IADIS International Journal on Computer Science and Information Systems
Vol. 6, No.1, pp. 58-75
ISSN: 1646-3692

FEATUREOUS: AN INTEGRATED ENVIRONMENT FOR FEATURE-CENTRIC ANALYSIS AND MODIFICATION OF OBJECT-ORIENTED SOFTWARE

Andrzej Olszak and Bo Nørregaard Jørgensen
The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
{ao, bnj}@mmmi.sdu.dk

ABSTRACT

The decentralized nature of collaborations between objects in object-oriented software makes it difficult to understand the implementations of user-observable program features and their respective interdependencies. As feature-centric program understanding and modification are essential during software maintenance and evolution, this situation needs to change. In this paper, we present Featureous, an integrated development environment built on top of the NetBeans IDE that facilitates feature-centric analysis of object-oriented software. Our integrated development environment encompasses a lightweight feature location mechanism, a number of reusable analytical views, and necessary APIs for supporting future extensions. The base of the integrated development environment is a conceptual framework comprising of three complementary dimensions of comprehension: perspective, abstraction and granularity. Together, these dimensions allow the analyst to focus the analysis at the right mode of comprehension during software evolution. We demonstrate applicability of our integrated development environment by conducting a case study of change adoption using the JHotDraw SVG.

KEYWORDS

Features, feature-centric analysis, software evolution

1. INTRODUCTION

Feature-centric analysis (Greevy, 2007) helps developers to perceive object-oriented software in terms of its user-observable behavior (Turner et al, 1999). The need for feature-centric analysis is constantly encountered during software evolution and maintenance, since users formulate their functional requirements, change requests and error reports in terms of features

(Turner et al, 1999)(Mehta and Heineman, 2002). The ability to relate these descriptions to relevant fragments of object-oriented source code is a prerequisite to feature-wise modification (Greevy et al, 2007), error correction (Cornelissen et al, 2009)(Röthlisberger et al, 2007), change impact assessment (Ryder and Tip, 2001) and derivation of new features from existing ones.

Relating features to their implementations is, however, a difficult task, since object-oriented programming languages provide no means for representing features explicitly. In object-oriented programs, features are implemented as inter-class collaborations crosscutting multiple classes as well as multiple architectural units (Murphy et al, 2001). This physical tangling and scattering of features over several units of code makes their implementations difficult to identify and understand (Turner et al, 1999)(Shaft and Vessey, 2006).

The complexity and size of feature-code mappings creates a need for a tool-supported analysis approaches. The role of tools is to guide the analysis process in a systematic fashion. Secondly, tool support is needed to automate repetitive and error-prone calculations, and thereby to ensure reproducibility and scalability of analytical activities. Finally, we deem it necessary to integrate tools for feature-centric analysis with contemporary software development environments, so that feature-centric analysis can be assimilated as part of standard activities during software evolution and maintenance.

In this paper, we present Featureous, our integrated development environment for feature-centric analysis of object-oriented programs. Our environment extends the NetBeans Java IDE (*http://netbeans.org*), and provides a lightweight dynamic feature location mechanism together with a set of APIs exposing the basic building blocks for supporting new feature-centric analytical views. In order to impose a conceptual structuring on possible views developed on top of our integrated development environment, we propose three dimensions for categorizing feature-centric views. Thus, each view can be represented as a point in the three-dimensional space of: perspectives, abstractions and granularity. To motivate the need for proposed conceptual framework during software evolution, we discuss the applicability of concrete configuration of views during individual phases of the change mini-cycle (Bennett and Rajlich, 2000).

In order to demonstrate applicability of feature-centric analysis using Featureous, we have implemented a selection of state-of-the-art feature-centric views. We show how these views can be applied in practice to gain insights into an unfamiliar mid-sized codebase. This is done in the context of a case study of feature-centric analysis and modification of the SVG application built on top of the JHotDraw framework (*http://jhotdraw.org*). Since Featureous is publically available from our website (*http://featureous.org*), the analytical views and procedures described in the case study can be immediately applied in third-party contexts.

The remainder of this paper is as follows. In Section 2, we present state-of-the-art on which we base our integrated development environment. In Section 3, we discuss our conceptual framework for feature-centric analysis and place it in the context of software evolution. Section 4 describes the design and APIs of Featureous. In Section 5, we apply feature-centric analysis in a case study of feature-centric modification. Finally, Section 6 concludes the paper.

2. STATE OF THE ART

Feature-centric analysis supports the understanding of object-oriented software by considering features as first-class analysis entities (Greevy, 2007). One of the basic elements of feature-centric analysis is the bi-directional traceability links between features and object-oriented source code. Tools that explicitly visualize this correspondence were shown to simplify discovering classes implementing a given feature and features implemented by a given class (Röthlisberger et al, 2007)(Kästner et al, 2008)(Robillard and Murphy, 2002).

By analyzing the established traceability links, it is possible to characterize features in terms of classes and characterize classes in terms of program features (Greevy and Ducasse, 2005). These characterizations can be used to investigate inter-feature relations in terms of implementation overlap. Furthermore, the static characterization based on classes can be complemented by views based on usage of objects by executing features (Salah and Mancoridis, 2004). This allows for examining run-time inter-feature dependencies.

The information contained in feature-code traceability links can be summarized by usage of software metrics. The approaches described in (Brcina and Riebisch, 2008)(Wong et al, 2000) have recognized applicability of the metrics traditionally associated with the separation of concerns to analyzing features. The two metrics proposed in (Brcina and Riebisch, 2008) - scattering and tangling - assess quantitatively the complexity of the relationships between features and computational units.

Finally yet importantly, feature location procedures are used by feature-centric analysis approaches for identification of source code fragments that contribute to implementations of program features (Wilde and Scully, 1995). The two major types of existing approaches based on static analysis (Chen and Rajlich, 2000), and dynamic analysis (Wilde and Scully, 1995)(Eisenberg and De Volder, 2005)(Olszak and Jørgensen, 2010) differ with respect to level of automation, accuracy, and repeatability. The location approach that we adopt in this paper is a dynamic, semi-automated technique defined in (Olszak and Jørgensen, 2010). Since it relies on tracing of program execution, it allows for resolving polymorphic invocations, detecting common usages of objects among multiple executing features, and takes into account the effect of branch instructions on control flow.

