Empir Software Eng
DOI 10.1007/s10664-014-9336-6

Quantifying structural attributes of system decompositions in 28 feature-oriented software product lines

An exploratory study

Stefan Sobernig · Sven Apel · Sergiy Kolesnikov · Norbert Siegmund

Abstract A key idea of *feature orientation* is to decompose a software product line along the features it provides. Feature decomposition is orthogonal to object-oriented decomposition—it crosscuts the underlying package and class structure. It has been argued often that feature decomposition improves system structure by reducing coupling and by increasing cohesion. However, recent empirical findings suggest that this is not necessarily the case. In this exploratory, observational study, we investigate the decompositions of 28 feature-oriented software product lines into classes, features, and feature-specific class fragments. The product lines under investigation are implemented using the feature-oriented programming language Fuji. In particular, we quantify and compare the internal attributes import coupling and cohesion of the different product-line decompositions in a systematic, reproducible manner. For this purpose, we adopt three established software measures (e.g., coupling between units, CBU; internal-ratio unit dependency, IUD) as well as standard concentration statistics (e.g., Gini coefficient). In our study, we found that feature decomposition can be associated with higher levels of structural coupling in a product line than a decomposition into classes. Although coupling can be concentrated in very few features in most feature decompositions, there are not necessarily hot-spot features in all product lines. Interestingly, feature cohesion is not necessarily higher than class cohesion, whereas features are more equal in serving dependencies internally than classes of a product line. Our empirical study raises critical questions about alleged advantages of feature decomposition. At the same time, we demonstrate how our measurement approach of coupling and cohesion has potential to support static and dynamic analyses of software product lines (i.e., type checking and feature-interaction detection) by facilitating product sampling.

Communicated by: Ebrahim Bagheri, David Benavides, Per Runeson and Klaus Schmid

S. Sobernig (✉)
WU Vienna, Wien, Austria
e-mail: stefan.sobernig@wu.ac.at

S. Apel · S. Kolesnikov · N. Siegmund
University of Passau, Passau, Germany

Published online: 29 October 2014

Keywords Software product lines · Feature-oriented programming · Fuji · Structural coupling · Structural cohesion · Software measurement

1 Introduction

A *software product line* is a family of software products derived from a shared code base, ideally in a widely automated manner. Each product is described in terms of a valid configuration of the product line's domain model (e.g., a feature model; Czarnecki and Eisenecker 2000). In a *feature-oriented product line*, the implementation assets implement features as cohesive units of functionality (Apel and Kästner 2009). A feature addresses a specific functional domain requirement, represents a design decision in the domain implementation, and often establishes a configuration option when deriving a product. *Feature-oriented programming* using AHEAD (Batory et al. 2004), Fuji[1] (Apel et al. 2012), and FeatureHouse (Apel et al. 2013b) aims at decomposing the code base into dedicated and tractable feature units that include all feature-specific code. Compared to alternative feature-implementation techniques (e.g., plug-ins, pattern-based designs, and code preprocessing), feature-oriented programming establishes an explicit and clean mapping (ideally, one to one) between the features in the domain model and the corresponding code.

In feature-oriented product lines, several structural decompositions co-exist, typically an object-oriented decomposition into classes and a feature-oriented decomposition into feature units. The extent to which a product line and its decompositions are accessible to developers (e.g., as chunks of cognitive processing; Lilienthal 2009) is affected by the level of mutual, functional dependencies between decomposition units (coupling) as well as their internal dependency structure (cohesion). Coupling and cohesion determine whether software developers can study decomposition units (features, classes) one at a time, for example, to make design and implementation decisions local to a decomposition unit (Kiczales and Mezini 2005; Lilienthal 2009; Kästner et al. 2011). The attributes of coupling and cohesion, in turn, reflect a number of intentions towards the structuring of a decomposition: First, decomposition units should have separated and unique (non-duplicated) functional responsibilities. Second, potential error-propagation paths between and within decomposition units should be reduced to a minimum and locatable unambiguously (Taube-Schock et al. 2011). Third, the level of structural fragmentation of a decomposition should be controlled. Fragmentation denotes the total number of decomposition units in relation to the sizes of decomposition units. Micro-modularization may yield a large number of decomposition units, each too small to facilitate any useful reasoning (Kästner et al. 2011).

Earlier, and in a convenience view, feature decomposition was thought of as leading to more modularly structured—that is, highly cohesive and loosely coupled—decomposition units (Kästner et al. 2011). However, more recent empirical findings suggest that feature decomposition does not necessarily achieve this objective (Apel and Beyer 2011; Kästner et al. 2011). Nevertheless, each alternative decomposition can be critical for different developer roles (e.g., domain or application developers) and for different engineering tasks (e.g., code reviews, feature promotion, and feature-specific refactorings). This adds to the conjectures about developers requiring tailorable and aggregating views on crosscutting object-oriented and feature-oriented decompositions (Kästner et al. 2011). Unfortunately, to this date, there is little empirical evidence on how alternative decompositions compare to

[1] http://fosd.net/fuji/, last accessed: 01.07.2014.

each other in terms of their internal attributes (fragmentation, coupling, cohesion). Addressing this issue will influence the development of language-based and tool-based approaches for feature-oriented product lines.

So far, research on component-oriented architectures (Bouwers et al. 2011), as well as on object-oriented (Sarkar et al. 2008) and feature-oriented programming (Apel and Beyer 2011) has investigated only one decomposition dimension in isolation (e.g., the decomposition into feature units) and only certain internal attributes: Bouwers et al. (2011) set out to quantify fragmentation while ignoring cohesion and coupling. Apel and Beyer (2011) investigated feature cohesion without incorporating coupling and fragmentation. Furthermore, earlier data sets (Apel and Beyer 2011) extracted from product-line code bases were based on the context-free syntax of feature implementations (introductions; e.g., method and field declarations), rather than incorporating also context-sensitive data (references; e.g., method calls, field accesses). While feature orientation is still in its infancy, it is a promising line of research (Apel et al. 2013a)—the time is ripe to back foundational research in this area with further empirical data.

Therefore, we set up an empirical investigation using software measurement on the internal attributes of 28 feature-oriented product lines, that is, their decompositions into classes, features, and feature-specific class fragments. The investigated product lines are implemented using Fuji, a Java language extension for feature-oriented programming (Apel et al. 2012). The selected product lines differ by target domains, by code size, by structural code complexity (i.e. number of introductions, decomposition units, and references between these units), and by structural feature-model complexity (e.g., number of leaf features and of optional features; Berger and Guo 2013).

We quantify internal attributes, such as decomposition size, import coupling, cohesion, and unit sizes, using software measures suggested by prior work on product lines to describe the different decompositions of a product line in a systematic, reproducible, and comparable manner (Montagud et al. 2012). Overall, we make the following contributions:

- We review and apply established software measures: *Coupling between units* (CBU) expresses the number of decomposition units (classes, features, feature-specific class fragments) that induce a dependency to a given decomposition unit. *Internal-ratio unit dependency* (IUD) captures the internal connectedness of a decomposition unit as the ratio actual references running between program elements owned by a given decomposition unit and its size. *External-ratio unit dependency* (EUD) captures the self-sufficiency of a decomposition unit as the ratio of references running between program elements of a decomposition unit and the unit's total, including external, references.
- To answer two research questions derived from product-line literature, we aggregate the direct, per-unit measurements using concentration statistics at the level of decompositions and of product lines. The Gini statistic (Vasilescu et al. 2011) signals how concentrated a quantity of interest (e.g., per-unit coupling) is in different subsets of decomposition units.
- We demonstrate how coupling and cohesion measurement using the above measure constructs can optimize product sampling for static and dynamic analysis techniques for product lines (i.e., variability-aware type checking and feature-interaction detection).

In a nutshell, we found that (1) feature decomposition can result in more densely coupled code structures than object-oriented decomposition. In addition, (2) feature decomposition

is more heterogeneous regarding the distribution of coupling over the individual feature implementations than object-oriented decomposition. Across the 28 product lines, both highly and loosely coupled feature implementations can take dominant shares in the overall coupling. At the same time, (3) feature decomposition can result in more self-contained units of functionality than object-oriented decomposition. However, this does not imply that syntax elements of feature implementations are internally more interconnected than in object-oriented decomposition. Finally, (4) we demonstrate how the adopted indicator measures for coupling and cohesion can be used to devise sampling techniques for product-line type checking and feature-interaction detection.

Based on our study, we discuss the implications and perspectives of our measurement methodology and experimental findings for future work on static and dynamic analysis of product lines. We back our discussion by two feasibility studies on type-checking product lines and feature-interaction detection.

Overall, our study is meant to contribute to a better understanding of the nature and merits of feature orientation. While feature orientation is still a comparatively novel software-development paradigm, it embodies a key insight of the practice of product-line engineering in the past: Features must be explicit (modular, if possible) in design and code, across the whole life cycle of a product line and the corresponding software products (Apel et al. 2013a). In this sense, our study provides insights that are meant to facilitate, motivate, and guide industrial adoption.

All data sets as well as the statistical tooling for reproducing our results are available as supplements to a detailed technical report (Sobernig et al. 2014).

2 Three Decompositions, Many Differences?

Typically, the code base of a feature-oriented product line is decomposed along two major dimensions: code units and feature units. In Fig. 1, we exemplify this for the canonical graph product line (GPL). GPL is a product line of graph libraries allowing a developer for tailoring data structures representing different graph types (weighted, directed) and for choosing from common graph traversal strategies. GPL was developed by Lopez-Herrejon and Batory (2001) as a standard problem suitable for evaluating and for comparing product-line techniques.

On the one hand, the code base can be structured into *code units* according to the decomposition mechanisms offered by the programming language. Consider, for example, a hierarchical object-oriented decomposition using Java, involving packages, nested classes, methods, and the containment relationships between them. GPL has a number of classes according to this decomposition, for example, `Graph`, `Edge`, and `Strength`. On the other hand, *feature units*, which embody the feature implementations in the code base, give rise to a second decomposition which is orthogonal to the object-oriented one. The feature decomposition can be hierarchical as well. GPL consists of three feature units: `Base` provides means to represent basic graphs, `Weighted` adds weights to edges, and `Measures` enables the computation of numeric graph characteristics (e.g., the strength measure sums the weights of the edges incident to a node). A third decomposition results from the intersection of the first two decompositions: code units (e.g., classes in the GPL) are divided into *code-unit fragments* (a.k.a. roles; Smaragdakis and Batory 2002) defining the structure and behavior of code units specific to single features. The set of code-unit fragments that belong to a single feature and that are scattered over multiple code units form a feature unit (a.k.a. collaborations; Smaragdakis and Batory 2002).

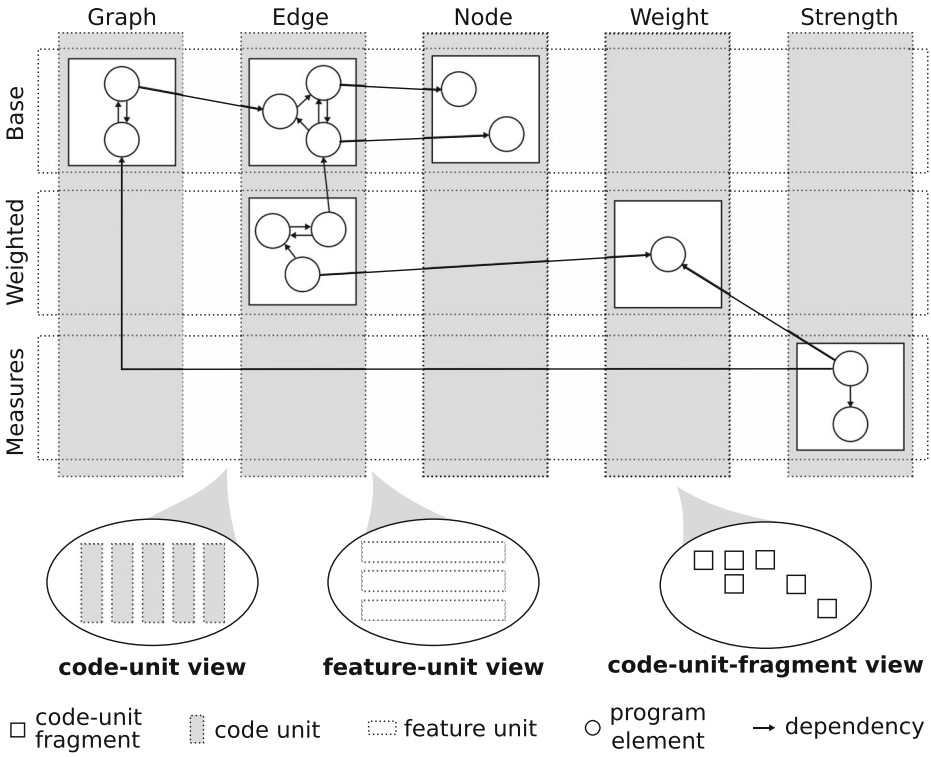


Fig. 1 The three decompositions of the graph product line into *code units*, *feature units*, and *code-unit fragments*. Each decomposition provides an alternative view on the product line to the developer. Code units, feature units, and code-unit fragments contain *program elements* (e.g., fields and methods) that may depend on other program elements

Different roles and tasks in product-line engineering benefit from different or all decompositions. A domain developer who maintains a given reusable product-line asset must frequently locate feature units on which the feature unit under review depends. In code reviews, domain architects and domain developers frequently evaluate the mapping between features and feature implementations by navigating through the code base following feature traces provided by, e.g., code annotations and feature-aware code editors (Kästner et al. 2012). From the viewpoint of an application developer, a view on the object-oriented decomposition representing the derived product is eligible to facilitate object-oriented development tasks (e.g., framework integration of the product into a final object-oriented system). To promote features from products to the product line (as done in the extractive approach; Clements and Krueger 2002), domain architects and application developers must locate feature-specific code in the object-oriented decomposition of the code base to refactor them into existing and new feature units (Apel et al. 2013a). Finally, application developers use all decompositions for reporting problems or defects experienced with a reused asset to the responsible domain developer. A feature decomposition is helpful to establish whether the defect is located in a particular feature unit or its neighbor units.

