in which we are interested.

[1] We do not consider this type of interactions in this paper, but our general framework can be applied to this type of interactions, too.

2.3 Quantifying Feature Interactions

In our study, we consider internal operational and structural feature interactions that, based on our experience, may have relation to type errors. Our basic model for internal interactions is a *feature-reference graph,* which represents dependencies between features. The nodes of the graph are features and the edges are references (dependencies) between features, as we illustrate in Figure 1. The references include inter-feature method calls, field accesses, and type accesses. Using this graph, we define measures that quantitatively capture the information about present internal interactions. For example, each of the feature references may lead to a type error, if the target of the reference is not present in a product. Thus, to measure the potential for a type error, we can count the number of references to optional features. The correlation coefficient between the number of type errors found in the source feature, which is the source of the references, and the number of outgoing references indicates a possible relation between the external and internal interactions.

It is clear that simple code measures, such as reference count, cannot capture all possible facets of internal feature interactions. Thus, we gradually enrich the basic reference-graph model with additional information gained from static program and feature-model analysis.

Using static program analysis, we collect structural information about features and represent it as *introduction sets*. An introduction set of a feature describes all program elements (i.e., types, methods, fields) that are introduced by the feature in question. If two features introduce (or refine) the same class or introduce the same method, then a piece of glue code may be needed to coordinate the collaboration of the program elements introduced by the different features. In other words, such introductions may result in internal feature interactions.

Furthermore, we analyze the feature model to learn about *optionality of the references* between features. The analysis assigns every pair of features to one of the three binary relations: always, maybe, and never. If two features are in always relation, then these features are always appear together in every valid product, and any reference between these features is never dangling. As a consequence, no corresponding product-line-specific type errors occur. If two features are in never relation, then there is no valid product containing both of these features. Thus, any reference between these two features inevitably leads to a type error (if the features are not dead). The maybe relation between two features denotes that there are valid products in which both features are present, but there are also valid products in which only one (or none) of the features is present. Thus, if, due to a reference, one feature always requires another feature, this leads to a type error in the products in which only the referring feature is present.

By combining operational (feature-reference graph), structural (introduction sets), and variability information about the product line (always, never and maybe relations), we define more complex measures that may have stronger relation to external feature interactions.

3. MEASURES FOR INTERNAL FEATURE INTERACTIONS

The main focus of our study is on the models representing operational and structural relations between features. The basic model is the feature reference graph that we gradually extend with additional structural and variability data. Besides measures defined on the basic and extended versions of the reference graph, we also studied how type errors correlated with established measures for cohesion and coupling. The rationale behind this was that high cohesion and low coupling improve the quality of the code in terms of modularity and maintainability and may reduce the number of possible type errors. Last but not least, we looked at how fragmentation of feature and class modules relates to the number of type errors, which is also reated to maintainability.

Next, we define the measures based on the basic and extended version of the reference graph, cohesion and coupling measures, and module fragmentation.

Feature-Reference Graph Degree (Ref). This is the most basic measure that is based on a feature-reference graph. The measure for a feature, which we calculate using this graph, is the degree of the corresponding node (i.e. the number of edges connected to the node).

The rationale for using this measure is that a reference to another feature may potentially lead to a type error (Figure 1). More references to different features may be the cause for more type errors. We expect positive correlation of this measure with the number of type errors.

Feature-Reference-and-Structure Graph Degree (RefS). This is also a degree-based measure. Although, before calculating the degree, we extend the basic reference graph with the structural information from the introduction set. We add an edge between two nodes, if the corresponding features introduce the same program elements (e.g., both features introduce the same method). Such structural feature interactions may require a manual resolution. If unresolved, they may brake contracts of other features, or bring in side effects not expected by other features. All these may result in type errors. We expect a positive correlation of this measure with the number of type errors.

Weighted Feature-Reference Graph Degree (RefW). For this degree-based measure, we count exactly how many references there are between two features.

A larger number of references between two features may indicate the higher complexity of the code. Consequently, this may negatively influence programmer's code comprehension and result in errors, including type errors. We expect positive correlation of this measure with the number of type errors.