Summing up, the existing approaches define a set of diverse methods for feature-centric analysis. Nevertheless, there is no common conceptual framework and technical platform for integrating them and exploring their mutual advantages. Moreover, for some of the mentioned approaches there remain questions about their practical applicability during software evolution and maintenance, both because of the lack of a conceptual framework for doing so and because of the lack of publically available tools implementing them.

3. CONCEPTUAL FRAMEWORK FOR FEATURE-CENTRIC ANALYSIS OF SOFTWARE

Feature-centric analysis is the process of analyzing programs by considering features as first-class analysis entities. What distinguishes features from other types of source code concerns is their inherent relationships to concepts in the problem domain. As the structure of object-oriented software rarely modularizes and represents features explicitly, any task related to change of program functionality is likely to crosscut multiple units of source code. Thus, a modification to one feature is likely to affect the correctness of other features which use the code fragments being modified.

Feature-centric analysis can be thought of as a special instance of a more general problem of *cross-decomposition analysis*. Software can be decomposed according to various criteria and thus made to modularize different dimensions of concerns in source code. However, the majority of modern programming languages only allow us to modularize one dimension of concerns at a time, called the *dominant dimension* (Tarr et al, 1999). The correspondence between modules of the dominant decomposition of a software program and one of its alternative decompositions is in general many-to-many. This is due to the phenomena of *scattering*, where a single module of an alternative decomposition is dislocated over a number of modules of the dominant decomposition, and *tangling*, where multiple modules of an alternative decomposition are interwoven in a single module of the dominant decomposition. This lack of isomorphic correspondence between the dominant and alternative decompositions affects changeability of programs, since different kinds of changes require different units of change. If the required change unit is modularized in the dominant decomposition, then the change can be performed in a localized manner. However, if the change unit is scattered over multiple modules, each of them will have to be modified to implement the change. It may also happen that change made to one of such under-represented concerns will result in an unforeseen modification of another one due to their tangling in terms of the same computational unit.

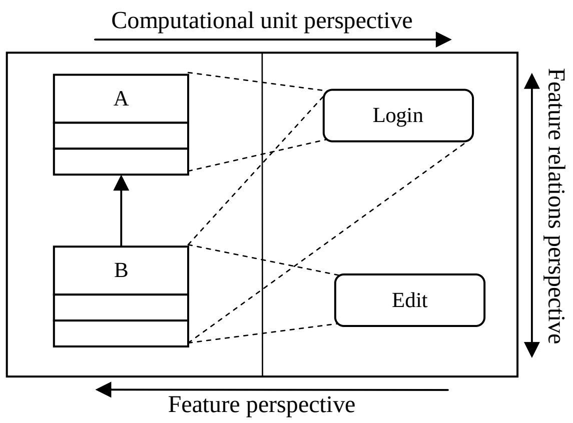


Figure 1. Perspectives on feature-code traceability links , based on (Greevy and Ducasse, 2005).

The mentioned situations occur for object-oriented legacy software, if tried to be perceived in terms of feature-oriented decomposition criteria. Example of such a situation is shown in Figure 1.

The correspondences between object-oriented and feature-oriented decompositions of software can be investigated from four *perspectives*, based on the concrete needs of a programmer. For instance, a programmer who is given a report about an error in a particular feature would be interested in inspecting the classes that implement this feature, hence she would use the feature perspective. After the error is corrected the programmer could use the computational perspective to reason if her modifications will affect the correctness of any other features in the program. The feature relations perspective can be used by programmers to assess the overlap of implementations of two features. In summary, the three perspectives are defined as follows:

1. *Computational unit perspective* shows how computational units like packages and classes participate in implementing features (Greevy and Ducasse, 2005).
2. *Feature perspective* focuses on how features are implemented. In particular, it describes features in terms of their usage of a software program's computational units (Greevy and Ducasse, 2005).

3. *Feature relations perspective* focuses on inter-feature relations that can be deduced from the feature-code mapping (Greevy, 2007).

We reckon that one of the major benefits of separating the analytical concerns by means of multiple perspectives, apart from imposing a structure on the analysis process, is reducing the complexity of analysis. This is because having multiple perspectives on the many-to-many correspondence between features and computational units allows us to avoid investigating this complex mapping directly. Instead, analysis is conducted on a number of one-to-many mappings, which are considerably easier to understand.

Within our framework, the perspectives are one of the *three dimensions* used for categorizing feature-centric analytical views. The other two dimensions are *abstraction* and *granularity*, as visualized in Figure 2.

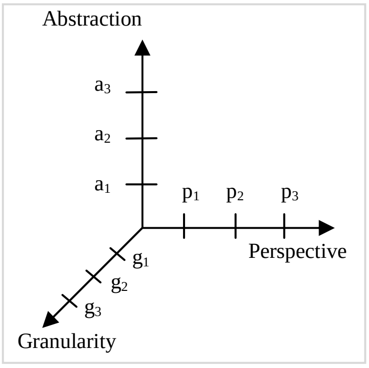


Figure 2. Three-dimensional conceptual framework.

The purpose of providing stratified levels of abstraction is to focus the analysis process by limiting the amount of information simultaneously presented to the analyst. Furthermore, stratified abstraction levels allow the complexity of a program's features to be investigated in an incremental fashion. We define three levels of abstraction:

1. *Characterization level* shows high-level diagrams that summarize the overall complexity of feature-code mappings.
2. *Correlation level* provides correlations between individual features and computational units.
3. *Traceability level* provides navigable traceability links between features and source code.

Correspondence between features and different granularities of computational units is supported by three levels of granularity: 1. *Package granularity*; 2. *Class granularity*; 3. *Method granularity*.

These three dimensions define a common categorization scheme that allows Featureous to be extended with a multitude of analytical views. Using this scheme it is possible to characterize feature-centric analytical views in terms of three coordinates. For instance, view $\{p_2, a_1, g_2\}$ could be a feature-class characterization in form of a plot, whereas view $\{p_1, a_2, g_1\}$ could provide a correlation view of features and packages in a program in form of a graph or a table. The need for concrete types of feature-centric views during software evolution is discussed in Section 3.1, whereas the set of the current views implemented by Featureous is described in Section 3.2.