The different decompositions of a feature-oriented product line can completely overlap or crosscut each other (Apel et al. 2008). Consider two different decomposition alignments of the GPL in Fig. 1. Feature unit `Measures` and code unit `Strength` contain the exact same set of program elements and the elements' dependencies. These elements form the code-unit fragment of `Strength` that is specific to feature `Measures`. There is a complete structural overlap of the three decomposition units, while each unit is part of a distinct decomposition. Consequently, the structural attributes, such as unit coupling and cohesion of the code-unit fragment, the code unit, and the feature unit, are strongly associated or even the same. In contrast, feature unit `Weighted` and code unit `Edge` crosscut each other. While code unit `Edge`, in total, comprises two code-unit fragments, feature unit `Weighted` contains only the one fragment specific to this feature. As a result, the feature unit and the code unit have distinct coupling and cohesion properties. This structural heterogeneity of decompositions cannot only arise for coupling and cohesion, but for any structural attributes, such as the fragmentation. Regarding unit sizes, one decomposition might show a relatively small number of decomposition units with imbalanced unit sizes (Fig. 2b), while the other counts a medium number of units of more balanced sizes (Fig. 2c).

3 Comparative Research Design

By means of an exploratory study, we want to gain insights into how unit sizes, fragmentation, coupling, and cohesion of the three different decompositions compare to each other. In particular, we want to answer the following two research questions:

> **Research question #1**: How do coupling structures of a feature-oriented product line differ between its decompositions into *classes*, *features*, and feature-specific *class fragments*?

Several studies hint at unequal distributions of coupling over the decomposition units in object-oriented designs (Taube-Schock et al. 2011). They suggest that object-oriented decompositions are dominated by a comparatively large number of lowly coupled decomposition units, with only a few highly coupled decompositions units. These latter, however, appear coupled over-proportionally, at extremes.

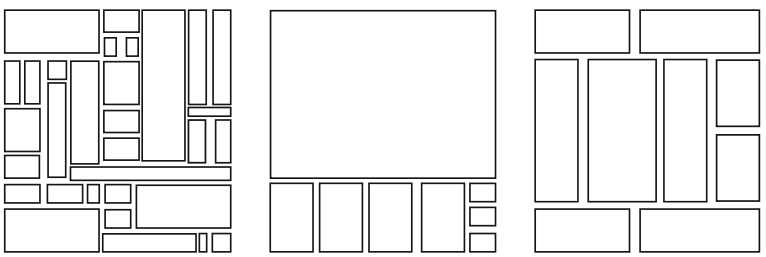


Fig. 2 Different fragmentations of product-line decompositions: **a** large decomposition size, non-uniform unit sizes; **b** small decomposition size, non-uniform and concentrated unit sizes; **c** medium decomposition size, balanced unit sizes

Similar observations have been reported for feature decompositions, though at the level of single features rather than entire product lines and for different notions of coupling (export vs. import coupling). Apel and Beyer (2011) introduce the notion of *provider* and *customer* features to discuss different roles in the dependency structure of a product line. Provider features offer data and behavior to customer features (export coupling). Customer features attach to provider features as part of the feature implementation (import coupling), but do not provide anything to other features. Likewise, Siegmund et al. (2012) report on *hot-spot features*, which are export-coupled with a comparatively large number of features.

Establishing whether all or some alternative decompositions of feature-oriented product lines show a characteristic tendency towards the presence of provider and hot-spot units would have several benefits. For example, product-line testing strategies could prioritize such decomposition units by running test cases on configurations guaranteed to include them, or by defining coverage criteria accordingly. Or, when building sampling-based prediction models for non-functional properties (Siegmund et al. 2012), a confirmed assumption of highly coupled feature units could be used to stratify the sampling of configurations, based on sub-populations that either include or exclude these decomposition units.

Research question #2: Do features as decomposition units form cohesive units of functionality? How do they compare with classes and class fragments in terms of cohesion?

Cohesion is the degree to which program elements of a decomposition unit (class, class fragment, and feature) depend on each other, for example, to operate on a shared set of data structures (e.g., communicational binding) or to perform a single function (functional binding). Apel and Beyer (2011) found that features predominantly depend on elements internal rather than external to them, and that features appear to be less cohesive than other, possibly smaller decomposition units. According to Apel and Beyer (2011), this follows from smaller features to be more cohesive than larger features. In addition, the feature units measured in their study depend only on comparatively few elements of the same feature. Therefore, the authors concluded that a refactoring into smaller, allegedly more cohesive units (e.g., code-unit fragments) should be considered. Still, it remains to be investigated whether there is a systematic relation between unit sizes and unit cohesion, and whether decompositions of smaller unit sizes turn out to be more cohesive, to establish such and similar guidelines.

3.1 Representing Decompositions

We model each of the three product-line decompositions as a dependency graph: $DG = (U, D)$. The nodes U denote decomposition units (viz., classes, class fragments, and features). The edges D represent usage dependencies between these decomposition units (e.g., one feature uses a method introduced by another feature).

We use two code excerpts taken from GPL (Fig. 3) to illustrate the models for different decompositions. The features `Base` and `Weighted` contain 12 uniquely identifiable program elements, such as type, method, and field definitions. In Fig. 3, every such element is described by a comment line indicating an element's identifier. The identifier consists of the enclosing feature, the element kind, and the fully qualified element name. For example, the field in Line 5 of feature `Base` is identified as `(Base, field, Graph.edges)`.

We call such uniquely identifiable program element an *introduction*, because it is incorporated by the corresponding feature into the code base of a product line at feature-

`Base` (excerpt)

```
1 /*** File: Base/Graph.java ***/
2 // (Base, type, Graph)
3 public class Graph {
4     // (Base, field, Graph.edges)
5     private java.util.List<Edge>
          edges;
6 }
7 /*** File: Base/Edge.java ***/
8 // (Base, type, Edge)
9 public class Edge {
10    // (Base, field, Edge.head)
11    private Node head;
12    // (Base, field, Edge.tail)
13    private Node tail;
14 }
15
16 /*** File: Base/Node.java ***/
17 // (Base, type, Node)
18 public class Node {
19        /* ... */
20 }
```

`Weighted` (excerpt)

```
1 /*** File: Weighted/Edge.java ***/
2 // (Weighted, type, Edge)
3 public class Edge {
4     // (Weighted, field, Edge.weight)
5     private Weight weight;
6     // (Weighted, ctor, Edge.Edge)
7     public Edge(int weight) {
8         this.weight = new Weight(
              weight);
9     }}
10 /*** File: Weighted/Weight.java ***/
11 // (Weighted, type, Weight)
12 public class Weight {
13     // (Weighted, field, Weight.value)
14     private int value;
15     // (Weighted, ctor, Weight.Weight)
16     public Weight(int v) { this.value
           = v; }
17 }
```

Fig. 3 Excerpts of the implementations of the features `Base` and `Weighted`

composition time. Introductions are basic building blocks for decomposition units. Usage dependencies between introductions define dependencies between decomposition units built of these introductions. Henceforth, we refer to such a usage dependency between two introductions as a *reference*. Note that there can be multiple distinct references between two introductions.

The dependency graph resulting from the introductions and their references found in the two code excerpts in Fig. 3 is illustrated in Fig. 4a. The twelve introductions are represented as the graph's nodes. The introductions are grouped according to the owning Java type, depicted as dashed rectangles (e.g., `Graph`). The references between the introductions are shown as edges. For example, the edge pointing from the field declaration `(Base, field, Graph.edges)` to the type declaration `(Base, type, Edge)` in Fig. 4a is recorded because this field has the type defined by `Edge` (we say the field uses the type; see Line 5 of `Base` in Fig. 3).

This dependency graph at the introduction level is the starting point for modeling the three decompositions we are interested in. Depending on the decomposition criterion, the introductions are grouped into distinct decomposition units: code units (Fig. 4b), feature units (Fig. 4c), and code-unit fragments (Fig. 4d).

In Fig. 4b, the grouping criterion is the Java type ownership of introductions. All introductions defined by a given Java type (e.g., an interface or a class) are grouped into one decomposition unit (`Graph`, `Node`, `Edge`, and `Weight`). This results in the code-unit decomposition or *class decomposition*. Note that nested classes and nested interfaces are considered decomposition units separate from their parent classes. As for the second decomposition, the grouping criterion applied to obtain the dependency graph in Fig. 4c is feature ownership of introductions. Hence, all introductions belonging to one feature are grouped into a feature unit. This results in the feature-unit decomposition or, for brevity, *feature decomposition*. The third decomposition combines the previous two decomposition criteria and builds groups of introductions according to both their type *and* their feature ownership. This is the code-unit fragment decomposition or, simply, *class-fragment decomposition* (Fig. 4d). When moving up between two decomposition levels (e.g., from single introductions to class fragments or from class to feature groupings), multiple equally directed

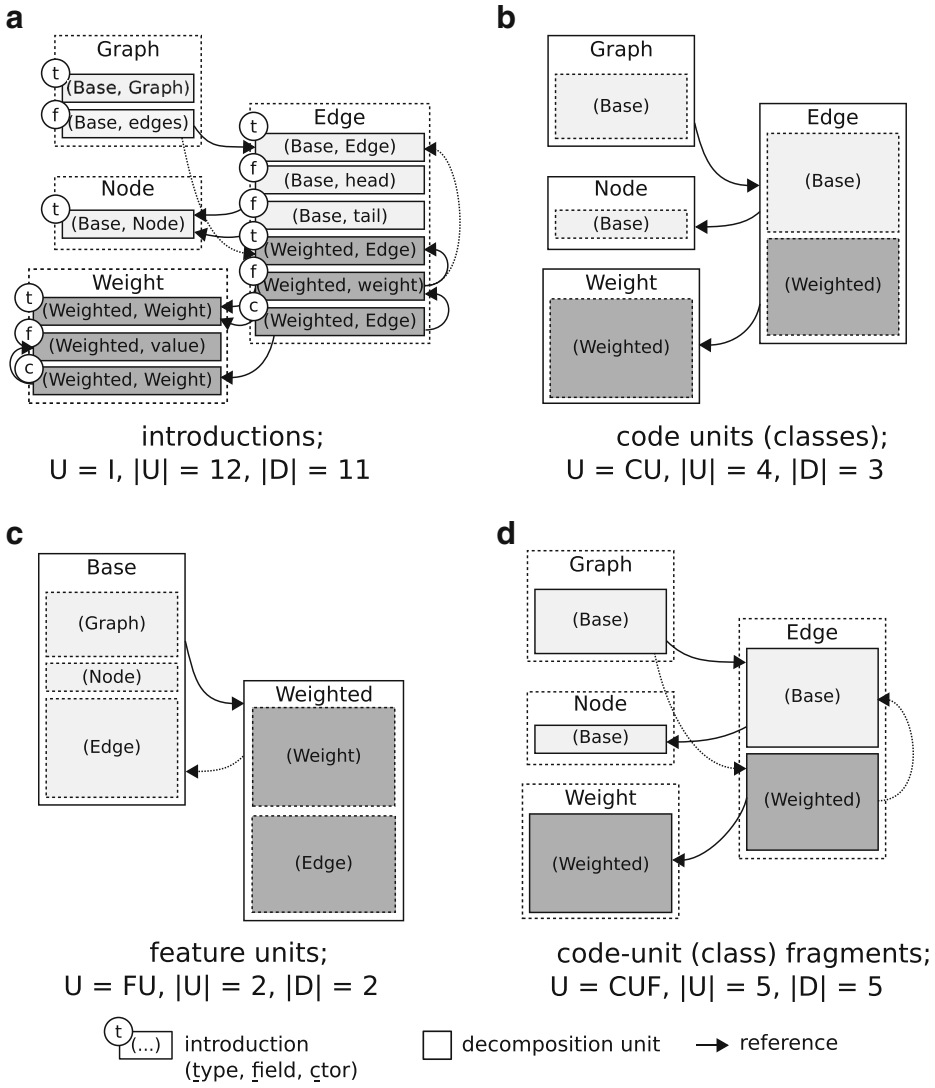


Fig. 4 Dependency graphs resulting from different decompositions of GPL; U: set of decomposition units in graph; D: set of dependencies; CU: set of code units (classes); CUF: set of code-unit (class) fragments; FU: set of feature units (feature modules)

references between two decomposition units (e.g., introductions, classes) are recorded as one reference between the corresponding units at the upper decomposition level (class fragments, features).

