Feature Cohesion in Software Product Lines: An Exploratory Study

Sven Apel
University of Passau, Germany

Dirk Beyer
University of Passau, Germany
Simon Fraser University, B.C., Canada

ABSTRACT

Software product lines gain momentum in research and industry. Many product-line approaches use features as a central abstraction mechanism. Feature-oriented software development aims at encapsulating features in cohesive units to support program comprehension, variability, and reuse. Surprisingly, not much is known about the characteristics of cohesion in feature-oriented product lines, although proper cohesion is of special interest in product-line engineering due to its focus on variability and reuse. To fill this gap, we conduct an exploratory study on forty software product lines of different sizes and domains. A distinguishing property of our approach is that we use both classic software measures and novel measures that are based on distances in clustering layouts, which can be used also for visual exploration of product-line architectures. This way, we can draw a holistic picture of feature cohesion. In our exploratory study, we found several interesting correlations (e.g., between development process and feature cohesion) and we discuss insights and perspectives of investigating feature cohesion (e.g., regarding feature interfaces and programming style).

Categories and Subject Descriptors: D.2.8 [**Software**]: Software Engineering—*Metrics*; D.2.11 [**Software**]: Software Architectures—*Domain-Specific Architectures*

General Terms: Design, Measurement

Keywords: Feature-Oriented Software Development, Software Product Lines, Feature Cohesion, Visual Clustering, FEATUREVISU

1. INTRODUCTION

A growing community of software-engineering researchers and practitioners theoretically investigates and industrially develops software systems as product lines. A *software product line* is a set of software-intensive systems of a domain that share a common set of features [13]. A *feature* is an end-user–visible program characteristic that is relevant to the stakeholders of the application domain. Features are used to describe the commonalities and variabilities of the products of a product line [17]. For example, in a database product line, individual database systems share a common set of features (e.g., basic data structures) but differ in other features (e.g., transaction management).

Recently, there have been many attempts to make features explicit in the product line's code base [1]. For example, compositional approaches such as component-based systems and feature-oriented programming encapsulate the code that belongs to a feature (not more, not less) in a cohesive and composable unit. Annotative approaches such as preprocessors and frame processors tag code that belongs to a feature with annotations and allow programmers to remove or include code conditionally.

Compositional and annotative approaches have many individual strengths and weaknesses [19], but they share the common goal of making features explicit in design and code, and exploit this property across the development process (e.g., for editing, generation, and type checking). Ideally, the features of a system align with the underlying system structure [27] (e.g., the class structure), although this is not entirely possible due to crosscutting features (e.g., transaction management in a database system) [3,25]. We define *feature cohesion* as the degree to which the elements (e.g., methods, fields, classes) of a feature depend on other elements of the same feature.

In the past, it has been assumed that decomposing a system according to its features improves naturally the quality of the system structure and yields benefits in terms of understandability and maintainability [1]. However, we argue that a misalignment of features and system structure can outweigh the benefits of feature decomposition. In the development and analysis of several product lines [2, 18, 20, 21, 25] we witnessed examples in which features align very well with the system's structure, that is, the average feature cohesion is relatively high, and other examples in which the opposite is true. However, little is known on how product lines are structured and how a product line's structure aligns with its features. But this is of special interest because —compared to engineering standalone software products— software product-line engineering particularly aims at variability and reuse.

To investigate the characteristics of feature cohesion, we conduct an exploratory study on forty software product lines of different sizes and domains. As a technological basis, we use our exploration and measurement tool FEATUREVISU, which computes all measures that we use in our exploratory study, and which can visually relate the structural elements of a product line to its features using layout-based clustering techniques. The idea is that, beside classic cohesion-based measures [11,30], layout-based clustering can provide additional insights into the structure of software product lines and, in particular, into feature cohesion. Our novel feature-cohesion–based measures consider not only the *number* of references between program elements but also their *distances* in a clustering layout (*cohesive* elements are drawn closely together in such layouts, *long-distance* references do not witness cohesion).

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

In our exploratory study, we are interested in a number of research questions concerning the characteristics of feature cohesion:

- How well does the system structure of a product line align with its features?
- Are there significant differences in feature cohesion between the features of a product line and between all product lines of our sample?
- Does the feature or system size in terms of lines of code or number of features correlate with feature cohesion?
- Does the development process (i.e., whether a product line has been developed from scratch or refactored from a legacy system) influence feature cohesion?

Providing answers to these questions can aid programmers, language designers, and tool builders in their quest to develop well-structured software, as we will discuss. To this end, we use FEATUREVISU in two ways: (1) interactively, to visualize and explore the structure of software product lines, and (2) for automatic processing, to conduct a quantitative analysis on feature cohesion.

Overall, we make the following contributions towards understanding feature cohesion:

- We adopt several classic cohesion-based measures to understand and assess *feature cohesion* in software product lines. To provide a more holistic view, we additionally introduce distance-based measures derived from clustering layouts.
- We offer the open-source tool FEATUREVISU, which computes values for all measures that we use in this work (classic and distance-based). Furthermore, FEATUREVISU can be used to *visually explore* the structure of product lines, especially with regard to feature cohesion.
- We study *forty software product lines* of different sizes and domains by exploring their structure and by investigating feature cohesion using classic measures and distance-based measures. Based on the collected and analyzed data, we answer a number of research questions regarding system structure, feature cohesion, and correlation factors.
- We discuss perspectives of our results and possible research directions, especially with regard to information hiding, programming guidelines, and tool support.

FEATUREVISU and all experimental data for reproducing our results are available at our supplementary project website: `http://www.fosd.de/FeatureVisu`.

2. PRELIMINARIES

In this section, we describe our holistic approach to assess feature cohesion using classic measures as well as distance-based measures that are derived from clustering layouts.

2.1 Feature Cohesion

Feature cohesion is the degree to which the elements (e.g., methods, fields, classes) of a feature depend on other elements of the same feature. This definition is derived from the classic notion of cohesion in chemistry, where cohesion is an attracting force between molecules that tries to keep the molecules together. *Feature coupling* is the degree to which the elements of a feature depend on elements outside the feature. Feature coupling acts against feature cohesion: if the coupling to elements outside the feature overcompensates the cohesion between the elements of the feature, then the structure of the system disappears; if the cohesion is stronger than the coupling, structure emerges in the system.

The cohesion-coupling ratio can be considered as a good indicator for structuredness. In the spirit of classic work on software structure [24,26], features with a high cohesion-coupling ratio (i.e., features whose elements depend on elements that are mostly of the same feature) have a positive effect on software quality, because program units with high cohesion and low coupling can be changed almost in isolation.

A simple approach to assess feature structure is to relate the number of dependencies between elements inside a feature (internal dependencies) to the number of dependencies to elements outside that feature (external dependencies). Another simple approach is to relate the number of dependencies inside a feature to the overall number of elements of that feature. We adopt these simple indicators to features and product lines but, in addition, we pursue a more holistic approach. The idea is to view the relations between features from a global perspective. The elements of features and their dependency relation form a dependency graph (which models calls, usage, inheritance, etc. as edges in the graph). Typically, the dependency graph is a highly connected network with edge degrees in the order of several hundreds. Furthermore, such a graph is generally not regular, but —if structure exists— the graph contains clusters (i.e., groups of elements that heavily depend on each other); such groups can be identified by layout-based clustering using light-weight tools [7]. In addition to classic measures, we use information obtained from a clustering layout to explore and assess feature cohesion. Before we define the measures that we use later as indicators for feature cohesion, we give a brief introduction to layout-based clustering.