3.1 Feature-centric Modification of Evolving Programs

In this subsection, we analyze the change mini-cycle, being a detailed model of stages of change adoption during software evolution (Bennett and Rajlich, 2000), and describe how the concepts of feature-centric analysis can be used to support evolutionary modification of programs. We do so by focusing on a situation in which a feature-centric change occurs in a software's problem domain, thus causing the solution domain artifacts (the source code of an object-oriented program) to be modified according to the *feature* as the *unit of change*. The presence of the discussed earlier mismatch between the unit of change and the unit of program decomposition contributes to cognitive overhead and thus hinders change adoption activities. In this context, we discuss how concrete elements from the three dimensions of our conceptual framework, i.e. perspectives, abstraction levels and granularities, can be used to support individual phases of the change mini-cycle, hereby establishing a model for a feature-centric change mini-cycle. This model serves as basis for designing concrete feature-centric views, which we present in Section 3.2, and as the underlying methodology for the case study of evolutionary modification presented in Section 6.

Table 1. Applicability of feature-centric techniques during change mini-cycle

	Request for change	Planning phase		Change implementation		Verification and validation	Re-documentation
		Program comprehension	Change impact analysis	Restructuring for change	Change propagation		
Perspective	n/a	p_2	p_1, p_2, p_3	p_1, p_2	p_1, p_2,	p_1	n/a
Abstraction	n/a	a_1, a_2	a_1, a_2, a_3	a_2, a_3	a_2, a_3	a_3	n/a
Granularity	n/a	g_1, g_2	g_1, g_2	g_1, g_2, g_3	g_2, g_3	g_2, g_3	n/a

Table 1 shows the correlation between the individual phases of the change mini-cycle and the levels of perspective, abstraction and granularity applicable for them. The general trends suggest that more abstract and more coarse-grained views are appropriate at earlier stages of change adoption, whereas the later stages require more concrete and fine-grained analysis. In the following, we discuss and motivate the presented correlation.

Request for change. Software's users initiate the change mini-cycle by communicating their changed expectations in the form of a request for change. During this process, they often use vocabulary related to features of software, since users perceive software through its identifiable functionality (Turner et al, 1999). Henceforth, such a feature-centric request for adding new features, enhancing existing features or correcting errors in existing features becomes a basis for feature-centric modification.

Planning phase. The goal of the planning phase is to evaluate feasibility of a requested change by understanding its technical properties and its potential impact on the rest of the software. This is not only a prerequisite for later implementation of the change, but can also be used to informed prioritization and scheduling of change implementation during multi-objective release planning (Saliu and Ruhe, 2007).

During the process of *program comprehension* one investigates a program in order to locate and understand the computational units participating in a feature of interest. In order to focus the comprehension process within the boundaries of the feature of interest, and thereby to reduce the search space, one can use the *feature perspective* on feature-code traceability links. The narrowed-down search space granted by feature-centric analysis allows for easier discovery of relevant *initial focus points* that can also be feature-wisely *built upon* by navigating along static relations between feature-intrinsic code units to gain understanding of a whole feature-*sub-graph* (Sillito et al, 2006). The concrete levels of *abstraction* (i.e. characterization or correlation) and *granularity* (i.e. package or class) should be chosen based on the precision of understanding required for performing change impact analysis.

After gaining sufficient understanding of the implementation of a feature, one performs *change impact analysis* in order to estimate the set of code units that will need to be modified during change implementation. This is performed by investigating how changes in the specification of a feature manifest themselves in computational units (feature perspective) and how other features can be affected by modifying shared computational units (computational unit perspective). Furthermore, it is necessary to consider both the logical and syntactical relations between features (feature relations perspective) in order to determine if modification of pre or post-conditions of a feature can affect its dependent features. Based on the desired precision of the estimated change impact the analysis can be performed on various levels of abstraction (i.e. characterization, correlation or traceability) and granularity (i.e. package or class). Ultimately, the thereby obtained results of change impact analysis should form a basis for an organizational-level estimation of the cost involved in a planned change implementation.

Change implementation. During change implementation, one modifies a program's computational units according to a corresponding request for change. This phase first prepares for accommodating a change by restructuring the computational units, and then propagates the change by modifying them respectively.

Restructuring involves reducing the scattering of feature-centric modifications by applying behavior-preserving transformations to source code in order to localize and isolate code units implementing a particular feature. Doing so decreases the number of computational units that have to be visited and modified, and reduces the impact on implementations of other features, so that features can evolve independently from each other. During restructuring the feature perspective is used to identify boundaries of a feature implementation under restructuring and the computational units perspective to identify portions of code shared by features that are candidates to splitting. Since restructuring involves modification of the source code, one should apply analytical views operating on the two lowest levels of abstraction (i.e. correlation and traceability). Moreover, the feature-code mappings have to be investigated at all granularities (i.e. package, class, method) in order to identify and address all tangling-introducing computational units.