3.2 Subject Product Lines

Our aim was to collect feature-oriented product lines implemented using the feature-oriented programming language Fuji (Apel et al. 2012), a Java language extension providing

Table 1 Overview of the data sets extracted for each product line

ID		SLOC	| I |	| R |	| CU |	| CUF |	| FU |
1	AHEAD	24 316	6 175	38 556	517	1 055	59
2	BCJak2Java	17 521	4 466	17 562	502	611	15
3	Jak2Java	18 035	4 590	18 847	505	643	16
4	Jampack	19 299	4 974	21 216	501	733	21
5	JREName	16 595	4 315	15 971	498	576	17
6	Mixin	17 765	4 576	18 626	500	632	17
7	MMatrix	17 639	4 484	16 620	504	608	13
8	UnMixin	17 049	4 347	15 822	500	582	12
9	AJStats	13 226	1 232	5 895	13	49	20
10	Bali2Jak	7 539	1 369	3 972	135	158	11
11	Bali2JavaCC	8 082	1 420	4 151	138	164	11
12	Bali2Layer	7 835	1 420	3 912	137	159	12
13	Bali	9 939	1 600	5 321	141	183	18
14	BaliComposer	6 791	1 253	3 594	128	152	10
15	BerkeleyDB*	45 000	9 379	53 035	408	892	99
16	EPL*	111	46	83	5	15	12
17	GameOfLife*	1 461	267	624	37	55	15
18	GPL*	1 930	461	2 892	16	57	20
19	GUIDSL	10 084	2 144	7 556	144	287	26
20	MobileMedia8*	4 189	982	3 026	60	170	47
21	Notepad	891	153	369	8	22	10
22	PKJab*	3 373	689	1 738	51	68	8
23	Prevayler*	5 268	1 275	2 398	158	170	6
24	Raroscope	316	79	125	3	12	5
25	Sudoku	1 422	281	1 001	26	51	7
26	TankWar*	4 845	757	3 208	22	88	30
27	Violet*	7 151	1 033	2 535	67	157	88
28	ZipMe	3 446	717	1 711	32	46	13

SLOC: # source lines of code; |I|: # introductions; |R|: # references; |CU|: # classes/interfaces; |CUF|: # class fragments; |FU|: # Fuji feature modules; *: feature model available

feature orientation. Our primary source was the repository[1] of 28 Fuji-based product lines set up by the Fuji language maintainers, but developed also by others.

The product-line code bases differ in terms of source lines of code (SLOC[2]). While the smaller-sized code bases contain a few hundred to less than 2 000 SLOC, the medium- to larger-sized ones account for more than 6 000 to about 20 000 SLOC. The most extensive code base (BerkeleyDB) is of 45 000 SLOC.

Although the product lines are mostly of medium size and used in academic contexts, they have been developed by different developers for different purposes, and they differ in various aspects, such as the target domain. For a relatively young paradigm, such as feature

[2] http://www.dwheeler.com/sloccount/, last accessed: June 26, 2014.

orientation, this is the best one can hope for. We refer the reader to Section 5 for a discussion on how the selection of subjects threatens external validity.

By instrumenting Fuji's internal syntax representations, we collected the introductions and the references for each product line. We obtained sets between 9 379 (BerkeleyDB) and 46 introductions (EPL) as well as sets between 53 035 (BerkeleyDB) and 83 references (EPL). From the introduction sets, we computed the populations of code units (3 to 517 classes), of code-unit fragments (12 to 1 055 class fragments), and feature units (5 to 59 Fuji feature modules). The descriptive data are summarized in Table 1.

For our study, we selected specific subsets of the total references to construct the dependency graphs depending on the internal attribute measured. As for structural coupling, we included method-call and constructor-call references, which reflect the common strategy of developers to find dependent decomposition units and program elements by navigating the control flow of a program (Bouwers et al. 2011). Coupling measurement also included field accesses as critical kinds of coupling (Briand et al. 1999). For measuring structural cohesion, we excluded any references (method calls and field accesses) originating from within constructor bodies, because they risk introducing a systematic bias (e.g., through initializing most or all fields; Briand et al. 1998).

The references were obtained from Fuji's *family-based* internal syntax representation of each product line. In a family-based strategy, the code base of a product line is analyzed as a whole by composing all feature units in the syntax representation and by amending the syntax representation to contain the corresponding variability information (Kolesnikov et al. 2013b). We then included this variability information when building the dependency graphs. At the time of performing the measurement, the Fuji repository provided feature models for nine product lines (marked by '*' in Table 1). From these models, we extracted presence conditions. A *presence condition* indicates whether, for all valid configurations, two features are *always*, *sometimes*, or *never* present together. Then, we excluded all references running between two feature units which are not (*never*) included in any valid configuration.

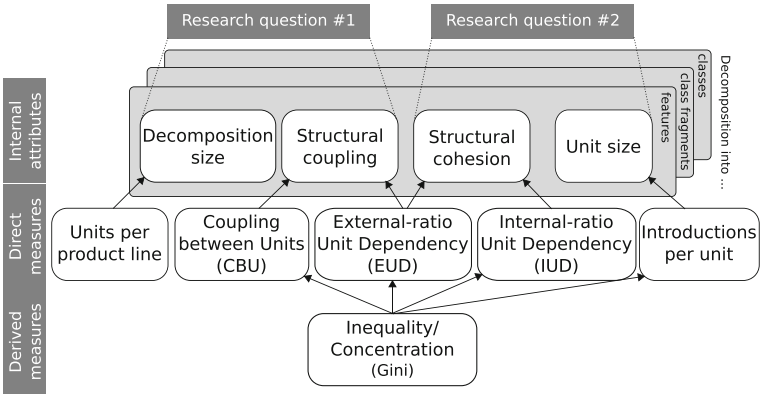


Fig. 5 Relationships between research questions, internal attributes of a product-line decomposition, indicator measures, and derived measures

3.3 Internal Attributes and Per-Unit Indicator Measures

As motivated in Section 2, we want to compare three decompositions of the 28 product lines regarding their decomposition sizes, the sizes of their decomposition units (class fragments, classes, feature modules), their structural coupling, and their structural cohesion (see Fig. 5). For these four internal attributes, we use five indicator measures: units per product line, coupling between units (CBU), external-ratio unit dependency (EUD), internal-ratio unit dependency (IUD), and introductions (program elements) per unit. The measure constructs are defined in terms of the underlying dependency graphs of the decompositions (see Section 3.1). These direct measure instantiations are then aggregated for each product line using derived measures (see Fig. 5).

Structural Coupling (Briand et al. 1999; Stevens et al. 1999) Coupling between decomposition units refers to the strength of association induced by structural references between two or more units. By taking design and implementation decisions that reduce (minimize) or increase the number of inter-unit references, coupling is said to be lowered or heightened, respectively. Structural coupling is measured by establishing the coupling between units in the dependency graph of a decomposition (see also Section 3.1): The *coupling between units* (CBU) measure collects the number of decomposition units that have direct dependencies to a given decomposition unit. This construct is defined as the number of decomposition units (a.k.a. couples) used by a given decomposition unit $u \in U$ in terms of structural references provided by these couple units to u (i.e., import coupling). In the dependency graph, $CBU(u)$ corresponds to the absolute out-degree of u.

Structural Cohesion (Briand et al. 1998; Stevens et al. 1999) Cohesion of decomposition units denotes the association strength between the program elements of a decomposition unit, established by intra-unit references. Assuming that a unit binds the program elements needed to fulfill a given function (code unit) or to implement a given feature (feature unit), an increasing (decreasing) number of intra-unit references indicates an improving (deteriorating) cohesion. We measure structural cohesion using *internal-ratio unit dependency* (IUD; Apel and Beyer 2011) which quantifies how interconnected the program elements of a decomposition unit are in terms of mutual import dependencies. This measure stands for the ratio of established (actual) references of a decomposition unit to the number of references that could potentially occur between all program elements of a decomposition unit. Self-references of an element are included.

To directly relate structural coupling and cohesion of a decomposition to each other in terms of references, we additionally compute the *external-ratio unit dependency* (EUD; Apel and Beyer 2011). The EUD measure captures to what extent a decomposition unit is self-contained in terms of two values: On the one hand, the number of import dependencies established internally between program elements contained by the decomposition is calculated. On the other hand, the external import dependencies between program elements of the decomposition unit and program elements of coupled decomposition units are counted. From these, the ratio of the number of actual references internal to a decomposition unit to the total (i.e., internal and external) number of actual references is computed.

Size To measure the *decomposition size*, we count the number of decomposition units observed in a given decomposition, that is, the number of class fragments, classes, and feature modules. The *unit size* of a decomposition unit is quantified by the number of introductions (program elements) of a decomposition unit. These two indicator measures allow

us to analyze the fragmentation of a product-line decomposition in terms of its scatteredness over decomposition units and the distinctiveness of the individual decomposition units. These two internal attributes and measurement points provide the analysis context for the two research questions on structural coupling and structural cohesion, respectively.

3.4 Aggregating Measures

From applying the five per-unit measures defined in Section 3.3 for each of the three decompositions, we obtain 15 direct measurements per decomposition unit and per product line. Given the number of decomposition units per product line (e.g., 1 051 class fragments for AHEAD), there are too many data points to permit a comparison of 28 product lines. Therefore, we apply data aggregation (Vasilescu et al. 2011) using concentration statistics. *Concentration* denotes how equal or how unequal values of a given measure (e.g., CBU) are distributed over the units of a decomposition (Vasilescu et al. 2011; Bouwers et al. 2011). This allows us to make statements such as "40 % of the import coupling of a product line is due to the bottom 60 % of decomposition units". This way, we can also address some of our motivating questions, for example, on the relative importance of certain decomposition units for coupling.

As a concentration statistic, we adopt the established Lorenz inequality (Kakwani 1980; Vasilescu et al. 2011). In our case, it relates the cumulative proportion of decomposition units in a product line ordered by ascending attribute value (e.g., from a low to a high CBU value) and the cumulative attribute value (e.g., cumulative sum of CBU values) measured for fractions of decomposition units. To summarize the Lorenz concentration between product lines, the Gini statistic G indicates the concentration degree.

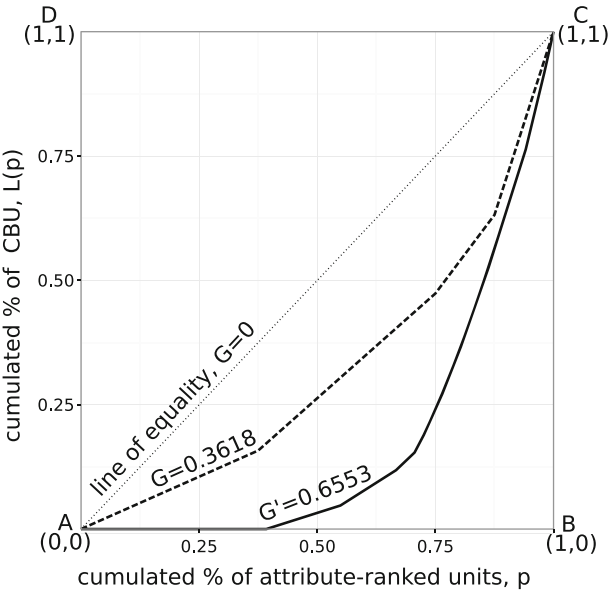


Fig. 6 Concentration degree and concentration symmetry of the per-class coupling (CBU per class, solid curve) and of the per-feature coupling (CBU per feature module, dashed curve) in PKJab; G,G': Gini coefficient

Visually, the Lorenz relationship can be depicted as a convex curve in a unit square ABCD, as shown in Fig. 6. On the x-axis, the cumulative proportion of decomposition units (or percentile p) not exceeding a specific attribute value x is plotted. On the y-axis, the corresponding cumulative share in the total sum of attribute values is printed. Each point of the Lorenz curve depicts a concentration. In Fig. 6, the Lorenz curve drawn using a solid line represents the concentration of CBU values in the class decomposition of the PKJab product line (as an example). At $x = 0.75$, the curve indicates that the bottom 75 % of classes are responsible for approx. 25 % of the summed CBU values. The inverse statement is equally valid: the top 25 % of classes have 75 % of the cumulative CBU sum.

The straight line of equality AC represents a reference at which each cumulative fraction of decomposition units (e.g., 60 % of classes in PKJab) is assigned a same-valued fraction in the cumulative attribute value (e.g., 60 % of the cumulative CBU values). In such a distribution, each decomposition unit has the same attribute value. A fully equal distribution would coincide with this line of equality. Conversely, any unequal distribution forms a convex curve under the line of equality. This is the case for both the per-class and per-feature CBU distributions plotted in Fig. 6.

Gini Coefficient (G; Vasilescu et al. 2011*)* This ratio represents the degree of distributional inequality (concentration) of an attribute among the decomposition units of a product line. The ratio takes a value between 0 and 1, with $G = 0$ denoting perfect equality: each unit fraction $n\%$ having a same-sized $n\%$ share in the cumulative attribute values, as found on the line of equality. This extreme implies that each decomposition unit has the same attribute value; for example, all classes having the same CBU value. Conversely, $G = 1$ denotes perfect inequality, with only one decomposition unit accounting for 100 % of the cumulative attribute value. This would signal a product line in which one class is responsible for all the import coupling in the product line. With $G = 0.3618$, coupling as measured by CBU is more equally distributed among the feature units of the PKJab product line than among its classes ($G = 0.6553$; see Fig. 6).