2.2 Layout-Based Clustering

Overview. *Clustering* partitions a set of elements into subsets according to certain properties. If we choose the nodes of the dependency graph of a software system as the set of elements to be clustered, and instruct the clustering algorithm to partition this set according to the graph structure (i.e., highly connected nodes shall be in the same cluster), then we obtain a decomposition of the software system according to the dependencies in the graph. Work on *layout-based clustering* (a.k.a. *visual clustering*) goes one step ahead and argues that it is not sufficient to *partition* a set into equivalence classes. Layout-based clustering considers *distances* between elements in a two-dimensional space, in which related elements have close positions and unrelated elements have distant positions. The mapping from nodes to positions is called *layout*. The advantage of a clustering layout is that it reveals a degree of relatedness: two elements can be very close (or not so close) in the same cluster, or belong to two different clusters that are neighbors (or distant clusters), or anywhere in between. Clustering criteria for layout-based clustering have been formally and thoroughly investigated by Noack [23], and the method was applied to software graphs in CCVISU [7, 10], which is the technological basis of FEATUREVISU.

Applied to features and product lines, layout-based clustering provides a holistic view on feature structure. Intuitively, a feature has a higher cohesion than coupling if its elements are close to each other, because then they are connected by many internal edges. If the elements that a feature introduces are scattered across the entire layout, then the cohesion of the feature is lower than its coupling to other features. Thus, we use the distances computed by a layout-based clustering algorithm to assess feature structure in software product lines, complementary to classic indicators for structure.

Technical Details. A clustering layout can be computed by force-directed graph drawing, in which an energy model ensures that the drawn layout fulfills the required clustering properties. The force-directed approach consists of two parts: an energy model that maps a layout to an energy value that is used for evaluation —the smaller the number, the better the layout— and an algorithm that computes a layout with minimal energy.

A layout of a graph G is a function p that maps each node of the graph to a position in the two-dimensional space. An energy model is a function U that assigns to each layout p a real number. The layout p is the best layout for G if $U(p)$ is the global minimum of function U. The energy model encodes the layout goal, that is, the user's choices of what is considered as good layout. For clustering, this means to produce layouts that provide separation of cohesive subgraphs and interpretable distances. FEATUREVISU uses Noack's clustering energy model, which has been successfully applied to layout-based clustering in a number of different domains [6, 7, 10, 23]:

$$U(p) = \sum_{\{v,w\}\in E} ||p(v)-p(w)|| + \sum_{\{v,w\}\in V^{(2)}} -\deg(v)\cdot\deg(w)\cdot\ln||p(v)-p(w)||,$$

where function $p : V \to \mathbb{R}^2$ is a layout, $U(p)$ is the energy of p, $||p(v)-p(w)||$ is the Euclidean distance of the nodes v and w in p, $\deg(v)$ is the edge degree of a node v (number of edges incident to v), and $V^{(2)} = \{\{v,w\} \mid v \in V \wedge w \in V \wedge v \neq w\}$ is the set of all possible undirected edges of nodes from V. The first term of the sum is interpreted as attraction between connected nodes, because its value decreases when the distance of such nodes decreases. The second term is interpreted as repulsion between all pairs of (different) nodes, because its value decreases when the distance between any two nodes increases. The repulsion of each node v is weighted by the edge degree $\deg(v)$ to avoid a bias to place nodes with heavy edge degree in the center of the layout. Noack has formally shown that such so-called LinLog energy models reveal clusters of a graph naturally. That is, if the graph contains clusters, then the clusters of the graph are visually identifiable in the layout as separated groups of highly connected nodes [23].

An energy minimizer —the second part of the layout-based approach— is an optimization algorithm that searches for a good approximation of the best layout. The energy minimizer starts with an initial layout, in which the positions of the nodes are randomly assigned. Then, in every iteration, the algorithm tries to improve the layout according to the energy model—by using the first derivation of the energy function (i.e., the force) to compute a direction and a distance for the new placement of each node.

Tool Support. For our exploratory study, we extended the existing visual-clustering tool CCVISU [7] by support for features and call this tool extension FEATUREVISU. Visual clustering and CCVISU have been used previously to decompose software graphs into subsystems [6, 10]. It has been shown that visual clustering can aid program comprehension by visualizing the software design based on distances in the clustering layout [6, 10, 23].

FEATUREVISU receives as input the dependency graph of a software product line and a mapping between element nodes and features. A node may belong to multiple features to resemble the situation in which a method contains statements of multiple features, or multiple features share single fields or methods, which may happen when features interact structurally [20]. The tool optimizes the layout of the dependency graph iteratively by grouping element nodes that depend on each other. The dependency graph spans a nontrivial network, in which many forces take effect simultaneously. This illustrates the global view that we pursue with the layout-based clustering approach: feature cohesion is affected not only by numbers of local references but by the global effects of forces caused by dependencies that manifest themselves in interpretable distances.

Example. In Figure 1, we show an excerpt of the clustered dependency graph of the product line BALI2JAK (for grammar processing) of our exploratory study, computed and rendered by FEATUREVISU. The discs (nodes of the graph) represent the fields and methods

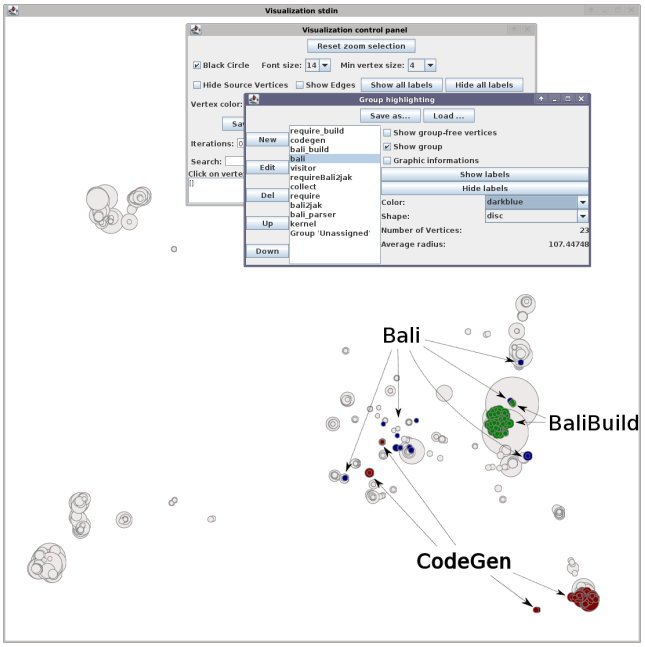

Figure 1: A clustering layout for the product line BALI2JAK (features Bali, BaliBuild, and CodeGen are highlighted with arrows and in blue, green, and red, respectively)