Feature-Reference-and-Optionality Graph Degree (RefO). For this measure, we enrich the basic reference graph with information about the optionality of the references between features. As we describe in Section 2.3, if two features are in MAYBE relation, then a reference between these two features may cause a type error. On the contrary, if the two features are in ALWAYS relation, then the references can not cause a type error. A NEVER reference always causes a type error, but this type of references is very seldom. We capture this information in the feature reference graph by assigning weights to the edges according to the relation in which the connected nodes stay. In the case of the MAYBE relation, we assign the weight of 10, and in the case of the ALWAYS relation, the weight of 1. In the case of the NEVER relation, we assign the weight 5 to the reference. We use these weight values only to order the references according to their potential of causing a type error. We do not assert that a MAYBE reference causes 10 times as much type errors as an ALWAYS reference (ordinal scale). We do not assign the weight 0 to always-references to be able to differentiate between feature-pairs that do reference each

other (and potentially may participate on a type error), and those that do not.

On calculating the degree of a node in the feature-reference-and-optionality graph, the weight of an edge is transformed in multiple edges (e.g., an edge with weight 10 is counted as 10 edges). We expect positive correlation of this measure with the number of type errors.

Combined Feature-Reference Graph Degree (RefSWO). The basis for calculating this degree-based measure is the superimposition of the graphs defined for the above measures. We assume that the previous degree-based measures cover different aspects of product lines that may be responsible for type errors. Thus, combining them should bring better results. We expect positive correlation of this measure with the number of type errors.

Module Fragmentation.

Feature modules are orthogonal to classes [1]. One class can participate in multiple features (e.g., a class from the base feature may be refined in several other features). At the same time, multiple classes can collaborate to implement the functionality of a single feature. In other words, a class can be fragmented by multiple features; or, looking from the other side, one feature may be fragmented by multiple classes. The level of such fragmentation may be related to the number of type errors in a feature.

Feature Fragmentation (FeatureFrag). This measure quantifies how many classes collaborate to implement the functionality of a feature. For feature-oriented implementations, it just counts the number of roles in a collaboration. The higher the number of roles, the more effort is needed from the side of the programmer, to keep track of all references of these roles to other features. The increased complexity may negatively influence the quality of the code and result in type errors.

Class Fragmentation (ClassFrag). This measure quantifies in how many features the given class participates. If one class is cut through by several features, then the class-internal references may become inter-feature references. In this case, the programmer has to switch mentally between these two orthogonal views on the code. The increased complexity of programming and maintenance tasks may lead to errors and, in particular, to type errors.

Feature Cohesion and Coupling.

Cohesion and Coupling are considered to be good indicators for code modularity. Rising cohesiveness and decreasing coupling of code units improves their maintainability and facilitate modular reasoning. Thus, improved modularity may positively influence the quality of code and result in reduced numbers of type errors. We expect a positive correlation for the cohesion measure and negative correlation for the coupling measures.

Internal-Ratio Feature Dependency (IFD). Internal-Ratio Feature Dependency is a measure for feature cohesion. It relates the number of actually existing internal feature references to the number of all possible internal feature references. It is defined as follows:

$$IFD(F) = \frac{intRef(F)}{posIntRef(F)}$$

$intRef(F)$ is the number of existing internal feature references and $posIntRef(F)$ is the number of all possible internal references. A feature with the highest cohesion has $IFD = 1$.

External-Ratio Feature Dependency (EFD). External-Ratio Feature Dependency is a measure for feature coupling. It relates the number of existing internal feature references to the number of all existing references of the feature (i.e., internal and external references). It is defined as follows:

$$EFD(F) = \frac{intRef(F)}{ref(F)}$$

$intRef(F)$ is the number of internal feature references (as in IFD) and $ref(F)$ is the number of all references of the feature. A feature with the highest coupling has $EFD = 1$, and a feature with the lowest coupling has $EFD = 0$.