4 Study Results

Our analysis rests on two data sets. The first describes the underlying code bases of the product lines (see Section 3.2). The second data set represents the measurements obtained from applying the indicator and aggregation measures (see Sections 3.3 and 3.4). In Section 4.1, we characterize the 28 product lines based on the first data set (see also Table 1). This provides the context for a discussion of our observations based on the measurement data in Section 4.2.

4.1 Descriptive Analysis

Comparing the 28 product lines within the data set, most product lines are small in terms of SLOC, references, and introductions as well as decomposition units (classes, class fragments, and feature modules), as indicated by the density peaks at the left end of the x-axes on the diagonal of Fig. 7. From plotting and correlating the basic variables (e.g., references, introductions, SLOC) against each other, we learn the following from the scatter plots in the lower segment of Fig. 7:

Introductions and references are positively and quasi-linearly associated ($r = 0.969$). The more introductions in a product-line code base, the more references are recorded. This also

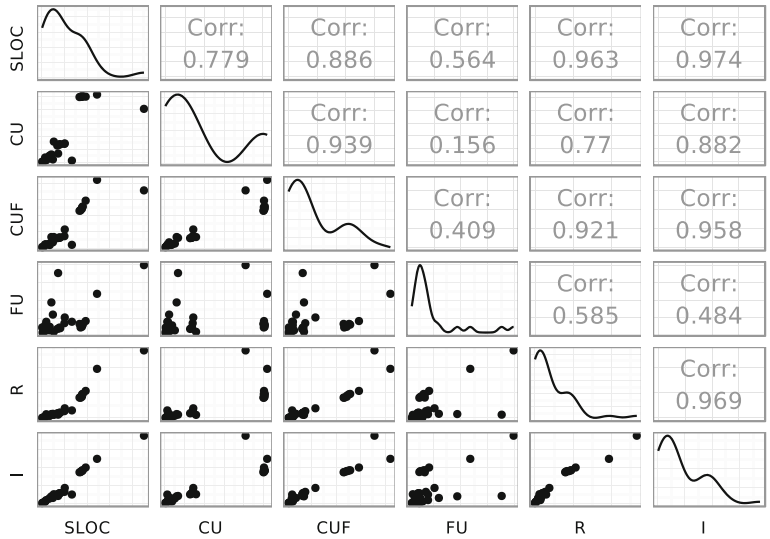


Fig. 7 Pairwise scatter plots (*lower segment*), Pearson correlation coefficients (*upper segment*), and density distributions (*diagonal*) for the 28 product lines; SLOC: source lines of code; CU: code units (classes); CUF: code-unit fragments (class fragments); FU: feature units; R: references, I: introductions

reflects the fact that there are only small shares of unreferenced (*dead*) introductions in all product lines. In 9 out of the 28 product lines, we find approximately 10–11 % of the total introductions that do not participate in any reference (viz., usage dependency between two introductions). The remaining 20 product lines exhibit much smaller shares. Note that, for measuring the internal attributes of decomposition size and unit size, dead introductions are considered. The coupling and cohesion measurement, however, explicitly excludes them by building on the observed references only.

The different sets of decomposition units appear related differently. On the one hand, the numbers of classes and class fragments are strongly and positively associated ($r = 0.939$). This results from the fact that class-fragment decompositions are directly dependent on the class decomposition in terms of the class count. Hence, product lines decomposed into a large (small) set of classes are also decomposed into a large (small) set of fragments. Moreover, this is an indicator for most feature modules being scattered over a majority of classes, resulting in an increase of per-feature class fragments depending on the number of classes present in a product-line code base. In contrast, the overall class counts and feature-unit counts are not associated. There are both product lines with comparatively many (few) feature modules and with few (many) classes ($r = 0.156$).

From the data sets on introductions and on references specific to each of the 28 product lines, we construct three dependency graphs per product line. Each dependency graph represents one decomposition of the product line: code units (CU), feature units (FU), and code-unit fragments (CUF; see Section 3.1). The node sizes (viz., the order) of the resulting dependency graphs correspond to the number of units in each of the decompositions (see Table 1). These decomposition sizes are plotted for the 28 product lines, for every decomposition, along the x-axes in Fig. 8. The number of edges connecting the decomposition units varies depending on the number of references between units and the aggregation

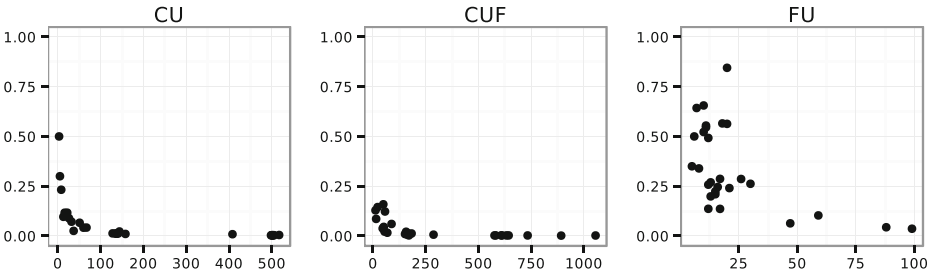


Fig. 8 Decomposition size (*x-axis*) and connectedness (*y-axis*) for the 28 product lines

of introductions (as source and target points of a reference) into decomposition units (see Section 3.1).

The resulting dependency graphs exhibit different degrees of connectedness (density), that is, the ratio of actually observed to the number of potential edges in the dependency graphs. The scatter plots in Fig. 8 contrast the connectedness (on the y-axes) to the decomposition size (on the x-axes). Generally, one might suspect that the more (fewer) units a product line is divided into, the more (fewer) units can become interconnected by references, potentially. The above observation that introductions and references are positively associated therefore suggests that larger decompositions (having many introductions) are more interconnected (having more references between units) than smaller ones.

This intuition is not supported, however. We find that decompositions (CU, CUF, and FU) containing comparatively small number of decomposition units are generally more densely connected than larger decompositions. This is indicated by the left-to-right, downward-sloping connectedness (y-axes) with increasing decomposition sizes (x-axes) in Fig. 8. Despite being the smallest decompositions, feature decompositions are more densely connected than decompositions based on classes and class fragments (see the rightmost scatter plot in Fig. 8). A major share in feature decompositions (17/28) have densities above 0.25 (i.e., more than 25 % of their potential references are realized). The majorities of the class and class-fragment decompositions account for less than 0.1.

4.2 Observations

Exploring the measurement data beyond the descriptive statistics on each structural attribute, we made five observations (O.1–O.5). We base these observations on boxplot and concentration statistics. The full detail on the statistical analyses, the underlying measurement data, and instructions on how to obtain and to reproduce our analyses are reported in Sobernig et al. (2014). Below, we iterate over each observation in isolation. In Section 4.3, we relate the observations to each other.

O.1—*Feature decompositions contain fewer uncoupled and fewer extremely high import-coupled feature units than class- and class-fragment decompositions.*

Typical CBU levels observed for the three decompositions of the 28 product lines fall into ranges of median CBU values of 2–4 for the median 60 %±14.7 classes, 1–9 for the median 65 %±16.2 class fragments, and 2–18 for the median 95 %±8.7 features. These typical CBU levels can be read as, for example, each of the median 60 %±14.7 classes relying on fields or methods introduced by between two and four other classes.

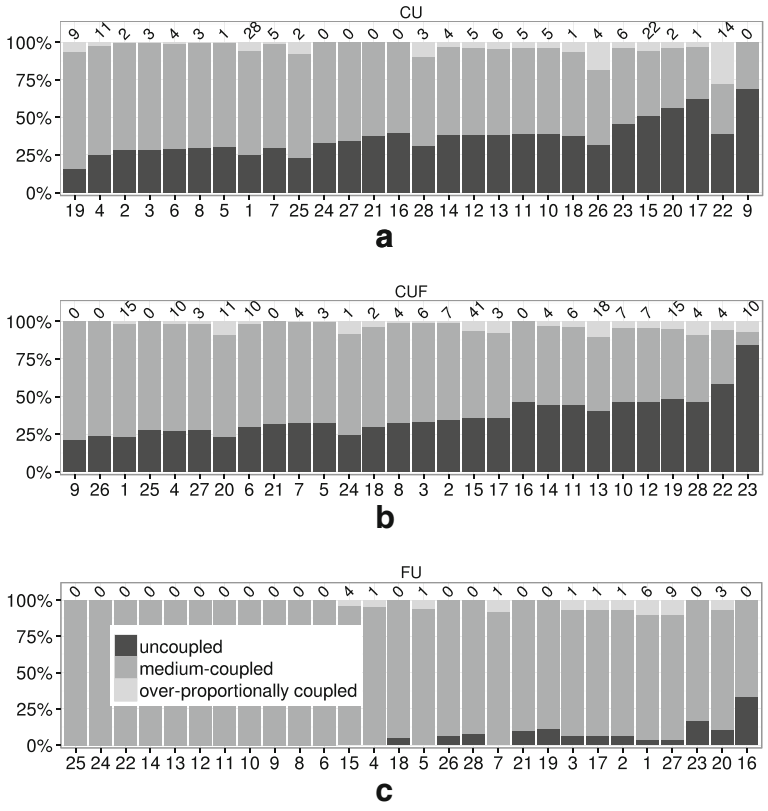


Fig. 9 Each stacked barplot relates the number of uncoupled units (CBU = 0, *isolates*), the number of over-proportionally coupled units (upper z-score outliers > 3.5), and the medium-coupled units as percentage shares (*y-axes*) per product line (*x-axes*). See Table 1 for the product-line identifiers (1–28). The *stacked bars* are ordered by the decreasing share of medium-coupled units per product line from left (*high share*) to right (*low share*). On top of each stacked bar, the absolute number of over-proportionally coupled units per product line is printed

CBU is distributed very differently between classes as well as between class fragments, on the one hand, and features, on the other hand. In both the class and class-fragment decompositions, there is a considerable number of uncoupled decomposition units (a.k.a. isolates). Approximately a median of 36 %±8.9[3] of the classes (see Fig. 9a) and a median of 34 %±13.8 of the class fragments (see Fig. 9b) are not import-coupled at all. In the feature decompositions, we did not find a single uncoupled feature in 15 out of the 28 product lines. In the remaining 13, there is only a minority share of uncoupled features (on avg. 5 %, with a maximum of 33.3 %; see Fig. 9c).

Feature decompositions contain features of over-proportionally high CBU values (outliers). In numbers, 10 product lines contain between 1 and 9 over-proportionally coupled feature units (see Fig. 9c). However, more outliers can be observed for classes and class

[3]We report the variance in terms of the *median absolute deviation from the median* (MADM) using the ± notation along with the median value.

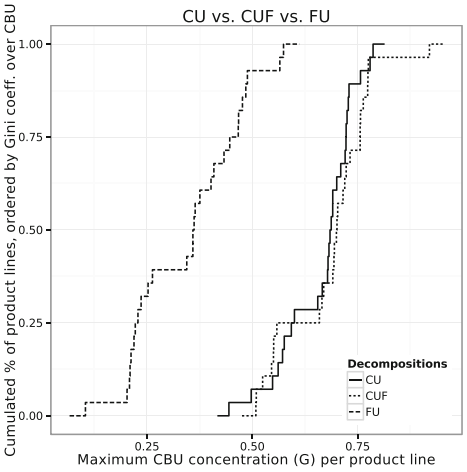
fragments (see Fig. 9a and b). This is in sharp contrast to observing that 18 feature decompositions do not have any import-coupling (CBU) outliers at all (see Fig. 9c). The class and the class-fragment decompositions report comparably small numbers of over-proportionally import-coupled classes (median 3.5±3.7 classes per product line) and class fragments (median 4±4.5 class fragments), respectively.

These differences in the number of isolates and the numbers of upper outliers help explain the observation on the differences in terms of connectedness (see Section 4.1), because isolates and outliers indicate the number of realized edges (i.e., extracted references between program elements) in the dependency graphs.

O.2—*The numbers of feature units providing a required reference target to a given feature unit (CBU) are more equally distributed between feature units than between classes and between class fragments.*

In their feature decompositions, the product lines—having a median Gini coefficient of 0.36±0.17—are less concentrated in their CBU distribution than in the other two decompositions. This is indicated by the FU curve in Fig. 10a clearly running left from the two other curves. The class and class-fragment decompositions show a strong and similar concentration pattern in their CBU distributions, as indicated by their co-running curves in Fig. 10a. All decompositions into classes and class fragments have concentration levels of more than 0.4 (median 0.69±0.06) and 0.5 (median 0.7±0.08), respectively.