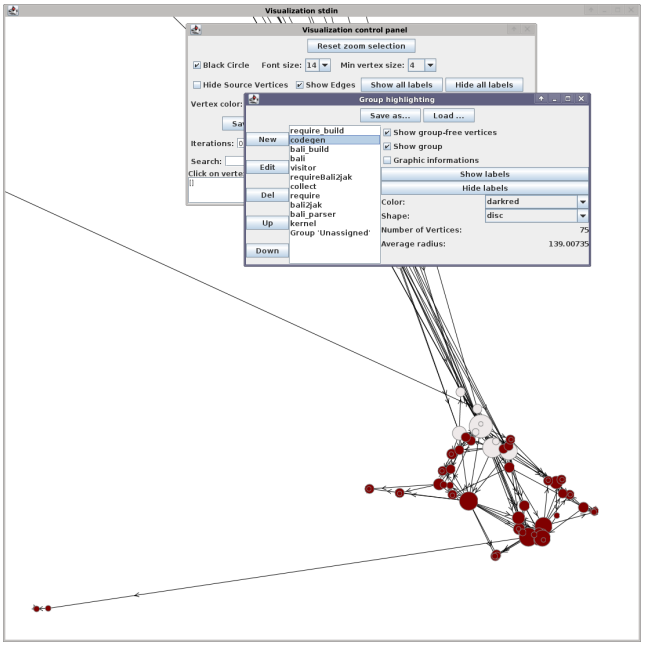

Figure 2: An excerpt of the clustering layout for the product line BALI2JAK including edges (feature CodeGen is highlighted in red; edges model dependencies)

of the system. The area of a disc for a node is proportional to the node's edge degree. If the discs form clusters, then the corresponding fields/methods heavily depend on each other. Initially, the color of all nodes is light gray in FEATUREVISU. For illustration, we have colored the nodes belonging to feature Bali in blue, to feature BaliBuild in green, and to feature CodeGen in red. To illustrate the forces that drive the clustering, we display in Figure 2 an excerpt of the layout of BALI2JAK, now including edges (dependencies). Both, node coloring and displaying edges are supported by FEATUREVISU — besides zooming, drag & drop, tool tips, etc.

Figures 1 and 2 illustrate how FEATUREVISU can be used to explore the feature structure in software product lines. The idea is to compare clusters and features (represented by colors). Features whose elements form clusters are more cohesive than coupled. For example, the elements of feature BaliBuild form a compact cluster; it is largely self-referential and thus cohesive. Conversely, features whose elements are scattered across the layout are highly coupled. For example, the elements of feature Bali are scattered across the entire layout; it shares many dependencies with other features and has thus a much smaller cohesion-coupling ratio than BaliBuild. Feature CodeGen is somewhere in between of Bali and BaliBuild, in terms of structuredness: it consists of a compact cluster and some elements scattered across the layout.

Visualizing the cross-linking between the elements of different features helps developers to explore the reasons for a particular clustering, for example, to get insights into why a feature is not cohesive and how to change that. Next, we formulate the relationship between clusters in the layout and the structuredness quantitatively.

2.3 Indicators for Feature Cohesion

Layout-based clustering can aid program comprehension [6, 10, 23]. However, only displaying the layouts of product lines is not sufficient to understand and compare feature cohesion systematically. Hence, we pursue a quantitative approach in addition. There is no measure that fully captures structuredness (cf. Sec. 4). Instead, there are various complementary indicators that address only particular aspects of cohesion or coupling. We contribute indicators for structuredness based on clustering layouts.

We represent a software system by its dependency graph $G = (V, E)$, where the set V of nodes represents the elements of the system (e.g., methods, fields, classes) and the set $E \subseteq V \times V$ of edges represents the dependency relation between elements of the system. Here, we are interested in feature-oriented product lines, so we use a mapping $elems : \mathscr{F} \rightarrow 2^V$ in order to assign to each feature $F \in \mathscr{F}$ a set $elems(F)$ of program elements (graph nodes) that F introduces to the software system. We assume that every feature introduces at least one element and that every element is introduced by at least one feature, possibly by multiple features.

Feature cohesion is the sum of attracting forces that try to keep the elements of a feature together. In the following, we are interested in the dependency relation between elements of a feature and from elements of a feature to elements outside the feature. Before we introduce a number of measures designed to indicate certain aspects of feature cohesion, we provide some auxiliary definitions.

Function $dd(v) = |dep(v)|$, with $dep(v) = \{(v,w) \mid (v,w) \in E\}$, measures the number of dependencies that have impact on element v (i.e., that relate v to other elements). This function was used before as indicator for structural problems and complex dependencies at the code level [9]; we adopt and extend it to features.

Function $ID(F) = |intdep(F)|$, with $intdep(F) = \{(v,w) \mid (v,w) \in E \wedge v \in elems(F) \wedge w \in elems(F)\}$, measures the number of dependencies (including self-references) that attract the elements of a feature together. In order to compare the values for different features, we normalize the measurement value in the following by the number of potentially possible dependencies in feature F, that is, $|\{(v,w) \mid v \in elems(F) \wedge w \in elems(F)\}| = |elems(F)|^2$.

Internal-ratio Feature Dependency (*IFD*). The first measure that we use in our exploratory study is the *internal-ratio feature dependency*, which measures the number of internal dependencies in relation to the total number of potentially possible *internal* dependencies of a feature:

$$IFD(F) = \frac{|intdep(F)|}{|elems(F)|^2} \tag{1}$$

Function $intdep(F)$ returns all dependency edges from elements of feature F to elements of feature F; $|elems(F)|^2$ is the maximum possible number of dependency edges. The intuition behind this measure is that the elements of a cohesive feature depend on many other elements of the same feature. So, for a feature F with three elements each depending on all elements of F (including self-references), we have $IFD(F) = 1$, which indicates that F is maximally cohesive. Conversely, for a feature F with three elements, none depending on any other element of F, we have $IFD(F) = 0$, which indicates that F is not cohesive. This indicator can be seen as a feature-oriented variant of established software measures that address relative cohesion in object-oriented and aspect-oriented systems [11, 30] (Sec. 4). The indicator *IFD* takes only internal references between elements of a feature into account. Hence, we use a further measure that relates the internal references to all references.

External-ratio Feature Dependency (*EFD*). The measure *external-ratio feature dependency* measures the number of internal dependencies in relation to the total number of actual dependencies (internal and *external*) of a feature:

$$EFD(F) = \frac{|intdep(F)|}{|dep(F)|} \tag{2}$$

Function $dep(F) = \sum_{v \in F} dep(v)$ returns all dependencies of elements of a feature F. If a feature F depends only on elements outside the feature, we have $EFD(F) = 0$, which indicates that F is not cohesive. Conversely, if F depends only on itself, we have $EFD(F) = 1$, which indicates that F is cohesive. This measure is an adaptation of previous work to features [11, 30] (Sec. 4).

