

A UNIFIED APPROACH TO FEATURE-CENTRIC ANALYSIS OF OBJECT-ORIENTED SOFTWARE

Andrzej Olszak and Bo Nørregaard Jørgensen
The Maersk Mc-Kinney Moller Institute
University of Southern Denmark
Odense, Denmark
{ao, bnj}@mmmi.sdu.dk

ABSTRACT

Feature-centric comprehension of software is a prerequisite to incorporating modifications requested by users during software evolution and maintenance. However, feature-centric understanding of large object-oriented programs is difficult to achieve due to size, complexity and implicit character of mappings between features and source code. In this paper, we address these issues through our unified approach to feature-centric analysis of object-oriented software. Our approach supports discovery of feature-code traceability links and their analysis from three perspectives and at three levels of abstraction. We further improve scalability of analysis by partitioning features into canonical groups. To demonstrate feasibility of our approach, we use our NetBeans-integrated tool Featureous for conducting a case study of feature-centric analysis of the JHotDraw project. Lastly, we discuss how Featureous supports program comprehension by means of concrete cognitive design elements.

KEY WORDS

Software Evaluation, Visualization, Software Maintenance, Feature-Centric Analysis.

1. INTRODUCTION

There exists a difference between a user's and a programmer's viewpoints on software. While a user perceives programs through their observable features, a programmer is primarily concerned with the underlying implementation. In other words, software users operate in a program's problem domain, whereas software programmers operate in a program's solution domain [1]. This duality is desired, as it allows parties to focus on their respective aspects of interest in a program. However, the sharp distinction between users, as operating in the problem domain, and programmers, as operating in the solution domain, does not always hold.

When modifying software, developers need to relate the task descriptions (e.g. functional requirements, change requests, error reports) that users formulate in terms of features in the program's problem domain [1] to their respective realizations in the program's solution domain (i.e. source code). That is, developers need to establish mental mappings between a program's observable features and their implementations in source code [1]. Understanding such mappings is a key factor during software evolution and

maintenance [2][3], since it is a prerequisite to feature-centric program modification [2], error correction [4][5] and derivation of new features from the existing ones.

In case of object-oriented programs, relating features to their implementations is, however, a difficult task, as object-oriented programming languages provide no means for expressing features explicitly. In object-oriented programs, features are implemented implicitly as inter-class collaborations crosscutting not only multiple classes but also multiple architectural units [6], e.g. layers in layered architectures [7]. This physical tangling and scattering of features over several source code units makes their implementations difficult to identify and understand [1][8].

In face of a lack of one-to-one correspondence between features and structure of object-oriented programs, feature-centric analysis [9] is needed in order to support feature-centric understanding. It is important that feature-centric analysis provides means for coping with the inherent complexity of feature-code mappings, so that it scales with respect to the number of a program's features and the size of a program's codebase. Finally, such a feature-centric analysis approach requires appropriate tool support to guide comprehension in a systematic fashion, automate repetitive calculations and ensure reproducibility of the process.

In this paper, we aim at providing scalable feature-centric analysis of object-oriented programs. We do this by introducing our feature-centric analysis tool called Featureous. The analytical framework behind Featureous provides three complementary perspectives on feature-code traceability links, which allows us to decouple investigations of the complex many-to-many feature-code relations into investigations of several one-to-many mappings. In order to control the amount of information presented to an analyst, we allow the perspectives to be examined at three levels of analytical abstraction. The scalability with respect to number of features is achieved through partitioning features based on canonical features, whereas scalability with respect to codebase size is supported by adjusting levels of computational units' granularity, i.e., package vs. class scope. Featureous, implemented as a plug-in for the NetBeans IDE [10], can be obtained from our web site [11], thereby allowing for replication of the analytical procedures described in this paper.

In order to demonstrate usage of our approach we present a case study of applying feature-centric analysis to the JHotDraw project [12]. Apart from demonstrating applica-

bility of our approach, we use this case study for discussing the details of our approach. Lastly, we evaluate the support of our approach for program comprehension with respect to the cognitive design elements defined in [13].

The remainder of this paper is organized as follows. In Section 2, we present the state of the art on which we base our approach. In Section 3, we give a high-level overview of our approach to feature-centric analysis. In Section 4, we discuss the elements of our approach through their application in the JHotDraw case study. In Section 5, we evaluate the support for program comprehension in our approach. Finally, Section 6 summarizes and concludes the paper.

2. STATE OF THE ART

Feature-centric analysis supports understanding of object-oriented software by considering features as first-class analysis entities [9]. One of the basic elements of feature-centric analysis is the bi-directional traceability links between features and object-oriented source code. As these links are often not easily deductible from legacy source code, they need to be discovered. This is done through feature location [14], which is a concrete example of the problem of concept assignment [15]. Feature location can be based on a diverse methods and data sources, ranging from static program analysis [16] to dynamic analysis and ranking heuristics [17]. The tradeoffs between different feature location approaches are: the set of required artifacts, level of automation, accuracy, reproducibility and availability of run-time-specific information (i.e. resolution of polymorphic invocations, branch conditions and object co-usage).

Making the feature-code traceability links explicit through visualization is known to improve the programmers' ability to discover classes implementing a given feature, as well as features being implemented by a given class [5][18][19]. Yet, the wide body of evidence from tools not related to features suggests that enriching traceability visualizations with more sophisticated analysis mechanisms brings additional comprehension improvements [20][21][4].

By analyzing the established traceability links, it is possible to characterize features in terms of classes and characterize classes in terms of program features [22]. These characterizations can be used to investigate inter-feature relations in terms of implementation overlap [22]. Furthermore, the static characterization based on classes can be complemented by dynamic views based on usage of concrete objects by executing features. This allows for examining run-time inter-feature dependencies.

The information contained in feature-code traceability links can be summarized by the usage of appropriate software metrics. The approaches described in [23][24] have recognized applicability of the metrics traditionally associated with the separation of concerns to analyzing features. The two metrics proposed in [23] - scattering and tangling - assess quantitatively the complexity of the relationships between features and computational units.