Coupling Between Features (CBF). Coupling Between Features counts to how many features the given feature is coupled. A feature is coupled to another one if it calls methods or accesses fields of that feature.

Coupling Between Objects (CBO). Coupling Between Objects is defined in the same way as Coupling Between Features, but instead of features, we look at classes. The rationale behind having coupling measures for features and classes is that the feature-oriented and object-oriented views at the system coexist in a product-line, and both of them should be considered.

4. EVALUATION

We conducted a evaluation of the measures under discussion by computing correlation coefficients between each measure and the number of type errors in features for a set of subject product lines. We chose Spearman's rank correlation coefficient, because the type error count is not normally distributed (Shapiro-Wilk test resulted in $p < 0.05$).

High correlation between a measure value and the number of type errors would indicate a strong relation between specific internal feature interactions (as captured by the measure) and external interactions (represented by type errors).

4.1 Subject Systems

For evaluation purposes, we selected a set of 15 feature-oriented, JAVA-based product lines. The set has been collected and prepared before for benchmarking purposes, and most of the systems have been already used in other studies [2, 4, 16]. The subject systems belong to different application domains, and have different sizes: in terms of lines of code and number of features. Table 1 summarizes relevant information about the systems.

We detected the type errors in the subject systems using FUJI [16]. FUJI is an extensible compiler for feature-oriented programming in JAVA that also provides a variability-aware type checker and a set of product-line analysis tools. Using these tools, we collected information about references between program elements and other structural information needed to calculate the measures. Information about the optionality of the references was collected using FeatureIDE [13].

4.2 Results

We summarize the results of our study in Table 2. For each subject system and each measure, we report the correlation coefficient between the number of errors in the features of the system and the corresponding indicator measures for these features. Correlations that are not statistically significant (p-value < 0.05) are not shown (replaced by a dash in the table).

Overall, we observe moderate correlations (correlation coefficients between 0.3 and 0.7) for the indicator measures under con-

sideration and the number of type errors in the features. Next, we will discuss possible reasons.

We have to note that the type errors that we used in this study are the hardest. That is, these are the type errors that left over after other code defects were fixed during the development and maintenance of the subject systems. The correlations may have been stronger if we had the historical data about all type errors that were found and fixed.

Furthermore, multiple correlation coefficients have insufficient statistical significance. This is mostly the case for the subject systems with small numbers of features, which apparently do not provide enough data to produce significant results (e.g., BANKACCOUNTTP and POKERSPL).

As for the measures based on the reference graph, we observe moderate correlations. We took the basic reference graph as the base case and compared to it all subsequent versions, which are enriched with additional information. In Figure 2, we observe that adding information about possible structural interactions does not result in much stronger correlations, in general. We can see a slightly stronger correlation for several subjects systems (e.g., BERKELEYDB, VIOLET, TANKWAR). At the same time, GUIDSL shows a decreased correlation, and BANKACCOUNTTP shows even a strong negative correlation. The outlier BANKACCOUNTTP and POKERSPL can be explained by a very small number of type errors that were found in these systems (five and one error respectively). These results indicate that the information about structural interactions may be not sufficient to draw statistically significant conclusions about external feature interactions.

We obtain a similar picture if we look at the measure *RefW*, which considers all possible references in the feature reference graph (Figure 3). We observe higher correlations for GPL, NOTEPAD-ROBINSON, and TANKWAR. At the same time, we see lower correlations for GUIDSL, MOBILEMEDIA, and VIOLET. Thus, we conclude that the pure number of references between features does not exhibit a significant relation to the number of type errors in these features.

In Figure 4, we observe that adding information about the variability of the references between features to the basic reference graph increases the correlation coefficients. We observe considerably higher correlations for EPL, PROP4J, and SUDOKU. At the same time, we see no considerable decrease for the rest of statistically significant results. This indicates that variability information combined with the information about references between features is related to product-line-specific type errors. This seems plausible considering that this is exactly the information that a variability-aware type checker uses to detect type errors. Furthermore, it is important to stress that this is exactly the kind of information we want to find with studies such as this one: We want to find what kinds of internal feature interactions have strong relations to external interactions of a certain kind.