When comparing the feature and class decompositions of each product line, the above observation between decompositions *across* all product lines is supported. CBU levels per unit are *less* concentrated (less equal) in the feature decomposition than in the class decomposition of a product line ($G_{FU} < G_{CU}$): In 26 out of 28 product lines, the CBU levels per



a

	CU	CUF	FU
Bottom 20%	0	0	10.53
20–40%	0.59	0	5.26
40–60%	7.69	1.35	21.05
60–80%	28.40	21.62	10.53
Top 20%	63.31	77.03	52.63
G	0.66	0.76	0.36
SE(G)	0.0464	0.0410	0.1420
S	0.8655	0.9023	1.3298

b

Fig. 10 **a** Three cumulative distribution curves (CDF), each representing one decomposition (CU: *solid curve*, CUF: *dotted curve*, FU: *dashed curve*). Each distribution curve relates a cumulated share of the 28 product lines (per decomposition, ordered by increasing CBU concentration) and a maximum CBU concentration value (G, Gini coeff.) observed for a given subset of product lines. See Sobernig et al. (2014) for the corresponding data set. **b** PKJab: Aggregated CBU shares hold by same-sized groups (quintiles) of decomposition units, ordered by increasing CBU; see the corresponding Lorenz curves in Fig. 6, Section 3.4; G: Gini coefficient; SE(G): Jackknife estimate of Gini standard error; S: Lorenz Asymmetry Coefficient. Please refer to Sobernig et al. (2014) for definitions of SE(G) and S

class are more unequally concentrated in the class decomposition than the CBU levels per feature unit. PKJab (as one out of the 26 product lines) exemplifies this difference between its two decompositions (see Fig. 10b). In the class decomposition of PKJab, the top 20 % group of most import-coupled units accounts for approx. 63 %, the bottom 40 % for less than 1 % of the total per-unit coupling (approximated by summing the per-unit CBU values). In the feature decomposition, however, the lower 4/5 of (lowly import-coupled) features take a comparatively greater share of 47 %, similar to the top 20 % features (53 %).

O.3—*Feature units are internally less connected units than classes. Generally, the internal connectedness of all decomposition-unit kinds is limited.*

An incohesive decomposition unit does not have a single internal reference realized (IUD = 0). To these decomposition units, all dependencies are provided externally. Feature decompositions give rise to fewer incohesive units (median 13 %±20.2) than class (median 38 %±6.6) and class-fragment decompositions (median 43 %±8.3).

As for over-proportionally cohesive units,[4] there are only very few units of over-proportionally high IUD in class decompositions (1 or 2 at most, with BerkeleyDB being an exception having 7 outliers) *and* feature decompositions (median 7 %±10.6). Class-fragment decompositions, conversely, exhibit large portions of over-proportionally high IUD values (median 13.1 %±18.5).

Between the extremes of incohesive and of overly cohesive units, we find median IUD levels of 0.03–0.07 for classes (maxima of up to 0.16), of 0.002–0.06 for class fragments (max. of 0.11), and of below the 1 % mark to 0.06 for features (max. of 0.17). These IUD levels apply to the median 61 %±6.2 of the classes, the median 40 %±22.3 of the class fragments, and the median 80 %±28.2 of the feature units, respectively.

O.4—*The per-unit cohesion in class and feature decompositions settles at medium levels of concentration. In class-fragment decompositions, IUD levels are distributed most unequally.*

The IUD concentration levels found for class fragments are the most pronounced among the three alternative decompositions, having Gini coefficients between 0.5 and 0.82 for all product lines. The IUD concentration among class fragments in 26 product lines exceeds the concentration among their classes (by a median difference in Gini coefficients by 0.12±0.04), in 21 product lines the concentration among the features (median Gini delta of 0.16±0.11). This is explained by the presence of comparatively high numbers of both incohesive class fragments and over-proportionally cohesive class fragments leading to an unbalanced distribution of IUD levels over class fragments.

The IUD distributions of class and feature decompositions are less concentrated, at IUD levels comparable to each other (median Gini coefficients of 0.52±0.12 and 0.53±0.2, respectively), than those of class-fragment decompositions (see above). A direct comparison of class and feature decompositions of each product line leaves a mixed picture, however: 10 product lines have more concentrated IUD distributions over features than classes (median Gini difference of 0.14±0.08), for 12 product lines it is the inverse (0.08±0.05). In the remainder of 6 product lines, IUD concentrations among features and classes are similar.

[4]We apply a standard technique of outlier identification: modified z-scores (Iglewicz and Hoaglin 1993).

O.5—*Features have comparatively higher shares in internal than external references. The extent to which a feature unit is self-contained (EUD) is more equally distributed among feature units than among classes or class fragments.*

In 24 out of 28 feature decompositions, more than 50 % of the references run *within* feature units (median 72 %±6.2 of total references). Similarly, class decompositions also exhibit a considerable share of internal references, though at a slightly lower level (median 57 %±9.4). Class-fragment decompositions have more externally than internally running references (median 43 %±11.4).

For the less-concentrated 75 % of the decompositions, feature decompositions have a more equal distribution of EUD levels (Gini coefficients ranging from 0.13 to 0.766). Class decompositions follow second (0.26–0.72), and class-fragment decompositions are the most concentrated ones (0.37–0.82). For the upper 25 % of the most concentrated decompositions, concentration converges to similar Gini levels.

4.3 Discussion

Research question #1: How do coupling structures of a feature-oriented software product line differ between its decompositions into *classes*, *features*, and feature-specific *class fragments*?

A cross-reading of our observations on coupling (CBU; see O.1 and O.2 in Table 2) tells us that **(1)** feature orientation is not necessarily associated with a more loosely coupled decomposition. Rather, the inverse holds: Feature modules are organized in more dense coupling structures than classes and class fragments. This adds to the similar finding on feature cohesion by Apel and Beyer (2011). To be precise, there appears to be an *inverse association*

Table 2 Summary of key observations on the three decompositions reported in Section 4.2

Measure		Decomposition kind			Attribute
		CU	FU	CUF	
CBU	O.1	low CBU density, low midrange CBU, few outliers, many isolates	medium/high CBU density, high midrange CBU, very few outliers, few isolates	low CBU density, medium midrange CBU, very few outliers, few isolates	RQ#1: Structural coupling
	O.2	mixed, medium/high CBU concentration	uniform, low/medium CBU concentration	mixed, medium/high CBU concentration	
EUD	O.5	medium EUD level; mixed, medium EUD concentration	medium/high EUD level; mixed, low/medium EUD concentration	low EUD level; mixed, medium/high EUD concentration	
IUD	O.3	low midrange IUD, very few outliers, many incohesives	very low midrange IUD, very few outliers, few incohesives	low midrange IUD, many outliers, many incohesives	RQ#2: Structural cohesion
	O.4	mixed, medium IUD concentration	mixed, medium IUD concentration	uniform, medium/high IUD concentration	

between decomposition size (i.e., the number of classes, class fragments, or feature modules) and the degree of import coupling (O.1; see Fig. 8). Regardless of the decomposition kind (CU, CUF, or FU), a product line decomposed into fewer units shows a comparatively higher level of import coupling than larger decompositions. Hence, the coupling structures of the three decompositions are similar to each other. More decomposition units in a product line do not necessarily correlate with higher import coupling between these units.

This leads to another important insight which tells us that **(2)** the coupling complexity of the class and class-fragment decompositions appear to scale better with increasing decomposition sizes. As for the coupling drivers, in class and class-fragment decompositions, the overall coupledness observed in dependence of the decomposition size is more related to the presence of coupling-free units (see O.1). For classes and class fragments, Fig. 11 illustrates this strong positive and quasi-linear association between decomposition size and uncoupled decomposition units (CBU = 0). The more units in a decomposition, the more are uncoupled. As a result, large class and class-fragment decompositions do not necessarily pair with dense coupling structures.

Feature decompositions are more interconnected, reaching up to and beyond 50 % of the potential coupling relationships actually realized, already for smaller-sized feature decompositions (of up to 25 feature modules). In feature decompositions, however, the observed association between decomposition size and non-coupled units is missing: **(3)** An increased total number of feature modules is not associated with an increased number of coupling-free feature modules. Rather, the larger a feature decomposition is in terms of feature modules, the more is the measured coupledness associated with feature modules of a small import-coupling degree: Most feature modules in these large feature decompositions tend towards a range between 2 and 6 coupled units per feature module. This holds even for the largest product lines in our sample (BerkeleyDB, Violet, AHEAD, and MobileMedia8). More generally, for most product lines, a major share in units of either decomposition does not exceed the CBU value of 5 or 6 (see O.1).

Furthermore, we found that **(4)** per-unit coupling (CBU) in class and class-fragment decompositions follows right-skewed distributions, for all product lines (O.2). This holds also for some feature decompositions (see AHEAD in Fig. 12a). In such a right, or positively, skewed distribution, the largest number of decomposition units takes a comparatively small CBU value. For AHEAD, most feature modules take a CBU value of less than 20 (see Fig. 12a). At the same time, there is a small number of extreme outliers having an over-proportional CBU. In AHEAD, for example, there is one feature module (`BaliJavaParser`) that is import-coupled to 56 out of 59 feature modules in this product line. These outliers represent candidate provider or hot-spot features.

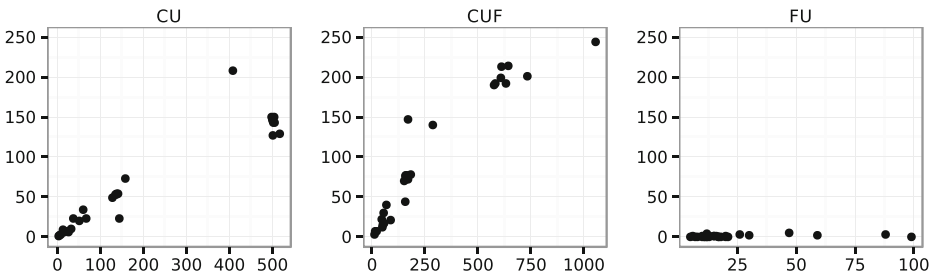


Fig. 11 Decomposition size (*x-axis*) and the number of units without any import coupling (*y-axis*; CBU = 0) for the 28 product lines

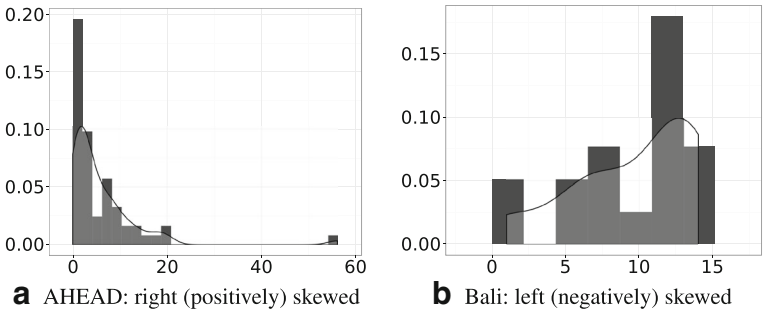


Fig. 12 Distributions of CBU for feature modules (histogram and estimated density plot)

However, **(5)** feature decompositions are described by *both* right- and left-skewed coupling distributions. The Bali product line in Fig. 12b is such an example. In this product line, a medium to large number of comparatively high-coupled feature modules can be found. This is in line with the related observation that feature modules appear to be more import-coupled than classes and class fragments (O.1). Furthermore, most feature modules take a more equal share in import coupling than typical classes and class fragments (O.2). This means that there are not necessarily extreme outliers of either high or low coupling. The dominance of highly-coupled feature modules and the lack of outliers requires us to reassess some working assumptions on provider and hot-spot features, originally established for non-functional properties (Siegmund et al. 2012), when it comes to the code structure of product lines. For example, product-sampling strategies based on the general assumption of hot spots should be only applied when the product line is tested positively for their existence. More generally, we must conclude that related empirical findings on coupling in object orientation (Taube-Schock et al. 2011) cannot be easily applied to coupling in feature orientation.

Research question #2: Do features as decomposition units form cohesive units of functionality? How do they compare with classes and class fragments in terms of cohesion?

Apel and Beyer (2011) found that feature modules depend mostly on their own elements, rather than on elements of other features. We can confirm this observation based on our reference data. Feature units tend to be self-contained in terms of self-provided dependencies, in particular, when compared to classes and class fragments (O.5). In addition, we show that feature modules appear to be more similar in terms of their self-reliance than classes and class fragments: Between classes and between class fragments, we find more varying levels of self-containment (O.5).

While Apel and Beyer (2011) report on product lines taking values in the entire spectrum of internal connectedness (i.e., number of observed to possible internal references) of feature modules (IUD), based on our reference data, we find that there is a generally low degree of internal connectedness regardless of the kind of decomposition. The maximum IUD value reported is 0.86 (MobileMedia8; classes), with most other product lines reporting maximal IUD values below 0.25 (especially for features and class fragments). Note that the different observations are not necessarily conflicting. Our observations are based on actual, type-level reference data, while Apel and Beyer (2011) used dependency graphs derived from data on program elements introduced by feature implementations. In addition, we

learn that the constituent elements of classes and feature modules are comparatively more connected than elements of class fragments (O.3). Note, however, that feature modules are not necessarily more densely connected, internally. While settling at low levels, generally, the internal connectedness is more equal between feature modules and between classes than between class fragments (O.4).

5 Threats to Validity

Construct Validity The definitions of the measures (CBU, IUD, EUD) deviate from their originating research designs. A major particularity of our setup are deviating notions of dependency: The references, on which the measure instantiations are computed, include all the variation of a product line, that is, they reflect all possible configurations. Measurements based on product-line references are different from measurements on the references found in a single product. The latter is the originating usage context of CBU. However, as we compare projects at the product-line level only, distortions from different reference kinds (product line vs. product) are excluded. Another difference stems from different sources of creating the underlying dependency graphs: The IUD and EUD measures of Apel and Beyer (2011) have been devised and applied to dependencies built from introduction data. As a result, the computed dependencies could not indicate any direction of dependency (source vs. target). In contrast, our references correspond to references as established and maintained by the Fuji/Java type system, including direction. As we apply the direction-aware IUD and EUD measures uniformly on each code base, all measurements would be similarly affected voiding any confounding effects in our comparative analysis.