As programming languages allow for representing programs at multiple units of granularity of source code (e.g.

methods, classes, packages), features can also be represented at a higher-level granularity by using canonical features [25]. The canonical-features approach reduces the number of features under analysis by finding features whose implementations are representative to a number of other features. This technique was shown to improve comprehension of feature-rich object-oriented software [26].

Summing up, the existing approaches define a set of diverse methods for feature-centric analysis. Yet there is no common point of view on how they can be integrated to facilitate scalable feature-centric comprehension.

3. UNIFIED APPROACH TO FEATURE-CENTRIC ANALYSIS

Feature-centric analysis operates on the traceability links between features and object-oriented source code. For establishing this traceability, our approach relies on the semi-automatic feature location mechanism defined in [27]. This mechanism uses dynamic analysis to discover the mappings between features and their corresponding classes, objects and methods. After a list of a program's features is recovered from its documentation, or its graphical user interface, each feature needs to have its feature entry points manually annotated in the source code. Feature entry points are the methods through which a program's control flow enters implementations of a feature. Annotated program is then transparently instrumented with a tracing aspect in order to collect the methods, classes and objects used by features executed at a program's run-time. After features are triggered in the user interface of the instrumented program, the information is then saved in form of feature-trace files, which serve as an input to further analysis.

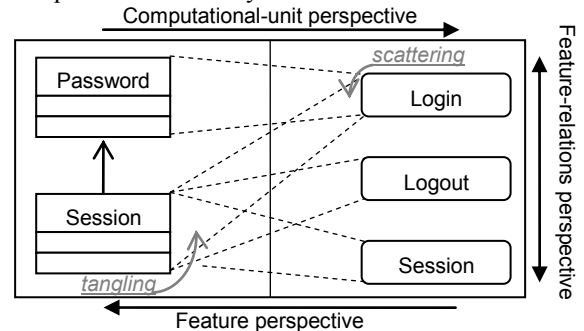


Figure 1. Perspectives on feature-code traceability links (based on [22]).

The relations between features and computational units are in general of the type many-to-many. An example of such relations is depicted in Figure 1, where a class exists that contributes to multiple features as well as a feature that is being implemented by multiple classes. The complexity of these mappings is caused by coexistence of the interleaving problem [28], which is caused by tangling of feature implementations in terms of computational units, and the delocalized plans [29], which are caused by scattering of feature implementations among multiple computational units. Therefore, we propose to reduce comprehension effort

of many-to-many mapping between features and code by decomposing it into two one-to-many mappings. We do that by adopting the three complementary analytical perspectives on feature-code relations first defined in [22][9].

The perspectives shown in Figure 1 are defined as:

- *Computational-unit perspective* shows how computational units, like packages and classes, participate in implementing features [22].
- *Feature perspective* focuses on how features are implemented. That is, it describes features in terms of their usage of a program’s computational units [22].
- *Feature-relations perspective* focuses on inter-feature relations deduced from the feature-code mapping [9].

However, after decomposing the many-to-many feature-code correspondence, there still remains the issue of a mapping’s size dependence on a number of features and computational units. In order to improve this situation, we propose to partition features around their respective *canonical features* [25]. A canonical feature is a feature that is representative to implementations of a number of other features. Therefore, usage of canonical features reduces the number of features under investigation with only minimal loss of information. Thus, it reduces the comprehension effort, as compared to traditional approaches [26].

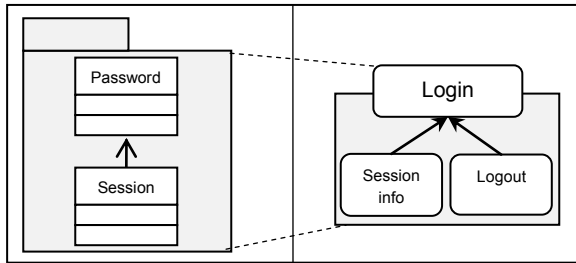


Figure 2. Adjustable granularity of traceability links.

To incorporate this method, our approach provides support for computing canonical features and grouping the remaining features around them into feature partitions [25]. As depicted in Figure 2, the partitions can then be treated as first-class analysis entities. It is possible to investigate a feature partition as a single analysis unit, as well as to analyze its internal features. As this mechanism can be applied recursively, it improves the scalability of our feature-centric analysis. In addition, we support adjusting the granularity of computational units to classes, alternatively or packages.

The separation of analytical concerns in terms of three analytical perspectives viewed at adjustable levels of granularity is complemented by a support for comprehension at multiple levels of *abstraction* within each of the three perspectives. The purpose of providing stratified levels of abstraction is to focus the analysis process by limiting the amount of information presented simultaneously. This allows for investigating the complexity of a program’s features in an incremental fashion at three abstraction levels:

- *Characterization level* shows high-level diagrams, which summarize the overall program complexity in the context of each analytical perspective.

- *Correlation level* refines the high-level characterizations into detailed correlations between features, computational units and objects.
- *Traceability level* provides navigable bi-directional traceability links between features and source code.

The division into three perspectives, three levels of abstraction and adjustable granularities constitutes the high-level structure of our unified approach to feature-centric analysis. Figure 3 presents the three analytical perspectives as vertical divisions and the three levels of abstraction as horizontal layers. The third axis represents granularity of feature partitions and computational units. Organizing our approach into the presented structure not only separates analytical concerns, but also helps guiding the comprehension process in a systematic fashion [30].