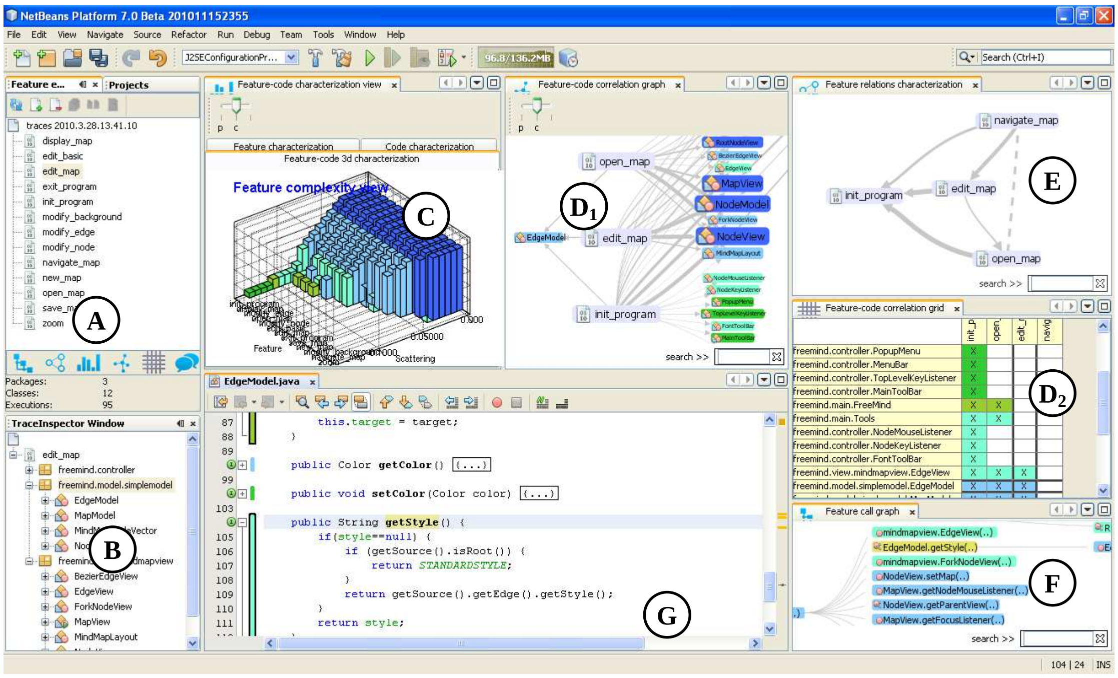
Change propagation deals with implementing a change and with ensuring syntactical consistency of a resulting program. In order to implement a change, a programmer needs to use the feature perspective to identify the concrete fine-grained (i.e. class and method granularity) parts of a feature's implementation that need to be modified as well as the parts that can be readily reused. This involves applying views at the levels of abstraction closest to source code – correlation and traceability. On the other hand, the computational units perspective is essential to facilitate feature-wise navigation over source code, to make feature-boundaries explicit and to give early indication of modifying feature-shared code (causes inter-feature change propagation) and of reusing single-feature code (increases evolutionary coupling of features and can be a sign of design erosion).

Verification and validation. The goal of verification and validation phase is to ensure that a newly modified feature is consistent with its description and that the remaining features of a program were not negatively affected by the change. By investigating the fine-grained (i.e. class and method granularity) changes made in the source code from the computational unit perspective it is possible to precisely determine which features were altered in the course of change implementation. This is done at the traceability level of abstraction, where detailed traceability links from code to features are available. Each of the features whose implementation was modified needs to be considered as candidates for validation. Such a derivation of a minimal set of features to be tested can result in significant savings of project resources in situations where the validation process is effort-intensive and time-consuming by its nature (e.g. manual testing, field-testing of control or monitoring systems).

Re-documentation. The re-documentation effort aims at updating the existing user-documentation of a program as well as developer-documentation. The goal of updating the latter is to capture information that can reduce the program comprehension effort during forthcoming evolutionary modifications. Tool support for feature-centric analysis can help developers to map features to existing use-case-based documentation (Olszak and Jørgensen, 2010). However, the real strength of supporting feature-centric analysis in an integrated development environment is that it practically eliminates any need for maintaining such mappings manually, as a feature-code mapping can be generated on-demand. Hence, the burden of maintaining textual documents that describe the implementations of a program's features in parallel with maintaining the source code vanishes.

3.2 Feature-centric Views

In its current version, Featureous is equipped with seven views that address the most important areas of our conceptual framework. An overview of the views in the user interface of the tool is presented in Figure 3.


NetBeans Platform 7.0 Beta 201011152355
Feature explorer
Projects
TraceInspector Window
Feature-code characterization view
Feature-code 3d characterization
Feature complexity view
Feature-code correlation graph
Feature relations characterization
Feature-code correlation grid
Feature call graph
EdgeModel.java
A
B
C
D1
D2
E
F
G

```
        this.target = target;
    }

    public Color getColor() {...}

    public void setColor(Color color) {...}

    public String getStyle() {
        if(style==null) {
            if (getSource().isRoot()) {
                return STANDARDSTYLE;
            }
            return getSource().getEdge().getStyle();
        }
        return style;
```

	init_p	open_	edit_	navig
freemind.controller.PopupMenu	X			
freemind.controller.MenuBar	X			
freemind.controller.TopLevelKeyListener	X			
freemind.controller.MainToolBar	X			
freemind.main.FreeMind	X	X		
freemind.main.Tools	X	X		
freemind.controller.NodeMouseListener	X			
freemind.controller.NodeKeyListener	X			
freemind.controller.FontToolBar	X			
freemind.view.mindmapview.EdgeView	X	X	X	
freemind.model.simplemodel.EdgeModel	X	X	X	

Figure 3. User interface of Featureous.

Feature explorer (marked as *A* in Figure 3) is the main window of Featureous. It displays features of a program, allows adding and removing features under analysis and grouping them into arbitrary hierarchies. From within *feature explorer* one can open the analytical feature-centric views of Featureous by selecting a set of target features and clicking on a view's icon.

Feature inspector (B) {p_2, a_3, $g_{1,2,3}$} provides navigable traceability links from features to concrete fragments of a program's source code. The feature inspector's window 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 view (G).

Feature-code 3D characterization (C) {$p_{1,2}$, a_1, $g_{1,2}$} serves as a coarse-grained summary of the complexity of feature-code relations. Depending on the chosen rotation angle of the 3D chart, the view gives either a summary of the distribution of feature implementations over computational units, or a characterization of computational units with respect to their participation in feature implementations. This view incorporates the feature-characterization and computational units characterization views (Greevy and Ducasse, 2005) as well as the metrics of *scattering* of feature implementations among a program's computational units and *tangling* of features in terms of computational units (Brcina and Riebisch, 2008).

Feature-code correlation graph (D_1) and *feature-correlation grid* (D_2) {$p_{1,2}$, a_2, $g_{1,2}$} enable a detailed investigation of the correspondence between computational units and features. The graph view is equipped with selection-driven re-layout functionality, so that it is possible to switch between the computational unit and feature perspectives by focusing the graph on a given node. These views incorporate the feature-implementation graph and the feature-class correlation (Greevy and Ducasse, 2005).

Feature relations characterization (E) {p_3, a_1, g_2} relates features to each other based on the dynamic dependencies between them, i.e. instantiation of classes and common usage of objects. This is done through a graph-based representation, where nodes represent features and edges represent dynamic relations between features producing objects, features using produced objects and features co-using common objects. This view is based on the feature-interaction graph (Salah and Mancoridis, 2004).