Table 1: Subject systems overview (LOC: Lines of code, #C: Number of classes in the system, #F: Number of features containing Java-code, # errors: Number of product-line-specific type errors).

Systems	LOC	#C	#F	# errors
BANKACCOUNTTP	132	3	8	5
BERKELEYDB	45000	283	99	198
EPL	111	12	12	42
GPL	1940	16	20	16
GUIDSL	11529	144	26	59
MOBILEMEDIA8	4189	51	45	142
NOTEPAD-ROBINSON	800	9	10	3
POKERSPL	283	8	10	1
PREVAYLER	5268	138	6	15
PROP4J	1531	14	14	490
SUDOKU	1422	26	7	17
TANKWAR	4845	22	30	66
UNIONFIND	210	4	8	19
VIOLET	7194	67	88	117
VISTEX	1608	8	16	12

Fragmentation measures. This group of measures also shows moderate correlations with the number of type errors (Figure 6). The class fragmentation measure shows higher correlation coefficients than the feature fragmentation measure. A possible explanation for this relation may be that a programmer considers a class as a whole and does not pay special attention to internal class references that cross feature boundaries. But these references are exactly the ones that are involved in type errors.

Cohesion and coupling measures. This group of measures also show moderate correlations. Surprisingly, the IFD cohesion measure has a positive correlation to the number of type errors, not negative, as expected. Although, only 4 out of 15 values are statistically significant. Thus, we can not draw any conclusion from these data.

As for the coupling measures (Figure 7), EFD shows moderate positive correlations for three subject systems and negative correlations for PREVAYLER. The rest of the correlations are not significant. CBO and CBF show pretty similar correlation coefficients. The half of the statistically significant values are at the border between weak and moderate correlation (i.e., coefficient of 0.3).

5. THREATS TO VALIDITY

The statistical significance of the correlation coefficients depend on the number of features in a product line and the number of type errors in these features. For the product lines with few features (e.g., SUDOKU) or few type errors (e.g., PREVAYLER), there are hardly significant correlations. For the product lines with few features and type errors (UNIONFIND, VISTEX), there are no significant results at all. Reproducing the study on larger product lines (in terms of number of features) and with historical data about type errors that have existed in the system would produce more generalizable results.

The measures in our study cover different aspects of the product lines that may be related to type errors (e.g., structure, variability, etc.). Depending on the application domain, development technique, and maintenance history of the systems, some of the aspects may dominate over others. Such nuances are hard to capture in one general measure. The problem can be mitigated by combining different measures (e.g., EFD and CBF), but the overlaps in the measures may distort the results (e.g., EFD and CBF both use external references count in the calculation). Alternatively, developing a measure that is specific to the given SPL can be considered [15].

6. RELATED WORK

Predicting defects using software measures has been successfully studied for years [9, 14, 20, 22]. Graph-based measures have been applied to make predictions about bug severity and defect count [7, 24].

In product-line domain, Siegmund et al. used information about internal operational interactions in product lines to predict external non-functional properties (in this case, performance) of indi-

vidual products [29]. However, our work is focused not on prediction of external quality attributes and feature interactions, but more on understanding the cause of the external feature interactions, and studying their relation to internal interactions. In a follow-up step, this knowledge can be used to improve existing prediction techniques, too.

Nguyen et al. developed a technique to analyze the influence of operational attributes of a software system on its functional properties (in this case, client-side output of a web application) [23]. Our work is similar to this one, but our ultimate goal is to target the whole spectrum of internal and external feature interactions and to study their relations in broader perspective.

7. CONCLUSION

We presented a preliminary study that examined the relations between internal and external feature interactions in feature-oriented product lines. The information about internal feature interactions was captured by a set of measures based on the data from static code analysis and feature-model analysis. The external feature interactions were represented by product-line-specific type errors.