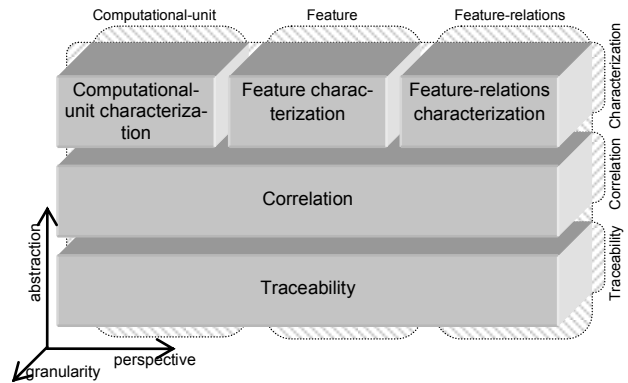


Figure 3. Proposed separation of analytical concerns.

Together, the three analytical perspectives and the three levels of abstractions contain five analysis elements. These elements provide the concrete means for performing feature-centric investigations of object-oriented software in terms of a given perspective and at a given level of abstraction within the analysis process. An overview of how this conceptual framework is realized in our tool Featureous is shown in Figure 4. Following, we describe each of these elements briefly, whereas their detailed descriptions are postponed to the running example in the next section.

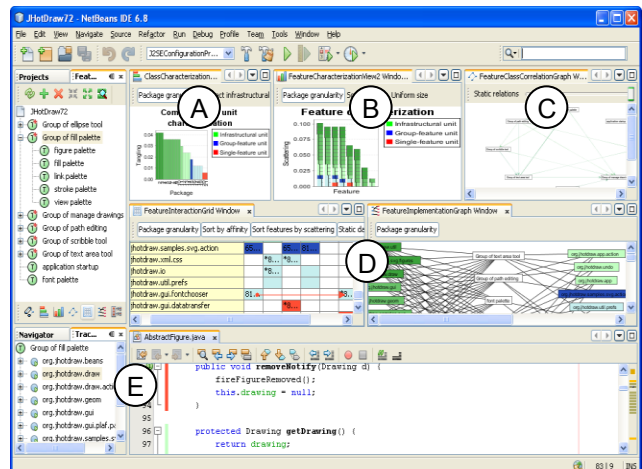


Figure 4. Feature-centric views in Featureous.

Computational-unit characterization (marked as *A* in Figure 4) gives a high-level summary of a program’s classes with respect to their participation in feature implementations. This element combines the affinity-colored class characterization view computed from a number of feature metrics [22] with a measure for tangling of features in terms of computational units [23].

Feature characterization (B) allows for assessment of the distribution of feature implementations over affinity-colored computational units. This element integrates the feature-characterization view [22] with a measure for scattering of feature implementations among a program’s computational units [23].

Feature-relations characterization (C) provides high-level information about dynamic and static dependencies between feature implementations. This element consists of two complementary views: the feature-interaction graph [31] for assessing dynamic (instance-based) inter-features relations and the feature-implementation-overlap graph that we propose to be the static counterpart of the feature-interaction graph. The relations in both graphs are depicted using an affinity-coloring scheme.

Correlation (D) element allows for detailed investigation of the correspondence between affinity-colored computational units, objects and features. This element offers both a matrix-based and a graph-based visualization. A correlation matrix is formed as a combination of the feature-class correlation matrix [22] and feature-interaction grid [31], allowing for a direct comparison between static and dynamic inter-feature relations. This representation is complemented by the affinity-colored feature-implementation graph [22]. The correlation element is common to all three analytical perspectives in our approach.

Traceability (E) element provides navigable bi-directional traceability links between features and concrete fragments of a program’s source code. This allows for fine-grained reasoning about how code implements features, how features use the code and how features relate to each other in terms of concrete code statements. The first mechanism employed by this element is the feature inspector that provides a navigable traceability from features to source code. The second mechanism is the coloring of source code in the NetBeans’ editor that characterizes individual fragments of source code by affinity coloring and the names of their corresponding features. The traceability element is common to all three analytical perspectives.

4. CONDUCTING FEATURE-CENTRIC ANALYSIS

In this section, we use a case study to explain our approach in detail. We do that by presenting an example of applying our approach in a scenario of top-down comprehension. The case study being discussed is a feature-centric analysis of a program built on top of the JHotDraw 7.2 framework called SVG [12]. The program consists of 62K lines of code and contains a significantly high number of features for the case study to be considered a realistic application. It is worth mentioning that prior to conducting this case study we had

no significant exposure to the design and implementation details of SVG.

As a prerequisite to conducting feature-centric analysis of SVG, we have located implementations of its features in the source code. Since appropriate documentation of program functionality was not available, we have had to discover the features. We have inspected elements of SVG’s graphical user interface (in particular: the main menu, contextual menus and toolbars) in order to enumerate available user-triggerable behaviors. For all the 29 features discovered, we have marked SVG’s source code with 91 feature-entry-point annotations in order to apply the feature-location approach defined in [27]. We have found it necessary to annotate more than one feature entry point per feature due to multiple possible ways in which a feature can be triggered (e.g. the ‘basic editing feature’ can be triggered both by invoking ‘copy’ and ‘paste’ commands in SVG’s main menu). Finally, we have produced feature traces of the discovered features, by manually triggering them at run-time in SVG program instrumented with the feature-tracing aspect [27].

In the following analysis, we start out by investigating SVG on the highest level of abstraction and incrementally explore more details as the analysis progresses to the lower levels. In order to keep the scope of the discussion manageable, we analyze only single examples of canonical features, features, inter-feature relations and classes. This enables us to present our approach without going too much into the details of the internal workings of SVG.

4.1 Computational-Unit Characterization

We begin the feature-centric analysis of SVG by investigating a high-level summary of how classes participate in implementations of features. This is done through visualizing all of the program’s classes, or alternatively its packages, and characterizing their contribution to program features. This contribution is indicated qualitatively by coloring of the affinity categories [22] as well as quantitatively by a measure of the tangling [23] of features in terms of computational units. Depending on the level of participation in implementing features, the three affinity categories determine whether a computational unit is an infrastructural unit (used by more than 50% of features), a group-feature unit (used by more than one, but less than 50% of features), or a single-feature unit [22]. The affinities are represented by their respective colors: green, blue and red [22]. We enhance the original representation of affinities by using the shades of affinity colors to show how strongly a computational unit belongs in an affinity category. The darker the shade of its affinity color, the more features a computational unit belongs to. Together with the shape of the tangling measure’s distribution, the affinity coloring indicates the level of reuse of computational units across features. The names of individual computational units can be displayed in the plot as tooltips by placing the mouse pointer over the computational unit. This view serves as a coarse-grained complexity indicator for understanding how a program’s computational units are used by features.