Both measures (*IFD* and *EFD*) take only the numbers of elements and references into account — not the distances in the clustering layout. Using *IFD* and *EFD* only, distance information obtained by a clustering algorithm is not considered. The clustering approach allows us to draw a more holistic picture of feature cohesion and the effects of feature decomposition on the system structure. Layout-based clustering is essentially about relative distances between nodes in a two-dimensional space. Dependencies are interpreted as forces, and geometric proximity represents feature cohesion. Consequently, we define a set of new measures that take these aspects into account.

Distance-based Internal-ratio Feature Dependency (IFD_W) and Distance-based External-ratio Feature Dependency (EFD_W). The first attempt is to modify the measures *IFD* and *EFD*, such that they incorporate distance information obtained from a corresponding clustering layout. Specifically, we penalize *long* references to internal and reward *long* references to external elements.

For *IFD*, we obtain the distance-based measure IFD_W:

$$IFD_W(F) = \frac{\sum_{(v,w) \in intdep(F)} \left(diag(p) - \left|\left| p(v) - p(w) \right|\right| \right)}{diag(p) \cdot |elems(F)|^2} \tag{3}$$

First, we replaced the numerator of Equation (1) by a term that considers the length of internal dependencies: $diag(p) - ||p(v) - p(w)||$ measures the length of dependency (v,w) and subtracts it from the maximal length of a reference (diagonal *diag* of p) in order to get large values if the internal reference is short. Second, the sum of these distances is normalized by the product of the longest possible reference (*diag*) and the number of potentially possible references.

Analogously, we define the distance-based measure EFD_W by replacing the terms $|intdep(F)|$ and $|dep(F)|$ by their distance-based counterparts:

$$EFD_W(F) = \frac{\sum_{(v,w)\in intdep(F)} \Big(diag(p) \; - \; \big|\big|p(v) - p(w)\big|\big| \Big)}{\sum_{(v,w)\in dep(F)} \Big(diag(p) \; - \; \big|\big|p(v) - p(w)\big|\big| \Big)} \quad (4)$$

Normalized Average Radius (*NAR*) and Normalized Maximum Radius (*NMR*). The distance-based variants of our measures take both the number of references and their lengths into account. To provide a further complementary perspective, we define two measures that are based solely on relative distances in the layout, without considering the actual numbers of references. To this end, we exploit the notion of a *dependency radius* of a feature in the clustering layout to make statements about its cohesion. The dependency radius of a feature F is based on the distances of the elements of F from the barycenter $bc(F)$, where $bc(F)$ is the arithmetic mean over all positions of the elements of F.

Specifically, we define the measures *normalized average radius* (*NAR*) and *normalized maximal radius* (*NMR*):

$$NAR(F) = 1 - \frac{\underset{v\in elems(F)}{\text{mean}} \big|\big|(p(v) - bc(F)\big|\big|}{diag(p)} \quad (5)$$

$$NMR(F) = 1 - \frac{\underset{v\in elems(F)}{\text{max}} \big|\big|p(v) - bc(F)\big|\big|}{diag(p)} \quad (6)$$

Both measures are *normalized* by dividing the distance-based term by the diagonal of the layout and then by subtracting the result from 1. Thus, the measure ranges from 0 to 1. A feature with only one single and well-separated cluster has a normalized radius close to 1, which indicates that the feature is cohesive. Conversely, a feature consisting of many elements scattered across the layout has a normalized radius close to 0, which indicates that the feature is not cohesive. We use both the average radius and the maximal radius to neglect or consider outliers. We refer to the two measures as *radius-based measures*.

2.4 Summary

We defined three classes of measures to quantify feature structuredness in software product lines. *IFD* and *EFD* are based solely on ratios of internal and external references. IFD_W and EFD_W are novel in that they additionally take into account the distances between nodes in a corresponding clustering layout, and *NAR* and *NMR* are novel in that they are completely based on distance information. All our measures are interval-scaled. They are based on a substantial body of previous work (cohesion measures and layout-based clustering), which provides evidence in the form of formal proofs and empirical studies that the measures are sound and useful (see Sec. 4). A novelty of our approach is to adapt the three classes of measures to the needs of software product-line engineering and to apply them in an exploratory study to draw a holistic picture of feature cohesion in software product lines.

3. AN EXPLORATORY STUDY ON FORTY SOFTWARE PRODUCT LINES

By means of a study on forty software product lines of different sizes and domains, we explore the characteristics of feature cohesion in software product lines as well as its correlation with factors such as feature size, system size, and development process. Note that we explore the structure of the code base of a product line, rather than the code of the individual products that have been generated from the product line's code base. The reason is that cohesion is relevant for program comprehension, so we have to consider what the developer sees and not the generated code that is deployed.

We briefly introduce our tool FEATUREVISU, outline the sample product lines, present and interpret the results of our quantitative analysis, and discuss threats to validity and perspectives.

3.1 FeatureVisu

FEATUREVISU extends the general-purpose visual-clustering tool CCVISU [7] by support for features. It expects as input the dependency graph of a product line as well as a mapping between program elements and features (both in relational standard format—RSF). This way, we can abstract from concrete product-line implementation techniques such as annotative approaches (e.g., the C preprocessor) and compositional approaches (e.g., feature-oriented programming).

FEATUREVISU supports many useful user interactions, for example, to adjust the number of iterations of the layout engine, change a feature's color, display edges and node names. In Section 2.2, we have already illustrated how FEATUREVISU is used to explore the structure of a product line. Next, we introduce the sample product lines of our study.

3.2 Sample Product Lines

We collected a sample of forty product lines of different domains and system sizes, developed by refactoring or developed from scratch. Table 1 provides information regarding domain, size, and development process. All sample product lines are available on the web. [1] All product lines are based on Java, developed with the product-line tools AHEAD [5], FEATUREHOUSE [2], and CIDE [18]. We extracted dependency graphs and the mapping between elements and features using the tools FEATUREHOUSE, DOXYGEN, and CCVISU, so the effort for collecting the input data was moderate.

Interestingly, some of the product lines are related. BCJAK2JAVA, JAK2JAVA, JAMPACK, JRENAME, MIXIN, MMATRIX, and UNMIXIN belong to the AHEAD tool suite, which is in fact a product line of product lines [4]. So, they share certain basic features such as parsers. Similarly, BALI2JAK, BALI2JAVACC, BALI2LAYER, and BALICOMPOSER belong to the BALI tool suite, which is also a product line of product lines [4]. Furthermore, CHATSYSTEM comes in eight different variants. The variants have been developed independently in a course on modern programming paradigms at the University of Magdeburg. The same applies to NOTEPAD, which comes in seven variants, independently developed in a course on feature-oriented design at the University of Texas at Austin. The relations between some sample product lines give us the opportunity to explore whether similarities between product lines manifest similar degrees of feature cohesion.