We calculated correlation coefficients between the measures and the number of type errors for 15 feature-oriented, JAVA-based product lines. We observed moderate correlations between the measures under discussion and the number of type errors (especially for *RefO*, *RefSWO*, and fragmentation measures). This result is not sufficient to draw final and generalizable conclusions, but motivates us to follow this direction and study other kinds of external interactions (e.g., non-functional interactions) using the same conceptual framework. Moreover, the framework may be useful to other researchers who want to study interdependencies between different kinds of feature interactions.

Acknowledgments

This work was supported by the DFG grants AP 206/4, AP 206/5, and AP 206/6.

8. REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[2] S. Apel and D. Beyer. Feature cohesion in software product lines: An exploratory study. In *Proc. ICSE*, 2011.

[3] S. Apel and C. Kästner. An overview of feature-oriented software development. *J. Object Technol.*, 2009.

[4] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access control in feature-oriented programming. *Sci. Comput. Program.*, 2012.

[5] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proc. FOSD*, 2013.

[6] D. Batory, P. Höfner, and J. Kim. Feature interactions, products, and composition. In *Proc. GPCE*, 2011.

[7] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proc. ICSE*, 2012.

[8] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Proc. SETSS*, 1989.

[9] N. Fenton and M. Neil. A critique of software defect prediction models. *IEEE T. Software Eng.*, 1999.

[10] B. Garvin and M. Cohen. Feature interaction faults revisited: An exploratory study. In *Proc. ISSRE*, 2011.

[11] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proc. FOSD*, 2011.

[12] C. Kästner, S. Apel, S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proc. SPLC*, 2009.

[13] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *Proc. ICSE*, 2009.

[14] B. Kitchenham, L. Pickard, and S. Linkman. An evaluation of some design metrics. *Software Engineering Journal*, 1990.

[15] S. Kolesnikov, S. Apel, N. Siegmund, S. Sobernig, C. Kästner, and S. Senkaya. Predicting quality attributes of software product lines using software and network measures and sampling. In *Proc. VaMoS*, 2013.

[16] S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel. A comparison of product-based, feature-based, and family-based type checking. In *Proc. GPCE*, 2013.

[17] D. Kuhn, D. Wallace, and A. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE T. Software Eng.*, 2004.

[18] J. Lee, K. Kang, and S. Kim. A feature-based approach to product line production planning. In *Prloc. SPLC*, 2004.

[19] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. ICSE*, 2010.

[20] B. Littleword. Forecasting software reliability. In *Software Reliability Modelling and Identification*, 1987.

[21] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. ICSE*, 2006.

[22] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. ICSE*, 2006.

[23] H. Nguyen, C. Kästner, and T. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proc. FSE*, 2014.

[24] R. Premraj and K. Herzig. Network versus code metrics to predict defects: A replication study. In *Proc. ESEM*, 2011.

[25] E. Reisner, C. Song, K. Ma, J. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proc. ICSE*, 2010.

[26] N. Siegmund, S. Kolesnikov, C. Kastner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proc. ICSE*, 2012.

[27] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Performance prediction in the presence of feature interactions. In *Software. Eng. J.*, 2014.

[28] N. Siegmund, M. RosenmüLler, C. KäStner, P. G. Giarrusso, S. Apel, and S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Inform. Software. Tech.*, 2013.

[29] N. Siegmund, A. von Rhein, and S. Apel. Family-based performance measurement. In *Proc. GPCE*, 2013.