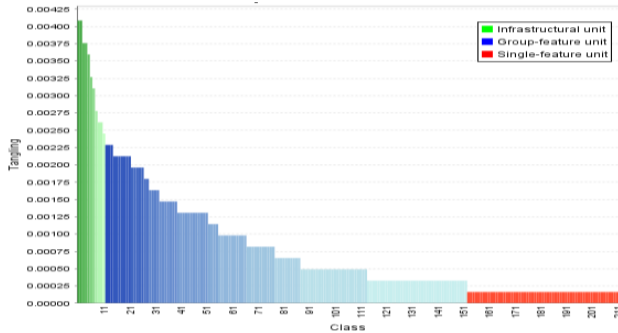


Figure 5. Class characterization of SVG.

The characterization of SVG’s classes, shown in Figure 5, indicates that majority of the classes of SVG participate in implementations of more than one feature. The high level of tangling of features in terms of classes indicates that a it would be difficult to modify features of SVG in isolation and a substantial effort would be needed to ensure that modifications to classes do not affect the correctness unintended features. The high degree of implementations sharing between features, at this point in the analysis, indicates that the architecture of SVG was not designed with separation of feature implementations in mind. Furthermore, it hints that some of the features of SVG are closely related and possibly implemented as variants of each other.

4.2 Feature Characterization

The feature-characterization element is based on feature characterization view [22], which is a bar chart summarizing implementations of features in terms of their contributing computational units. We represent each feature by a separate bar, whose height indicates the scattering [23] of a feature over computational units. The coloring of bars shows the distribution of the computational units within the affinity-based categories [22]. The names of the computational units, i.e., classes or packages, can be shown as tooltips. The coloring scheme of the affinity-based categories is the same as the one used in the computational-unit characterization element. In addition, this information is shown within bars also as a distribution profile plot. This fine-grained information on the characterization of contributing computational units gives an impression on how difficult it would be to change the implementation of a given feature without affecting the rest of a program’s functionality. I.e. changing a red-colored unit will only affect the feature itself, whereas changing green or blue units will affect other features as well. Lastly, the feature bars are sorted by the values of scattering in order to focus the analyst’s attention on the features whose relation to code is most intricate.

The concrete results of feature characterization obtained for SVG are shown in Figure 6. It can be seen that there exists a high degree of code sharing among the features of SVG. The features contain relatively small amounts of feature-specific classes, and most of the code is shared in terms of group-feature classes, rather than the infrastructural classes. Interestingly, there exist features that contain no

feature-specific classes. As we shall see in the next subsection, this observation supports our earlier hypothesis about the existence of a number of features that are possibly implemented as variants of each other.

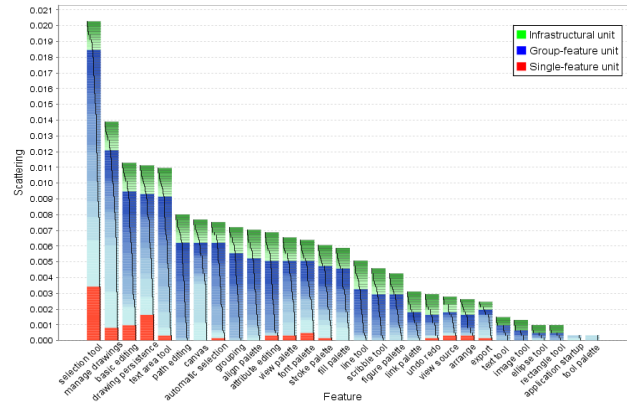


Figure 6. Feature characterization of SVG in terms of classes.

4.3 Partitioning Features

In order to group highly-related features together, our approach supports computation of canonical features [25] and their usage as centroids for grouping features into partitions. Partitioning based on canonical features not only reduces the amount of information simultaneously presented to an analyst [26], but also allows for localized reasoning about inter-feature relations in individual feature partitions.

Our approach supports automatic creation of feature partitions in terms of canonical features, while allowing for arbitrary manual modification of the resulting partitions into alternative hierarchies. For computing canonical features we have made the following implementation decisions. The similarity matrix [25], which is necessary for calculating canonical features, is constructed using the Jaccard’s measure of inter-method call-edge similarity [25], since the compact nature of gathered feature traces does not allow for using the graph-matching technique described in [25]. Based on the similarities, we find the canonical-feature set by using the criteria of minimal similarity among features in the set and maximal similarity between the features in the set and the features outside the set [25]. This is performed by using a genetic algorithm [32] (we use binary encoding of chromosomes for representing status of canonical-feature-set membership of features) to optimize the set according to the mentioned criteria. Lastly, we iteratively assign features to feature partitions based on their Jaccard similarity to the partitions’ canonical centroids.

Figure 7 shows how the 29 features of SVG have been arranged around canonical features to establish 8 feature partitions. The relatively high degree of grouping of the SVG’s features supports our observations of the high number of highly-related features that are possibly variants of each other.

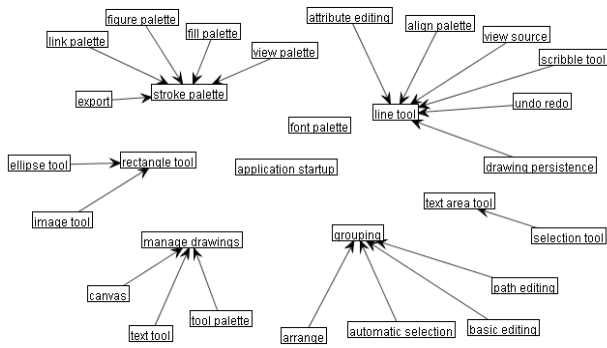


Figure 7. Partitioning features based on canonical centroids.