3.3 Measurements

In a first step, we loaded the data (dependency graphs and feature mappings) of each sample product line into FEATUREVISU and computed a clustering layout (two-dimensional). Then, we colored nodes of individual features and displayed selected sets of edges (cf. Fig. 1). This way, we got an impression of the differences between individual features and individual product lines with regard to feature cohesion (and there are considerable differences, as our analysis confirms). In a second step, we conducted a quantitative analysis. To this end, we extended FEATUREVISU such that, based on a clustering layout, it is able to calculate the measures defined in Section 2.3. We collected data for all features of all

[1] `http://www.fosd.de/FeatureVisu/`

No.	Product Line	Domain	LOC	FS	DP
1	AHEAD/BCJak2Java	progr. tool	32 326	15	S
2	AHEAD/Jak2Java	progr. tool	32 934	16	S
3	AHEAD/Jampack	progr. tool	34 326	21	S
4	AHEAD/JREName	progr. tool	31 120	17	S
5	AHEAD/Mixin	progr. tool	32 493	17	S
6	AHEAD/MMatrix	progr. tool	32 228	13	S
7	AHEAD/UnMixin	progr. tool	31 658	12	S
8	AJStats	analysis tool	15 311	20	S
9	Bali/Bali2Jak	grammar tool	13 527	11	S
10	Bali/Bali2JavaCC	grammar tool	14 139	11	S
11	Bali/Bali2Layer	grammar tool	13 811	12	S
12	Bali/BaliComposer	grammar tool	12 197	10	S
13	BerkeleyDB	database system	64 652	99	R
14	ChatSystem/Burke	network client	614	6	S
15	ChatSystem/Dreiling	network client	938	5	S
16	ChatSystem/Becker	network client	651	7	S
17	ChatSystem/Weiss	network client	931	7	S
18	ChatSystem/Schink	network client	873	7	S
19	ChatSystem/Luong	network client	862	9	S
20	ChatSystem/Rehn	network client	760	6	S
21	ChatSystem/Thuem	network client	544	8	S
22	EPL	expression eval.	99	12	S
23	GameOfLife	game	1 656	14	R
24	GPL	graph library	791	27	S
25	GUIDSL	config tool	13 573	26	S
26	MobileMedia8	multimedia	5 278	51	R
27	Notepad/Quark	text editor	1 397	10	R
28	Notepad/Delaware	text editor	1 654	6	R
29	Notepad/Wellington	text editor	1 522	4	R
30	Notepad/Svetoslav	text editor	1 627	6	R
31	Notepad/Wehrman	text editor	1 716	5	R
32	Notepad/Guimbarda	text editor	1 586	9	R
33	Notepad/Robison	text editor	1 404	10	R
34	PKJab	network client	4 994	8	R
35	Prevayler	database system	6 867	6	R
36	Raroscope	compression lib	428	5	R
37	Sudoku	game	1 850	7	R
38	TankWar	game	3 184	15	S
39	Violet	model editor	9 789	88	R
40	ZipMe	compression lib	5 479	35	R

Table 1: Overview of the sample product lines (LOC: number of lines of code; FS: number of features; DP: development process—S: from scratch; R: refactored)

product lines. Due to the sheer amount of raw data, we can provide only a condensed view. However, the raw data are available on FeatureVisu's website.

In Figure 3, we present values for the measurements of *IFD* and *EFD*. On the x-axis we display the product lines (indexes correspond to column 'No.' in Table 1) and on the y-axes we present their average *IFD* and *EFD* values (i.e., we calculated the mean of all feature values per product line). Analogously, we present numbers for the measurements of IFD_W and EFD_W in Figure 4 and of *NAR* and *NMR* in Figure 5.

For illustration, Table 2 contains the measured values for the features Bali, BaliBuild, and CodeGen from our example in Figure 1. All three features have a low or no internal feature dependency. Exploring the layout and the code, we found that they are 'providers' in the sense that they bundle data and functions for other features. External feature dependency is higher than internal feature dependency for all three features. An exceptional case is feature CodeGen that has no external references and is thus isolated. Looking at the layout and source code, we found that the functionality of this feature is not used by other features except from the outside. Finally, the radius-based measures are as expected, when looking at Figure 1 and Table 2.

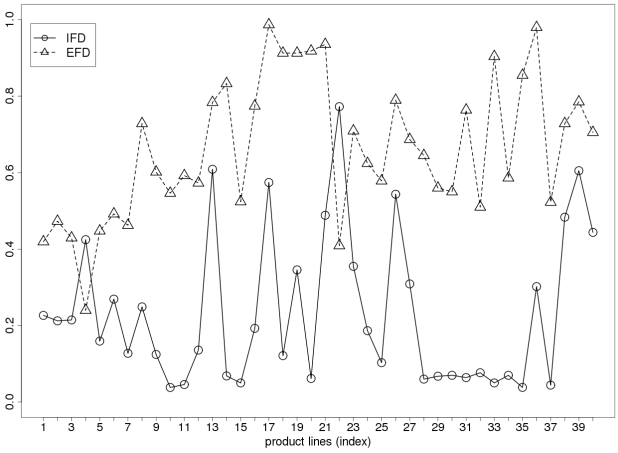

Figure 3: Average *IFD* and *EFD* values for the samples

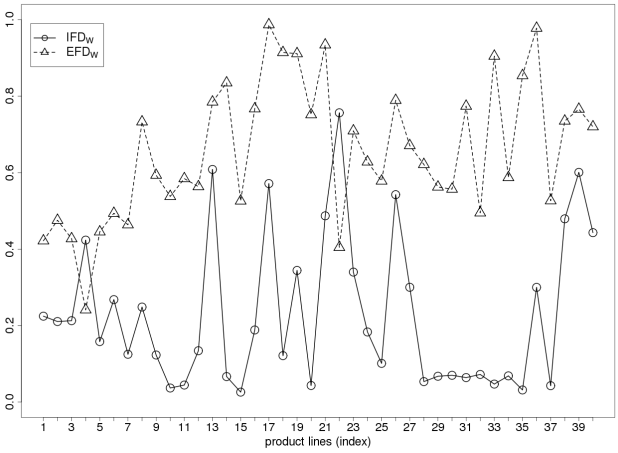

Figure 4: Average IFD_W and EFD_W values for the samples

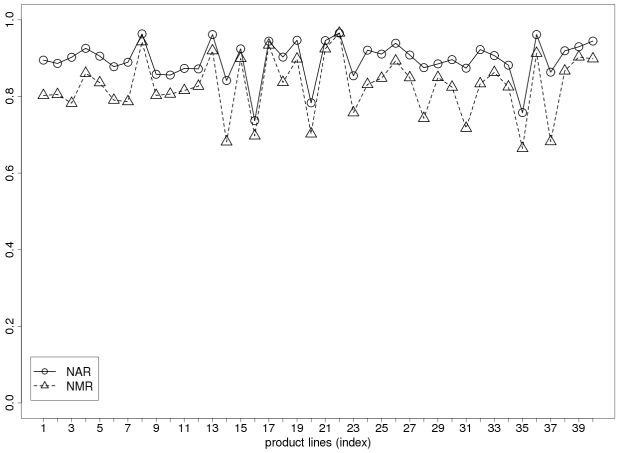

Figure 5: Average *NAR* and *NMR* values for the samples

Feature	*IFD*	IFD_W	*EFD*	EFD_W	*NAR*	*NMR*
Bali	0.02	0.01	0.5	0.38	0.61	0.57
BaliBuild	0	0	0.3	0.3	0.99	0.99
CodeGen	0.02	0.02	1	1	0.79	0.7

Table 2: Measurements of the features Bali**,** BaliBuild**, and** CodeGen **of the example of Figure 1**

	IFD	IFD_W	*EFD*	EFD_W	*NAR*	*NMR*
IFD	1.00	1.00	0.27	0.28	0.46	0.51
IFD_W		1.00	0.27	0.28	0.47	0.51
EFD			1.00	0.99	-0.15	-0.08
EFD_W				1.00	-0.11	-0.07
NAR					1.00	0.87
NMR						1.00

Table 3: Pearson's correlation coefficients (measure vs. measure)

3.4 Statistical Analysis

Before we interpret the results, we conduct and discuss a number of significance and correlation tests (all using the statistics tool R [2]).