Feature call graph (F) {p_2, a_3, $g_{2,3}$} displays the call hierarchy of methods implementing a given feature. By visualizing the call-relations between methods and their containing classes, this view aims at enabling inspection of feature implementations according to their run-time execution flow. Nodes of the view, representing computational units, are automatically folded to visualize only the currently investigated path in the call hierarchy.

Feature-aware source code editor (G) {p_1, a_3, $g_{2,3}$} extends the default source code editor of the NetBeans IDE with a sidebar visualizing participation of computational units in implementing features. This gives a programmer precise fine-grained information about relevance of a fragment of source code to a feature of interest and the degree of feature tangling involved. Furthermore, we have modified the default folding mechanism of the editor to provide automatic folding of methods that do not contribute to a given feature of interest. Thereby, it is possible to hide irrelevant code during comprehension of code subject to modification when changing a concrete feature.

4. DESIGNING FEATUREOUS

Featureous is implemented on top of the NetBeans Rich-Client Platform (RCP) and tightly integrated with Java IDE capabilities of the platform. The usage of the module system and the lookup mechanism of the NetBeans RCP allowed us to achieve extensibility of Featureous with respect to adding new analytical views without any need for re-compilation. Tight integration with the IDE makes it possible to seamlessly combine Featureous with other programming tools commonly used in contemporary software development.

The infrastructure provided by Featureous is centered around a feature location mechanism, an extension API that exposes the discovered feature-code traceability links and a set of other APIs that can be exploited by feature-centric views.

4.1 Feature-location Infrastructure

Feature-centric analysis operates on the traceability links between features and object-oriented source code. For establishing this traceability, our approach relies on the feature location mechanism defined in (Olszak and Jørgensen, 2010). This approach requires annotating *feature entry points* in the source code of an investigated program. Feature entry points are the methods through which the execution flow enters the implementations of features. In case of GUI programs, in which features are triggered through GUI elements, feature entry points will most often be the *actionPerformed* methods of event-handling classes. Feature entry point annotations placed on method declarations have to be parameterized by string-based identifiers of their corresponding features. Based on annotations inserted by a programmer in the source code, Featureous locates feature implementations by tracing the program's execution flow while a user interact with it. During program execution a tracing agent registers any object innovations encountered within the control context of feature entry points. The tracing process is transparent for the user and it does not introduce any significant performance or memory overhead, since the tracing agent does not register information about timing and ordering of captured method invocations. The feature location capability of Featureous is provided in the NetBeans IDE by two execution buttons placed on the main toolbar of the IDE: "Run traced project" and "Test traced project". Hence, it is possible to perform user-driven triggering of features for arbitrary legacy programs regardless of the presence of appropriate feature-triggering test suites, and similar to gradually implement such test suites in order to incorporate feature location as a part of a team's development practices.

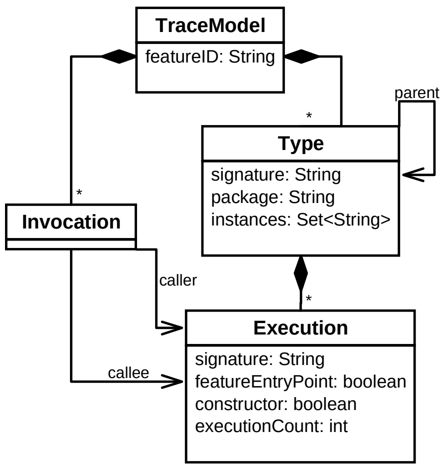


Figure 4. Feature trace model of Featureous.

The feature location mechanism produces a set of feature traces that contain a mapping between features and source code of a program. The model that we use to represent a trace of a single feature is shown in Figure 4. Feature trace models, being inputs to the feature-centric analysis, contain the information about packages, methods, constructors, classes, instances, and inter-method invocations that occurred at run-time in implementations of features. This data, as well as some additional utilities are then exposed through a set of APIs to feature-centric views created on top of our infrastructure.

4.2 Extension API

The most important API of Featureous is the API that allows for accessing feature traces and for registering third-party feature-centric views. An illustrative example of using this API is shown in Figure 5.

```
@ServiceProvider(service=AbstractTraceView.class)
public class ExampleView extends AbstractTraceView {

  public ExampleView() {
    setupAttribs("ExampleView", "", "pkg/icon.png");
  }

  public TopComponent createInstance() {
    return new ExampleView();
  }

  public void createView() {
    Controller c = Controller.getInstance();
    Set<TraceModel> ftms = c.getTraceSet().getAllTraces();
    String msg = "No. of traces loaded: " + ftms.size();
    JLabel status = new JLabel(msg);
    this.add(status, BorderLayout.CENTER);
  }

  public void closeView() { ... }
}
```

Figure 5. Extending Featureous with a new feature-centric view.

The access to feature trace models is obtained through the *Controller* singleton class contained in the *core* module of Featureous. Apart from providing the access to trace set, *Controller* exposes a number of helper methods, such as: loading and unloading of traces, splitting and merging traces and accessing the *affinity categorization of computational units* discussed in the next section.

In the example view implemented in Figure 5 the *AbstractTraceView* abstract class defined in Featureous' core is being extended to create a new view that displays the number of feature traces currently loaded in the tool. The three abstract methods (i.e. *createInstance*, *createView*, *closeView*) are used as a part of template method pattern in the base class and will be called by it upon the creation, opening and closure of the view. A class implemented in the presented way can be enclosed in a separate NetBeans module to be dynamically looked-up by Featureous at run-time thanks to the registration as a *provider* of the AbstractTraceView *service*. Each of the found views is then represented by a button in the main toolbar of the *feature explorer* window of Featureous. In the provided example, the new service is registered by annotating the *ExampleView* class with the *ServiceProvider* annotation.

4.3 Affinity API

All the feature-centric analytical views bundled with Featureous make use of the affinity API that defines a common coloring scheme for computational units. The purpose of the affinity scheme is to be an indicator of the *reuse type* of participation of a computational unit among features of a program. We propose our affinity scheme as an alternative to the affinity scheme proposed by Greevy and Ducasse, in order to distinguish different categories of reuse of code between features more precisely than by using arbitrary threshold values, as it is done in (Greevy and Ducasse, 2005).

