Stefan Sobernig and Sven Apel and Sergiy Kolesnikov and Norbert Siegmund

Quantifying Structural Attributes of System Decompositions in 28 Feature-oriented Software Product Lines: An Exploratory Study

Paper

Quantifying Structural Attributes of System Decompositions in 28 Feature-oriented Software Product Lines

An Exploratory Study

(Technical Report)

STEFAN SOBERNIG[*]
SVEN APEL[**]
SERGIY KOLESNIKOV[**]
NORBERT SIEGMUND[**]

[*]WU VIENNA
[**]UNIVERSITY OF PASSAU

Background. 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 (reduced coupling, increased cohesion). However, recent empirical findings suggest that this is not necessarily the case, which is the motivation for our empirical investigation.
Aim. In fact, there is little empirical evidence on how the alternative decompositions of feature orientation and object orientation compare to each other in terms of their association with observable properties of system structure (coupling, cohesion). This motivated us to empirically investigate and compare the properties of three decompositions (object-oriented, feature-oriented, and their intersection) of 28 feature-oriented software product lines.
Method. In an exploratory, observational study, we quantify internal attributes, such as import coupling and cohesion, to describe and analyze the different decompositions of a feature-oriented product line in a systematic, reproducible, and comparable manner. For this purpose, we use three established software measures (CBU, IUD, EUD) as well as standard distribution statistics (e.g., Gini coefficient).
Results. First, feature decomposition is associated with higher levels of structural coupling in a product line than a decomposition into classes. Second, although coupling is concentrated in feature decompositions, there are not necessarily hot-spot features. Third, the cohesion of feature modules is not necessarily higher than class cohesion, whereas feature modules serve more dependencies internally than classes. Fourth, coupling and cohesion measurement show potential for sampling optimization in complex static and dynamic product-line analyses (product-line type checking, feature-interaction detection).
Conclusions. Our empirical study raises critical questions about alleged advantages of feature decomposition. At the same time, we demonstrate how the measurement of structural attributes can facilitate static and dynamic analyses of software product lines.

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 variability model (e.g., a feature model [Czarnecki and Eisenecker, 2000]). In a *feature-oriented software 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., 2013] 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 software 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 software 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 [Kästner et al., 2011] may yield a large number of decomposition units, each too small to facilitate any useful reasoning.

Earlier, and in a convenience view, feature decomposition was thought of as leading to more

[1]`http://fosd.net/fuji/`, last accessed: 12.05.2013.

modularly structured—that is, highly cohesive and loosely coupled—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., the domain or the application developers) and for different engineering tasks, such as 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 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).

This motivated us to empirically investigate and compare the structural attributes of different decompositions of 28 feature-oriented software product lines implemented using Fuji. While feature orientation is still in its infancy, it is a promising line of research [Apel et al., 2013]—the time is ripe to back foundational research in this area with empirical data.

In particular, we aim at quantifying internal attributes, such as decomposition size, import coupling, cohesion, and unit sizes, using software measures to describe the different decompositions of a product line in a systematic, reproducible, and comparable manner [Montagud et al., 2012]. This report makes the following contributions:

- We review and apply established software measures to characterize and compare structural coupling, structural cohesion, and fragmentation of three different decompositions of feature-oriented software product lines.
- We derive quantitative observations using established statistical techniques from a data set of 28 feature-oriented product lines, available from the Fuji repository. The selected product lines differ in size and by their target domains.
- Based on the quantitative observations, we answer a number of research questions, posed in the literature, on coupling, cohesion, and fragmentation in feature-oriented

product lines.

- We also look at the possible impact of coupling and cohesion measurement on optimizing static and dynamic analysis techniques for product lines.

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. Throughout 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 inter-connected 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.

All experimental data as well as the statistical tooling for reproducing our results are available at: `http://www.infosun.fim.uni-passau.de/spl/projects/decomposition/data.zip`.

2. Three Decompositions, Many Differences?

Typically, the code base of a feature-oriented product line is decomposed along two major dimensions. In Figure 1, we exemplify this for the canonical graph product line (GPL) [Lopez-Herrejon and Batory, 2001]. On the one hand, the code base is 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]).

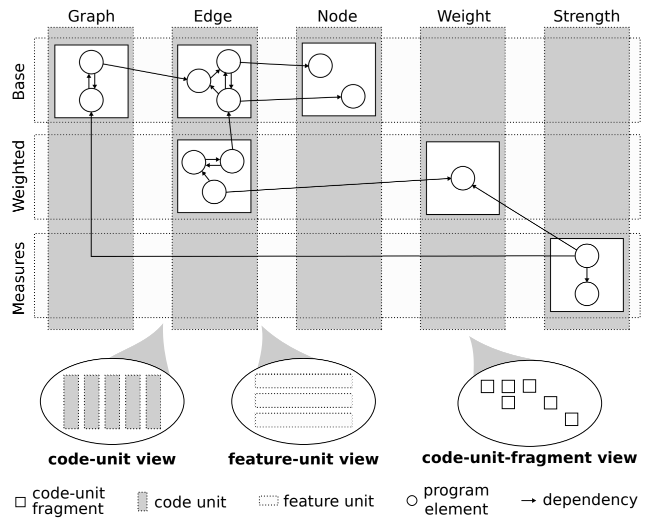


Figure 1: The three decompositions of the graph product line into *code units*, *feature units*, and *code-unit fragments*. Each decomposition provides an alternative view on the product line to the developer. Code units, feature units, and code-unit fragments contain *program elements* (e.g., fields 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 code annotations, declaration cues, 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., 2013]. 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 Figure 1. Feature unit `Measures` and code unit `Strength` contain the exact same set of program elements and their dependencies. These elements form the code-unit fragment of `Strength` which 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` only contains 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 (Figure 2b), while the other counts a medium number of units of more balanced sizes (Figure 2c).

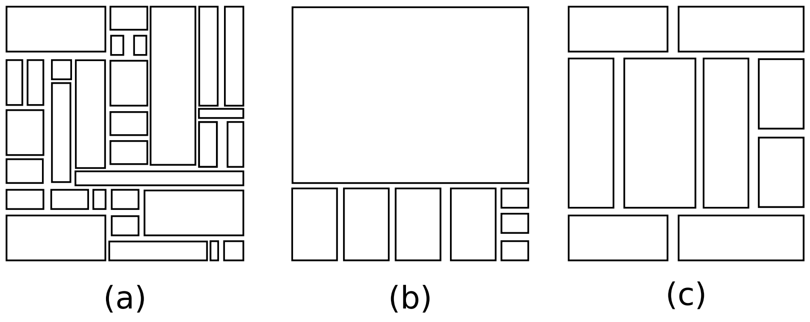


Figure 2: Different fragmentations of product-line decompositions: (a) large decomposition size, non-uniform unit sizes; (b) small decomposition size, non-uniform and concentrated unit sizes; (c) medium decomposition size, balanced unit sizes.

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:

How do coupling structures of a feature-oriented software product line differ between its decompositions into *code units*, *feature units*, and *code-unit 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.

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, for example, by running test cases on configurations guaranteed to include them, or by defining coverage criteria accordingly. Or, when building sampling-based prediction systems 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.

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 features. 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.