Interestingly, a vast majority of the features belonging to each of the eight feature partitions are not only related by overlap in implementation, which is the reason for their co-membership in feature partitions, but also related semantically. The property of semantic coherence of feature partitions is important, as it allows treating feature partitions themselves as features. Such higher-order features can then be used as a useful abstraction mechanism, because they make it possible both to understand the meaning of feature partitions in terms of a program’s domain (semantic coherence property) and reason about feature partitions in isolation (low inter-partition overlap property).

4.4 Revisiting the Characterization Views

The initial assessment of the feature characterization and the computational-unit characterization allowed us to recognize a need for partitioning features. Since feature partitions became the new first-class analysis entities, we need to revisit the mentioned characterization elements.

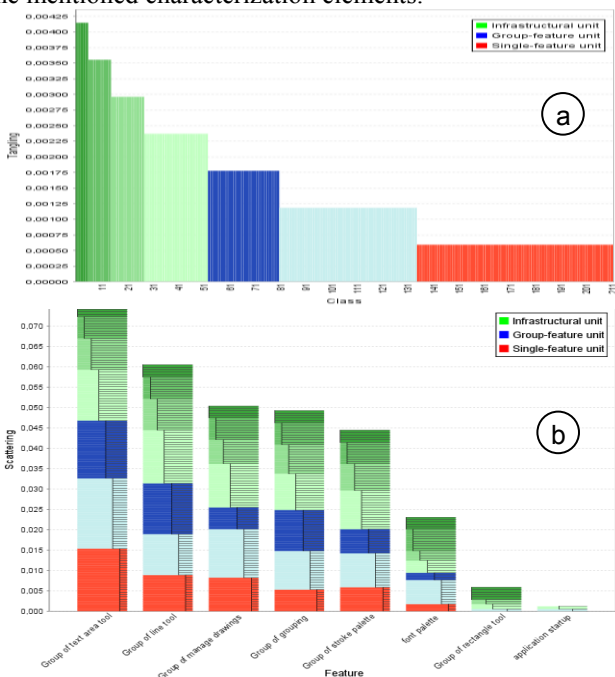


Figure 8. Class (a) and feature (b) characterizations after partitioning features into canonical features.

A comparison between Figure 8 and Figure 6 reveals that the introduction of feature partitions helped to reclassify most of the group-feature classes as infrastructural or single-feature classes. This transition indicates that functionality in SVG is implemented around a core that forms a common base for implementing features [1]. Equally important, the decrease of the information amount and the simplification of relations between features improve focus and reduce effort of further analysis.

At this stage, we use the two characterization diagrams to define the focus of further investigations. We choose the ‘Group of text area tool’ feature partition (the most scattered feature partition) and the *org.jhotdraw.draw.AbstractFigure* class (the most tangled class).

4.5 Feature-Relations Characterization

Summarizing the inter-feature relations in feature-relations element is done through two views, addressing both the dynamic and the static aspects of these relations.

The first view – feature-interaction graph [31] - characterizes how features depend on each other with respect to instantiation of classes and how they interact at run-time by sharing objects. Features, denoted as vertices, are connected by dashed edges in case of common usage of objects at run-time. A directed, solid edge is drawn if a pair of features is in a producer-consumer dependency relation, meaning that objects that are instantiated in one of them are used in another. The edge is directed according to the UML conventions, i.e. towards the producer.

The second view characterizes the static inter-feature relations. In this, we summarize the sharing of classes between feature implementations. Relations are depicted as undirected edges between feature vertices. This view complements the feature interaction graph for cases where static relations do not result in dynamic relations between features. One instance of such a case is static methods (since static methods are not executed on the class instances, but on classes themselves); another is two features that use disjoint sets of instances of a class.

For both views, we propose that the thickness of an edge reflects the number of shared instances/classes – the more instances/classes shared, the thicker the edges. Furthermore, we define the color of an edge as the mean value of the affinity colors of the classes whose objects contribute to the edge’s relation. The amount of data shown in a graph can be limited by selecting a subset of features and again narrowed further by setting an object/class-count threshold for hiding weak relations.

The two graphs in Figure 9 show the dynamic and static relations between features contained in the ‘Group of text area tool’ partition. At the same time, these diagrams present an important feature of the Featureous tool – the ability to operate on contents of the established partitions. Although, we present this in the context of the feature-relation diagrams, an adjustable feature-partition focus is available for all analysis elements of our approach. On-demand switching

between multiple levels of features' granularity of chosen feature partitions is aimed at enabling partial comprehension, which is known to be important when working with legacy code [33]. Finally, to denote the relations between a partition's internals and externals, all the features not contained by a partition are summarized as one artificial '[externals]' feature.

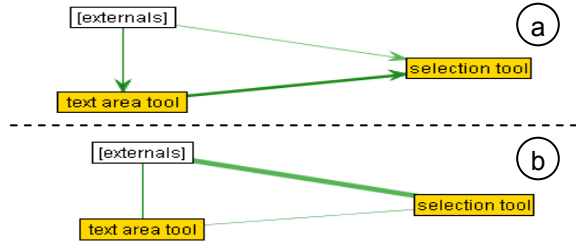


Figure 9. Characterization of (a) dynamic and (b) static inter-feature relations in the ‘Group of text area tool’.

The particular inter-feature relations in SVG, shown in Figure 9, reveal that there exists external dependence on instances created by both the ‘text area tool’ and the ‘selection tool’ features. Especially, the ‘selection tool’ feature seems to be a producer of a significant number of objects that are then used by both the ‘text area tool’ and other features of SVG. Static relations inside the focused partition reveal an interesting situation – the ‘selection tool’ feature is more related to the rest of SVG’s features than to its canonical feature. We hypothesize that this is due to the ‘selection tool’ feature’s importance in implementing a number of other features in SVG, since almost any manipulation of figures on canvas requires manipulation targets to be selected. From a closer investigation, it turns out that the ‘selection tool’ contributes, among others, to: ‘basic editing’, ‘path editing’ and ‘automatic selection’.