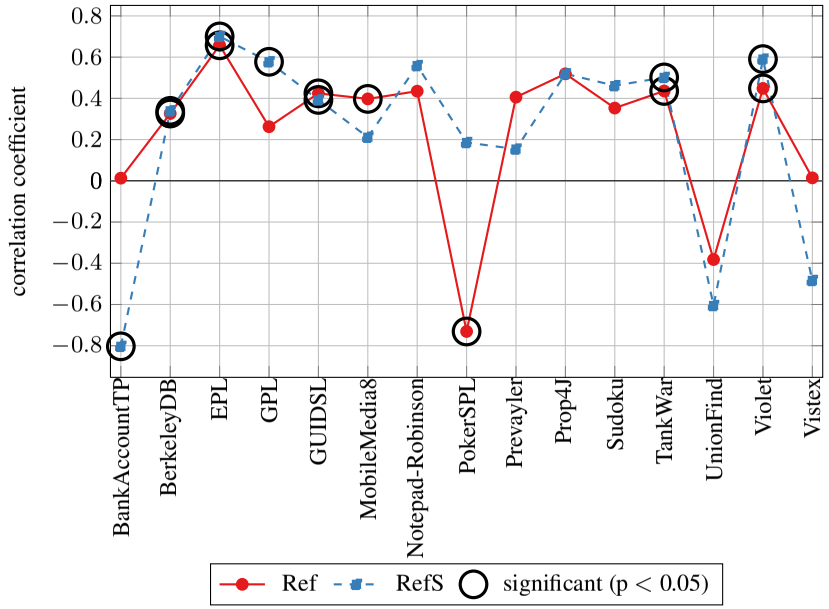

Figure 2: Comparison between the correlation coefficients for the Feature-Reference Graph Degree measure and the Feature-Reference-and-Structure Graph Degree measure.

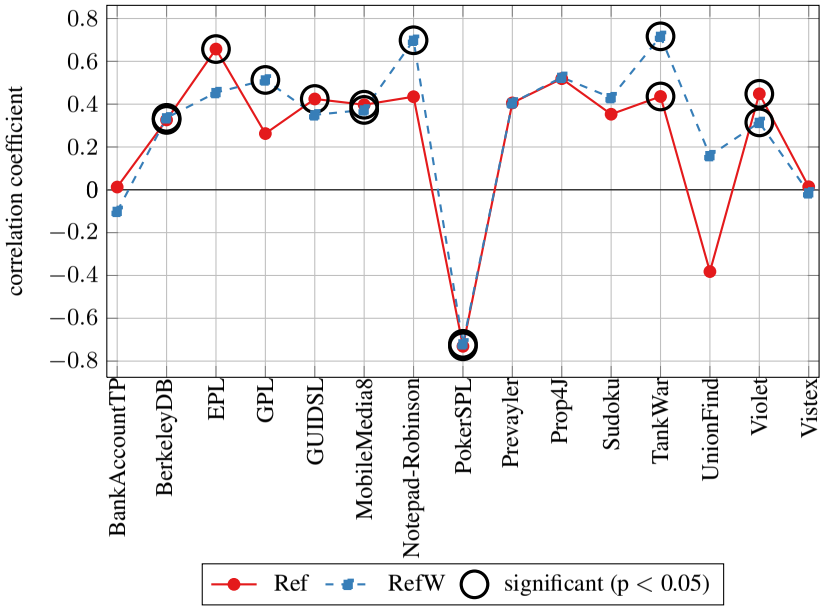

Figure 3: Comparison between the correlation coefficients for the Feature-Reference Graph Degree measure and the Weighted Feature-Reference Graph Degree measure.