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 }
```

Figure 3: Excerpts of the implementations of the features `Base` and `Weighted`

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 (Figure 3) to illustrate the models for different decompositions. Features `Base` and `Weighted` contain 12 uniquely identifiable program elements, such as type, method, and field definitions. In Figure 3, every such element is described by a comment line indicating an element's identifier. The identifier consists of the containing 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-composition time. Introductions are basic building blocks for decomposition units. Usage dependencies between introductions define dependencies between decomposition units built of these introductions. Hereafter, we refer to such a usage dependency between two introductions as a *reference*. Note that there can be multiple distinct references running between two introductions.

The dependency graph resulting from the introductions and their references found in the two code excerpts in Figure 3 is illustrated in Figure 4a. The twelve introductions are represented as the graph's nodes. The introductions are grouped according to the owning Java

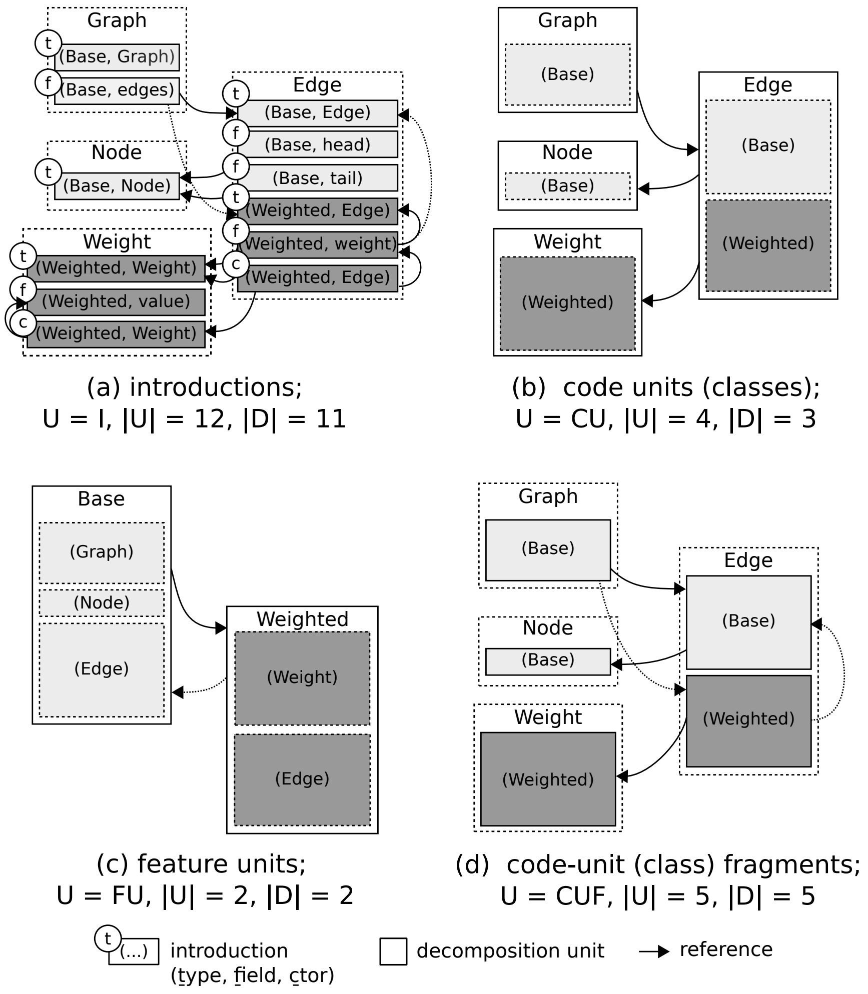


Figure 4: Dependency graphs resulting from different decompositions of GPL; U: set of decomposition units in graph; D: set of dependencies; CU: set of code units (classes); CUF: set of code-unit (class) fragments; FU: set of feature units (feature modules).

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 Figure 4a is recorded because this field is typed by `Edge` (we say the field uses the type; see Line 5 of `Base` in Figure 3).

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

In Figure 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 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 Figure 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 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 or, simply, *class-fragment decomposition* (Figure 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 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

For our exploratory study, we extracted data from the 28 feature-oriented product lines available in the Fuji repository.[1] Our selection criterion was to collect all feature-oriented product lines that we were able to locate and for which we have technical means to calculate the measures of interest. The Fuji repository has been set up exactly for this purpose.

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 for different purposes and differ in various aspects, such as the target

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

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

Table 1: Overview of the data sets extracted for each product line. SLOC: # source lines of code; |I|: # introductions; |R|: # references; |CU|: # classes/interfaces; |CUF|: # class fragments; |FU|: # Fuji feature modules; *: product-line model available.

domain. For a relatively young paradigm, such as feature orientation, this is the best one can hope for.

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]).

Furthermore, we include variability information when building the dependency graphs. The Fuji repository[1] provides 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.

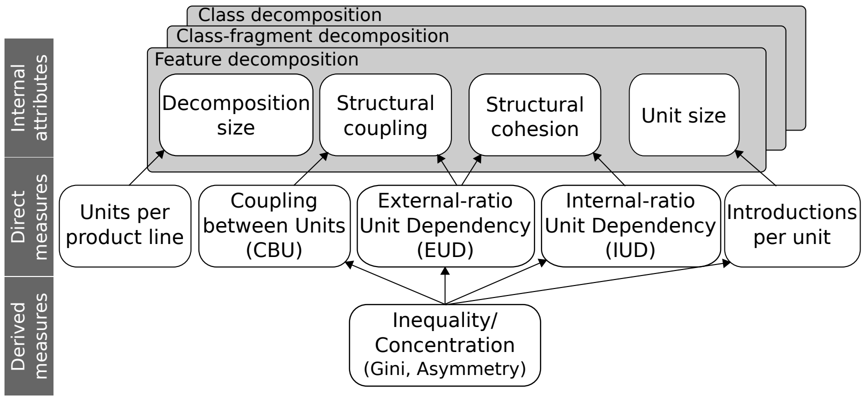


Figure 5: Relationships between internal attributes of a product-line decomposition, the indicator measures, and the derived measures.

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 Figure 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 Figure 5).

Structural coupling [Briand et al., 1999; Stevens et al., 1999]. Coupling between decomposition units means 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. couple) 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 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.

Sizes. 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 its distinctiveness of the individual decomposition units. These two internal attributes and measurement points relate to the question of the chunking effectiveness of a decomposition, and provide the analysis context for the two motivating coupling and cohesion issues (see Section 2).

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]. 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.

Visually, the Lorenz relationship can be depicted as a convex curve in a unit square ABCD, as shown in Figure 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

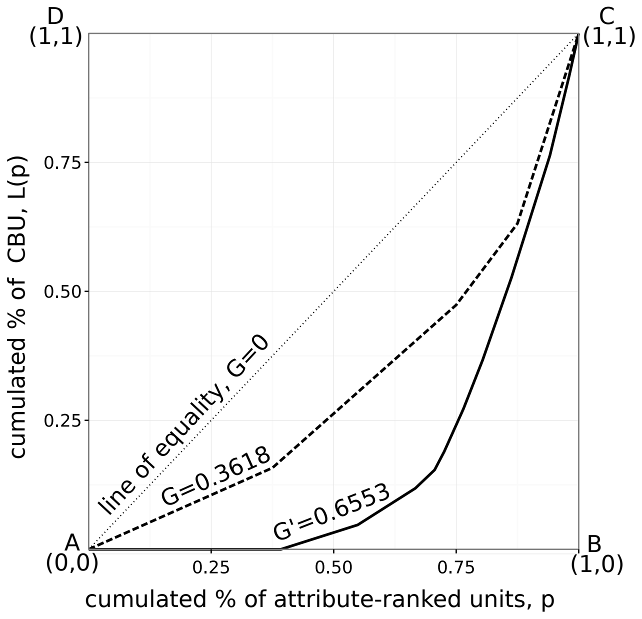


Figure 6: Concentration degree and concentration symmetry of the per-class coupling (CBU per class, solid curve) and of the per-feature coupling (CBU per feature module, dashed curve) in PKJab; G,G': Gini coefficient

Lorenz curve depicts a concentration. In Figure 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 Figure 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 Figure 6).

Lorenz Asymmetry Coefficient (S; [Kakwani, 1980; Damgaard and Weiner, 2000]). This summary statistic of the Lorenz relationship which describes how symmetric the concentration (inequality) is spread between the top and bottom decomposition units, at a given concentration degree. This addresses the issue of whether an observed concentration is more accentuated between the lower-ranked (higher-ranked) units than the higher-ranked (lower-ranked) decomposition units. The coefficient S ranges between values of 0 and 2. At $S = 1$, the Lorenz relationship (the concentration) is fully symmetric (see Figure 7). The concentration is equal within the bottom group and within the top group of decomposition units.

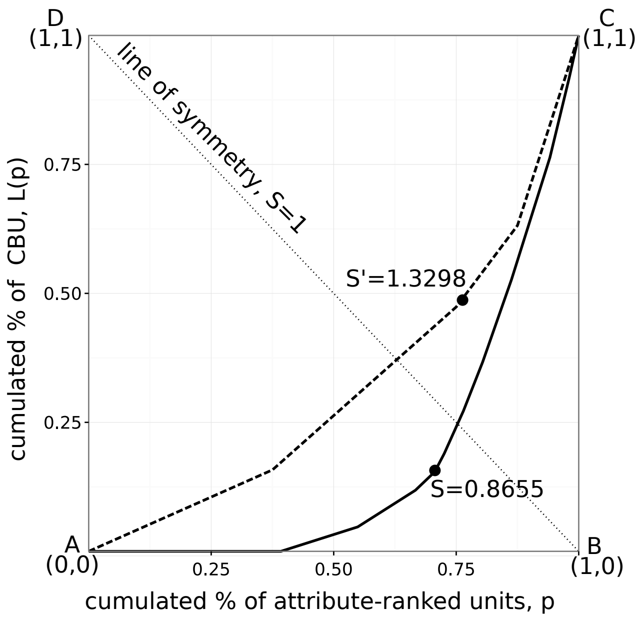


Figure 7: Concentration degree and concentration symmetry of the per-class coupling (CBU per class, solid curve) and of the per-feature coupling (CBU per feature module, dashed curve) in PKJab; S,S': Lorenz Asymmetry coefficients

$S > 1$ indicates that the concentration is more drastic within the group of higher-ranked units whereas lower-ranked decomposition units are more equally concentrated. Consider, for example, the dashed Lorenz curve (AS'C) representing the CBU concentration of the PKJab feature decomposition in Figure 7. With $S' = 1.3298$, it indicates that each percent of the highest-coupled (top) feature units adds substantially more to the cumulative attribute value (e.g., sum of CBU values) than the previous percent. Conversely, each percent of the lowest-coupled feature units adds similar-sized increments to the cumulative CBU value. For PKJab, these differences in increments can signal that the per-feature coupling is dominated by extremely high-coupled feature units (outliers) with large deltas between their CBU values.

Conversely, $S < 1$ represents that the bottom group of decomposition units is more unequal, internally, than the top group of higher-ranked units. We find this scenario for the per-class CBU concentration (ASC in Figure 7), with the increments of each percent of lowest-coupled (bottom) classes growing over-proportionally, while the increments of each percent of the heaviest-coupled (top) classes are of less growing sizes. This concentration asymmetry can reflect that, for PKJab, there are (few) classes of comparatively low and dissimilar CBU coupling, while there is a (larger) group of similarly high-coupled units.

4. Study Results

For our analysis, two data sets are available. 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 sets the context for a discussion of our observations based on the measurement data in Section 4.2.

4.1. Descriptive Analysis

The 28 product lines in our data set are comparatively small in terms of SLOC, references, and introductions as well as decomposition units (classes, class fragments, and feature modules). This is visually indicated by the characteristic shape of the density plots with peaks in the lower value ranges for each variable (CU, FU etc.) on the diagonal of Figure 8. For classes (CU) and class fragments (CUF), however, we see a second, lower peak in the respective density plots, formed by the product lines 1–8 (see IDs in Table 1) having medium to large numbers of units. 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 Figure 8:

- The SLOC and the number of introductions per product line are associated in a positive and a quasi-linear manner ($r = 0.974$).

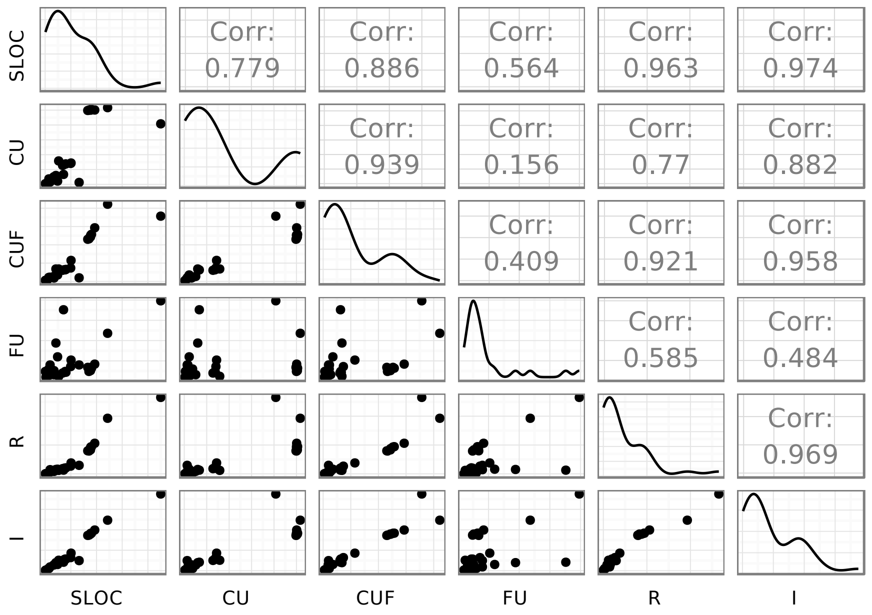


Figure 8: Pairwise scatter plots (lower segment), Pearson correlation coefficients (upper segment), and density distributions (diagonal) for the 28 product lines; SLOC: source lines of code; CU: code units (classes); CUF: code-unit fragments (class fragments); FU: feature units; R: references, I: introductions.

- The SLOC and the number of references per product line also follow a positive and quasi-linear trend ($r = 0.963$).
- 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 reflects that there are only small shares of unreferenced (*dead*) introductions in all product lines. In eight 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 (see also Table 4 in the Appendix). 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 by 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 contract, 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 ($r = 0.156$).
- When contrasting the correlated variables of SLOC, introduction, and reference counts to the numbers of different decomposition units, we find that an increasing number of SLOC, introductions, and references aligns with a quasi-linear increase in classes ($r > 0.77$) and class fragments ($r > 0.88$). This hints at a relatively balanced distribution of SLOC over classes and class fragments. A comparable relation between SLOC, introduction, and reference counts, on the one hand, and feature units, on the other hand, cannot be observed. Feature units in different product lines take varying shares in the total SLOC.

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

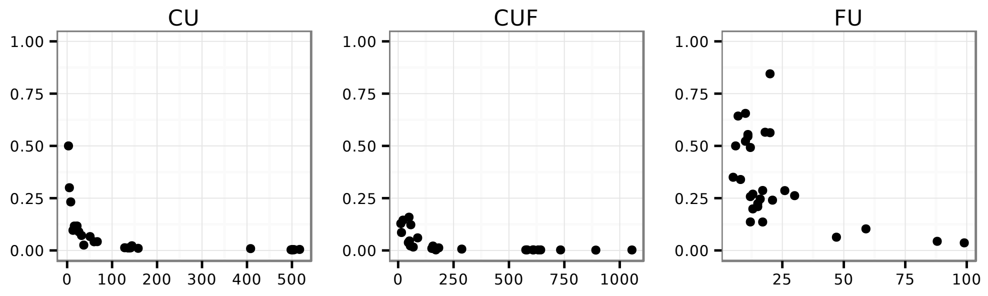


Figure 9: Decomposition size (x-axis) and connectedness (y-axis) for the 28 product lines.

decomposition sizes are plotted for the 28 product lines, for every decomposition, along the x-axes in Figure 9. The number of edges connecting the decomposition units varies depending on the number of references between units and the aggregation 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 scatterplots in Figure 9 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 (many introductions) are more interconnected (more references between units) than smaller ones.