A further threat to validity is the assumption of direct dependencies. In our analysis, a dependency is established between two elements iff there is a direct reference (e.g., a direct method call) between the two. Transitive references between two elements do not give rise to an indirect dependency. This notion of direct dependency, while intuitive and backed by literature on software measures (Briand et al. 1999), is linked to the unit size. In decomposition units large in element numbers (feature modules), cohesion measurement based on direct dependencies may understate the cohesiveness of a unit. Conversely, cohesiveness of smaller units may be overstated. As this assumption is uniformly applied in our comparative analysis, it can limit, if at all, only the generalizability of our observations.

Finally, we consider only unique references between program elements. To include multiple, recurring references, edge-weighted dependency graphs would be required as representation. However, any weighting risks introducing an ambiguous qualification, depending on the weighting strategy. Empirical evidence on appropriate weighting strategies (e.g., reference frequencies, unique source/target pairs) and on their representation condition is missing. Furthermore, we build upon indicator measures defined for unweighted dependency graphs (Apel and Beyer 2011; see also Section 3.3).

Using a family-based syntax representation to extract references (Kolesnikov et al. 2013b), on the one hand, we abstract from the heterogeneity in structural feature-model complexity of product lines (e.g., the number of leaf features and of optional features). On the other hand, a family-based strategy risks recording references which can never realize in any valid configuration of a product line. To reduce this risk, we incorporated presence conditions derived from the feature models of the product lines to exclude any unrealizable references (see Section 3.2). However, at the time of performing the study, validated feature models were only available for 9 of the 28 product lines (see also

Table 1). For the remainder of 19 product lines, the number of collected references may therefore be overstated.

Internal Validity Our analysis design and procedure could have caused our observations not to follow directly from the data collected (i.e., the code bases implementing the product lines and processed to obtain the reference data). To begin with, data pre-processing, statistical analysis, and visualization are performed by approximately 3000 lines of R code. To control this threat, critical steps, such as parsing reference data as provided by Fuji into R data sets, building data subsets (e.g., by filtering based on presence conditions), and the implementation of measure constructs (e.g., CBU, IUD), while developed by one author, have been reviewed by a second author.

To avoid bias introduced by heterogeneous code bases written in different feature-oriented programming languages, we deliberately limited ourselves to the product lines available from the Fuji repository. Furthermore, the product lines in this repository have been developed in various contexts, by different developers (e.g., to exclude learning effects), and for different application domains. To assess the internal attributes (e.g., cohesion, coupling), we always devised several—mostly two—measure constructs. This way, we mitigate the risk of single measures being influenced by an unknown variable. For aggregating the direct measurements for each product line, we employed established statistical techniques (Vasilescu et al. 2011).

External Validity As to be expected for this kind of study, the selection of 28 subject product-lines threatens external validity. This is because, first, our analysis design remains exploratory by nature: We interpreted quantitative observations in the light of conjectures on different decompositions of feature-oriented product lines found in the current state of literature. As usual for an exploratory study, these interpretations remain to be tested and confirmed in a controlled setting. Second, our analysis is based on a single and coherent sample of product lines to increase internal validity. To mitigate the threats to external validity, we made sure that the product lines stem from distinct domains (e.g., gaming, DBMS, model authoring), and have not been developed for the purpose of this study. While the sample does not allow for transferring our observations to product lines developed in alternative implementation techniques (code pre-processors, plug-in frameworks), it overlaps widely with samples used in earlier studies on feature-oriented product lines (Apel and Beyer 2011; Siegmund et al. 2011; Apel et al. 2013b). With our study, we provide an important prerequisite to perform meta-studies generalizing over the individual study findings in follow-up work. In addition, our analysis design (including the statistical tooling) is repeatable for other product-line code bases.

Finally, we would like to comment on the role of feature orientation in the practice of product-line engineering, which threatens external validity. Clearly, feature orientation is a comparatively novel development paradigm, which did not see broad industrial adoption, so far. Certainly, this is partly due to its novelty (companies must adopt to a new way of thinking and new tools), but also due to the lack of evidence and understanding of the paradigm in action. This is exactly were our study comes into play: It provides insights into the nature of feature orientation, shall guide further developments in the area, and ultimately facilitate and guide industrial adoption. So, the study is not meant to provide results that are immediately transferable to product lines that are developed right now in industry. In this sense, the extent to which we can transfer our results to current industrial product lines is clearly limited, but still, the overarching goal of better understanding the merits of feature modularity has been reached.

6 Perspectives

Our approach of analyzing different product-line decompositions using concentration statistics and the corresponding empirical results have immediate and profound implications on research areas relying on static and dynamic program analyses of feature-oriented product lines: variability-aware type checking (Apel et al. 2010; Kästner et al. 2012; Kolesnikov et al. 2013b) and detection of non-functional feature interactions (Siegmund et al. 2013).

6.1 Static Program Analysis: Product-Line Type Checking

Type-checking and other static program analysis techniques for feature-oriented product lines, and for variable programs in general, are challenging because a whole family of programs must be analyzed. This increases the effort substantially and requires intelligent techniques to acquire type-checking results in reasonable time (Apel et al. 2010; Kästner et al. 2012; Kolesnikov et al. 2013b). Approaches to this variability challenge involve checking every possible program variant (product), incorporating variability to examine all program variants at once (Thüm et al. 2014), and applying *product sampling*.

Sampling aims at systematically selecting a subset of variants of a feature-oriented product line for analysis according to a sampling criterion. Such a sampling considered advantageous because it overcomes computation barriers inherent to product-based approaches and because it allows for reusing existing, variability-unaware analysis tooling. However, the choice of a sampling strategy (especially, a sampling criterion) presents a challenge on its own: Oster proposed a method based on pairwise sampling (Oster et al. 2010). The approach finds a minimal set of configurations, in which all pairwise combinations of features are present. However, software defects can involve up to 14 features (Garvin and Cohen 2011). Thus, checking only feature pairs may not be sufficient. The appropriate way of grouping (sampling) code units according to indicator measures (e.g., coupling measures) of internal attributes (e.g., structural coupling) is a field of active research on its own (Shatnawi et al. 2010; Ferreira et al. 2012). Any sampling strategy must be evaluated regarding its efficiency and its effectiveness in terms of the number of errors found in relation to the invested detection effort (Apel et al. 2013d).

Measuring structural coupling based on CBU concentration provides an alternative sampling criterion, yielding variant samples beyond feature pairs. By grouping code units according to their structural coupling as indicated by CBU, one can select a portion of decomposition units likely to be responsible for type-checking defects. This assumption is backed by strong evidence that structural coupling is a suitable indicator for defect probability (Emam et al. 2001; D'Ambros et al. 2010).

We ran the Fuji type checker (Kolesnikov et al. 2013b) for 13 product lines of the 28 Fuji product lines (see Section 3.2) — the ones providing a corresponding feature model. Actual type-checking errors were reported for 7 product lines (see Table 3). We then extracted the decomposition units (classes and Fuji feature modules) involved in the reported type-check errors from the available type-checking trace data. The retrieved decomposition units of each product line (classes, features, and class fragments) are then ranked according to their CBU values (i.e., their CBU *rank* in ascending order). Then, this sorted list of decomposition units were split into several, approximately equally sized groups based on purely statistical grouping using quintiles. Thus, the first group contains the bottom 20 % units having the *lowest* CBU and the fifth group contains the top 20 % units of *highest* CBU.

For each product line and for each decomposition kind, Table 3 shows the 5 resulting groups of CBU-ranked decomposition units and their corresponding error counts. We find

Table 3 Results of a feasibility study on the potentials of sample-based type-checking. For each product line and each decomposition type (feature, class fragment, class), the table presents five groups of decomposition units and their corresponding error counts. Each data row represents a group containing approx. 20 % of the decomposition units, ranked by increasing CBU

Decomposition unit	Group	Subject product line (type-checking error count)						
		BerkeleyDB	EPL	GPL	GUIDSL	Prevayler	Sudoku	Violet
Feature								
	Bottom 20 %	3	16	17	0	0	0	30
	20–40 %	6	4	0	2	4	0	4
	40–60 %	14	1	0	0	10	2	4
	60–80 %	2	4	22	151	0	11	87
	Top 20 %	176	38	2	1	1	4	0
Class fragment								
	Bottom 20 %	3	8	0	5	0	0	8
	20–40 %	6	2	4	5	0	2	14
	40–60 %	13	10	17	15	0	3	76
	60–80 %	50	5	10	56	0	1	27
	Top 20 %	129	38	10	73	15	11	0
Class								
	Bottom 20 %	0	2	0	5	0	1	12
	20–40 %	0	0	0	5	0	0	0
	40–60 %	0	9	0	0	0	2	16
	60–80 %	12	14	21	74	3	2	48
	Top 20 %	189	38	20	70	12	12	49

that the top-40 % of the most coupled classes are responsible for considerable portions of type-checking errors, ranging from 78 % (Violet) to even 100 % (BerkeleyDB, GPL, Prevayler). If most errors in these product lines were concentrated in highly coupled classes, then a sampling of this portion of classes would facilitate type-checking tasks considerably. Note that this observation—while promising—is only tentative and requires further investigation (see Section 8).

6.2 Dynamic Program Analysis: Feature-Interaction Detection

Dynamic program analysis requires to actually execute the program to examine it. An important area in this domain is the detection of non-functional feature interactions. Detecting feature interactions that influence external attributes, such as time/space efficiency and energy consumption, is of paramount importance, as they can lead to generating non-optimal program variants, violations against quality-of-service requirements, unsatisfactory user experience, and so forth (Siegmund et al. 2012).

Feature-interaction detection is tremendously costly in the product-line context because we can run only one program variant at a time, and there may be exponentially many of

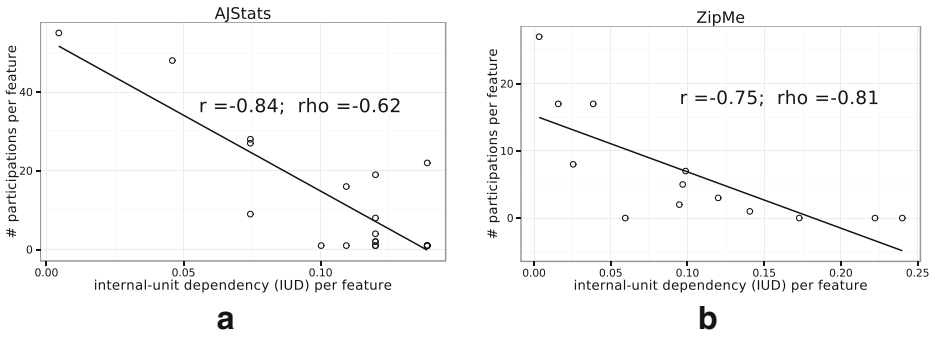


Fig. 13 Plotting the static measurement of internal-unit dependency (IUD; *x-axis*) per feature versus the dynamic measurement of interaction participations (*y-axis*); r: Pearson correlation coefficient; rho: Spearman rank correlation coefficient

them (Apel et al. 2013a). Furthermore, unlike static program analysis, it is particularly difficult to make a dynamic program analysis aware of variability. The reason is that one needs a running, operative program, which in turn represents only a single valid configuration of the product line. There are ambitions to execute the whole family of programs simultaneously (Siegmund et al. 2013; Nguyen et al. 2014), it yet has to be shown whether this is possible for different product-line implementation techniques. Therefore, *product sampling* is, at the current state of the art, the only practical solution to enable dynamic program analysis for product lines. The testing community developed several sampling approaches under the umbrella of combinatorial testing using covering arrays which can be employed in the product-line context to test t-way feature interactions using a minimal number of tests (Yilmaz et al. 2006). Given that software defects can involve up to 14 features (Garvin and Cohen 2011), combinatorial testing can nevertheless result in a huge number of required tests.

To drive a sampling strategy, finding and validating indicators for possible feature interactions derived from static program analyses are critical (Apel et al. 2013c). We performed a feasibility study to explore whether a prior analysis of structural cohesion (IUD) has potential to predict performance feature interactions by integrating data sets generated independently from two different studies. On the one hand, in previous work, we collected data on external performance feature interactions in terms of method execution timings for the product lines AJStats and ZipMe (Siegmund et al. 2013). Using a variant simulator (Apel et al. 2011), we traced method executions between the originating (calling) feature of the method call and the (called) feature introducing the method implementation. This way, we counted for each feature how often it participates in performance-critical method calls and what quantity of execution time it consumed. On the other hand, from our main study in this paper, we took the corresponding cohesion (IUD) measurements for these two product lines (see Section 4).

To explore whether the two quantities (feature interactions, IUD/EUD) show an association, we applied a correlation analysis using the Pearson (`r`) and the Spearman rank correlation (`rho`) coefficients (see Fig. 13a for AJStats and Fig. 13b for ZipMe). Each data point represents a single feature of the respective product line. The higher along the y-axis, the more often the corresponding feature has participated in an performance-critical interaction. The more positioned to the right, along the x-axis, the higher the cohesion of the feature.

We observe that a feature participating in many performance interactions (method calls) is of a comparatively weak cohesion. Conversely, features of high IUD do not contribute substantial interaction counts. This observation becomes manifest as a strong negative association between feature-interaction counts and IUD per feature. As a tentative result, we can make the conjecture that cohesive units (as measured by IUD) might be omitted when drawing a variant sample, therefore substantially narrowing down the search space in a sampling strategy.