Our affinity scheme is based on finding the so-called canonical features in the feature set of a program (Kothari et al, 2006). Then, the features are grouped around their canonical centroids thus forming groups of implementation-wisely similar features. Based on this grouping, we distinguish four categories of computational units with respect to their usage among features and canonical groups:

- *Single-feature units* are the computational units that participate in implementing only one feature. We visualize them in green.
- *Core units* are computational units reused by all canonical groups. The high degree of reuse of such units among diverse feature-groups means that they are a part of the reusable core of the software. We visualize these units in blue.
- *Single-group units* are computational units used exclusively within a single feature-group. Such units form a local core reused within a group of functionally related features, as opposed to the global core built of *core units*. We visualize these units in light green.
- *Inter-group units* are computational units reused among multiple, but not all, feature-groups. This reuse category is half-step between the *core* and *single-group* categories, and hence includes classes that in the future can either be promoted to being *core* or be confined to single feature-groups by correcting cases of inappropriate inter-group reuse. We visualize these units in light blue.

The affinity information can be accessed through the *Controller* object by getting its aggregated *AffinityProvider* object. *AffinityProvider* interface exposes the information about the affinity characterization and affinity coloring of concrete packages, classes and methods in a program. By hiding the concrete affinity providers behind a common interface, it is possible for third parties to easily replace the default provider and thereby extend Featureous with customized domain-specific affinity schemes.

4.4 Selection API

As a part of the Featureous, we provide a global selection API. On the technical side, this API allows feature-centric views for listening and reacting to changes in selection of features and computational units in other views. On the conceptual side, the selection API enables context-sensitivity of feature-centric views, so that they can dynamically re-adjust themselves according to the features and units that a user investigates.

The API is provided by *SelectionManager* class and is to be used by implementing the provided *SelectionChangeListener* interface and registering created listeners within *SelectionManager*. Each of the registered listeners will get notified whenever there occurs a change in the set of selected features or computational units. In addition, it is possible for any view to use *SelectionManager* to alter the sets of selected entities and thus change the current focus of other feature-centric views.

In the current version of Featureous focus in the feature dimension is driven by selecting features in the *feature explorer* view, whereas focus in computational units dimension is driven by either selecting computational units in the *feature inspector* view or moving the caret of the *feature-aware source code editor* to classes and methods of interest. Most of the views are made to dynamically react to such events by highlighting the selected entities or, in the case of graph-based views, dynamically re-layouting their visualizations.

5. FEATURE-CENTRIC ANALYSIS AND MODIFICATION – A CASE STUDY

In this section, we present a case study of feature-centric analysis performed during a change adoption task in an evolving object-oriented program. The methodology applied in this study followed the feature-centric change mini-cycle presented in Section 3.1.

The program under investigation was a vector graphic-drawing editor for Java called SVG, which is an instantiation of the JHotDraw 7.2 framework (*http://jhotdraw.org*). The program consists of 62K lines of code and contains significantly many features for the case study to be considered a realistic application scenario.

The *request for change* that we chose as a starting point of our investigations was to modify the *export* feature of SVG so that a *watermark text* is added to any drawing file being exported.

To creating a basis for adopting the request for change, we had to establish traceability links between features and source code of SVG. To achieve this, we needed to recover the list of features of SVG, since no requirement specification documents were available. In order to identify features of SVG, we have inspected the executing application. We have performed this by investigating user-triggerable functionality in graphical user interface elements like the main menu, contextual menus, and toolbars. We have identified 28 features, whose 91 feature entry point methods we have annotated in the program's source code. By manually triggering each identified feature at run-time in the instrumented SVG program, we obtained a set of feature traces that became the input to feature-centric analysis.

In the *planning phase*, we wanted to get an impression of the effort needed to perform the intended modification task. This was done by investigating our target *export* feature, which is responsible for exporting drawings from the program's canvas to various file formats, through the *feature-code 3D characterization* view {$p_{1,2}$, a_1, $g_{1,2}$}. Feature-code 3D characterization view, through its *feature perspective*, visualizes the scattering of a feature's implementation over computational units of a program while revealing the complexity of the contributing computational units it the terms of tangling that they exhibit. This is done in three-dimensional bar chart, where each feature is represented by a number of bars corresponding to its computational units. The number of bars displayed for each feature depicts the scattering of its implementation, which corresponds to the size of the search space during a feature-centric change task. The height of the individual bars represents their

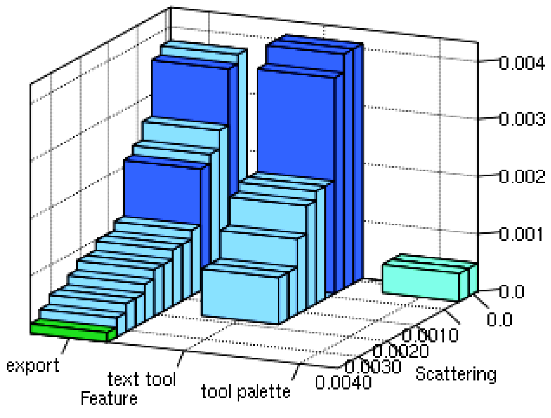


Figure 6. Feature characterization of SVG.

degree of tangling, thereby indicating the complexity of an occurring feature overlap. Furthermore, the affinity-based coloring of the bars characterizes the individual computational units with respect to the way they are reused among features. This shows the *potential impact* of modifying a part of the implementation of a given feature on the remaining features. I.e. modifying a single-feature, green-colored unit within a feature's implementation will only affect the feature itself, whereas changing a blue-colored core unit might affect, and possibly break, multiple other features.

The resulting feature-code 3D characterization obtained for some of the features of interest in SVG at the granularity of classes is shown in Figure 6. It can be seen that the *export* feature is scattered over a fairly low number of classes, 15 in total, which is the maximum number of classes that we will have to investigate and understand in order to introduce a watermark text to exported files. However, this feature contains only one feature-specific class, which indicates a high chance for our modification to propagate to implementations of other features in unanticipated ways, hence increasing the overall effort of change propagation. Another feature of interest shown in Figure 6 is the *text tool* feature. Since this feature is responsible for drawing the text-based shapes on the editor's canvas, we hypothesized that it contained classes that we needed to reuse to programmatically draw a watermark in an exported drawing. The relatively low extent of scattering of this feature indicated that we would not have to visit many classes in order to find the ones that we will reuse.