- Opposing to the above observation, we find that decompositions (CU, CUF, and FU) of a 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 Figure 9.

- Despite being the smallest decompositions, feature decompositions are more densely connected than decompositions based on classes and class fragments (see the rightmost scatterplot in Figure 9). 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 (10%).

The following, in-depth look at the actual measurements on the dependency graphs (in particular, CBU, IUD, EUD) helps explain these two general observations.

4.2. Observations

Exploring the measurement data beyond the descriptive statistics on each structural attribute, we made seven observations. We base these observations on boxplot and concentration statistics, which are reported in full detail in a statistical annex (see Section A). The measurement data are given in Tables 5 and 9 of the Appendix. We elaborate on the implications of these observations in Section 4.3.

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

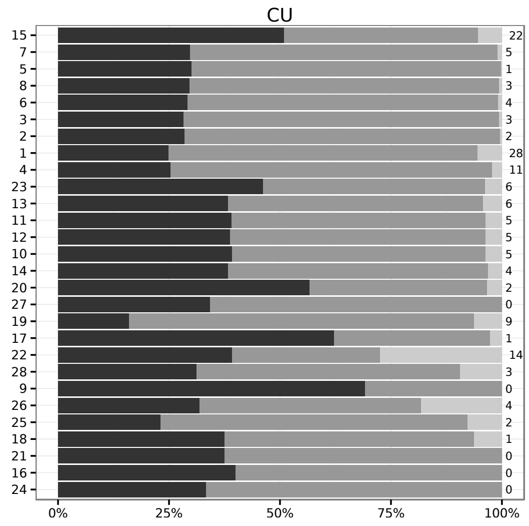
The coupling between units (CBU), expressed as the number of units that have a dependency to a given unit, is distributed very differently between classes as well as between class fragments, on the one hand, and features, on the other hand. In Figure 10, we identify three subsets of decomposition units per product line and the relative sizes of the three subsets (in % of the total number of decomposition units): The first subset comprises entirely uncoupled units (of CBU = 0; *isolates*). The third subset contains units that are coupled overproportionally[3] within their decomposition and product lines. The second and intermediate subset identifies those units which are import-coupled at medium level, falling between the lower bound (CBU = 0) and the upper bound (i.e., the data outliers).

- In both the class and class-fragment decompositions, there is a considerable number of uncoupled decomposition units. Approximately a median of 36%$\pm$8.9[4] of the classes (see Figure 10a) and a median of 34%$\pm$13.8 of the class fragments (see Figure 10b) are not import-coupled at all.
- In the feature decompositions, we did not find a single uncoupled feature in 15 out of the 28 product lines. In the remaining 13, there is only a minority share of uncoupled features (on avg. 5%, with a maximum of 33.3%; see Figure 10c).
- The number of units having an overproportional import coupling settle at comparable and low levels for the class decompositions (median 3.5$\pm$3.7 classes per product line) and class-fragement decompositions (median 4$\pm$4.5 class fragments).
- The majority of feature decompositions (18/28) do not have any import-coupling (CBU) outliers. The remainder of 10 product lines contain just one or up to a maximum of 9 over-proportionally coupled feature units (see Figure 10c).

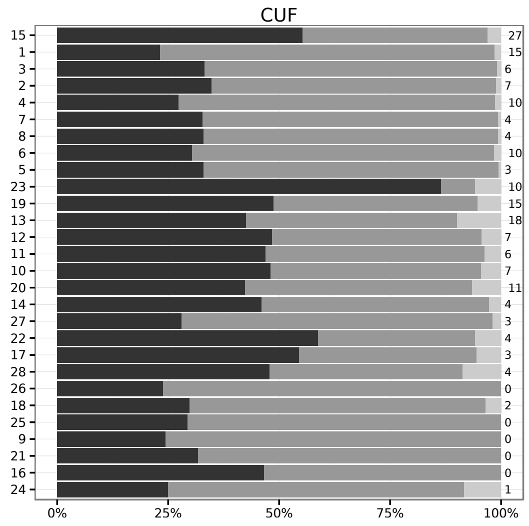
The median 60%$\pm$14.7 of classes, the median 65%$\pm$16.2 of class fragments, and the median 95%$\pm$8.7 of the feature units have medium import-coupling levels. These CBU levels fall into

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

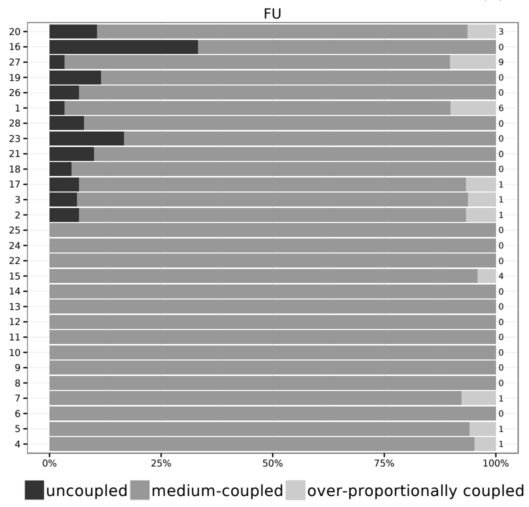
[4]We report the variance in terms of the *median absolute deviation from the median* (MADM) using the $\pm$ notation along with the median value.



(a)



(b)



(c)

Figure 10: 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 absolute number of uncoupled units per product line. Right next to each stacked bar, the absolute number of over-proportionally coupled units per product line is printed.

ranges of median CBU values of 2–4 for classes, 1–9 for class fragments, and 2–18 for features. Note, however, that the absolute CBU values—being dependent on the decomposition size—must not be compared directly.

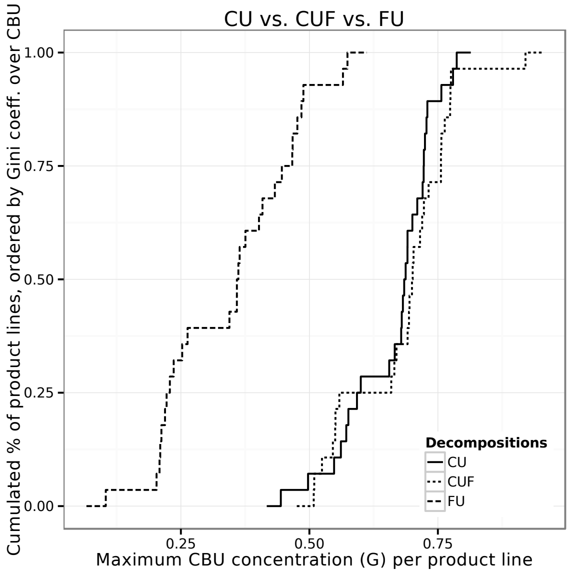
The differences in the number of isolates and the numbers of upper outliers help explain the initial observation on the differences in terms of connectedness (see Section 4.1): With feature decomposition having fewer uncoupled and fewer highly coupled feature units, their dependency graphs are more complete in terms of edges (references) observed. As a result, we found that more than 25% of their potential references are realized. The inverse holds for class and class fragment decompositions.

This also reveals that, for feature decompositions, there are outliers of over-proportionally high CBU values, more of them for classes and class fragments than for features. However, the mere count of CBU outliers does not give us an insight on how critical these outliers are for the overall import coupling (e.g., the overall connectedness) in a decomposition of a product line. Consider the two examples of PKJab (ID: 22) and JREName (ID: 5) in their class decompositions in Figure 10c. While PKJab has 14 CBU outliers, JREName contains just one (at comparable numbers of uncoupled classes). This one JREName class outlier has a CBU of 348 (!) (with a maximum CBU possible of 497, i.e., one minus the decomposition size). The 14 PKJab outliers take CBU values in a range between 7 and 14 (with a maximum CBU of 50). Still, PKJab is more densely connected in its class-decomposition dependency graph (6% of realized references as compared to 0.3% for JREName). To learn more about the relative importance of such outliers in each decomposition, we conduct a concentration analysis next (see O.2).

O.2—*The numbers of feature units providing a required reference target to a given feature unit (i.e., import-coupling or CBU levels) are more equally distributed between feature units than between classes and between class fragments.*

The CBU measurement yields a range of CBU values per decomposition unit for the three decompositions of a per product line. The way the unique CBU values (i.e., CBU levels) measured are distributed over the decomposition units (i.e., the number of occurrences of a given CBU level) describes how individual units or groups of them contribute to the import-coupling structure, overall. The concentration of CBU levels is summarized for the three decompositions (over the 28 product lines each) in Figure 11a.

- 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 Figure 11a. All decompositions have concentrations of more than 0.4 (classes; median 0.69 ± 0.06) and 0.5 (class fragments; median 0.7 ± 0.08)
- In their feature decompositions, the product lines—having a median Gini coefficient of



	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

(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 Table 5, 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 Figure 6, Section 3.4; G: Gini coefficient; SE(G): Jackknife estimate of Gini standard error; S: Lorenz Asymmetry Coefficient

Figure 11

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 Figure 11a clearly running left from the two other curves. Starting from a minimum concentration (Gini) of 0.1, the concentration of the most concentrated feature decomposition (EPL; G = 0.57) falls into the range of the lowest concentrated class and class-fragment decompositions.

This adds to the earlier observation of fewer uncoupled feature units and fewer outliers in feature decompositions when compared to classes and class fragments (see O.1). At the same time, the increased concentration coefficients of class and class-fragment decompositions provide a hint at higher CBU levels being concentrated in (small) subsets or even a few classes and class fragments, respectively.

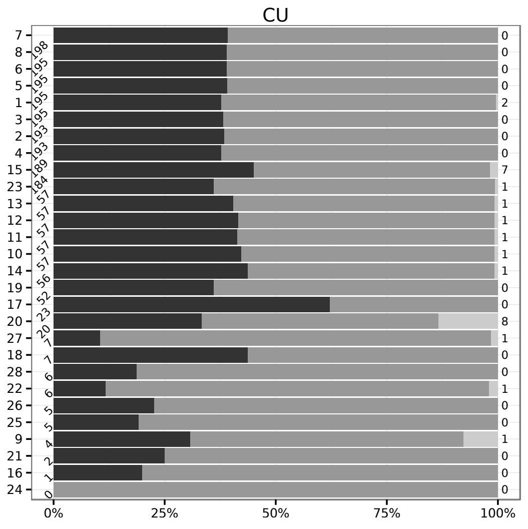
When comparing the three decompositions for each product line, we identify two different alignments of CBU concentration, in support of the picture from Figure 11a. Given the similarity of CU and CUF concentration, the comparison below is limited to CU vs. FU:

- Import-coupling (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 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 Table 6). 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 four fifths of (lowly import-coupled) features take a comparatively greater share of 47%, similar to the top 20% features (53%).
- Import-coupling (CBU) levels per unit are similarly concentrated (similarly equal/unequal) in the feature decomposition and in the class decomposition ($G_{FU} \simeq G_{CU}$): This holds only for 2 product lines (EPL and Raroscope).

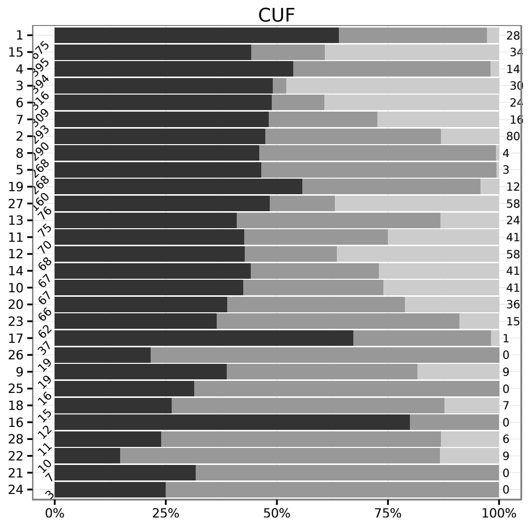
O.3—*Feature modules are internally less connected units than classes. Generally, the internal connectedness of all decomposition-unit kinds is limited.*

The measurement of internal-unit dependencies (IUD) shifts emphasis to the level of connectedness internal to a decomposition unit. The internal connectedness is computed as the ratio of actual references running between introductions owned by a given decomposition and the potential internal references. The latter is determined by the number of introductions that form a decomposition unit (class, class fragment, or feature unit). In Figure 12, we group the decomposition units into three subsets (per product line and per decomposition): incohesive units, over-proportionally[5] cohesive units and units of medium cohesion.

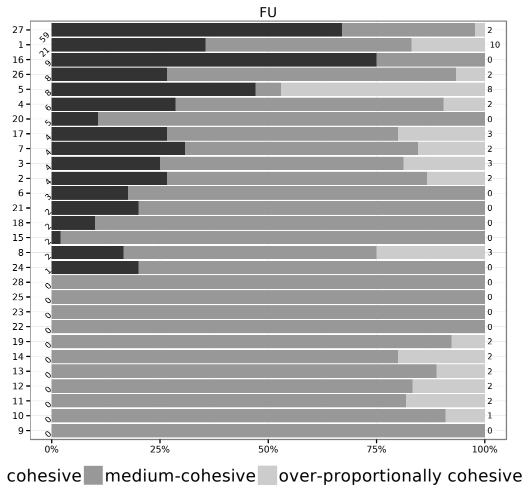
[5] We apply a standard technique of outlier identification: modified z-scores [Iglewicz and Hoaglin, 1993].



(a)



(b)



(c)

Figure 12: Each stacked barplot relates the number of incohesive units (IUD = 0, *incohesives*), the number of over-proportionally cohesive units (upper z-score outliers > 3.5), and the medium-cohesive 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 absolute number of incohesive units per product line. Right next to each stacked bar, the absolute number of over-proportionally cohesive units per product line is printed.

- Large numbers of classes (median 38%±6.6; see Figure 12a) and class fragments (median 43%±8.3; see Figure 12b) do 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; see Figure 12c).
- 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, feature a considerable partition of over-proportionally high IUD values (see Figure 12b), with a median share of 13.1%±18.5.

As a result, medium CBU values are found for a median of 61%±6.2 of the classes, 40%±22.3 of the class fragments, and 80%±28.2 of the feature units. By medium IUD levels, we refer to median IUDs 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). Note that, despite class fragments being the comparatively smallest decomposition units (see O.1), they do not necessarily form more cohesive units in terms of IUD than classes.

