FEATUREOUS: AN INTEGRATED APPROACH TO LOCATION, ANALYSIS AND MODULARIZATION OF FEATURES IN JAVA APPLICATIONS

PH.D. THESIS IN SOFTWARE ENGINEERING

BY

ANDRZEJ OLSZAK

SEPTEMBER 3RD, 2012

UNIVERSITY OF SOUTHERN DENMARK

ABSTRACT

To remain useful for their users, software systems need to continuously enhance and extend their functionality. Thereby, the individual units of software functionality, also known as *features*, become the units of source code comprehension, modification and work division. To support these activities, it is essential that features are properly modularized within the structural organization of software systems.

Nevertheless, in many object-oriented applications, features are not represented explicitly. Consequently, features typically end up *scattered* and *tangled* over multiple source code units, such as architectural layers, packages and classes. This lack of modularization is known to make application features difficult to locate, to comprehend and to modify in isolation from one another.

To overcome these problems, this thesis proposes Featureous, a novel approach to location, analysis and modularization of features in Java applications. Featureous addresses these concerns in four steps. Firstly, a dynamic method based on so-called feature-entry points is used to establish traceability links between features and their corresponding source code units. Secondly, this traceability forms a basis for a feature-oriented analysis method that visualizes and measures features. Thirdly, scattering and tangling of features can be physically reduced using feature-oriented remodularization method that is based on multi-objective optimization of Java package structures. Finally, automated detection of so-called feature seed methods is proposed, to enable large-scale feature-oriented quality assessment.

Featureous is implemented as a plugin to the NetBeans IDE. This implementation was used to evaluate the constituent methods of Featureous by applying them to several medium and large open-source Java systems. The gathered quantitative and qualitative results suggest that Featureous succeeds at efficiently locating features in unfamiliar codebases, at aiding feature-oriented comprehension and modification, and at improving modularization of features using Java packages.

RESUME

Denne afhandling foreslår Featureous, en ny tilgang til lokation, analyse og modularisering af funktionelle krav (så kaldte features) i Java-programmer. Featureous adresser spredning og sammenfiltring af features i kildekoden af objekt-orienterede applikationer på forskellige måder. For det første er en dynamisk metode baseret på såkaldte feature-indgange, som bruges til at etablere sporbarhed mellem funktionelle krav og deres tilsvarende kildekode enheder. For det andet bruger Featureous sporbarheden til en feature-orienteret analyse, der visualiserer og måler funktionelle krav. For det tredje kan spredning og sammenfiltring af features blive fysisk reduceret ved hjælp af feature-orienteret remodularization, der er baseret på multi-objektiv optimering af Java pakke strukturer. Endelig foreslås en automatisk metode til at detektere såkaldte feature seed metoder. Featureous er implementeret som et plugin i NetBeans IDE og er valideret i form af anvendelse i flere mellemstore og store open-source Java-systemer.

PREFACE

Human beings can tackle the most complex problems by approaching them one aspect at a time. By carefully decomposing problems into smaller and simpler parts, we are able to gradually understand and solve what initially seems beyond our capabilities. Whether it is a computer program to be developed, a building to be constructed or a language to be learned, humans intuitively seek to substitute the *complexity of the complete* with the *simplicity of the partial*.

There, however, exists no single and universal recipe for decomposing problems into smaller and simpler parts. Quite the contrary, most problems can be decomposed in several alternative ways – not all of which will be equally beneficial. For instance, a script of a theatrical play can be decomposed either along the boundaries of individual scenes, or along the boundaries of individual roles. The first decomposition would be beneficial for a stage designer, who considers each scene an independent unit of work, regardless of the particular characters acting. Hence, a scene designer does not have to know anything about the roles or even about the overall plot of the play, as long as he or she can locate descriptions of the sceneries and their order of appearance. On the other hand, decomposing the script to separate the roles would benefit the actors, who are interested in learning their own roles and in identifying their direct interactions with other characters of the play.