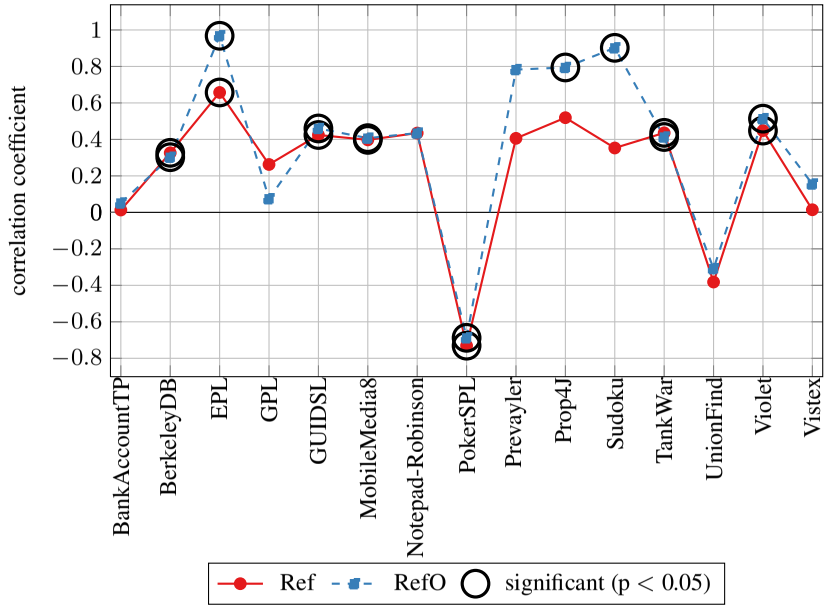

Figure 4: Comparison between the correlation coefficients for the Feature-Reference Graph Degree measure and the Feature-Reference-and-Optionality Graph Degree measure.

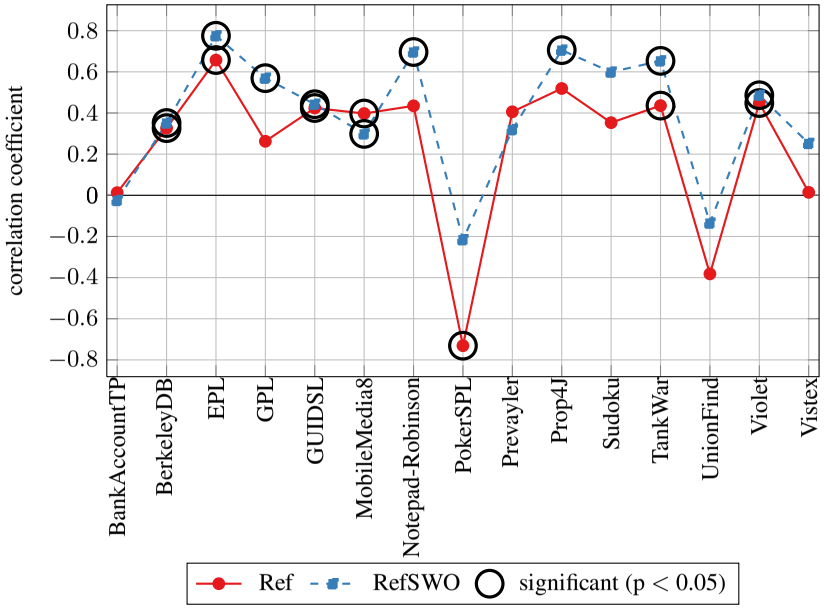

Figure 5: Comparison between the correlation coefficients for the Feature-Reference Graph Degree measure and the Combined Feature-Reference Graph Degree measure.

Table 2: Summary of the results. For each subject system and each measure a correlation coefficient between the number of errors in the features of the system and the corresponding measure values for these features is given. Correlations that are not statistically significant (p-value < 0.05) are not shown (depicted by a dash in the table)

	Ref	RefS	RefW	RefO	RefSWO	ClassFrag	FeatureFrag	IFD	EFD	CBO	CBF
BANKACCOUNTTP	–	–0.803	–	–	–	–	–	–	–	–	–
BERKELEYDB	0.326	0.339	0.336	0.302	0.351	0.526	0.366	–	0.241	0.402	0.313
EPL	0.657	0.701	–	0.968	0.775	–	–	0.613	–	–	–
GPL	–	0.577	0.513	–	0.569	0.798	–	–	–	0.632	0.471
GUIDSL	0.424	0.393	–	0.460	0.443	0.412	0.433	–	–	0.286	–
MOBILEMEDIA8	0.397	–	0.373	0.407	0.298	0.486	0.377	0.411	0.458	0.472	0.561
NOTEPAD–ROBINSON	–	–	0.698	–	0.696	–	–	–	–	–	–
POKERSPL	–0.731	–	–0.719	–0.688	–	–	–0.645	–	–	–	–0.640
PREVAYLER	–	–	–	–	–	0.284	–	–	–0.899	–	–
PROP4J	–	–	–	0.794	0.705	–	–	–	–	–	–
SUDOKU	–	–	–	0.902	–	–	–	–	–	0.632	–
TANKWAR	0.436	0.502	0.717	0.413	0.653	0.791	0.413	0.718	–	0.706	0.805
UNIONFIND	–	–	–	–	–	–	–	–	–	–	–
VIOLET	0.448	0.591	0.315	0.514	0.486	–	0.247	0.426	0.504	–	0.310
VISTEX	–	–	–	–	–	–	–	–	–	–	–