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 presence of comparatively high numbers of both incohesive class fragments and over-proportionally cohesive class fragments leads to an unbalanced distribution of IUD levels over the units in class-fragment decompositions. Therefore, the IUD concentration levels found for class fragments are the most pronounced for all aggregations of decompositions, having Gini coefficients of between 0.5 and 0.82 (see the right-hand CUF curve in Figure 13). At a generally lower level of concentration, class and feature decompositions share similarly concentrated IUD distributions (i.e., median Gini coefficients of 0.52±0.12 and 0.53±0.2, respectively).

Comparing the three decomposition kinds for each and every product line reflects the picture drawn by Figure 13.

- 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).
- A direct comparison of class and feature decompositions leaves a mixed picture: 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.

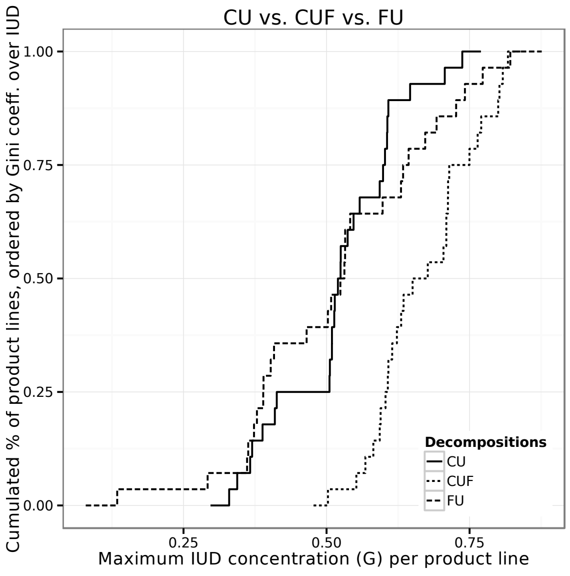


Figure 13: Three cumulative distribution curves (CDF), each representing one decomposition (CU: solid curve, CUF: dotted curve, FU: dashed curve). Each distribution curve relates a cumulated share of the 28 product lines (per decomposition, ordered by increasing IUD concentration) and a maximum IUD concentration value (G, Gini coeff.) observed for this subset of product lines. See Table 5, for the corresponding data set on Gini coefficients.

To summarize: Feature units show lower levels of per-unit cohesion than classes and class fragments. However, in feature (and class) decompositions as a whole, there are fewer incohesive units, and the level of cohesion—as measured by IUD—is distributed more equally between features (and classes) than between class fragments.

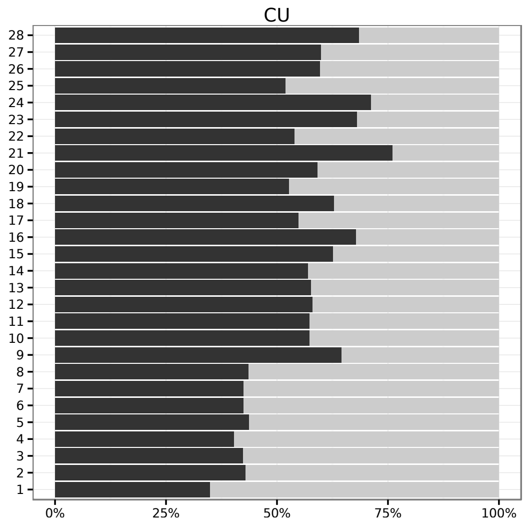
O.5—*Features have comparatively higher shares in internal than external references. The extent to which a feature unit is self-contained (the ratio of internal to total import-coupling references; EUD) is more equally distributed among the feature units when compared to classes or to class fragments.*

The measurement of external-unit dependencies (EUD) integrates the otherwise isolated views of CBU (coupling in terms of coupled units) and of IUD (cohesion in terms of the internal interconnectedness). An observation consistent with the prior CBU-based and IUD-based observations is that the feature decompositions have comparatively higher ratios of internal references to their totally realized (internal plus external) references than class and class-fragment decompositions. Figure 14c illustrates that in 24 out of 28 feature decompositions more than 50% of the references encountered run *within* feature units; more precisely a median share of 72%$\pm$6.2 in total references. Note that class decompositions also exhibit a considerable share of internal references (median 57%$\pm$9.4; see Figure 14a), though at a slightly lower level. Class-fragment decompositions have more externally than internally running references (median internal-reference share of 43%$\pm$11.4; see Figure 14b).

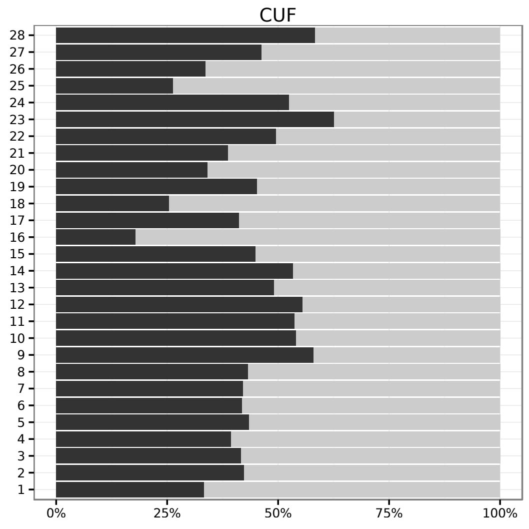
The way the external references in relation to all references are distributed among the units of each decomposition is depicted in Figure 11a. For the lower-concentrated 75% of decompositions, feature decompositions have a more equal distribution of EUD levels among features (Gini coefficients ranging from 0.13 to 0.766; see the left-handed, dashed FU curve in Figure 11a). Class decomposition follow second (0.26 <= Gini <= 0.72) and class-fragment decompositions are the most concentrated ones (0.37 <= Gini <= 0.82). For the upper 25% of the most-concentrated decompositions, the concentration levels converge.

This ordering of concentration (i.e., Gini(FU) < Gini(CU) < Gini(CUF)) can only be found in 12 out of 28 product lines. AJStats is one of them. In AJStats, the Gini coefficient of 0.22 for its feature decomposition captures that the top 20% of the most-coupled feature units account for 36% of the observed EUD, with EUD shares between 10% and 20% for the lower four fifths. Recall from Figure 14c that AJStats (9) has only 9% (472) of its total references (4855) running externally. These nine percent then distribute across the feature units according to these subset frequencies. Both, AJStats' class and class-fragment decomposition are more unequal in this distribution between subsets of classes and between subsets of class fragments, respectively (see Table 15b). For the majority of 16 product lines, we find mixed orders of concentration levels.

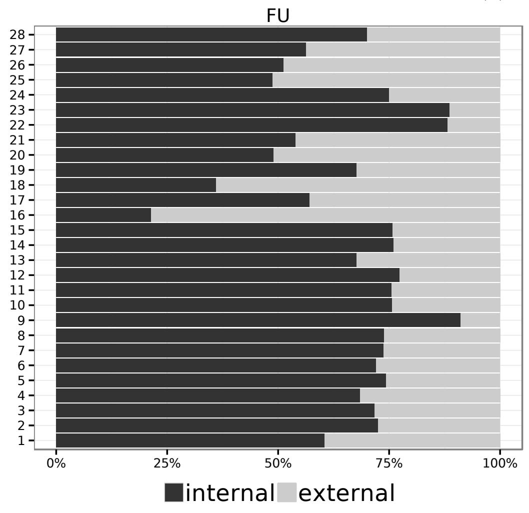
In our study, we have additionally investigated the condition of the fragmentation of the three



(a)

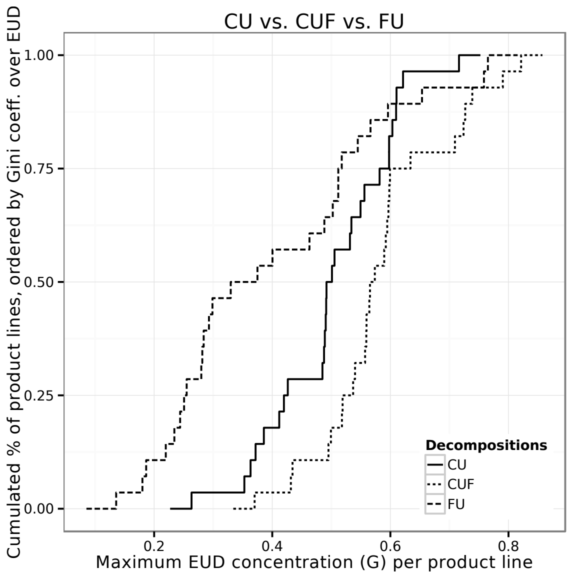


(b)



(c)

Figure 14: Each stacked barplot relates the number of internal and external references as percentage shares (y-axes) in total references per product line (x-axes). See Table 1, for the product-line identifiers (1-28).



(a) The 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 EUD concentration) and a maximum EUD concentration value (G, Gini coeff.) observed for this subset of product lines. See Table 5 for the corresponding data set.

	CU	CUF	FU
Bottom 20%	0	0	12.82
20–40%	0.96	0.82	14.79
40–60%	17.04	8.36	17.22
60–80%	25.18	10.36	19.06
Top 20%	56.82	80.47	36.12
G	0.56	0.74	0.22
SE(G)	0.1067	0.0371	0.0603
S	0.7065	1.0491	1.3357

(b) AJStats: Aggregated EUD shares hold by same-sized groups (quintiles) of decomposition units, ordered by increasing EUD; G: Gini coefficient; SE(G): Jackknife estimate of Gini standard error; S: Lorenz Asymmetry Coefficient

Figure 15

decompositions. While the resulting observations form the background of the five major results (O.1–O.5; see Section 4), we report these auxiliary observations in full detail for the sake of completeness.

O.6—*There are more feature decompositions containing comparatively large decomposition units in terms of program elements introduced than class and class-fragment decompositions. Feature decompositions exhibit a stronger dispersion of decomposition-unit sizes than class and class-fragment decompositions.*

In the Figures 16a through 16c, five groups of SPLs having comparable median unit sizes *within* a given decomposition are represented by circles of different diameter, from *very small* to *very large*. In their class decompositions, 22 out of the 28 SPLs have very small or small median unit sizes. By (very) small, we refer to one half of the (smaller) classes introducing less than and the other half of the (larger) classes introducing more than between five and 20 program elements. This compares with just one product line (AJStats) having a median class size of 57 program elements. For the class-fragment and feature decompositions, we observe a shift towards medium and large median unit sizes. In their class-fragment decompositions, 18 out of the 28 product lines have small to medium unit sizes (between three and six program elements). When decomposed into their feature units, each of 13 product lines contain feature units of which the bottom 50% introduce less and the top 50% introduce more than between 30 and 50 program elements.

When comparing the median unit sizes *between* the three decompositions, we find that the 13 decompositions of (very) large median unit size also represent the decompositions containing the largest decomposition units, globally. This observation, however, is sharply contrasted by the other 15 feature decompositions having median unit sizes less than or equal to 20 program elements per feature (see the product lines marked *very small* and *small* in Figure 16c). These small-sized feature decompositions, therefore, directly compare with the 26 class decompositions of very small to large median unit sizes. This hints at an important subset of product lines, in which feature units—conventionally thought of as coarser-grained functional units or collaborations between classes—do *not* tend to contain more introductions than classes as a convenient norm. A closer look at each of the 28 SPLs reveals that there are eight SPLs (IDs 9, 15, 16, 20, 24, 26–28 in Table 1) having larger median class sizes than median feature-unit sizes. For example, AJStats (ID 9) decomposes into 13 classes of a median unit size of 20 and into 20 feature units of a median size of 6 introductions. Note that this observation is not specific to product lines having more features than classes per se.

We quantified the dispersion of unit sizes as the median deviation of the number of introductions compared to a reference value (viz., the median unit size of each product line).[6] The

[6]This is also referred to as the median absolute deviation from the median (MADM). Dividing the MAD by the median unit size yields a coefficient of variance (said the relative MADM) suitable for comparing dispersions between

resulting dispersions are shown along the y-axes in Figures 16a–16c. Consider the example of BerkeleyDB's class decomposition having a median unit size of 11 and a MADM of 10.37 introductions.[7] This indicates a median spread of approximately 10 introductions above and below the median class size of 11 (viz., 11 ± 10.37; see the corresponding data point in the bottom right corner of Figure 16a). Differently put, for the less deviating 50% of decomposition units (classes), the MADM represents the maximum deviation observed in terms of program elements introduced. For the upper 50%, it is the minimum deviation from the median unit size. For BerkeleyDB, this gives us a relative MADM of approx. 0.94 ($10.37/11 \simeq 0.94$), that is, unit sizes float above and below the median unit size by 94% of the median unit size.

When contrasting the three decompositions, we find feature decompositions at comparatively higher relative MADM levels than class and class-fragment decompositions. Most visibly, in Figure 16c, several data points are concentrated in the upper-left corner of the plot, above a relative MADM $\simeq 1$. To gain more insights, Figure 16d plots the cumulative feature- and class-specific distributions of the relative MADM against each other. At the bottom part of the plot, the solid, class-specific distribution curve is right from the dashed, feature-specific one. This area indicates that, for the bottom (approx.) 15% of class and feature decompositions, ordered by their relative MADM, the unit-size dispersion is more accentuated for class decompositions. Above the (approx.) 17% mark of all feature decompositions (ordered by their relative MADM rank), the feature-specific distribution curves shifts rightwards. This indicates that the upper 75% of feature decompositions are more diverse than the upper 75% of the class decompositions, that is, the feature-unit sizes vary more strongly around the median unit sizes than the class-unit sizes.[8]