While enumerating several alternative decompositions of a single problem is an uncomplicated task, choosing the best one often proves difficult. This is well exemplified by the history of the field of manufacturing, where it was not until the era of Henry Ford that the best problem decomposition was identified. There, the traditional approach used to be to assign a single worker to a single item being produced, so that the worker was responsible for carrying out a sequence of manufacturing steps related to his or her item. This traditional decomposition of work was eventually found inefficient, and it was replaced with the assembly line approach. The proposed change was a structural one; the assembly line approach postulated to decompose work around the individual manufacturing steps, rather than around individual items. Hence, instead of handling a single item through several diverse processing steps, it was postulated that each worker focuses on applying a single specialized manufacturing step to a series of incoming items. As it turned out over time, this alternative decomposition of work revolutionized manufacturing.

Fortunately for manufacturing, the principle of assembly lines is as valid today as it was a hundred years ago. This is because its physical *medium*, i.e. the items, parts and tools, is only involved in one type of *interaction* – the manufacturing process. However, there also exist problems in which the physical medium is involved into several different types of interactions. In the mentioned example of a script of a theatrical play, the same medium (the script) is used in at least two different interactions (stage design and acting). These different interactions require

fundamentally different decompositions of the underlying medium – each dedicated to supporting its specific interaction.

Given this observation, several questions arise. What are the important types of interactions with a given medium? When do they occur? Which decompositions of the medium support them? How to represent and manage multiple alternative decompositions? How to flexibly transform the medium from one decomposition to another, when a particular type of interaction is required?

This thesis focuses on exactly these topics. The concrete medium of focus is the *source code of software applications*, and the concrete interactions of focus are the *evolutionary changes performed by software developers*.

ACKNOWLEDGEMENTS

It is my pleasure to thank people who made this thesis possible.

First of all, I would like to thank my supervisor Associate Prof. Bo Nørregaard Jørgensen for his indispensable guidance, continuous support and securing the funding for conducting and publishing this work.

I would like to thank my colleagues Hans Martin Mærsk-Møller, Martin Lykke Rytter Jensen and Jan Corfixen Sørensen for numerous stimulating discussions and for sharing the joys and sorrows of the Ph.D. life.

My thanks go to Eric Bouwers and Joost Visser from the Research Department of Software Improvement Group for making my research stay in Amsterdam a great experience.

I thank Geertjan Wielenga and Jaroslav Tulach from the NetBeans Team at Oracle for their support and for our joint JavaOne talks.

Last but not least, I would like thank my girlfriend Karolina Szymbor, my friends Piotr Zgorzalek, Bartosz Krasniewski, and my parents Marianna Olszak and Leszek Olszak for being supportive, and tolerant in the times of intense occupation with this thesis.

Andrzej Olszak
Odense, September 2012

PUBLICATIONS

Main Publications

[TOOLS'12] Olszak, A., Bouwers, E., Jørgensen, B. N. and Visser J. (2012). Detection of Seed Methods for Quantification of Feature Confinement. In TOOLS'12: Proceeding *of the TOOLS Europe: Objects, Models, Components, Patterns*, Springer LNCS, Vol. 7304, pp 252-268.

[CSMR'12] Olszak, A. and Jørgensen, B. N. (2012). Modularization of Legacy Features by Relocation and Reconceptualization: How Much is Enough? In CSMR'12: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press, pp. 171-180.

[SCICO'12] Olszak, A. and Jørgensen, B. N. (2012). Remodularizing Java Programs for Improved Locality of Feature Implementations in Source Code. *Science of Computer Programming*, Elsevier, Vol. 77, no. 3, pp. 131-151.

[CSIS'11] Olszak, A. and Jørgensen, B.N. (2011). Featureous: An Integrated Environment for Feature-centric Analysis and Modification of Object-oriented Software. *International Journal on Computer Science and Information Systems*, Vol. 6, No. 1, pp. 58-75.

[SEA'10] Olszak, A. and Jørgensen, B. N. (2010). A unified approach to feature-centric analysis of object-oriented software. In SEA'10: *Proceedings of the IASTED International Conference Software Engineering and Applications*, ACTA Press, pp. 494-503.

[AC'10] Olszak, A. and Jørgensen, B. N. (2010). Featureous: Infrastructure for feature-centric analysis of object-oriented software. In AC'10: P*roceedings of IADIS International Conference Applied Computing*, pp. 19-26.