7 Related Work

Throughout the paper we already discussed related work on quantifying the internal attributes of different system decompositions using component orientation (Bouwers et al. 2011), object orientation (Sarkar et al. 2008), and feature orientation (Apel and Beyer 2011).

To this date, research on quality attributes of product lines has already seen some development. Montagud et al. (2012) identified 97 different process-, resource-, and structure-related quality attributes of product lines that have been investigated by published research. These include both quality attributes characteristic to product lines (e.g., variability, reusability) and more generic ones (reliability, time/space efficiency). It is noteworthy that none of the research papers screened by Montagud et al. (2012) provides a systematic comparison of different product-line decompositions regarding internal attributes, as delivered in this paper. In the following, we concentrate on related work that emphasizes the internal attributes of structural coupling and structural cohesion.

7.1 Software Measures for Software Product Lines

In a systematically sampled corpus of 35 research papers, Montagud et al. (2012) found 165 distinct software measures applied on product lines. 144 out of these (i.e., 87 %) quantify internal attributes of various product-line artifacts, such as the code assets and the variability model. In the following, we iterate over a selection of research contributions with emphasis on measurement constructs on code assets and specific to three key implementation techniques for feature-oriented product lines: feature-oriented programming (FOP), aspect-oriented programming (AOP), and annotation-based or preprocessor-based techniques (ANN); see Table 4 for an overview.

In a longitudinal study on a commercial product line in the telecommunications sector, Ajila and Dumitrescu (2007) collected data quarterly on the product-line size, the product size, and the code-churn size in terms of LOC over a period of several years and several release cycles. In addition, they recorded the number of modules, each implementing a feature. The data entered an analysis of quarterly growth rates in the product-line size, among others. We do not consider LOC-based constructs in our research design because they do not qualify as an indicator measure for structural coupling and structural cohesion. However, we included SLOC counts to document the sizes of the 28 Fuji product lines.

In earlier work, Sobernig (2010) explored means to quantify dependency structures between feature units based on code-level dependencies. The approach puts forth the abstraction of *feature interaction networks* and the computation of network-statistical measures over those dependency networks. Measure constructs are scattering (node degree), scatteredness (density), and scattering concentration (degree concentration). When analyzing the overall coupling structure in the three decompositions, we resorted to a density-based analysis in Section 4.1.

Table 4 Overview of related work on software measures for product lines (in chronological order); FOP: feature-oriented programming; AOP: aspect-oriented programming; ANN: annotation; 1-mode: measurement of element-element or feature-feature relations; 2-mode: measurement of feature-element relations; local scope: measures describe the condition of one program element or feature; global scope: measures describe the condition of the entire system or a subsystem

	Implementation technique			Units			Scope		Measurement constructs
	FOP	AOP	ANN	1-mode: element XOR feature	2-mode: element AND feature	other	local	global	
Wong et al. (2000)			✓		✓		✓		Disparity, Concentration, Dedication
Sant'Anna et al. (2003)		✓	✓		✓	LOC	✓		CDC, CDO, CDLOC
Ajila and Dumitrescu (2007)	✓	✓	✓	✓		LOC		✓	lines count, code churn, module count
Eaddy et al. (2007) and Eaddy et al. (2008)			✓		✓			✓	DoS, DoF, DoT
Lopez-Herrejon and Apel (2007)	✓	✓			✓	LOC	✓	✓	FCD, ACD, HQ, PHQ
Figueiredo et al. (2008)		✓	✓	✓	✓	LOC	✓		CDO, CDLOC, feature interlacing
Liebig et al. (2010)			✓	✓	✓	LOC	✓	✓	SD, TD, GRAN, ND
Sobernig (2010)	✓				✓		✓	✓	Scatteredness, Scattering Concentration
Apel and Beyer (2011)	✓			✓			✓		IUD, EUD
Revelle et al. (2011)			✓	✓	✓	syntax tokens		✓	SFC, term vectors, document matrices
This paper	✓			✓			✓	✓	CBU, IUD, EUD

Liebig et al. (2010) applied several code-level measure constructs on conditional compilation directives (#ifdefs). Among others, they describe scattering (SD) and tangling (TD) of features in expressions of conditional compilation directives. In addition, measure constructs for the scoping of compilation directives (GRAN) and for their nesting (ND) are provided. In their study, Liebig et al. (2010) concentrated on #ifdef-based programs. By contrast, we require measures capable of capturing conditions of FOP-based product lines. The constructs proposed by Liebig et al. (2010) have more recently been adopted in a study by Berger and Guo (2013) to correlate measurements on variability-model size and on code size. The proposed variability-model measures (e.g., number of top and leaf features) are orthogonal to our code-level measures and the study design by Berger and Guo (2013) is an important point of future extension for our work.

Apel and Beyer (2011) employed the visual-clustering tool FeatureVisu to decompose dependency graphs and a mapping between feature and program elements of a product line into a clustered graph, according to interdependencies between features. Based on the clustered graph, they conducted measurements using internal-ratio feature dependency (IFD), external-ratio feature dependency (EFD), as well as distance-based variants of the former two constructs. The clustering and measurement approach was applied to data sets of 40 product lines, including the 28 product lines investigated in this paper. Apel and Beyer (2011) arrived at important observations that motivated our study, namely that a feature-oriented decomposition alone does not guarantee improved feature cohesion, and that feature implementations take different roles, which yield different dependency structures. In our work, we extend the reach of the FeatureVisu study by including a novel data set on the product lines (type-system references rather than feature-code mappings), by adding a view on structural coupling (CBU), and by drawing a connection to product-line analysis.

There is an extensive body of research aiming at investigating internal attributes (and beyond) of aspect-oriented code bases, including product lines (see, e.g., Burrows et al. 2010). A key difference to our work is that we investigate the code bases including the variation for entire product lines (rather than single products). In addition, our observations relate to code bases implemented using feature-oriented programming techniques (Fuji), which target predominantely heterogeneous and statically crosscutting product-line designs (Lopez-Herrejon and Apel 2007; Apel et al. 2008).

Figueiredo et al. (2008) contrast two variability implementation techniques (AOP, syntax preprocessor) by looking at the further development of two product lines: a variant of the MobileMedia product line and a gaming product line called BestLap. In their study, the authors collected data—among others—about the implementation structure and feature dependencies over several release cycles. Each release cycle represents an implementation of a specific functional scenario (feature addition). The comparison between AOP and syntax preprocessing leads the authors to the conclusion that the former is preferable when it comes to implementing alternative or optional features, and that the latter has advantages when adding or removing mandatory features. They measured concern diffusion over components (CDO), the concern diffusion over operations (CDC, including advices), and concern diffusion over lines of code (CDLOC), thus extending the measure suite proposed by previous work (see also below; Sant'Anna et al. 2003). Feature dependencies are derived from the feature-code data in terms of feature interlacing (component or operation sharing between two features) and feature overlapping (co-ownership of components or operations between two features). This compares with the measure constructs proposed by Sobernig (2010).

Lopez-Herrejon and Apel (2007) present a measure suite aiming at quantifying aspect-oriented program structures (e.g., feature and aspect counts, aspect-code fractions in terms of LOC) and feature crosscutting. For the latter, the authors propose four different measure constructs. The *feature crosscutting degree* (FCD) counts the number of classes that are tangled by a feature-implementing aspect (aspects, ITDs). The *advice crosscutting degree* (ACD) limits this tangling count on advices only. The *homogeneity quotient* differentiates how much of the tangling is caused by advices or by ITDs. The *program homogeneity quotient* (PHQ) aggregates the latter for all the features into a global indicator. The measures are employed on four AspectJ product lines, two of which are also featured in our study: AHEAD and Prevayler.

Another strand of research (Wong et al. 2000; Sant'Anna et al. 2003; Eaddy et al. 2008) on quantifying separation and composition of concerns relates to techniques of virtually separating concerns in feature-oriented software development (Kästner et al. 2012) and annotation-based or preprocessor-based implementation techniques of product lines. The key difference to our measure suite is that their measure constructs relate features and code units to evaluate, for example, activities in concern or feature location (Robillard and Murphy 2007). Besides, the proposed measures are only local to the given measurement units.

Wong et al. (2000) present a suite of three measures and their measure interactions to indicate the *closeness* between functional, but higher-level concerns (such as features) and code units. 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). These are primary examples of measures based on absolute attributes, namely code slices.

The approach of *concern diffusion measures*, proposed by Sant'Anna et al. (2003), counts code units that are 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.

Eaddy et al. (2007, 2008) present two refinements over the closeness measures of Wong et al. (2000). 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). These measure instruments are devised as frequency and dispersion statistics based on code-unit links to concerns. 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. These local measures are only suitable for characterising a feature in isolation.

Montagud et al. (2012) remark critically that measure constructs are seldomly reused throughout the literature corpus and the different empirical research designs. Consequently, the comparability of research findings is limited and the empirical validation of measures remains an issue. We advance the field by reusing product-line code bases and measures already considered in earlier studies (Apel and Beyer 2011; Siegmund et al. 2011; Bouwers et al. 2011; Apel et al. 2013b). Reporting our indicative observations on potentials for static and dynamic analyses of product lines (see Section 6) are a first step towards empirically validating the CBU and the IUD measures with respect to external attributes.

7.2 Tailed Distributions in Empirical Software Data

Empirical research on software engineering has been showing strong interest in how empirical quantities generated from code bases (e.g., hierarchical and non-hierarchical relationships between classes) are structured (Taube-Schock et al. 2011; Louridas et al. 2008; Potanin et al. 2005; Marchesi et al. 2004; Wheeldon and Counsell 2003). This way, we have learnt that many empirical software quantities do not conveniently cluster around typical values (e.g., the mean value) and do not follow straightforward distributional shapes (i.e., a Gaussian distribution). It is more common to find quantities that place a critical number of observations so far from any typical value that reporting this typical value and derived statistics (e.g., means, standard deviations) stops fulfilling the representation condition and becomes misleading. Power-law distributions have attracted particular attention over the last two decades (Clauset et al. 2009; Louridas et al. 2008). Beyond power laws, many distributional shapes having heavy tails of some sort—which indicate important fractions of observations taking over-proportionally large portions of the measured entity—have been found (see below). On the one hand, complex distributions bring a host of challenges for any empirical software researcher, beginning with creating appropriate research designs, establishing measurement plans, applying reporting guidelines, and extending to processing measurement results using appropriate (often unconventional) statistical techniques (Vasilescu et al. 2011). On the other hand, when mastered successfully, observations derived from such heavy-tailed distributions are some of the most interesting ones for software engineering research (Louridas et al. 2008).

Taube-Schock et al. (2011) selected 97 software systems written in Java provided by the Qualitas corpus, including Tomcat, PicoContainer, and Weka. Note that the Qualitas corpus, at the time of this writing, did not contain any feature-oriented code bases. Taube-Schock et al. (2011) extracted data on between-class and within-class dependencies from these systems and processed the link data into degree distributions. Using doubly logarithmic histogram plots as statistical "smoke tests", the authors verified whether the 97 data sets follow some power-law distribution by estimating the scale parameter. For the between-class dependencies, their measurement plan revealed small fractions of classes being highly coupled at a generally low level of coupledness in the dependency structures. Our findings support these results because we find similar distribution structures for the class decompositions of the 28 feature-oriented product lines. Taube-Schock et al. (2011) discuss the role of preferential attachment in object-oriented programming and software reuse in the face of these dependency structures, without any empirical backing of these claims, however. For the same reason, we cannot draw any conclusions about processes leading to the concentrated class decompositions in product lines.

Louridas et al. (2008) investigated the distribution structures of inter-dependencies (fan/in, fan/out) between diverse building blocks of 17 software systems (e.g., Java classes in J2SE SDK, Perl CPAN packages, Pascal modules underlying TEX). The data set collection was processed to test whether the data distributions obeyed some power-law distributions (using histogram plots on doubly logarithmic scales only) and to estimate the key parameters (i.e., the scaling parameter) of such a fitting power-law distribution. While the power-law hypothesis did not hold for all data sets, the authors found distributions with long, heavy tails in every case. They discuss implications on software reuse (e.g., observed preferences towards already highly reused artefacts), software testing (e.g., test prioritisation), and software optimisation (e.g., move-to-front re-orderings according to popularity and access patterns). While covering a large array of different systems, feature-oriented product lines are not included in their data collection. An indicative finding of

our study (Section 6.1) is that focusing on the top dependent classes may cover critical portions of variability-specific type-checking errors. This finding provides first empirical evidence for feature-oriented product lines on the authors' conjecture of potentials in test prioritization.

8 Conclusion

Feature orientation offers an additional dimension of decomposition. Decomposing a system along its provided features typically crosscuts the underlying object-oriented decomposition, which has implications for fundamental structural properties such as cohesion and coupling. In literature, feature decomposition is supposed to improve the modular structure in terms of increasing cohesion and decreasing coupling, but little is known on whether this is actually the case in feature-oriented systems.