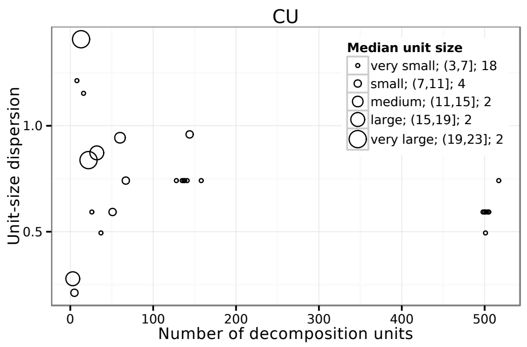
Note, however, that this observation does not necessarily allow one to conclude that any given SPL contains features of more varying size than the product line's classes. When comparing the relative MADM of the feature and class decompositions per product line, we find—similar to the median unit sizes—two groups: 6 product lines (incl., AJStats and MobilMedia8) vary more strongly in their class decompositions, 22 have more varying feature sizes (incl. AHEAD, BerkeleyDB, and ZipMe). We also find that 21 out of the 22 product lines of higher feature-size dispersion fall into the upper 75% segment of product lines identified in Figure 16d. This further supports the observation that feature decomposition vary more strongly in terms of unit sizes than the other two decompositions.

From this observation, we gain the insight that, for the 28 observed product lines, the observed feature units tend to be larger in size (# introductions per feature unit) than classes

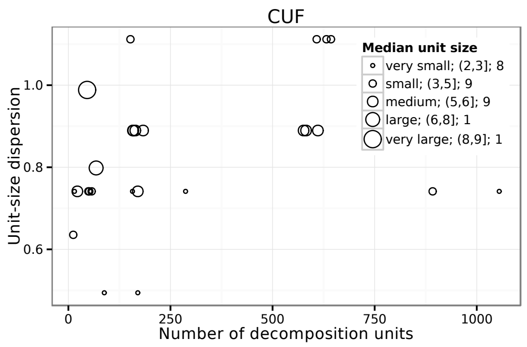
decompositions of a product line.

[7]The MADM reported here is normalized by a constant factor (1.4826) to guarantee equality with the standard (mean) deviation in the case of an underlying normal distribution of the data. For BerkeleyDB, the non-normalized MAD is eventually 7 so that $7 * 1.4826 \simeq 10.37$.

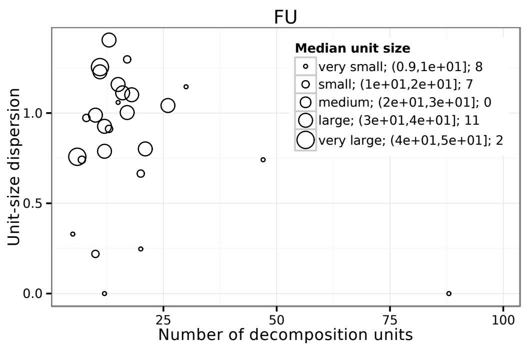
[8]We omit the CUF decomposition in Figure 16d because it follows the pattern of the class (CU) decomposition.



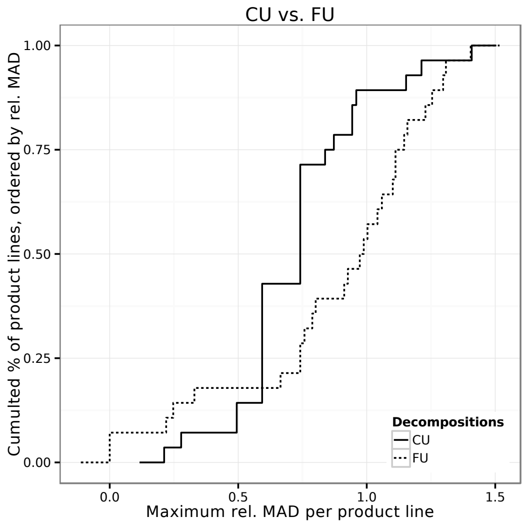
(a)



(b)



(c)



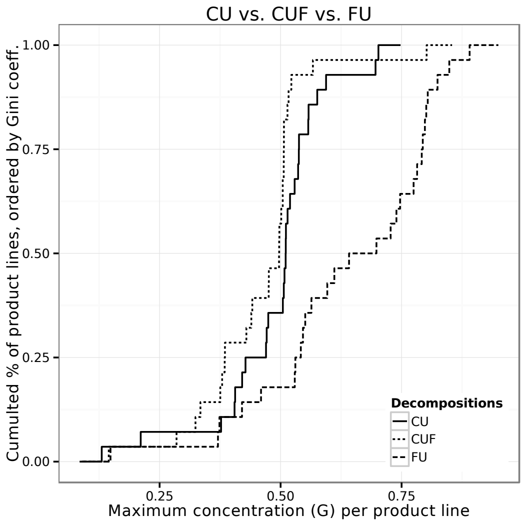
(d)

Figure 16: Key indicators for the decomposition fragmentation per product line; 1) the number of decomposition units (x-axis); 2) the relative median dispersion of unit sizes (y-axis, relative MAD); 3) the median unit size (diameter of the data points). Subfigure 16c illustrates that feature decompositions tend to have fewer, yet larger decomposition units whose unit sizes vary more pronouncedly than those of classes and class fragments. Subfigure 16d displays two cumulative distribution curves (CDF), each representing one decomposition (CU: solid curve; FU: dashed curve). Each distribution curve gives the cumulated share in the 28 product lines (per decomposition, ordered by increasing rel. MADM) and a maximum dispersion (rel. MADM) observed for a given subset of product lines.

and class-fragment units. Moreover, while also larger, feature units are more heterogeneous in terms of their unit size than classes and class-fragments.

O.7—*Feature decompositions are the most concentrated decompositions of the three decompositions regarding unit sizes: Fewer (large) feature modules are responsible for a comparatively larger share in total introductions.*

To assess the fragmentation of each of the three decomposition kinds, we quantify to which extent different groups of decomposition units (e.g., the comparatively larger or the comparatively smaller units) are responsible for important shares in the total introductions. This adds to the insights from O.6. Having described how unit sizes and unit-size dispersions of either decomposition are related, we look at the relative coverage of introductions by specific groups of or even single decomposition units within each decomposition and within each product line. This shows how ownership of introductions, at a *given* level of unit sizing (O.6), is distributed among decomposition units.



(a)

Figure 17: The three cumulated rank distributions (ECDF), each representing one decomposition (CU: solid curve, CUF: dotted curve, FU: dashed curve). Each distribution curve gives the increasing, cumulated share of the 28 product lines (per decomposition, ordered by increasing concentration) and the maximum concentration value (G, Gini coefficient) observed the corresponding subset of product lines. See Table 5 for the corresponding data set.

We observe that there is a certain concentration of unit sizes for the absolute majority of decompositions ($G > 0.3$ for 25 out 28, or approx. 90%). There are no occurrences of perfectly or close-to-perfectly equal distributions of unit sizes ($G \simeq 0$). That is, there is no single decomposition (whether CU, CUF, or FU) having units of equal or close-to-equal size

(# introductions). All three decomposition kinds contain many small decomposition units and only a few large ones. In Figure 17a, this is documented as the quasi-horizontal run of the three distribution curves at the bottom, followed by the steep ascent of upper segments.

A second observation from Figure 17a is that, while the class (solid line) and class-fragment (dotted line) decompositions maintain comparable levels of concentration for the upper 25 product lines, the 25 most concentrated feature decompositions (dashed line) exhibit higher concentration values than the 25 most concentrated class- and class-fragment decompositions. This is signalled by the FU distribution curve taking the rightmost position in Figure 17a.

	CU	CUF	FU
Bottom 20%	2.48	3.22	0.51
20–40%	5.69	4.95	0.83
40–60%	10.33	8.55	1.62
60–80%	18.02	17.47	4.41
Top 20%	63.48	65.82	92.62
G	0.59	0.61	0.90
SE(G)	0.0213	0.0159	0.1353
S	1.0452	1.0144	1.0707

(a) BerkeleyDB

	CU	CUF	FU
Bottom 20%	2.05	2.61	1.69
20–40%	5.43	5.36	4.59
40–60%	11.57	12.63	9.95
60–80%	15.38	17.85	12.00
Top 20%	65.56	61.54	71.77
G	0.61	0.57	0.60
SE(G)	0.0388	0.0360	0.0410
S	1.1465	1.1101	1.0726

(b) Bali

	CU	CUF	FU
Bottom 20%	1.73	5.45	4.95
20–40%	2.48	9.65	9.65
40–60%	4.46	12.13	14.11
60–80%	12.38	22.03	25.74
Top 20%	78.96	50.74	45.54
G	0.72	0.43	0.41
SE(G)	0.0392	0.0249	0.0405
S	1.0078	1.0669	1.0906

(c) GPL

Figure 18: Shares in introductions for decomposition units (CU, CUF, FU) for three product lines, grouped by their unit size into quintile groups (top fifth, bottom fifth, etc.)

The last observation, however, does not imply that the measured unit-size concentration in the feature decomposition of any product line exceeds the concentration of its class decomposition. This is because in Figure 17a, we compare the three decomposition kinds (CU, CUF, FU) globally rather than within each product line.

At a per-product-line basis, we identify three different alignments of concentration levels between the class and feature decompositions:

- Unit sizes are *more* concentrated (less equal) in the feature decomposition than in the class decomposition ($G_{FU} > G_{CU}$): In 16 product lines, introduction ownership is more unequally distributed among features than among classes. BerkeleyDB falls into this group ($0.59 < 0.90$). More than 90% of the introductions are owned by the 20% of feature units while the top 20% of classes only account for approx. 64% of the introductions (see Table 18a). Also note that, in its feature decomposition, the lower four fifths of feature units fall clearly below the corresponding shares in the class decomposition.

- Unit sizes are similarly concentrated (similarly equal/unequal) in the feature decomposition and in the class decomposition ($G_{FU} \simeq G_{CU}$): In seven product lines, the feature and class decompositions show very similar to nearly equal levels of unit-size concentration. Bali showcases this in Table 18a. With the top 20% of classes and features settling between 65–72% of the introductions, and the middle 50% of classes and features between

10–15%, Gini coefficients of approx. 0.6 are reported for both decompositions.

- Unit sizes are *less* concentrated (more equal) in the feature decomposition than in the class decomposition ($G_{FU} < G_{CU}$): In five product lines, including GPL (see Table 18c), we find distributions of unit sizes more heavily concentrated in classes than their features. In GPL, the top 20% of classes represent approx. 79% of the total introductions compared to approx. 46% contained by the top 20% of its feature units. This discrepany also holds for the lower unit fifths of the two decompositions. The resulting Gini coefficient is 0.72 for the GPL class decomposition, which exceeds the feature decomposition's one (0.41).

Comparing class and class-fragment decompositions draws a different picture: In 14 product lines, class fragments have more concentrated unit sizes; in 11, the inverse holds; and in 3, there are comparable concentration levels. The generally observed higher concentration of unit sizes in feature decompositions in Figure 17a, however, signals a clear difference between CU vs. CUF and CU vs. FU. Feature decompositions, which either exceed or fall below the concentration levels of the corresponding class decompositions (in 23 SPLs), do so more strongly than the diverging unit-size concentration between CU and CUF (in 25 SPLs).

4.3. Discussion

How do coupling structures of a feature-oriented software product line differ between its decompositions into *code units* (classes), *feature units* (feature modules), and *code-unit fragments* (class fragments)? A cross-reading of our observations on coupling (CBU; see O.1 and O.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* between decomposition size (i.e., the number of classes, class fragments, or feature modules) and the degree of import coupling (O.1; see Figure 9). 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

Direct measures		Decomposition kind: CU	FU	CUF	Attribute
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, few outliers, many isolates	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	Structural cohesion
	O.4	mixed, medium IUD concentration	mixed, medium IUD concentration	uniform, medium/high IUD concentration	
Number of decomposition units, number of introduc-	O.6	medium decompositions, (very) small units, medium unit-size variance	small decompositions, (very) small *and* (very) large units, large variance of unit sizes	large decompositions, (very) small to medium units, medium unit-size variance	Decomposition fragmentation (decomposition size,
	O.7	mixed, medium unit-size concentration	uniform, high unit-size concentration	mixed, medium unit-size concentration	

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

presence of coupling-free units (see O.1). For classes and class fragments, Figure 19 illustrates this strong positive and quasi-linear association between decomposition size and uncoupled decomposition units of (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 a dense coupling structure.