4.6 Correlation

The correlation element refines the information provided by the characterizations of computational units, features and inter-feature relations. This element consists of a correlation matrix and a correlation graph.

The correlation matrix is based on the feature-class correlation [22] and the feature-interaction grid [31]. The matrix associates features, computational units and objects using the established affinity-coloring scheme. The arrows drawn on top of the matrix denote instantiation and usage of the classes’ instances. The sorting of the matrix’s rows and columns resembles the arrangement of analysis units in the feature-characterization and computational-unit-characterization charts.

Shown in Figure 10a, the correlation between features and classes of SVG reveals new insights into the role of the *AbstractFigure* class in both features of our focus partition. The ‘text area tool’ feature instantiates the *AbstractFigure* class, whereas the ‘selection tool’ feature uses the created instances. Most likely, this is due to the creation of text area figures on SVG’s canvas by the ‘text area tool’ feature.

These figures are then selected in order to move them, edit their attributes etc.

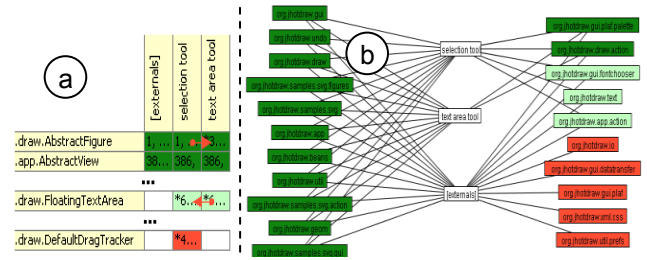


Figure 10. Correlation views: (a) excerpt from the correlation matrix, (b) feature-package-correlation graph.

The second view of the correlation element, shown in Figure 10b, is based on the feature-implementation graph [22], but here enhanced with affinity coloring. The purpose of this view is to simplify the reasoning about relations between feature classes in the correlation matrix. Selecting a feature-vertex in the graph, highlights its relations with other features in terms of concrete computational units. The highlighting algorithm is based on code sharing, or alternatively on instance sharing (dynamic inter-feature relations). Apart from simplifying the reasoning about the data contained in the correlation matrix, this view can be used as a heuristic for visualizing likely paths of change propagation during feature-centric program modification. However, in the present case study, we have no need for applying this view, since the chosen subset of features and classes was sufficiently small to be fully examined directly from the correlation matrix.

4.7 Traceability

Using the traceability element from the traceability abstraction level of our framework, we investigate the bodies of individual classes and methods in SVG’s source code. Bi-directional, navigable traceability links between features and source code are provided by two mechanisms: the feature inspector and the editor coloring.

The feature inspector’s window, depicted in Figure 11a, contains a hierarchy of nodes symbolizing the packages, classes and methods that implement a feature. The nodes symbolizing classes and methods can be used for automatic navigation to their corresponding source-code fragments in the NetBeans editor. The methods annotated as feature entry points [27] are marked in the hierarchy tree by a distinct icon, due to their special role in implementations of features.

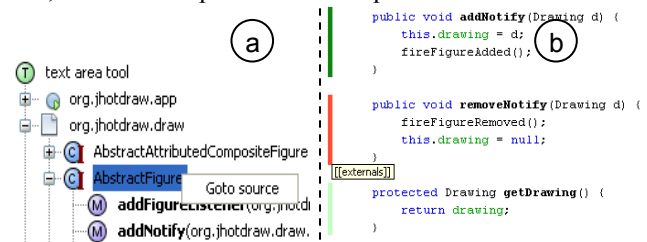


Figure 11. Traceability views: (a) navigable feature inspector and (b) editor coloring.

The Featureous plug-in enhances the default code editor of the NetBeans IDE to provide feature-centric information. Figure 11b shows how Featureous uses color bars to visualize the affinity of the viewed methods and classes. The bars allow for immediate assessment of source code with respect to its participation in implementing features. Hence, the functionality provided by the editor coloring not only supports understanding of source code in terms of the features it implements, but also simplifies the reasoning about possible consequences of source-code modifications on the correctness of a program’s functionality. Furthermore, logical traceability from source code to concrete features is provided in the form of tooltips, listing the features in which a class or method participates, that are associated with the color bars.

The feature-centric analysis of the `AbstractFigure` at the correlation level revealed a number of important details. `AbstractFigure`, being a class that provides base implementation of figures in SVG, is used by most of SVG’s features. Yet, there exist methods that are used exclusively by the ‘selection tool’ feature. These methods deal with selection-specific functionalities, like handling mouse events, dragging figures, getting a list of available figure’s actions, etc. The presence of selection-specific methods in a base class suggests that the implementation of selection mechanisms was decided to be added to the core of SVG. We reckon this is due to selection being a mandatory building block for a number of other features.

Another interesting observation is that the three most widely shared methods are concerned with notifying a figure’s observers about invalidation of a figure’s area. This indicates that this responsibility is not local to a single feature, making these notifications a concern that crosscuts implementations of multiple features in SVG.

Finally, yet importantly, we have discovered an interesting source comment preceding the method that registers a figure’s observers. This comment is a list of names of SVG’s functionalities: “drawing”, “shape and bounds”, “attributes”, “editing”, “connecting”, “composite figures”, “cloning”, “event handling”. These names resemble closely the names of SVG’s features. We reckon that this comment is a trace of the developers’ knowledge of features that observe figures. Using the code coloring tool, we have found that this method was, indeed, used by features that correspond to the listed names. This comment-based ad hoc method for capturing feature-code correspondence used by developers of JHotDraw sharply shows the need for explicit representation of traceability links between feature and source code.