During *change implementation* we investigated the implementations of *export* and *text tool* features in a greater detail in order to identify the concrete classes that we needed to modify and reuse to implement the change request.

To analyze the implementations of both features we applied the fine-grained traceability from features to computational units offered by the *feature call tree* view $\{p_2, a_3, g_{2,3}\}$. The feature call tree view, depicted in Figure 7, is a graph containing a hierarchy of nodes symbolizing class-name-qualified methods. The edges between method-nodes stand for feature-wise inter-method *call*-relations recorded at run-time. Such a design allows one to follow the control flow of a feature incrementally from the point it was triggered in its feature entry points by selectively navigating relevant call-relations. If requested, it is also possible to automatic navigate in the NetBeans code editor to the source code fragments corresponding to a concrete node being investigated. In the context of our case study, we found the control flow-based method of browsing of feature implementations to let us understand features faster, as compared to the alternative structure-based browsing strategy offered by the *trace inspector* view. Nevertheless, we reckon that the structure-based approach can be more appropriate in situations where the design of software has to be taken into account, e.g. during refactorings or migrations of monolithic programs to component architectures.

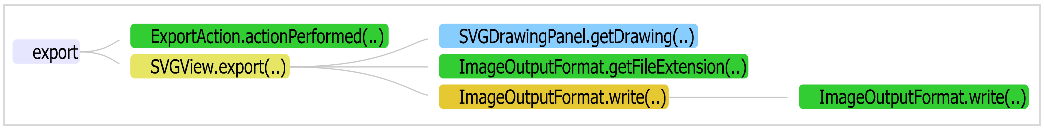

export
ExportAction.actionPerformed(..)
SVGView.export(..)
SVGDrawingPanel.getDrawing(..)
ImageOutputFormat.getFileExtension(..)
ImageOutputFormat.write(..)
ImageOutputFormat.write(..)

Figure 7. Navigable feature call tree view

We used the feature call tree view on the *export* feature to find the class that saves a drawing to a file on the disk, in order to modify the drawing before it is being saved. We discovered that the export feature consists of a chain of invocations involving many classes, most of which could be a possible point of equipping a drawing with a watermark. In order to confine the impact of our change, we chose to modify the *SVGView* class. This class implements a visual rendition over a domain model of a SVG drawing and contains *export* method that saves the underlying model to a file. Hence, *export* is a good candidate for implementing our change by inserting a watermark into the model before the drawing is being saved and removing the watermark afterwards. It is worth mentioning that the usage of feature-centric analysis was of significant help during localization of the *SVGView* class and the *export* method, not only because it narrowed down the search space, but also because it helped us to decipher the polymorphism-based relations between the JHotDraw framework and the SVG application built on top of it. We reckon that without the support of Featureous the indirection created by polymorphism-based variation points of the framework would be a significant obstacle to finding the concrete framework classes that carry out the export-related functionality in SVG.

After locating the to-be-modified method in the *export* feature, we investigated the *text tool* feature in order to identify a domain class suitable for representing a programmatically-insertable watermark text. By traversing the call tree of the feature and analyzing the roles of classes and methods, we have located the candidate class named *SVGTextFigure*, which is a domain class for representing textual figures in JHotDraw.

Based on the gained understanding, we have implemented the change request. As shown in Figure 8, we have done this by modifying the *export* method, so that the state of a drawing being exported is modified prior to saving. We added a simple watermark to the drawing's model (lines 359, 360) before it is written to the output file by *format.write(...)* and removed the watermark afterwards (line 362), so that insertion of a watermark is completely transparent to a user of SVG. The change implementation was supported by the *feature-aware source code editor* {p_1, a_3, $g_{2,3}$}, which enhances the standard NetBeans IDE's editor with fine-grained traceability from individual fragments of source code to features they implement. Figure 8 demonstrates the three feature-centric capabilities of the editor: color bars annotating source code with affinities, tooltips providing traceability from source code to concrete features and code-folding focusing the editor on code units relevant to a given feature. All these enhancements aim at narrowing the search space to the implementation of a feature of interest during feature-centric comprehension and modification of source code.

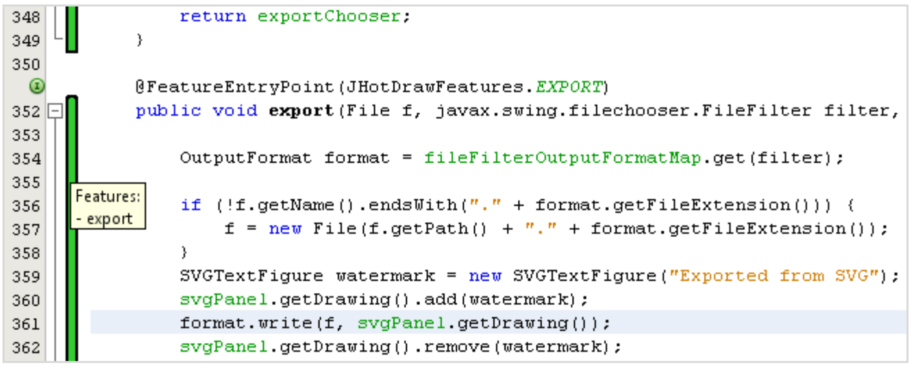

```
348             return exportChooser;
349         }
350
            @FeatureEntryPoint(JHotDrawFeatures.EXPORT)
352         public void export(File f, javax.swing.filechooser.FileFilter filter,
353
354             OutputFormat format = fileFilterOutputFormatMap.get(filter);
355
356             if (!f.getName().endsWith("." + format.getFileExtension())) {
357                 f = new File(f.getPath() + "." + format.getFileExtension());
358             }
359             SVGTextFigure watermark = new SVGTextFigure("Exported from SVG");
360             svgPanel.getDrawing().add(watermark);
361             format.write(f, svgPanel.getDrawing());
362             svgPanel.getDrawing().remove(watermark);
```

Figure 8. The performed modification in colored code editor.