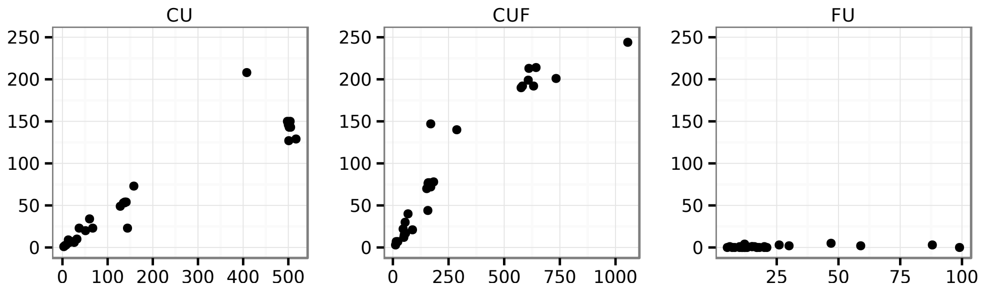
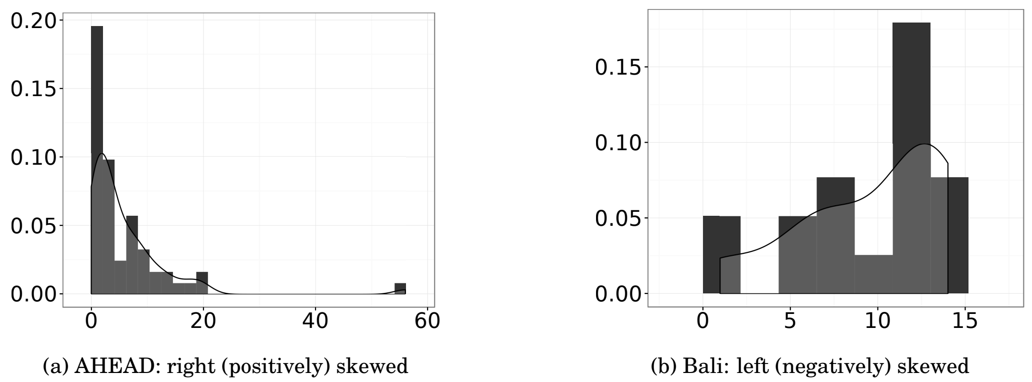


Figure 19: Decomposition size (x-axis) and the number of units without any import coupling (y-axis; CBU = 0) for the 28 product lines.

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 in terms of feature modules, the more the measured coupledness associates 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.2 and the boxplot in Figure 21 in Section A).

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 Figure 20a). 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 Figure 20a). 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.



(a) AHEAD: right (positively) skewed

(b) Bali: left (negatively) skewed

Figure 20: 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 Figure 20b is such an example. In this product line, a medium to large number of comparatively high-coupled feature modules can be found (O.3). 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.3). 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 re-assess 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.

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, first, 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).

Do the three decompositions differ in the sizes of chunks of interdependent decomposition units?

There are more chunks of feature modules considered suitable for local decision-making and short-term recall present than chunks of classes and class fragments. To solve an engineering task in one unit, a developer must consider groups (chunks) of decomposition units as a whole. Short-term memorizability of such cognitive chunks is a requirement primarily resulting from the practical needs of many engineering tasks (e.g., code spelunking, navigating the control and data flow). Furthermore, there are debated conjectures on a reduced error proneness of short-term memory as compared to long-term memory. Memorizable chunks are considered as limited to a comparatively small number of units per chunk, for example, 7+/-2 units per chunk as discussed by Bouwers et al. [2011]. In addition, chunks are to represent coherent and non-duplicated units of functional responsibility [Lilienthal, 2009].

We observe that feature modules form chunks of inter-dependent feature modules which are closer to a size considered beneficial (6 to 10 units, including the feature module under observation) than chunks of classes and class fragments (see O.6 and O.7).

These chunk sizes, however, must be considered in a broader context. Feature modules are much larger decomposition units than classes and class fragments in terms of introduced elements (see O.6 below). Chunk sizes of feature modules might indicate an improved memorization of dependency structures only. Not only are feature modules the comparatively largest decomposition units, the introductions of a product line are also concentrated in comparatively fewer units than in class and class-fragment decompositions (O.7). A small number of large feature modules dominate.

Besides not all product lines show an allegedly favourable fragmentation of feature decompositions. To begin with, the feature decomposition of a product line is not necessarily the smallest-sized decomposition. In some cases (e.g., Raroscope, AJStats, TankWar, Notepad), there are more feature modules than classes. Then, for larger product lines, a considerable number of feature modules can be involved (47–99 feature modules). The concentration of feature-module sizes indicates an unwanted fragmentation with a subset of over-proportionally large feature modules, also for small feature decompositions. A potentially large decomposition size (see above) may deteriorate the fragmentation even further. As for a beneficial fragmentation (medium number of decomposition units, balanced but distinct unit sizes; see also Figure 2c), none of the three decomposition kinds presents a clear advantage: While classes and class fragments deviate more from the baseline chunk sizes, feature decompositions have comparatively larger and concentrated unit sizes. Nevertheless, being aware of different decomposition fragmentations of a product line can help developers to improve defect location in product-line code bases and performance testing. If less and more balancedly sized feature units than classes are present, first testing at the feature-unit level can narrow down the location; and vice versa.

5. Threats to Validity

Construct validity. The definition 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 Section 3.3).

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., 2013]. 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.

6. 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 software 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 which emphasizes the internal attributes of structural coupling and structural cohesion.

6.1. 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 software 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 software product lines: feature-oriented programming (FOP), aspect-oriented programming (AOP), and annotation-based or preprocessor-based techniques (ANN); see Table 3 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.

Apel and Beyer [2011] employed the visual-clustering tool FeatureVISU to decompose dependency graphs and a mapping between feature and program elements of a software 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-basted 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] study 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 static 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 [Sant'Anna et al., 2003, see below]. 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, aspects code fraction 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], performs counts of code units required to implement a concern. Concern diffusion measures reflect the number of operations (i.e., the *concern diffusion over operations* CDO) and components (i.e., the *concern diffusion over components* CDC) required to implement a given concern. Comparatively higher (lower) concern diffusion counts are read as indicators for a low (high) cohesion.

In Eaddy et al. [2007, 2008], two refinements over the closeness measures of Wong et al. [2000] are presented. On the one hand, the authors suggest variance-based aggregates of concentration for a given concern as the *degree of scattering* (DoS). Similarly, the variance of dedications for all program components is discussed as the *degree of focus* (DoF; or its inversion, the *degree of tangling*; DoT). 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., 2013].

	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, 2008]			✓		✓			✓	DoS, DoF, DoT
Lopez-Herrejon and Apel [2007]	✓	✓			✓	LOC	✓	✓	FCD, ACD, HQ, PHQ
Figueiredo et al. [2008]		✓	✓	✓	✓	LOC	✓		CDO, CDLOC, feature interlacing
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

Table 3: Overview of related work on software measures for software 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: measure(s) describe the condition of one program element or feature; global scope: measure(s) describe the condition of the entire system or a subsystem.

6.2. 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, does 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 software product lines. Taube-Schock et al. [2011] discuss the role of preferential attachment in OO 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 software 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. In an extensive discussion, 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 software product lines are not included in their data collection.

7. Conclusion

Feature orientation imposes an additional dimension of decomposition. Decomposing a system along the features it provides, crosscuts typically the underlying object-oriented decomposition, which has implications for fundamental structural properties, such as cohesion and coupling. In the 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. We found that convenient claims on feature orientation, such an improved modular code structure, do not hold unconditionally. We observed that, first, feature modules can form highly coupled code structures. Second, the degrees of per-unit coupling are distributed among the feature units of product lines unequally. However, there are not necessarily hot-spot features, which has implications for product-line analysis [Thüm et al., 2014]. Third, feature units do not always form the most cohesive units of functionality when compared to classes and class fragments, although this is one of the key goals of feature orientation. These observations add to the critical debate on modularity and feature orientation [Kästner et al., 2011].

Based on our study, we discussed the implications and perspectives of our measurement methodology and experimental findings for future work on static and dynamic analysis of product lines. For this purpose, we conducted two feasibility studies on type-checking product lines and feature interaction detection. Our studies show that there are correlations of important product-line characteristics (e.g., type errors) with our software measures. This opens new research directions and, based on our data, there are strong hypotheses we can empirically evaluate (e.g., regarding the predictive power of individual measures for detecting feature interactions).

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., 2013].

Acknowledgements

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

References

Ajila, S. and R. Dumitrescu (2007). Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *J. Syst. Software 80*(1), 74–91.

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

Apel, S. and D. Beyer (2011). Feature cohesion in software product lines: An exploratory study. In *Proc. ICSE*, pp. 421–430.

Apel, S. and C. Kästner (2009). An overview of feature-oriented software development. *J. Object Technology 8*(5), 49–84.

Apel, S., C. Kastner, and C. Lengauer (2013). Language-independent and automated software composition: The FeatureHouse experience. *IEEE Trans. Softw. Eng. 39*(1), 63–79.

Apel, S., S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich (2012). Access control in feature-oriented programming. *Sci. Comput. Program. 77*(3), 174–187.

Apel, S., T. Leich, and G. Saake (2008). Aspectual feature modules. *IEEE Trans. Softw. Eng. 34*(2), 162–180.

Batory, D., J. Sarvela, and A. Rauschmayer (2004). Scaling step-wise refinement. *IEEE Trans. Softw. Eng. 30*(6), 355–371.