Significance. Looking at the data, we observed that the sample product lines differ considerably in terms of our indicators. To confirm that the differences between them are indeed significant (rather than being coincidental), we used the Kruskal-Wallis test (a statistical significance test for variance analysis of multiple samples) because the data are generally not normally distributed (tested with the Shapiro-Wilk test). Specifically, we applied the Kruskal-Wallis test (as well as the Shapiro-Wilk test) on a per-measure basis to the entire set of measured values of the individual features. We found that the data sets of all measures pass the test (all p-values are lower than 10^{-10}). That is, the individual product lines indeed differ with regard to the individual measures.

Measure vs. Measure. A further interesting question is whether the individual measures correlate. Of course, it is easy to see that *IFD* and *EFD* are quite similar to their distance-based counterparts IFD_W and EFD_W, and that the radius-based measures provide fairly different results. But, to address this issue more systematically, we calculated for each pair of measures a correlation coefficient (we use Pearson's correlation coefficient, which is a standard method to measure the association between two measured quantities that are, at least, interval-scaled). In Table 3, we report the calculated correlation coefficients. All correlations except between EFD_W and *NMR* are significant (p-values not shown). Most of the coefficients are below 0.3, which indicates almost no correlation. Some are above 0.7, which indicates a strong correlation; this includes the correlations between the classic measures and their distance-based counterparts, as well as the correlation between *NAR* and *NMR*. Also, there is a weak correlation (0.3–0.7) between the internal feature dependency (classic and weighted) and a feature's normalized radius (average and maximal).

Size vs. Cohesion. Furthermore, we are interested in whether the size of a feature or of an entire product line correlates with its cohesion. We tested the correlation between the number of lines of code (LOC) of a feature and the individual measures, between the number of features of a product line and the individual measures, as well as between the number of lines of code (LOC) of an entire product line and the individual measures (again, using Pearson's correlation coefficient). In Table 4, we report the calculated correlation coefficients. Many pairs of samples do not correlate (> -0.3 and < 0.3). Interestingly, there are weak but significant correlations between the number of features and some of our measures (*IFD*, IFD_W, *NAR*, *NMR*). That is, the more features a product line has, the higher is their internal feature dependency. Another interesting observation is that there is a weak negative but also significant correlation (< -0.3) between *EFD* (and EFD_W) and the numbers of lines of code of a product line. That is, the larger the code base of a product line is, the weaker is the effect of external references of features. Finally, there is a weak negative but significant correlation between the number of lines of code of a feature and its radius-based cohesion values (*NAR* and *NMR*). That is, the smaller a feature is, the higher is its normalized radius-based cohesion.

[2] `http://www.r-project.org/`

	IFD	IFD_W	*EFD*	EFD_W	*NAR*	*NMR*
# LOC (PL)	0.15	0.16	-0.44	-0.43	0.14	0.07
# features	0.55	0.56	0.07	0.08	0.36	0.36
# LOC (F)	-0.2	-0.2	0.05	0.05	-0.34	-0.3

Table 4: Pearson's correlation coefficients (size vs. measure); PL: product line, F: feature

	IFD	IFD_W	*EFD*	EFD_W	*NAR*	*NMR*
# from scratch	23.6	23.3	58.1	57.7	90.0	83.5
# refactored	45.4	45.1	73.3	75.0	93.0	87.4

Table 5: Influence of the development process on feature cohesion (all differences are significant: $p < 0.006$)

Process vs. Cohesion. Furthermore, we tested whether feature cohesion depends on the development process of a product line (i.e., developed from scratch or by refactoring). Since we have two discrete levels in our data samples (refactored and from scratch), we performed a Mann-Whitney-U test (a statistical significance test for two non-normally distributed samples) to test whether there are significant differences between product lines developed from scratch and by refactoring in terms of the feature cohesion measures. We performed the test for each measure based on the data of all features. We found that there are significant differences between product lines developed from scratch and by refactoring with regard to each individual measure (all p-values are lower than 0.006). We show the results in Table 5. On average, the cohesion is for all four measures greater in product lines developed by refactoring (e.g., *IFD* is 45) than in product lines developed from scratch (e.g., *IFD* is 24).

Product-Line Families. Finally, we are interested in whether there are significant differences between the product lines of the AHEAD family and of the BALI family, respectively. Also we are interested in whether there are significant differences between the different variants of NOTEPAD and of CHATSYSTEM. To this end, we used a Kruskal-Wallis test to determine whether the differences are significant. For the AHEAD and BALI product lines we did not find significant differences; the same applies to the variants of NOTEPAD and CHATSYSTEM, so we omit the numbers.

3.5 Interpretation

Looking at the data and the statistical analysis, we made six notable observations:

(1) For each measure, there are statistically significant differences between the individual sample product lines.

(2) In most cases, the measures do not correlate with other measures. On average, external feature dependency (*EFD* and EFD_W) is higher than internal feature dependency (*IFD* and IFD_W) and lower than the radius-based measures (*NAR* and *NMR*).

(3) The difference between the classic measures *IFD* and *EFD* and their distance-based counterparts is marginal.
(4) Internal feature dependency (*IFD* and IFD_W) correlates with the number of features of a product line; external feature dependency (*EFD* and EFD_W) correlates negatively with the number of lines of code of a product line; the radius-based measures (*NAR* and *NMR*) correlate negatively with the number of lines of code of a feature.
(5) Product lines developed by refactoring have significantly higher internal and external feature dependencies than product lines developed from scratch; the same applies to the radius-based measures.
(6) The product lines of the AHEAD family do not differ significantly; the same applies to BALI. The variants of NOTEPAD do not differ significantly either; the same holds for CHATSYSTEM.

(1) The first observation suggests an important insight: Only by using feature orientation (i.e., by separating features in design and code, or by making them otherwise explicit, e.g., by using annotations), one does not necessarily attain a proper feature cohesion. We found considerable differences between individual features and entire product lines (effectively, covering the entire spectrum of possible values of our measures). For example, the internal feature dependency is 0.61 for BerkeleyDB and 0.04 for Prevayler; the external feature dependency is 0.78 for BerkeleyDB and 0.86 for Prevayler. That is, BerkeleyDB has highly cohesive and loosely coupled features, whereas Prevayler has loosely coupled features that are not very cohesive.