4.8 Summary of the Case Study

In the presented case study, we have conducted a feature-centric analysis of the JHotDraw application SVG. Application of our feature-centric approach allowed us to understand selected aspects of feature-code mapping in the SVG application. Our experience is that the hypotheses and observations made were useful for understanding SVG from

feature-centric point of view and they provide a good starting point for modifying or correcting a feature, implementing new features as variants of the existing ones, or refactoring of feature implementations in order to improve their separation in the source code. To generalize these observations with respect to other subject programs, a larger number of software developers and various types of software modification tasks, we plan to conduct a more extensive empirical evaluation in our future work.

5. SUPPORT FOR PROGRAM COMPREHENSION

In this section, we evaluate our approach’s support for feature-centric comprehension. We do this by assessing our NetBeans plug-in Featureous in terms of its support of cognitive design elements for constructing mental models during software visualization [13].

The taxonomy of cognitive design elements presented in [13] is defined in terms of a hierarchy. At the top-most level, the hierarchy divides the cognitive design elements into two categories: improving program comprehension and reducing the maintainer’s cognitive overhead. At the current stage, where the core structure of our approach has been established, but minor adjustments in visualization techniques are expected to be introduced as Featureous evolves, we find it most relevant to evaluate our tool against the first category of cognitive elements. This evaluation is summarized in Table 1.

Table 1. Support for cognitive design elements.

Cognitive design elements for improving program comprehension		Support in Featureous
Enhance bottom-up comprehension	Indicate syntactic and semantic relations between software objects (E1)	Traceability element
	Reduce the effect of delocalized plans (E2)	Traceability element
	Provide abstraction mechanisms (E3)	Feature partitions
Enhance top-down comprehension	Support goal-directed, hypothesis-driven comprehension (E4)	Not supported
	Provide an adequate overview of the system architecture, at various levels of abstraction (E5)	Three levels of abstraction
Integrate bottom-up and top-down approaches	Support the construction of multiple mental models (domain, situation, program) (E6)	Three perspectives
	Cross-reference mental models (E7)	- Correlation element - Affinity coloring - Identifiers

Bottom-up comprehension, which we have left out from our case study, involves investigating a program’s source code in order to incrementally build high-level abstraction (being either structural abstractions, or functional abstractions), until program understanding is achieved [13]. The traceability element of our approach plays an important role in this comprehension strategy as it allows expressing con-

crete source-code statements in the context of features as well as features in terms of source-code statements (E1). Reduction of delocalized plans (E2) is supported two-fold. Firstly, from the point of view of source code units, features are delocalized plans. A reduction of this effect is done by explicit visualization through the editor coloring. Secondly, from the point of view of features as first-class analysis entities, a program's computational units are delocalized plans. This, in turn, is handled by the feature inspector. Finally, bottom-up comprehension requires a support for building high-level abstractions from already-comprehended lower-level entities (E3). In our approach, we support this by supporting adjustable granularity of computational units and manual partitioning of features into arbitrary hierarchies. Here, grouping of features into partitions helps not only to improve scalability of our approach with respect to the number of features, but also in an important mechanism for composing features into higher-level abstractions.

Top-down comprehension requires application of domain knowledge or a previous exposure to a program, based on which goals and hypotheses can be formulated to drive the comprehension process [13]. In the course of the analysis, these goals and hypotheses are refined into sub-goals and sub-hypotheses as the analysis progresses from higher levels of abstraction to the more detailed ones. Featureous does not provide any mechanisms for managing and documenting goals and hypotheses (E4) that an analyst may develop. This, however, does not prevent goal-driven and hypothesis-driven usages of our approach. As we have presented in our case study, top-down analysis is possible as long as the analyst himself keeps track of the aims of the investigations. Since the hypotheses need to be refined and concretized, our approach supports viewing programs at multiple abstraction levels (E5). This is done in terms of characterization, correlation and traceability levels of abstraction. These three levels can be used to match the level required for evaluating a given goal or hypothesis. If an evaluation cannot be made due to a too high level of abstraction, a goal or a hypothesis needs to be decomposed into separately-analyzable sub-parts that can be evaluated on the lower levels of analytical abstraction.

Finally, the integration of *bottom-up and top-down approaches* is motivated by the observation that programmers tend to alternate between these two comprehension strategies in an as-needed fashion [13]. This is due to a need for creating multiple mental models (E6) that can be switched during software comprehension. Our approach provides support for multiple mental models through the computational-unit perspective (which corresponds to the program model from [13]), the feature perspective (the situation model in [13]) and the feature relations perspectives (also situation model). In terms of individual perspectives, mental models are facilitated by viewing source code in terms of features, features in terms of code, viewing run-time information about usage of class instances by features, feature metrics and affinity coloring. It is interesting that our feature-based approach supports usage of the domain mental model [13]. As features represent functional concepts from a

program's domain, feature-centric analysis allows for understanding source code in terms of a program's domain, thus supporting the construction of domain mental models. To enhance alternation between the supported mental models (E7), Featureous provides three mechanisms for model cross-referencing. These are the correlation element, affinity coloring and identifiers of features and computational units. The correlation element supports relating the three perspectives to each other. Affinity coloring allows for relating all analysis elements to the characterization of computational units. Identifiers of feature and computational units cross-reference the three abstraction levels as well as the three perspectives of our approach.

Based on the soundness of the presented argumentation, we claim that the structure of our approach supports the core cognitive design elements required by the bottom-up, top-down and as-needed comprehension strategies. This is an important property with respect to an applicability of the feature-centric analysis during software evolution and maintenance, since diverse nature of program-modification tasks creates need for multiple comprehension strategies [34].

6. CONCLUSION

Object-oriented software is perceived by its users in terms of the features it provides, not in terms of its implementation. This is a problem particularly during evolution of large programs, because the complex implementations of existing features are not evident from the object-oriented source code. Hence, there is an urgent need for tool-supported scalable analysis approaches, which will help developers to comprehend the correspondence between features and source code.