We conducted an empirical study on 28 feature-oriented product lines to compare feature-oriented and object-oriented decomposition with regard to structural attributes, such as decomposition size, import coupling, cohesion, and unit sizes. Our study is based on a comprehensive data set—which is an improvement over previous studies—based on actual, structural references obtained from a product-line type system. Our observations add to the critical debate on modularity and feature orientation (Kästner et al. 2011):

1. Feature units can form highly coupled code structures.
2. Degrees of per-unit coupling are unequally distributed among the feature units of product lines. However, there are not necessarily hot-spot features, which has implications for product-line analysis (Siegmund et al. 2013; Thüm et al. 2014).
3. Feature units do not always represent the most cohesive units of functionality when compared to classes and class fragments, although this is one of the key goals of feature orientation.

Against the background of these observations, we discussed implications on static and dynamic analyses of product lines. For this purpose, we conducted two feasibility studies on type-checking product lines and on detecting non-functional feature interactions. Our studies show that there are associations between important product-line characteristics (i.e., type errors, performance interactions incurred by inter-feature method calls) and measurements obtained from applying measure constructs used in our study (e.g., CBU, IUD). This opens new research directions, and we formulated strong hypotheses on the predictive power of individual measures to be empirically confirmed.

In further work, we will systematically integrate the findings of this study with earlier empirical evidence in terms of a meta-study. We will also explore empirical research designs integrating our findings to improve prediction systems of non-functional properties (Kolesnikov et al. 2013a).

In the long run, feature orientation as a paradigm of thinking about and guiding software development will see broader industrial adoption, as we firmly believe. This is because it took up on key lessons learned in the practice of product-line engineering: Features must be explicit (modular, if possible) in design and code, across the whole life cycle of a product line and the corresponding software products. The results of our study provide valuable insights into the nature and merits of feature orientation, to pave the way for and to guide industrial adoption—be it in the form of tools like Fuji or alternative ways (Apel et al. 2013a).

Acknowledgments This work has been supported by the German Research Foundation (AP 206/2, AP 206/4, AP 206/5, and AP 206/6).

References

Ajila S, Dumitrescu R (2007) Experimental use of code delta, code churn, and rate of change to understand software product line evolution. J Syst Softw 80(1):74–91. doi:10.1016/j.jss.2006.05.034

Apel S, Beyer D (2011) Feature cohesion in software product lines: an exploratory study. In: Proc. ICSE. ACM, pp 421–430. doi:10.1145/1985793.1985851

Apel S, Kästner C (2009) An overview of feature-oriented software development. J Object Technol 8(5):49–84. doi:10.5381/jot.2009.8.5.c5

Apel S, Leich T, Saake G (2008) Aspectual feature modules. IEEE Trans Softw Eng 34(2):162–180. doi:10.1109/TSE.2007.70770

Apel S, Kästner C, Größlinger A, Lengauer C (2010) Type safety for feature-oriented product lines. Autom Softw Eng 17(3):251–300. doi:10.1007/s10515-010-0066-8

Apel S, Speidel H, Wendler P, von Rhein A, Beyer D (2011) Detection of feature interactions using feature-aware verification. In: Proc. ASE. IEEE CS, pp 372–375. doi:10.1109/ASE.2011.6100075

Apel S, Kolesnikov S, Liebig J, Kästner C, Kuhlemann M, Leich T (2012) Access control in feature-oriented programming. Sci Comput Program 77(3):174–187. doi:10.1016/j.scico.2010.07.005

Apel S, Batory DS, Kästner C, Saake G (2013a) Feature-oriented software product lines: concepts and implementation. Springer. doi:10.1007/978-3-642-37521-7

Apel S, Kästner C, Lengauer C (2013b) Language-independent and automated software composition: the FeatureHouse experience. IEEE Trans Softw Eng 39(1):63–79. doi:10.1109/TSE.2011.120

Apel S, Kolesnikov SS, Siegmund N, Kästner C, Garvin B (2013c) Exploring feature interactions in the wild: the new feature-interaction challenge. In: Proc. FOSD. ACM, pp 1–8. doi:10.1145/2528265.2528267

Apel S, von Rhein A, Wendler P, Größlinger A, Beyer D (2013d) Strategies for product-line verification: case studies and experiments. In: Proc. ICSE. IEEE, pp 482–491. doi:10.1109/ICSE.2013.6606594

Batory D, Sarvela J, Rauschmayer A (2004) Scaling step-wise refinement. IEEE Trans Softw Eng 30(6):355–371. doi:10.1109/TSE.2004.23

Berger T, Guo J (2013) Towards system analysis with variability model metrics. In: Proc. VaMoS. ACM, pp 23–23. doi:10.1145/2556624.2556641

Bouwers E, Correia J, van Deursen A, Visser J (2011) Quantifying the analyzability of software architectures. In: Proc. WICSA. IEEE CS, pp 83–92. doi:10.1109/WICSA.2011.20

Briand L, Daly J, Wüst J (1998) A unified framework for cohesion measurement in object-oriented systems. Empir Softw Eng 3(1):65–117. doi:10.1023/A:1009783721306

Briand L, Daly J, Wüst J (1999) A unified framework for coupling measurement in object-oriented systems. IEEE Trans Softw Eng 25(1):91–121. doi:10.1109/32.748920

Burrows R, Ferrari FC, Garcia A, Taïani F (2010) An empirical evaluation of coupling metrics on aspect-oriented programs. In: Proc. WETSoM. ACM, pp 53–58. doi:10.1145/1809223.1809231

Clauset A, Shalizi C, Newman M (2009) Power-law distributions in empirical data. SIAM Rev 51(4):661–703. doi:10.1137/070710111

Clements P, Krueger C (2002) Point – counterpoint: being proactive pays off – eliminating the adoption. IEEE Software 19(4):28–31

Czarnecki K, Eisenecker U (2000) Generative programming – methods, tools, and applications, 6th edn. Addison-Wesley

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: Proc. MSR. IEEE, pp 31–41. doi:10.1109/MSR.2010.5463279

Eaddy M, Aho AV, Murphy GC (2007) Identifying, assigning, and quantifying crosscutting concerns. In: Proc. ACoM. IEEE CS. doi:10.1109/ACOM.2007.4

Eaddy M, Aho A, Antoniol G, Gueheneuc Y (2008) CERBERUS: tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: Proc. ICPC. IEEE, pp 53–62 doi:10.1109/ICPC.2008.39

Emam KE, Melo WL, Machado JC (2001) The prediction of faulty classes using object-oriented design metrics. J Syst Softw 56(1):63–75. doi:10.1016/S0164-1212(00)00086-8

Ferreira K, Bigonha M, Bigonha R, Mendes L, Almeida H (2012) Identifying thresholds for object-oriented software metrics. J Syst Softw 85(2):244–257. doi:10.1016/j.jss.2011.05.044

Figueiredo E, Cacho N, Sant'Anna C, Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Filho F, Dantas F (2008) Evolving software product lines with aspects: an empirical study on design stability. In: Proc. ICSE. ACM, pp 261–270. doi:10.1145/1368088.1368124
Garvin B, Cohen M (2011) Feature interaction faults revisited: an exploratory study. In: Proc. ISSRE. IEEE, pp 90–99. doi:10.1109/ISSRE.2011.25
Iglewicz B, Hoaglin DC (1993) How to detect and handle outliers, vol 16. ASQC Quality Press
Kakwani N (1980) Income inequality and poverty. Oxford University Press
Kästner C, Apel S, Ostermann K (2011) The road to feature modularity? In: Proc. FOSD. ACM, pp 5:1–5:8. doi:10.1145/2019136.2019142
Kästner C, Apel S, Thüm T, Saake G (2012) Type checking annotation-based product lines. ACM Trans Softw Eng Methodol 21(3):14:1–14:39. doi:10.1145/2211616.2211617
Kiczales G, Mezini M (2005) Aspect-oriented programming and modular reasoning. In: Proc. ICSE. ACM, pp 49–58. doi:10.1145/1062455.1062482
Kolesnikov S, Apel S, Siegmund N, Sobernig S, Kästner C, Senkaya S (2013a) Predicting quality attributes of software product lines using software and network measures and sampling. In: Proc. VaMoS. ACM, pp 25–29. doi:10.1145/2430502.2430511
Kolesnikov S, von Rhein A, Hunsen C, Apel S (2013b) A comparison of product-based, feature-based, and family-based type checking. In: Proc. GPCE. ACM, pp 115–124. doi:10.1145/2517208.2517213
Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An analysis of the variability in forty preprocessor-based software product lines. In: Proc. ICSE. ACM, pp 105–114. doi:10.1145/1806799.1806819
Lilienthal C (2009) Architectural complexity of large-scale software systems. In: Proc. CSMR. IEEE, pp 17–26. doi:10.1109/CSMR.2009.16
Lopez-Herrejon R, Apel S (2007) Measuring and characterizing crosscutting in aspect-based programs: basic metrics and case studies. In: Proc. FASE. Springer, pp 423–437. doi:10.1007/978-3-540-71289-3_32
Lopez-Herrejon R, Batory D (2001) A standard problem for evaluating product-line methodologies. In: Proc. GCSE. Springer, pp 10–24. doi:10.1007/3-540-44800-4_2
Louridas P, Spinellis D, Vlachos V (2008) Power laws in software. ACM Trans Softw Eng Methodol 18(1):2:1–2:26. doi:10.1145/1391984.1391986
Marchesi M, Pinna S, Serra N, Tuveri S (2004) Power laws in Smalltalk. In: Proc. ESUG joint event, ESUG
Montagud S, Abrahão S, Insfran E (2012) A systematic review of quality attributes and measures for software product lines. Softw Qual J 20(4–5):425–486. doi:10.1007/s11219-011-9146-7
Nguyen H, Kästner C, Nguyen TN (2014) Exploring variability-aware execution for testing plugin-based web applications. In: Proc. ICSE. ACM, pp 907–918. doi:10.1145/2568225.2568300
Oster S, Markert F, Ritter P (2010) Automated incremental pairwise testing of software product lines. In: Proc. SPLC. Springer, pp 196–210. doi:10.1007/978-3-642-15579-6_14
Potanin A, Noble J, Frean M, Biddle R (2005) Scale-free geometry in OO programs. Comm ACM 48(5):99–103. doi:10.1145/1060710.1060716
Revelle M, Gethers M, Poshyvanyk D (2011) Using structural and textual information to capture feature coupling in object-oriented software. Empir Softw Eng 16(6):773–811. doi:10.1007/s10664-011-9159-7
Robillard M, Murphy G (2007) Representing concerns in source code. ACM Trans Softw Eng Methodol 16(1):3–38. doi:10.1145/1189748.1189751
Sant'Anna C, Gracia A, Chavez C, Lucena C, von Staa A (2003) On the reuse and maintenance of aspect-oriented software: an assessment framework. In: Proc. BSSE
Sarkar S, Kak A, Rama G (2008) Metrics for measuring the quality of modularization of large-scale object-oriented software. IEEE Trans Softw Eng 34(5):700–720. doi:10.1109/TSE.2008.43
Shatnawi R, Li W, Swain J, Newman T (2010) Finding software metrics threshold values using ROC curves. J Softw Maint-Res Pr 22(1):1–16. doi:10.1002/smr.404
Siegmund N, Rosenmüller M, Kästner C, Giarrusso P, Apel S, Kolesnikov S (2011) Scalable prediction of non-functional properties in software product lines. In: Proc. SPLC. IEEE, pp 160–169. doi:10.1109/SPLC.2011.20
Siegmund N, Kolesnikov S, Kästner C, Apel S, Batory D, Rosenmüller M, Saake G (2012) Predicting performance via automated feature-interaction detection. In: Proc. ICSE. IEEE, pp 167–177. doi:10.1109/ICSE.2012.6227196
Siegmund N, von Rhein A, Apel S (2013) Family-based performance measurement. In: Proc. GPCE. ACM, pp 95–104. doi:10.1145/2517208.2517209
Smaragdakis Y, Batory D (2002) Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans Softw Eng Methodol 11(2):215–255. doi:10.1145/505145.505148
Sobernig S (2010) Feature interaction networks. In: Proc. SAC. ACM, pp 2360–2364. doi:10.1145/1774088.1774574

Sobernig S, Apel S, Kolesnikov S, Siegmund N (2014) Quantifying structural attributes of system decompositions in 28 feature-oriented software product lines. Available at http://epub.wu.ac.at/id/eprint/4186, Technical Reports, Institute for Information Systems and New Media, WU Vienna, 2014/01

Stevens W, Myers G, Constantine L (1999) Structured design. IBM Syst J 38(2/3):231–256. doi:10.1147/sj.132.0115

Taube-Schock C, Walker R, Witten I (2011) Can we avoid high coupling? In: Proc. ECOOP. Springer, pp 204–228. doi:10.1007/978-3-642-22655-7_10

Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014) A classification and survey of analysis strategies for software product lines. ACM Comput Surv 47(1):6:1–6:45. doi:10.1145/2580950

Vasilescu B, Serebrenik A, van den Brand M (2011) You can't control the unfamiliar: a study on the relations between aggregation techniques for software metrics. In: Proc. ICSM. IEEE, pp 313–322. doi:10.1109/ICSM.2011.6080798

Wheeldon R, Counsell S (2003) Power law distributions in class relationships. In: Proc. SCAM. IEEE, pp 45–54. doi:10.1109/SCAM.2003.1238030

Wong W, Gokhale S, Horgan J (2000) Quantifying the closeness between program components and features. J Syst Softw 54(2):87–98. doi:10.1016/S0164-1212(00)00029-7

Yilmaz C, Cohen M, Porter A (2006) Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Trans Softw Eng 32(1):20–34. doi:10.1109/TSE.2006.8