(2) The second observation tells us about the kind of cohesion that features exhibit. Many product lines have loosely coupled features in the sense that they depend mostly on their own elements, rather than on elements of other features (*EFD* and EFD_W). This suggests that there is a potential for interfaces and information hiding to encapsulate a feature's elements (see Sec. 3.7). But our data indicate that, internally, features are less cohesive in that a feature's elements depend only on few other elements of the same feature (*IFD* and IFD_W). This suggests that there is room for refactoring features into smaller pieces (see Sec. 3.7). Finally, compared to the other measures, the normalized radii of features are more homogeneous, but still reveal considerable differences between individual features and product lines. That is, the elements of some features are much more scattered across the clustering layout than others. The fact that the normalized radii do not necessarily correlate with the other measures illustrates that radius-based measures provide an alternative view on feature cohesion. Feature GZIP of ZIPME is a good example. It consists of 18 elements. Its internal feature dependency (classic and distance-based) is low because most of its elements refer only to a few of its other elements. Its external feature dependency is medium because there are roughly as many external dependencies as internal dependencies. However, the distances between the feature's elements are very low in the layout (i.e., *NAR* and *NMR* are very high), which indicates that the inner attraction is by far higher than the attraction from the outside. The three pieces of information draw a holistic picture that would be less informative without distance information.

(3) The third observation is that the distance-based measures draw a similar picture as their classic counterparts (they correlate strongly). But in certain cases, they provide more information than the classic measures. Both distance-based measures penalize long internal dependencies and reward long external dependencies. It seems that this is not frequently effective (i.e., classic measure and distance-based measure have similar values), but still we found several situations in which there is a considerable difference between classic and distance-based measures. In these situations, features are rated less cohesive compared to the traditional measures because their elements are pulled apart because of attractions to other features. Feature Rot13 of CHATSYSTEM/REHN is an extreme example: $EFD(\text{Rot13}) = 100$ and $EFD_W(\text{Rot13}) = 0$. We found the reason for the difference when examining the layout of the product line: The feature consists of three elements that form two clusters at different regions of the layout. *EFD* is maximal because the elements refer only to themselves. EFD_W is minimal because the elements do not form a connected graph and thus are at different positions in the layout. Although this is an extreme example (recall, on average, there is no significant difference between *EFD* and EFD_W), it shows that the weighted measures produce indeed different results in special cases that may point to exceptional situations and code smells. For feature Rot13, the developers may consider to split it into two separate features.

(4) The forth observation is that there are correlations between feature cohesion, and feature and system size. We observed a negative correlation between radius-based cohesion values and a feature's size. That is, the smaller a feature is in the layout, the more cohesive it is. This result is intuitive because large features often share many references with other features. Furthermore, we observed a positive correlation between internal feature dependency and a feature's normalized radii with the number of features of a product line, and we observed a negative correlation between external feature dependency and the number of lines of code of a product line. The question is why do they correlate? We believe that the larger the product line, the larger is the number of dependencies between program elements, and, consequently, the lower is the external feature dependency. But, we cannot explain why the internal feature dependency increases with the number of features.

(5) The fifth and probably most surprising observation is that the features of product lines developed by refactoring have significantly higher cohesion values (for all measures) than the features of product lines developed from scratch. This is surprising because one would expect that it should be easier for programmers to develop cohesive features when planning and designing a product line from scratch, rather than when struggling with a given legacy design. This issue needs more investigation in the future.

(6) The sixth observation is that, although the different variants of NOTEPAD and CHATSYSTEM have been developed independently, the individual variants of each system do not differ significantly. This is easy to understand because NOTEPAD has been developed by refactoring a common code base and the independent developments of the CHATSYSTEM variants started from the same core. The similarities between the product lines of the AHEAD family is not surprising either, since they share certain features; the same applies to the BALI family. These observations are interesting, as they suggest that product lines of a single narrow domain (possibly sharing features) have similarly cohesive features.

3.6 Threats to Validity

Internal Validity. To minimize threats to internal validity, we controlled a number of confounding variables. In particular, we selected only Java-based product lines to rule out effects due to different languages, and we selected a large sample size to minimize influences of confounding variables such as programming experience and domain. In general, a key idea of our study is to assess feature cohesion in different ways to provide a holistic picture. This way, we circumvent the problem that a certain measure is influenced by an unknown confounding variable. Additionally, we used established statistical methods to assess significance and correlations.

External Validity. A common issue is to what extent the external validity of our study relies on the selection of samples. Can we generalize to other product lines, application domains, and languages? The fact that we used only product lines written in Java increases internal validity at the cost of external validity. As in every empirical study, there is a trade-off between the two. For our study, we decided to maximize internal validity to control confounding variables and to live with the fact that we cannot generalize to all kinds of languages. However, to still increase external validity to an acceptable level, we collected as many feature-oriented product lines as we were able to locate and to process with our tools (e.g., to infer the dependency graphs and feature mappings), deliberately excluding too small and artificial examples. [3] Although a larger sample size would increase external validity, we argue that the selected programs represent the state-of-the-art in feature-oriented product-line engineering (with Java) because they are of substantial size, of different domains, and have been developed mostly by others and for different purposes.

3.7 Perspectives

A key insight of our exploratory study is that, concerning different cohesion indicators, there are significant differences between individual features of a product line and across individual product lines. The point is that all sample product lines have been developed with feature-oriented techniques. One goal of feature orientation is to make features explicit in design and code, and to encapsulate code belonging to a feature into a single, addressable unit [1, 5, 22]. So, one would expect that features are implemented mostly cohesively, but this seems not to be generally the case, as our data suggest.

A further issue is that popular feature-oriented approaches such as AHEAD do not provide proper encapsulation mechanisms for features. That is, a programmer can separate feature code but there is no feature-specific interface mechanism to hide internal details of a feature. A feature-specific interface mechanism is clearly desirable because this would facilitate modular type and model checking as well as separate compilation of features [16]. Our data suggest that by far not all features and all product lines in this field are ready for such mechanisms. Or it may be the other way around: proper information hiding and interface mechanisms are missing in feature-oriented approaches such that the result is a suboptimal feature cohesion.

Another issue is that there is no established catalog of guidelines or patterns of how to program software product lines in a feature-oriented way. So, for example, inspecting manually the layouts of individual product lines, we found features that are "providers" for other features and features that are "customers"; we found features that are used by almost all other features and features that are used only by one or two other features. So, the study suggests that there are different styles of feature-oriented programming. Features can be large subsystem-like structures (e.g., a parser in MIXIN), fine-grained extensions (e.g., adding a button in VIOLET), or everything in between. Furthermore, features can be heavily crosscutting in nature (e.g., latches in BERKELEYDB), almost self-contained units (e.g., the SAT solver in GUIDSL), or everything in between. The question is in which situation is which style favorable? We believe that tools like FEATUREVISU and analyses like the one we performed, can help to answer this question in the future.

Finally, we believe that layout-based clustering approaches can help to understand large software systems such as product lines. They can point to design and code smells such as indicating a low feature cohesion and help programmers to understand why is that the case (e.g., too large, non-cohesive features should be divided by refactoring). We envision tools that visualize changes and refactorings of a software system in its layout online, to provide immediate feedback to the programmer [8].