In this paper, we have presented our solution: an approach to feature-centric analysis of object-oriented software that unifies and enhances a number of existing analysis techniques. This approach has been implemented as the Featureous plug-in to the NetBeans IDE. Featureous allows for recovery and analysis of traceability links between features and the Java source code. To decompose the complexity of the analysis process and improve its scalability, we have systematized our approach into three analytical perspectives, three abstraction levels and multiple granularities of features partitions. In the enclosed case study, we have demonstrated how to apply feature-centric analysis while reducing its complexity by recognizing similarity among features and partitioning them around their canonical features. As the last part, we have evaluated Featureous against a set of cognitive design elements in order to assess its support for program comprehension.

We believe that the provided tool Featureous will help facilitating feature-centric comprehension of legacy Java software during evolution and maintenance. Furthermore, we hope that our systematic and reproducible analysis approach can lead to a discovery of general characteristics of features in object-oriented software and to improving the current practices of implementing features in object-oriented software.

ACKNOWLEDGEMENT

We would like to thank Kemal Pajevic for providing the feature analysis software, which inspired the development of Featureous.

REFERENCES

- [1] C.R. Turner, A. Fuggetta, L. Lavazza, and A.L. Wolf, "A conceptual basis for feature engineering," *Journal of Systems and Software*, vol. 49, 1999, pp. 3–15.
- [2] O. Greevy, T. Girba, and S. Ducasse, "How Developers Develop Features," *CSMR*, 2007, pp. 265–274.
- [3] A.V. Mayrhauser and A.M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, vol. 28, 1995, pp. 44–55.
- [4] B. Cornelissen, A. Zaidman, B. Van Rompaey, and A. van Deursen, "Trace Visualization for Program Comprehension: A Controlled Experiment," *ICPC*, 2009, pp. 100–109.
- [5] D. Röthlisberger, O. Greevy, and O. Nierstrasz, "Feature driven browsing," *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, New York, NY, USA: ACM, 2007, pp. 79–100.
- [6] G.C. Murphy, A. Lai, R.J. Walker, and M.P. Robillard, "Separating Features in Source Code: An Exploratory Study," *ICSE*, 2001, p. 0275.
- [7] D. Garlan and M. Shaw, *An Introduction to Software Architecture*, Carnegie Mellon University, 1994.
- [8] T. Shaft and I. Vessey, "The Role of Cognitive Fit in the Relationship Between Software Comprehension and Modification," *MIS Quarterly*, 30(1), 2006. pp. 29–55.
- [9] O. Greevy, "Enriching Reverse Engineering with Feature Analysis," PhD thesis, University of Bern, 2007.
- [10] NetBeans IDE, <http://netbeans.org>
- [11] Featureous tool, <http://ecosoc.sdu.dk/coe/Featureous>
- [12] JHotDraw framework, <http://jhotdraw.org>
- [13] M. Storey, F. Fracchia, and H. Mueller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization," *Program Comprehension, International Workshop on*, 1997, p. 17.
- [14] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code," *Proceedings of International Conference on Software Maintenance*, 1992, pp. 200–205.
- [15] T.J. Biggerstaff, B.G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," *ICSE*, 1993, pp. 482–498.
- [16] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," *IWPC '00*, 2000, p. 241.
- [17] A.D. Eisenberg and K. De Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *ICSM '05*, 2005, pp. 337–346.
- [18] M.P. Robillard and G.C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," *ICSE '02*, 2002, pp. 406–416.
- [19] A. Egyed, G. Binder, and P. Grunbacher, "STRADA: A Tool for Scenario-Based Feature-to-Code Trace Detection and Analysis," *Proceedings of 29th International Conference on Software Engineering*, 2007, pp. 41–42.
- [20] M. Storey, K. Wong, and H. Muller, "How Do Program Understanding Tools Affect How Programmers Understand Programs," *WCRE*, 1997, p. 12.
- [21] M. Lanza, "CodeCrawler - Lessons Learned in Building a Software Visualization Tool," *CSMR*, 2003, p. 409.
- [22] O. Greevy and S. Ducasse, "Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach," *CSMR*, 2005, pp. 314–323.
- [23] R. Brcina and M. Riebisch, "Architecting for evolvability by means of traceability and features," *Automated Software Engineering - Workshops*, 2008, pp. 72–81.
- [24] W.E. Wong, S.S. Gokhale, and J.R. Horgan, "Quantifying the closeness between program components and features," *J. Syst. Softw.*, vol. 54, 2000, pp. 87–98.
- [25] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh, "On Computing the Canonical Features of Software Systems," *Proceedings of the 13th Working Conference on Reverse Engineering*, 2006, pp. 93–102.
- [26] J. Kothari, T. Denton, A. Shokoufandeh, and S. Mancoridis, "Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence," *Proceedings of the 15th International Conference on Program Comprehension*, 2007, pp. 17–26.
- [27] A. Olszak and B.N. Jørgensen, "Remodularizing Java programs for comprehension of features," *Proceedings of the First International Workshop on Feature-Oriented Software Development*, ACM, 2009, pp. 19–26.
- [28] S. Rugaber, K. Stirewalt, and L.M. Wills, "The Interleaving Problem in Program Understanding," *Reverse Engineering, Working Conference on*, 1995, p. 166.
- [29] S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software*, vol. 3, 1986, pp. 41–49.
- [30] M.P. Robillard, W. Coelho, and G.C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *Transactions on Software Engineering*, 2004, pp. 889–903.
- [31] M. Salah and S. Mancoridis, "A Hierarchy of Dynamic Software Views: From Object-Interactions to Feature-Interactions," *ICSM*, 2004, pp. 72–81.
- [32] J.H. Holland, *Adaptation in natural and artificial systems*, Cambridge, MA, USA: MIT Press, 1992.
- [33] A. Lakhotia, "Understanding someone else's code: analysis of experiences," *J. Syst. Softw.*, vol. 23, 1993, pp. 269–275.
- [34] M. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," *13th International Workshop on Program Comprehension*, 2005, pp. 181–191.