Bouwers, E., J. Correia, A. van Deursen, and J. Visser (2011). Quantifying the analyzability of software architectures. In *Proc. WICSA*, pp. 83–92.

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

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

Burrows, R., F. C. Ferrari, A. Garcia, and F. Taïani (2010). An empirical evaluation of coupling metrics on aspect-oriented programs. In *Proc. WETSoM*, pp. 53–58.

Clauset, A., C. Shalizi, and M. Newman (2009). Power-law distributions in empirical data. *SIAM Rev. 51*(4), 661–703.

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

Czarnecki, K. and U. Eisenecker (2000). *Generative Programming – Methods, Tools, and Applications* (6th ed.). Addison-Wesley.

Damgaard, C. and J. Weiner (2000). Describing inequality in plant size or fecundity. *Ecology 81*(4), 1139–1142.

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

Eaddy, M., A. V. Aho, and G. C. Murphy (2007). Identifying, assigning, and quantifying crosscutting concerns. In *Proc. ACoM*.

Figueiredo, E., N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas (2008). Evolving software product lines with aspects: an empirical study on design stability. In *Proc. ICSE*, pp. 261–270.

Iglewicz, B. and D. C. Hoaglin (1993). *How to Detect and Handle Outliers*, Volume 16. ASQC Quality Press.

Kakwani, N. (1980). *Income Inequality and Poverty*. Oxford University Press.

Kästner, C., S. Apel, and K. Ostermann (2011). The road to feature modularity? In *Proc. FOSD*, pp. 5:1–5:8.

Kästner, C., S. Apel, T. Thüm, and G. Saake (2012). Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. 21*(3), 14:1–14:39.

Kiczales, G. and M. Mezini (2005). Aspect-oriented programming and modular reasoning. In *Proc. ICSE'05*, pp. 49–58.

Kolesnikov, S., S. Apel, N. Siegmund, S. Sobernig, C. Kästner, and S. Senkaya (2013). Predicting quality attributes of software product lines using software and network measures and sampling. In *Proc. VaMoS*, pp. 25–29.

Lilienthal, C. (2009). Architectural complexity of large-scale software systems. In *Proc. European Conf. Software Maintenance and Reengineering (CSMR)*, pp. 17–26.

Lopez-Herrejon, R. and S. Apel (2007). Measuring and characterizing crosscutting in aspect-based programs: Basic metrics and case studies. In *Proc. Int. Conf. Fundamental Approaches to Software Engineering (FASE)*, pp. 423–437.

Lopez-Herrejon, R. and D. Batory (2001). A standard problem for evaluating product-line methodologies. In *Proc. GCSE*, pp. 10–24.

Louridas, P., D. Spinellis, and V. Vlachos (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol. 18*(1), 2:1–2:26.

Marchesi, M., S. Pinna, N. Serra, and S. Tuveri (2004). Power laws in smalltalk. In *Proc. Smalltalk Joint Event, ESUG*.

Montagud, S., S. Abrahão, and E. Insfran (2012). A systematic review of quality attributes and measures for software product lines. *Softw. Qual. J. 20*(4-5), 425–486.

Potanin, A., J. Noble, M. Frean, and R. Biddle (2005). Scale-free geometry in OO programs. *Comm. ACM 48*(5), 99–103.

Revelle, M., M. Gethers, and D. Poshyvanyk (2011, December). Using structural and textual information to capture feature coupling in object-oriented software. *Empir. Softw. Eng. 16*(6), 773–811.

Robillard, M. and G. Murphy (2007). Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol. 16*(1), 3–38.

Sant'Anna, C., A. Gracia, C. Chavez, C. Lucena, and A. von Staa (2003). On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proc. BSSE*.

Sarkar, S., A. Kak, and G. Rama (2008). Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Trans. Softw. Eng. 34*(5), 700–720.

Siegmund, N., S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake (2012). Predicting performance via automated feature-interaction detection. In *Proc. ICSE*, pp. 167–177.

Siegmund, N., M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov (2011). Scalable prediction of non-functional properties in software product lines. In *Proc. SPLC*, pp. 160–169.

Smaragdakis, Y. and D. Batory (2002). Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol. 11*(2), 215–255.

Sobernig, S. (2010). Feature interaction networks. In *Proc. SAC*, pp. 2360–2364.

Stevens, W., G. Myers, and L. Constantine (1999). Structured design. *IBM Syst. J. 38*(2/3), 231–256.

Taube-Schock, C., R. Walker, and I. Witten (2011). Can we avoid high coupling? In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pp. 204–228.

Thüm, T., S. Apel, C. Kästner, I. Schaefer, and G. Saake (2014). A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*. Accepted for publication Jan 30, 2014.

Vasilescu, B., A. Serebrenik, and M. van den Brand (2011). You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *Proc. Int. Conf. Software Maintenance (ICSM)*, pp. 313–322.

Wheeldon, R. and S. Counsell (2003). Power law distributions in class relationships. In *Proc. SCAM*, pp. 45–54.

Wong, W., S. Gokhale, and J. Horgan (2000). Quantifying the closeness between program components and features. *J. Syst. Software 54*(2), 87–98.

A. Boxplot Statistics

We computed various statistics on the indicator measures (introduction count, CBU, IUD, EUD) of the internal attributes investigated (unit size, import coupling, cohesion; see Section 3.4). This annex provides the boxplot statistics which back the discussion in Section 4.2.

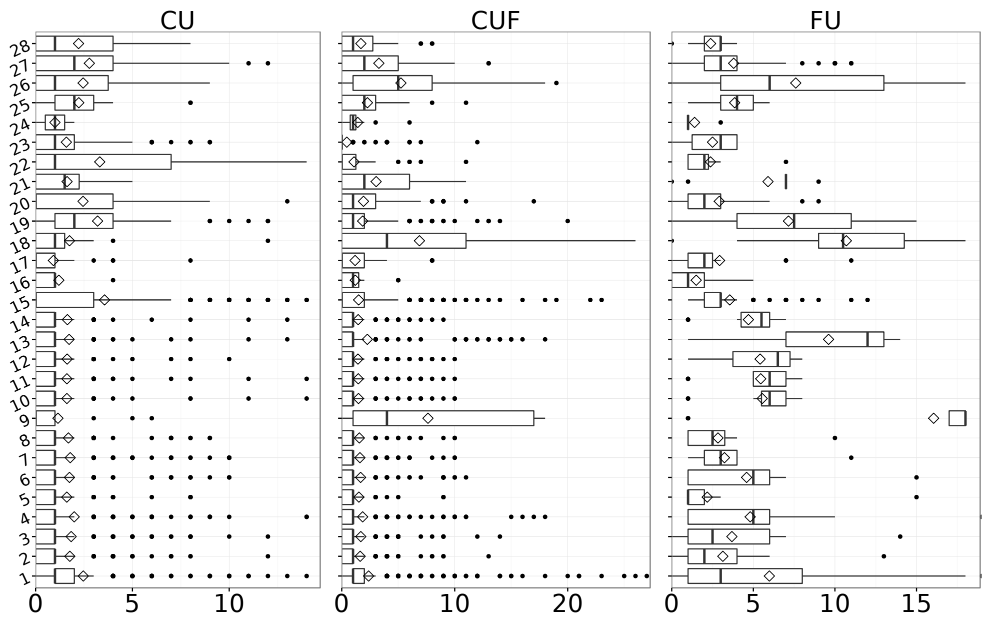


Figure 21: Boxplot statistics for coupling between units (CBU; x-axis) per product line (y-axis); see O.1 and O.2

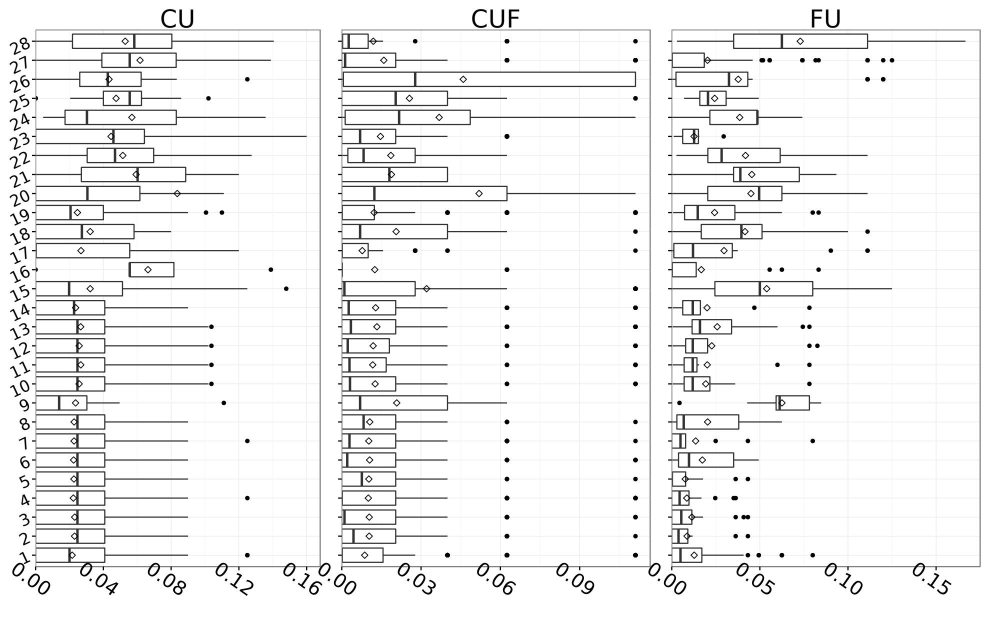


Figure 22: Boxplot statistics for internal-unit dependency (IUD; x-axis) per product line (y-axis); see O.3 and O.4

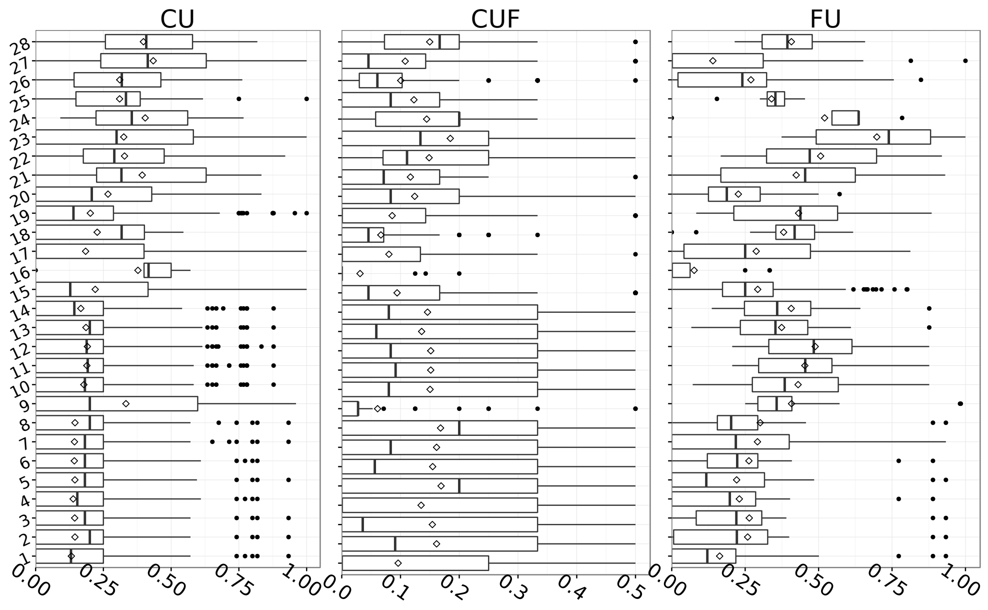


Figure 23: Boxplot statistics for external-unit dependency (EUD; x-axis) per product line (y-axis); see O.5

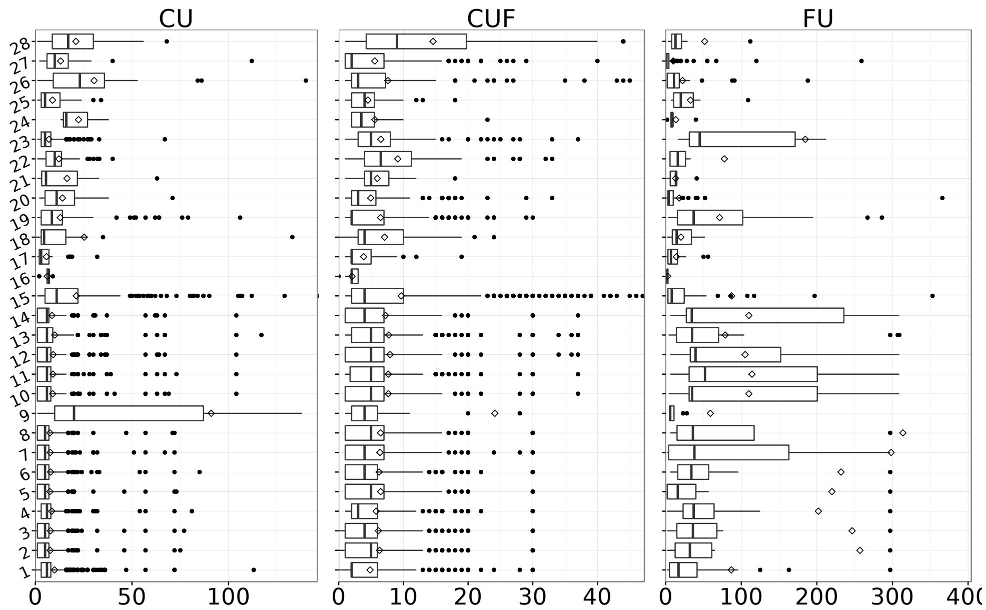


Figure 24: Boxplot statistics for unit sizes (x-axis) per product line (y-axis); see O.6 and O.7

B. Data

	I	I%	CU	CU%	CUF	CUF%	FU	FU%
AHEAD	481	8	4	1	4	0	0	0
BCJak2Java	452	10	3	1	3	0	0	0
Jak2Java	455	10	3	1	3	0	0	0
Jampack	470	9	4	1	4	0	0	0
JREName	451	10	3	1	3	0	0	0
Mixin	455	10	4	1	4	1	0	0
MMatrix	456	10	3	1	3	0	0	0
UnMixin	452	10	3	1	3	0	0	0
AJStats	37	3	2	15	2	4	0	0
Bali2Jak	127	9	5	4	5	3	0	0
Bali2JavaCC	129	9	6	4	6	4	0	0
Bali2Layer	134	9	5	4	5	3	0	0
Bali	133	8	6	4	6	3	0	0
BaliComposer	126	10	4	3	4	3	0	0
BerkeleyDB	340	4	126	31	272	30	0	0
EPL	5	11	0	0	0	0	0	0
GameOfLife	20	8	16	43	16	29	0	0
GPL	9	2	0	0	0	0	0	0
GUIDSL	156	7	1	1	1	0	0	0
MobileMedia8	60	6	10	17	43	25	0	0
Notepad	5	3	0	0	0	0	0	0
PKJab	34	5	0	0	0	0	0	0
Prevayler	109	8	18	11	21	12	0	0
Raroscope	4	5	0	0	0	0	0	0
Sudoku	18	6	1	4	1	2	0	0
TankWar	24	3	0	0	0	0	0	0
Violet	44	4	0	0	0	0	0	0
ZipMe	59	8	1	3	1	2	0	0

Table 4: The numbers of introductions (I), code-units (classes, CU), code-unit fragments (class fragments, CUF), and feature units (FU) per product line which do not participate in any observed reference. The gray-shaded columns report the corresponding relative shares in the total number of introductions (I%), of code units (CU%), of code-unit fragments (CUF%), and feature units (FU%), respectively, as reported in Table 1.

	Unit size			CBU			IUD			EUD		
	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU
AHEAD	0.5944	0.5301	0.8179	0.7099	0.7010	0.5653	0.5472	0.7641	0.6923	0.5340	0.7093	0.5959
BCJak2Java	0.5719	0.5808	0.8077	0.6844	0.7194	0.4766	0.5052	0.6071	0.6724	0.4877	0.5191	0.5664
Jak2Java	0.5724	0.5777	0.8003	0.6908	0.7153	0.4883	0.5095	0.6304	0.6301	0.4913	0.5402	0.5176
Jampack	0.5674	0.5579	0.7874	0.6797	0.6911	0.4328	0.5245	0.6770	0.6338	0.5011	0.5947	0.5117
JREName	0.5753	0.5850	0.8497	0.6819	0.6938	0.4674	0.5094	0.5930	0.7419	0.4905	0.4952	0.6536
Mixin	0.5806	0.5771	0.8070	0.6909	0.6948	0.3756	0.5144	0.6346	0.5413	0.4918	0.5369	0.4630
MMatrix	0.5763	0.5836	0.7930	0.7004	0.7026	0.3590	0.5199	0.6147	0.7265	0.5054	0.5180	0.5448
UnMixin	0.5718	0.5824	0.7833	0.6873	0.6996	0.4020	0.5058	0.5945	0.5980	0.4890	0.4993	0.4882
AJStats	0.7123	0.8350	0.8511	0.7795	0.5085	0.1039	0.6058	0.6230	0.1341	0.5560	0.7387	0.2197
Bali2Jak	0.5958	0.5813	0.5545	0.7278	0.7744	0.2027	0.6063	0.7121	0.5080	0.5978	0.5572	0.2928
Bali2JavaCC	0.5861	0.5726	0.5355	0.7249	0.7634	0.2121	0.5988	0.7044	0.5312	0.5978	0.5596	0.2440
Bali2Layer	0.5947	0.5880	0.5482	0.7223	0.7738	0.2526	0.6022	0.7094	0.5325	0.6034	0.5592	0.2507
Bali	0.6079	0.5720	0.6033	0.7295	0.7754	0.2360	0.5929	0.7144	0.4652	0.5817	0.5896	0.2798
BaliComposer	0.5976	0.5878	0.5683	0.7227	0.7559	0.2191	0.6080	0.7093	0.5324	0.6097	0.5655	0.2815
BerkeleyDB	0.5923	0.6082	0.9022	0.7866	0.7320	0.3589	0.6461	0.8022	0.3898	0.6104	0.6341	0.3297
EPL	0.1935	0.2151	0.1478	0.6000	0.6593	0.5741	0.3665	0.8000	0.7730	0.2633	0.8214	0.7652
GameOfLife	0.5214	0.4211	0.5711	0.7568	0.7228	0.4667	0.7066	0.8170	0.6435	0.7162	0.7904	0.5118
GPL	0.7218	0.4283	0.4101	0.6786	0.5583	0.2098	0.5369	0.6081	0.4024	0.4850	0.5737	0.2342
GUIDSL	0.5234	0.5738	0.5389	0.5613	0.7568	0.3644	0.5579	0.8081	0.5024	0.6215	0.7268	0.2987
MobileMedia8	0.4419	0.4862	0.7263	0.7203	0.6653	0.4845	0.7372	0.7124	0.3611	0.5495	0.5982	0.3749
Notepad	0.6097	0.3786	0.4023	0.5481	0.5502	0.2085	0.4130	0.5025	0.3732	0.4263	0.5967	0.4004
PKJab	0.4025	0.4266	0.7599	0.6553	0.7560	0.3618	0.3700	0.5818	0.4082	0.3858	0.4343	0.2839
Prevayler	0.4586	0.4347	0.6453	0.6660	0.9204	0.3444	0.5248	0.6504	0.3893	0.5315	0.5993	0.1865
Raroscope	0.2488	0.4540	0.4657	0.4444	0.5245	0.2286	0.5135	0.6031	0.3631	0.3718	0.3701	0.2551
Sudoku	0.4960	0.3482	0.4894	0.4973	0.5507	0.2222	0.3298	0.5679	0.2922	0.4118	0.5922	0.1357
TankWar	0.5179	0.5736	0.6513	0.5926	0.5094	0.4088	0.4096	0.5521	0.5242	0.4197	0.5644	0.5024
Violet	0.4181	0.5606	0.8016	0.5758	0.5457	0.4464	0.3439	0.7703	0.8212	0.3527	0.7239	0.7585
ZipMe	0.4256	0.4718	0.7403	0.5717	0.6695	0.2630	0.3881	0.7498	0.3783	0.3633	0.4315	0.1801

Table 5: Gini coefficients; see Section 3.4

	Unit size			CBU			IUD			EUD		
	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU
AHEAD	0.0770	0.0912	0.0865	0.1060	0.1193	0.0601	0.0175	0.0118	0.0373	0.0190	0.0131	0.0502
BCJak2Java	0.1161	0.1141	0.0396	0.1998	0.1970	0.0957	0.0182	0.0174	0.0710	0.0204	0.0195	0.0866
Jak2Java	0.1125	0.1117	0.0443	0.1820	0.1896	0.0752	0.0180	0.0174	0.0637	0.0201	0.0189	0.0860
Jampack	0.1039	0.1072	0.0716	0.1675	0.1734	0.0745	0.0179	0.0161	0.0615	0.0197	0.0173	0.0814
JREName	0.1198	0.1173	0.0365	0.2396	0.2534	0.2441	0.0184	0.0181	0.0642	0.0206	0.0203	0.0806
Mixin	0.1104	0.1119	0.0435	0.2005	0.2056	0.0827	0.0182	0.0181	0.0769	0.0203	0.0191	0.0816
MMatrix	0.1141	0.1126	0.0494	0.1846	0.2109	0.1117	0.0183	0.0175	0.0691	0.0203	0.0197	0.0855
UnMixin	0.1197	0.1171	0.0464	0.2140	0.2302	0.1110	0.0184	0.0178	0.0879	0.0206	0.0201	0.0786