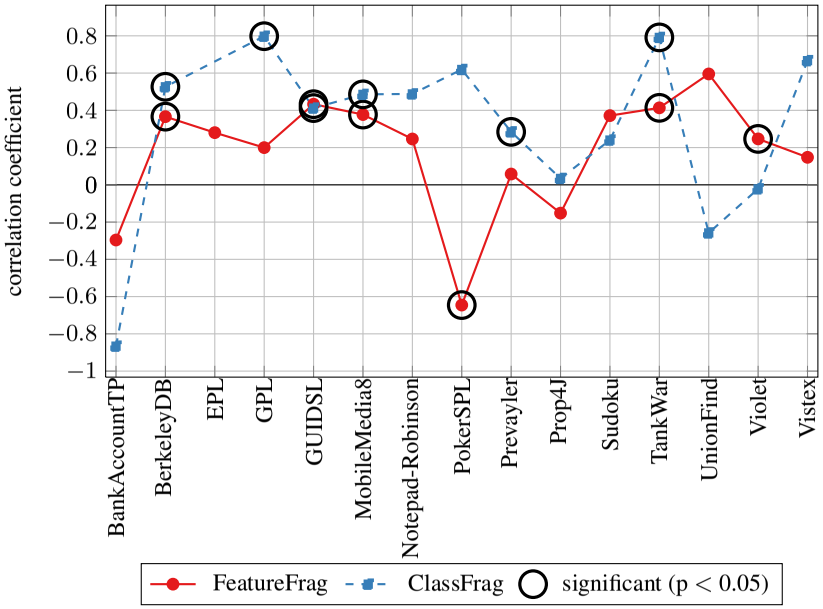

Figure 6: Comparison between the correlation coefficients for the module fragmentation measures.

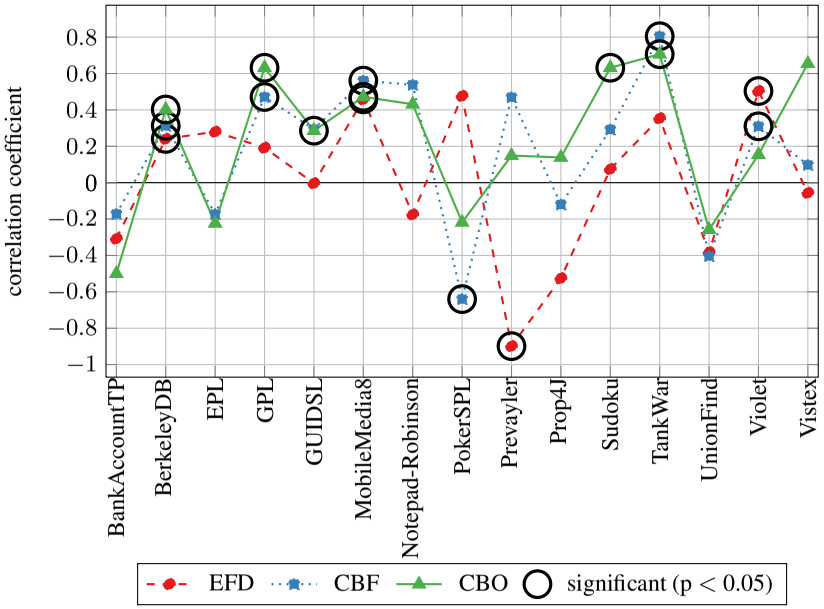

Figure 7: Comparison between the correlation coefficients for the coupling measures.