[3] We included EPL because it implements the expression problem — a widely-used benchmark for modularity and composition techniques [28].

4. RELATED WORK

Cohesion Measures. The notion of feature cohesion in general and the definition of the measures *IFD* and *EFD* in particular are based on previous work discussing and measuring cohesion in software systems [3, 11, 22, 26, 30].

An early definition of cohesion can be found in the work of Stevens et al. on structured design [26]. This work defines cohesion informally as the degree to which relationships among program elements tend to be inside individual modules rather than across module boundaries. Our informal definition is inspired by this work.

The notion of cohesion has been used in object-oriented design to assess the system structure; Briand et al. provide a comprehensive overview of the subtle differences in the definitions of different cohesion measures for object-oriented systems [11]. All measures are centered around key object-oriented abstraction mechanisms such as methods, fields, classes, and inheritance. Most measures concentrate on the cohesion of single classes, which resembles our measure *IFD*. Measures that consider also inter-class relations such as external method invocations resemble our measure *EFD*. In any case, our model and measures are more abstract (and thus language- and paradigm-independent) in that they are defined over dependency graphs.

Zhao and Xu apply previous work on object-oriented cohesion measures to aspect-oriented programming [30]. Specifically, they define two kinds of cohesion measures, one considering the cohesion of elements within one module and one considering the cohesion across multiple modules. The former is an instance of our measure *IFD* and the latter is an instance of our measure *EFD*, both applied to aspect-oriented programming. Zhao and Xu proved that the two measures satisfy all properties necessary to define a measure. Furthermore, numerous empirical studies (e.g., [14]) rely on these two measures, so we are confident that our measures and our results are sound.

Finally, there is some work that stresses the importance of feature cohesion in software product lines [3, 22]. Most notably, Lopez-Herrejon et al. compare different programming languages with respect to their ability to implement cohesive feature modules [22]. However, their notion of feature cohesion is defined only informally. Hence our measures and our experiments advance the field.

Product-Line Measures. There is a large body of work aiming at assessing and predicting the quality of product lines, especially of product-line architectures. There are too many pieces of work to name them all, so we refer the reader to representative examples. Specifically, we would like to point to the work of van der Hoek et al. [29], who propose a number of measures to assess the variability of a product-line architecture, to the work of Her et al. [15], who developed a framework for evaluating the reusability of a product line's core assets, and to the work of Cheng et al. [12], who propose several measures based on architectural drivers. Feature cohesion at the level of code has not been considered, nor has layout-based clustering been used as a foundation.

5. CONCLUSION

To assess the characteristics of feature cohesion in software product lines, we conducted an exploratory study on forty software

product lines of different sizes and domains. We defined and used different cohesion measures including classic measures and measures that are based on layout-based clustering. The idea of using multiple measures that are based on different foundations is to draw a holistic picture of feature cohesion. In our study, we made several interesting observations such as that individual features and individual product lines differ significantly in their cohesion and that there are correlations between feature and system size and feature cohesion as well as between development process and feature cohesion. These observations open up interesting perspectives. For example, the role of interfaces in feature orientation is directly related to feature cohesion but did not receive much attention in the past. Another example is that there are different styles of implementing features that lead to different cohesion degrees. Currently, there is no agreement on a common catalog of programming guidelines or design patterns for feature-oriented development, which seems to be important when looking at the data. Finally, as we made good experience with using FEATUREVISU, it is interesting to systematically explore how layout-based clustering can aid product-line engineering, for example, in terms of comprehension and maintenance.

Acknowledgments

We thank J. Feigenspan and C. Kästner for fruitful discussions on earlier versions of this paper. This research was supported in part by the German DFG grants AP 206/2-1 and AP 206/4-1, and by the Canadian NSERC grant RGPIN 341819-07.

6. REFERENCES

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *J. Object Technology (JOT)*, 8(5):49–84, 2009.

[2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. ICSE*, pages 221–231. IEEE, 2009.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

[4] D. Batory, J. Liu, and J. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *Proc. FSE*, pages 48–57. ACM, 2003.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[6] D. Beyer. Co-Change Visualization Applied to PostgreSQL and ArgoUML. In *Proc. MSR*, pages 165–166. ACM, 2006.

[7] D. Beyer. CCVISU: Automatic Visual Software Decomposition. In *Proc. ICSE*, pages 967–968. ACM, 2008.

[8] D. Beyer and A. Fararooy. CHECKDEP: A Tool for Tracking Software Dependencies. In *Proc. ICPC*, pages 42–43. IEEE, 2010.

[9] D. Beyer and A. Fararooy. A Simple and Effective Measure for Complex Low-Level Dependencies. In *Proc. ICPC*, pages 80–83. IEEE, 2010.

[10] D. Beyer and A. Noack. Clustering Software Artifacts Based on Frequent Common Changes. In *Proc. IWPC*, pages 259–268. IEEE, 2005.

[11] L. Briand, J. Daly, and J Wüst. A Unified Framework for Cohesion Measurement. In *Proc. METRICS*, pages 43–53. IEEE, 1997.

[12] S. Chang, H. La, and S. Kim. Key Issues and Metrics for Evaluating Product Line Architectures. In *Proc. SEKE*, pages 212–219, 2006.

[13] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[14] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. ICSE*, pages 261–270. ACM, 2008.

[15] J. Her, J. Kim, S. Oh, S. Rhew, and S. Kim. A Framework for Evaluating Reusability of Core Asset in Product Line Engineering. *Information and Software Technology*, 49(7):740–760, 2007.

[16] D. Hutchins. *Pure Subtype Systems: A Type Theory for Extensible Software*. PhD thesis, School of Informatics, University of Edinburgh, 2009.

[17] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) — Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, 1990.

[18] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. ICSE*, pages 311–320. ACM, 2008.

[19] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. GPCE*, pages 157–166. ACM, 2009.

[20] C. Kästner, S. Apel, S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proc. SPLC*, pages 181–190. SEI, 2009.

[21] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. ICSE*, pages 105–114. ACM, 2010.

[22] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. ECOOP*, *LNCS 3586*, pages 169–194. Springer, 2005.

[23] A. Noack. Energy Models for Graph Clustering. *J. Graph Algorithms Appl.*, 11(2):453–480, 2007.

[24] D. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058, 1972.

[25] M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data & Knowledge Engineering*, 68(12):1493–1512, 2009.

[26] W. Stevens, G. Myers, and L. Constantine. Structured Design. *IBM Systems J.*, 13(2):115–139, 1974.

[27] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. ICSE*, pages 107–119. IEEE, 1999.

[28] M. Torgersen. The Expression Problem Revisited. In *Proc. ECOOP*, *LNCS 3086*, pages 123–143. Springer, 2004.

[29] A. van der Hoek, E. Dincel, and N. Medvidovic. Using Service Utilization Metrics to Assess the Structure of Product Line Architectures. In *Proc. METRICS*, pages 298–308. IEEE, 2003.

[30] J. Zhao and B. Xu. Measuring Aspect Cohesion. In *Proc. FASE*, *LNCS 2984*, pages 54–68. Springer, 2004.