After implementing the change, we needed to *validate* that our modification was correct and that it did not affect the correctness of other features in the program. Firstly, we verify that a watermark is added when exporting images by simply running the new version of the SVG program and then triggering the *export* feature again. Secondly, we investigated whether the correctness of some of the remaining features of SVG could have been affected by the change (e.g. the *drawing persistence* feature, which could in principle reuse the *export* method in some way). This was done by inspecting the affinity of the modified *export* method in the affinity-coloring facility of the feature-aware code editor. We have observed that even though four other features are reusing the method's enclosing *SVGView* class, the modified method is solely used by the *export* feature. This indicates that no other features could have been affected by the implemented change.

6. CONCLUSION

As mentioned previously, there exists a lack of isomorphic correspondence between the users' and the programmers' perception of object-oriented programs. This becomes problematic during software maintenance and evolution, as the implementations of the features users want to have changed are not evident from the object-oriented source code. Hence, there is an urgent need for tool-supported analysis approaches, which can help developers to understand the correspondence between features and code.

In this paper, we have presented Featureous a novel integrated development environment that provides a conceptual framework for implementing feature-centric analytical views of legacy object-oriented programs. Featureous is implemented as a plug-in to the NetBeans IDE. Featureous provides a lightweight mechanism for recovering the feature-code traceability links and APIs for implementing third-party extensions, like additional analytical views and affinity coloring schemes. Currently, Featureous comes with implementations of a number of state-of-the-art feature-centric views, three of which we have applied in our case study. The case study performed on JHotDraw SVG demonstrates how to use feature-centric analysis to support source code modification during software maintenance and evolution. It was our experience that the usage of feature-centric analysis during the presented modification task reduced the extent of required investigations of unfamiliar source code and allowed us to reason about the impact of the performed modification for the overall correctness of the program.

We hope that the integrated development environment that Featureous provides will help researchers to experiment with new ideas for feature-centric analysis of software. Motivated by our experiences reported in this paper, we believe that usage of feature-centric analysis tools such as Featureous can improve the performance of adopting functionality-related changes during software evolution. In a long perspective, we hope that improved understanding of feature-code relations may improve the practices of implementing features in object-oriented programs.

REFERENCES

Bennett, H.K. and Rajlich, V., 2000. Software maintenance and evolution: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, 73-87.

Brcina, R. and Riebisch, M., 2008. Architecting for evolvability by means of traceability and features. Automated Software Engineering - Workshops. ASE Workshops 2008. 23rd IEEE/ACM International Conference on. pp. 72-81.

Chen, K. and Rajlich, V., 2000. Case Study of Feature Location Using Dependence Graph. IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension. p. 241 IEEE Computer Society, Washington, DC, USA.

Cornelissen, B. et al, 2009. Trace Visualization for Program Comprehension: A Controlled Experiment. In: Marcus, A. and Koschke, R. (eds.) Proceedings of the 17th International Conference on Program Comprehension (ICPC'09). pp. 100–109 IEEE Computer Society, Washington, DC, USA.

Eisenberg, A.D. and De Volder, K., 2005. Dynamic Feature Traces: Finding Features in Unfamiliar Code. ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance. pp. 337–346 IEEE Computer Society, Washington, DC, USA.

Greevy, O. and Ducasse, S., 2005. Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach. CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering. pp. 314–323 IEEE Computer Society, Washington, DC, USA.

Greevy, O. et al, 2007. How Developers Develop Features. CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. pp. 265–274 IEEE Computer Society, Washington, DC, USA.

Greevy, O., 2007. Enriching Reverse Engineering with Feature Analysis. PhD thesis. University of Bern.

Kästner, C. et al, 2008. Granularity in Software Product Lines. Proceedings of the 30th International Conference on Software Engineering (ICSE). pp. 311–320 ACM, New York, NY, USA.

Kothari et al,, 2006. On Computing the Canonical Features of Software Systems, Proceedings of the 13th Working Conference on Reverse Engineering, pp. 93–102.

Mehta, A. and Heineman, G.T., 2002. Evolving legacy system features into fine-grained components. ICSE '02: Proceedings of the 24th International Conference on Software Engineering. pp. 417–427 ACM, New York, USA.

Murphy, G.C. et al, 2001. Separating Features in Source Code: An Exploratory Study. ICSE 2001: International Conference on Software Engineering, , p. 0275.

Olszak, A. and Jørgensen B.N., 2010, Remodularizing Java programs for improved locality of feature implementations in source code, Science of Computer Programming, In Press, Available online 6 November 2010, ISSN 0167-6423.

Robillard, M.P. and Murphy, G.C., 2002. Concern graphs: finding and describing concerns using structural program dependencies. ICSE '02: Proceedings of the 24th International Conference on Software Engineering. pp. 406–416 ACM, New York, NY, USA.

Röthlisberger, D. et al, 2007. Feature driven browsing. ICDL '07: Proceedings of the 2007 international conference on Dynamic languages. pp. 79–100 ACM, New York, NY, USA.

Ryder, B.G. and Tip, F., 2001. Change impact analysis for object-oriented programs. Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 46–53 ACM, New York, USA.

Salah, M. and Mancoridis, S., 2004. A Hierarchy of Dynamic Software Views: From Object-Interactions to Feature-Interactions. ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance. pp. 72–81 IEEE Computer Society, Washington, DC, USA.

Saliu, M.O. and Ruhe, G., 2007. Bi-objective release planning for evolving software systems. In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07). ACM, New York, NY, USA, 105-114.

Shaft, T. and Vessey, I., 2006. The Role of Cognitive Fit in the Relationship Between Software Comprehension and Modification. MIS Quarterly, 30(1). pp. 29-55.

Sillito, J. et al, 2006. Questions programmers ask during software evolution tasks. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14). ACM, New York, NY, USA, 23-34.

Tarr, P. et al, 1999. N degrees of separation: multi-dimensional separation of concerns. ICSE '99: Proceedings of the 21st international conference on Software engineering. New York, NY, USA: ACM, pp. 107-119.

Turner, C.R. et al, 1999. A conceptual basis for feature engineering. In *Journal of Systems and Soft.*, vol. 49, pp. 3–15.

Wilde, N. and Scully, M.C. 1995. Software reconnaissance: mapping program features to code, Journal of Software Maintenance, vol. 7, pp. 49–62.

Wong, W.E. et al, 2000. Quantifying the closeness between program components and features, J. Syst. Softw., vol. 54, pp. 87–98.