AJStats	0.1605	0.1540	0.5193	0.0938	0.0510	0.0635	0.1105	0.0575	0.0434	0.1067	0.0371	0.0603
Bali2Jak	0.0392	0.0408	0.0663	0.1016	0.0750	0.0885	0.0516	0.0298	0.1157	0.0329	0.0340	0.0640
Bali2JavaCC	0.0396	0.0406	0.0655	0.0997	0.0767	0.0861	0.0498	0.0310	0.0898	0.0314	0.0334	0.0459
Bali2Layer	0.0366	0.0380	0.0477	0.1009	0.0783	0.0768	0.0513	0.0308	0.0727	0.0311	0.0338	0.0422
Bali	0.0388	0.0360	0.0410	0.0864	0.0347	0.0609	0.0494	0.0260	0.0478	0.0321	0.0299	0.0509
BaliComposer	0.0427	0.0449	0.0765	0.1090	0.0888	0.0900	0.0588	0.0299	0.1021	0.0346	0.0349	0.0635
BerkeleyDB	0.0213	0.0159	0.1353	0.0134	0.0123	0.0872	0.0223	0.0088	0.0254	0.0189	0.0135	0.0229
EPL	0.1134	0.0655	0.0275	0.1470	0.0803	0.0869	0.1570	0.1069	0.1124	0.1794	0.0895	0.1197
GameOfLife	0.0470	0.0367	0.0581	0.0586	0.0457	0.0780	0.0643	0.0482	0.0644	0.0625	0.0511	0.0880
GPL	0.0392	0.0249	0.0405	0.1154	0.0493	0.0493	0.1077	0.0472	0.0592	0.1183	0.0389	0.0695
GUIDSL	0.0315	0.0298	0.0458	0.0475	0.0436	0.0601	0.0307	0.0154	0.0460	0.0272	0.0207	0.0313
MobileMedia8	0.0376	0.0211	0.1607	0.0484	0.0261	0.0892	0.0365	0.0221	0.0492	0.0464	0.0273	0.0363
Notepad	0.0839	0.0554	0.1392	0.1317	0.0780	0.1208	0.1264	0.0884	0.1014	0.1188	0.0685	0.1154
PKJab	0.0299	0.0269	0.3418	0.0464	0.0410	0.1420	0.0428	0.0331	0.0967	0.0389	0.0351	0.0586
Prevayler	0.0324	0.0235	0.1375	0.0294	0.0166	0.1576	0.0468	0.0243	0.1311	0.0301	0.0288	0.0475
Raroscope	0.1176	0.1288	0.2191	0.2222	0.1260	0.2000	0.0881	0.0886	0.1686	0.1215	0.1220	0.1836
Sudoku	0.0384	0.0370	0.1298	0.0693	0.0483	0.0808	0.0694	0.0510	0.0818	0.0703	0.0621	0.0647
TankWar	0.0669	0.0257	0.0783	0.0654	0.0369	0.0482	0.0758	0.0409	0.0584	0.0731	0.0297	0.0595
Violet	0.0670	0.0195	0.0610	0.0425	0.0397	0.0648	0.0345	0.0218	0.0318	0.0331	0.0275	0.0386
ZipMe	0.0421	0.0354	0.1869	0.0560	0.0495	0.0767	0.0604	0.0380	0.0716	0.0585	0.0544	0.0323

Table 6: Jackknife estimate of Gini standard error (SE); we computed a standard leave-one-out Jackknife estimate on the Gini concentration statistic to quantify the degree of statistical insecurity inherent in our data-generating process.

	Unit size			CBU			IUD			EUD		
	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU
AHEAD	0.5939	0.5299	0.8018	0.7093	0.7007	0.5578	0.5463	0.7639	0.6870	0.5331	0.7091	0.5854
BCJak2Java	0.5715	0.5805	0.7887	0.6838	0.7190	0.4104	0.5042	0.6065	0.6490	0.4867	0.5183	0.5354
Jak2Java	0.5720	0.5775	0.7727	0.6902	0.7149	0.4444	0.5085	0.6298	0.6055	0.4903	0.5395	0.4855
Jampack	0.5669	0.5577	0.7468	0.6791	0.6907	0.3985	0.5235	0.6765	0.6155	0.5001	0.5941	0.4852
JREName	0.5749	0.5847	0.8414	0.6813	0.6932	0.3178	0.5084	0.5923	0.7257	0.4895	0.4943	0.6319
Mixin	0.5802	0.5769	0.7794	0.6903	0.6943	0.3330	0.5135	0.6341	0.5126	0.4908	0.5362	0.4287
MMatrix	0.5758	0.5833	0.7768	0.6998	0.7022	0.2806	0.5189	0.6141	0.6968	0.5044	0.5172	0.5069
UnMixin	0.5714	0.5821	0.7665	0.6867	0.6991	0.3218	0.5048	0.5938	0.5615	0.4880	0.4985	0.4417
AJStats	0.5952	0.8326	0.5651	0.7611	0.4982	0.0602	0.5527	0.6151	0.1083	0.5190	0.7271	0.1786
Bali2Jak	0.5858	0.5763	0.5158	0.7258	0.7730	0.1433	0.6034	0.7064	0.4172	0.5948	0.5544	0.2488
Bali2JavaCC	0.5762	0.5681	0.4972	0.7229	0.7619	0.1542	0.5958	0.7026	0.4657	0.5949	0.5569	0.2150
Bali2Layer	0.5854	0.5840	0.5160	0.7203	0.7724	0.2045	0.5992	0.7073	0.4905	0.6005	0.5564	0.2297
Bali	0.5998	0.5683	0.5855	0.7266	0.7742	0.1984	0.5900	0.7104	0.4411	0.5788	0.5873	0.2501
BaliComposer	0.5860	0.5808	0.5293	0.7205	0.7543	0.1594	0.6049	0.7065	0.4516	0.6066	0.5626	0.2379
BerkeleyDB	0.5899	0.6072	0.9016	0.7861	0.7317	0.3516	0.6452	0.8020	0.3839	0.6094	0.6337	0.3243
EPL	0.0880	0.1717	0.1279	0.5000	0.6349	0.5354	0.2191	0.7857	0.7524	0.1000	0.8086	0.7438
GameOfLife	0.5109	0.4137	0.5454	0.7487	0.7177	0.4131	0.6985	0.8136	0.6180	0.7083	0.7865	0.4769
GPL	0.7008	0.4196	0.3862	0.6110	0.5504	0.1783	0.5060	0.6011	0.3710	0.4506	0.5661	0.1939
GUIDSL	0.5166	0.5681	0.5220	0.5582	0.7559	0.3389	0.5548	0.8074	0.4851	0.6188	0.7258	0.2867
MobileMedia8	0.4359	0.4834	0.7225	0.7156	0.6633	0.4733	0.7319	0.7107	0.3472	0.5418	0.5958	0.3613
Notepad	0.5603	0.3542	0.3154	0.4835	0.5288	0.1244	0.3291	0.4788	0.3036	0.3443	0.5775	0.3337
PKJab	0.3933	0.4193	0.4304	0.6484	0.7523	0.2379	0.3573	0.5755	0.3370	0.3735	0.4259	0.2407
Prevayler	0.4538	0.4301	0.4928	0.6639	0.9200	0.2224	0.5218	0.6483	0.2943	0.5286	0.5957	0.1461
Raroscope	0.0593	0.3575	0.2297	0.1833	0.4439	0.0250	0.3203	0.5670	0.2159	0.1894	0.3129	0.0895
Sudoku	0.4779	0.3363	0.3666	0.4772	0.5417	0.1520	0.3030	0.5592	0.2329	0.3879	0.5841	0.0800
TankWar	0.4839	0.5633	0.6268	0.5732	0.5038	0.3884	0.3815	0.5469	0.5078	0.3921	0.5594	0.4853
Violet	0.4115	0.5575	0.8000	0.5694	0.5428	0.4401	0.3339	0.7688	0.8191	0.3429	0.7217	0.7557
ZipMe	0.4091	0.4626	0.5968	0.5579	0.6621	0.2128	0.3684	0.7442	0.3356	0.3427	0.4189	0.1628

Table 7: Lower Gini error bounds (.025 quantile); we computed the empirical .025 quantile based on a standard leave-one-out Jackknife resampling to set the lower error bounds when comparing Gini coefficients of different product lines or different decompositions.

	Unit size			CBU			IUD			EUD		
	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU
AHEAD	0.5950	0.5304	0.8230	0.7105	0.7013	0.5711	0.5481	0.7647	0.7000	0.5349	0.7100	0.6032
BCJak2Java	0.5727	0.5814	0.8268	0.6851	0.7201	0.4935	0.5060	0.6080	0.6966	0.4886	0.5199	0.5914
Jak2Java	0.5731	0.5783	0.8142	0.6915	0.7159	0.5082	0.5103	0.6312	0.6485	0.4922	0.5410	0.5385
Jampack	0.5681	0.5584	0.7957	0.6804	0.6916	0.4468	0.5254	0.6777	0.6528	0.5020	0.5955	0.5287
JREName	0.5760	0.5856	0.8712	0.6827	0.6944	0.4768	0.5103	0.5939	0.7676	0.4914	0.4960	0.6795
Mixin	0.5813	0.5777	0.8188	0.6916	0.6954	0.3913	0.5153	0.6355	0.5652	0.4927	0.5378	0.4814
MMatrix	0.5770	0.5842	0.8212	0.7010	0.7032	0.3772	0.5207	0.6156	0.7551	0.5063	0.5188	0.5693
UnMixin	0.5725	0.5831	0.8114	0.6880	0.7002	0.4223	0.5067	0.5954	0.6329	0.4899	0.5002	0.5062
AJStats	0.7399	0.8386	0.8557	0.8306	0.5169	0.1091	0.6401	0.6333	0.1385	0.5924	0.7454	0.2265
Bali2Jak	0.5981	0.5833	0.5780	0.7309	0.7770	0.2185	0.6101	0.7154	0.5285	0.6012	0.5605	0.3094
Bali2JavaCC	0.5884	0.5746	0.5568	0.7278	0.7658	0.2283	0.6024	0.7077	0.5490	0.6009	0.5628	0.2574
Bali2Layer	0.5969	0.5901	0.5604	0.7253	0.7764	0.2696	0.6059	0.7128	0.5508	0.6065	0.5625	0.2628
Bali	0.6101	0.5737	0.6149	0.7323	0.7773	0.2463	0.5964	0.7171	0.4770	0.5848	0.5925	0.2894
BaliComposer	0.6001	0.5900	0.6001	0.7258	0.7587	0.2385	0.6122	0.7127	0.5548	0.6135	0.5690	0.2982
BerkeleyDB	0.5931	0.6085	0.9044	0.7879	0.7326	0.3613	0.6473	0.8028	0.3924	0.6116	0.6347	0.3317
EPL	0.2292	0.2241	0.1505	0.6500	0.6875	0.6023	0.4151	0.8571	0.8349	0.3114	0.8720	0.8312
GameOfLife	0.5278	0.4247	0.5889	0.7726	0.7322	0.4796	0.7235	0.8290	0.6641	0.7328	0.8022	0.5383
GPL	0.7369	0.4317	0.4180	0.7040	0.5660	0.2162	0.5675	0.6168	0.4159	0.5160	0.5791	0.2441
GUIDSL	0.5254	0.5748	0.5509	0.5637	0.7585	0.3745	0.5609	0.8099	0.5132	0.6244	0.7286	0.3055
MobileMedia8	0.4463	0.4875	0.7330	0.7297	0.6681	0.4916	0.7423	0.7152	0.3676	0.5566	0.6008	0.3798
Notepad	0.6483	0.3886	0.4349	0.5995	0.5714	0.2308	0.4492	0.5233	0.3940	0.4586	0.6151	0.4339
PKJab	0.4070	0.4300	0.7687	0.6655	0.7639	0.3866	0.3748	0.5870	0.4365	0.3905	0.4385	0.3026
Prevayler	0.4603	0.4362	0.6701	0.6691	0.9248	0.4000	0.5276	0.6531	0.4373	0.5343	0.6018	0.2029
Raroscope	0.2430	0.4721	0.5000	0.5000	0.5455	0.2500	0.4635	0.6341	0.4317	0.3892	0.4012	0.3107
Sudoku	0.5022	0.3517	0.5161	0.5120	0.5582	0.2424	0.3403	0.5765	0.3173	0.4237	0.5999	0.1515
TankWar	0.5313	0.5762	0.6611	0.6099	0.5138	0.4176	0.4238	0.5575	0.5356	0.4341	0.5676	0.5136
Violet	0.4219	0.5625	0.8050	0.5822	0.5483	0.4495	0.3472	0.7734	0.8284	0.3563	0.7270	0.7660
ZipMe	0.4331	0.4778	0.7520	0.5845	0.6805	0.2798	0.3974	0.7572	0.3959	0.3718	0.4386	0.1882

Table 8: Upper Gini error bounds (.975 quantile); we computed the empirical .975 quantile based on a standard leave-one-out Jackknife resampling to set the upper error bounds when comparing Gini coefficients of different product lines or different decompositions.

	Unit size			CBU			IUD			EUD		
	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU	CU	CUF	FU
AHEAD	1.1149	0.9579	1.1215	1.1543	1.1750	0.8956	0.6035	0.6833	0.8794	0.5766	0.7216	0.7491
BCJak2Java	1.1763	1.1077	1.0019	1.1054	1.1352	1.1278	0.4712	0.5537	0.9303	0.4868	0.5342	0.9767
Jak2Java	1.1538	1.1266	1.0233	1.0820	1.1327	0.9129	0.5273	0.5691	1.0341	0.4973	0.5595	1.0332
Jampack	1.2383	0.9342	1.0762	1.0067	1.1131	0.7231	0.5747	0.6034	0.8913	0.4968	0.6121	0.9962
JREName	1.2029	1.0826	0.9720	1.2023	1.2479	1.2807	0.4725	0.5468	0.9240	0.4765	0.5021	0.6936
Mixin	1.1765	1.1125	1.0180	1.1284	1.1939	0.7480	0.5406	0.5673	0.8723	0.4606	0.5655	0.9401
MMatrix	1.1882	1.0990	1.0113	1.1208	1.1801	1.0123	0.5036	0.5610	1.0027	0.5106	0.5282	1.0304
UnMixin	1.2145	1.0898	0.9808	1.1483	1.2206	0.8489	0.4665	0.5408	0.8670	0.4886	0.5150	1.1801
AJStats	1.0544	1.0805	1.1181	0.8572	0.8262	0.2031	0.7575	0.7292	1.1240	0.7065	1.0491	1.3357
Bali2Jak	1.1151	1.1317	0.9840	1.0385	1.0607	0.4907	0.5973	0.7037	1.1277	0.5499	0.6387	0.9720
Bali2JavaCC	1.1793	1.1316	1.0176	1.0347	1.0715	0.6504	0.5838	0.6908	1.2330	0.5555	0.6454	0.9146
Bali2Layer	1.1661	1.1169	1.0278	1.0216	1.0780	0.7052	0.6040	0.6886	1.2529	0.5582	0.6189	0.9744
Bali	1.1465	1.1101	1.0726	0.9984	1.0647	0.7242	0.5875	0.7090	1.0517	0.5431	0.6738	0.9826
BaliComposer	1.1475	1.1866	0.9584	1.0352	1.0672	0.5721	0.5667	0.6884	1.2154	0.5709	0.6442	1.0550
BerkeleyDB	1.0452	1.0144	1.0707	0.8772	0.7523	1.3468	0.6672	0.8501	0.7393	0.6637	0.6641	1.0095
EPL	0.7432	1.1937	1.1400	1.1911	0.9911	0.9861	1.1193	0.8800	0.8585	0.5888	0.8833	0.8588
GameOfLife	1.0355	1.0164	1.0323	0.6735	0.8565	1.2228	0.6964	0.7583	0.9321	0.6964	0.7084	0.7673
GPL	1.0078	1.0669	1.0906	1.0279	0.6269	0.9214	0.6779	0.8031	0.8093	0.5568	1.0141	0.4839
GUIDSL	1.0338	1.0289	0.8782	0.9818	0.7999	0.7437	0.6211	0.8064	0.8468	0.8400	0.7565	0.8028
MobileMedia8	0.9047	1.0441	1.0612	0.6908	0.7475	0.7758	1.1173	0.8332	0.6096	0.7080	0.6975	0.8522
Notepad	0.9784	0.9074	0.7573	0.7512	0.8508	0.3955	0.6997	0.6418	1.0361	0.9997	0.8966	0.7663
PKJab	1.0101	0.9504	1.1324	0.8655	0.9023	1.3298	0.9473	0.8312	1.0471	0.8777	0.8873	0.9787
Prevayler	1.0502	0.9602	1.0856	0.7180	0.8732	0.8083	0.6026	0.7975	0.8961	0.6292	0.7350	0.9656
Raroscope	1.3587	1.2078	1.3161	0.6667	1.1867	1.4971	1.1546	1.0328	0.7961	1.1501	0.5849	0.5908
Sudoku	1.0809	1.0964	1.1461	0.8046	0.8555	0.9373	0.6417	0.7316	1.0416	0.6087	0.8551	0.7076
TankWar	1.0362	1.0510	1.1676	0.8489	0.6652	0.8961	0.7709	0.7858	0.7232	0.7217	1.0735	0.8304
Violet	1.0163	0.9260	0.9822	0.6379	0.8126	1.0453	0.9071	0.8285	0.8138	0.8607	0.8042	0.7533
ZipMe	1.0224	0.8894	1.1108	0.8836	0.7186	0.7864	0.6745	0.9701	0.9099	0.7249	0.6691	1.1270

Table 9: Lorenz Asymmetry Coefficients; see Section 3.4