[FOSD'09] Olszak, A. and Jørgensen, B. N. (2009). Remodularizing Java programs for comprehension of features. In FOSD'09: *Proceedings of the First International Workshop on Feature-Oriented Software Development*, ACM, pp. 19-26.

Short Papers and Tool Demo Papers

[WCRE'11a] Olszak, A., Jensen, M. L. R. and Jørgensen, B. N. (2011). Meta-Level Runtime Feature Awareness for Java. In WCRE'11: *Proceedings of the 18th Working Conference on Reverse Engineering*, IEEE Computer Society Press, pp. 271-274.

[WCRE'11b] Olszak, A. and Jørgensen, B. N. (2011). Understanding Legacy Features with Featureous. In WCRE'11: *Proceedings of the 18th Working Conference on Reverse Engineering*, IEEE Computer Society Press, pp. 435-436.

[ICPC'10] Olszak, A. and Jørgensen, B. N. (2010). Featureous: A Tool for Feature-Centric Analysis of Java Software. In ICPC'10: *Proceedings of IEEE 18th International Conference on Program Comprehension*, IEEE Computer Society Press, pp. 44-45.

GLOSSARY

Application – software designed to fulfill specific needs of a user; for example, software for navigation, payroll or process control [1]

Cohesion – the manner and degree to which tasks performed by a single module relate to one another [1]

Concern – an aspect of interest or focus in software [2], [3]

Control flow – the sequence in which operations are performed during the execution of software [1]

Coupling – The manner and degree of interdependence between software modules [1]

Dynamic analysis – the process of evaluating a system or component based on its behavior during execution [1]

Feature – a unit of user-identifiable software functionality [4]; an instance of *concern*; it can be further divided into three concepts: feature name, *feature specification* and *feature implementation*

Feature implementation – source code units implementing a given user-identifiable functionality

Feature specification – textual or diagrammatic description of a user-identifiable functionality of software

Fragment [of a *source code unit*] – a semantically-related part of a source code unit, often separated using a mechanism for advanced separation of concerns, i.e. aspects, mixins; also known as refinement, increment, derivative

Functional requirement – a requirement that specified a function that a system or system component must be able to perform [1]

Modularity – the degree to which software is composed of discrete components so that a change to one component has minimal impact on other components [1]

Modularization (as noun) – A concrete decomposition of software into modules; also known as modular decomposition [5]

Modularization (as verb) – the process of breaking software into components to facilitate design and development; also known as modular decomposition [1]

Module – a logically separable part of a software system [1]

Object-oriented design – a software development technique in which a system or component is expressed in terms of objects and connections between those objects [1]

Pareto optimality – a property of a solution to multi-objective optimization, where no objectives can be improved without degrading any other objectives

Refactoring – a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [6]; an instance of *restructuring*

Remodularization – the process of changing how software's source code is decomposed into modules [2]; an instance of *restructuring*

Restructuring – a transformation of source code from on representation form to another at the same relative abstraction level, while preserving the software's external behavior [7]

Software comprehension – a set of activities directed at understanding software's source code; also known as software understanding

Software evolution – the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment; also known as software maintenance [8], [1].

Source code unit – a logically separable syntactical part of software, such as package, class or method [1]

CONTENTS

1. INTRODUCTION

This chapter presents the motivation and states the research problem of behind the Featureous approach. Furthermore, this chapter discusses the used research method and outlines the contributions and the organization of the thesis.

1.1 MOTIVATION

E-type software systems, also known as *applications*, are embedded in the real world, and therefore they need to change together with it to remain useful to their users [8], [9]. In particular, existing applications need to change in response to evolving operational conditions and ever-expanding requirements. This process is known as *software evolution* or *software maintenance* [8]. According to several independent studies, the overall cost of the software evolution can amount to anything between 60% and 90% of the total software costs in organizations [10], [11], [12], [13].

The Lehman's sixth law of software evolution, known as the law of *continuing growth* [8], postulates that an important class of evolutionary changes are the changes concerned with the *functional content* of software. During such changes, the individual units of user-identifiable functional content, which are referred to as *features* [4], are added, enhanced or corrected according to the requests formulated by the users. In order to address such feature-oriented requests of the users, developers have to inspect and evolve the software's source code in a feature-oriented fashion [4], [14].

Evolving software in a feature-oriented fashion is non-trivial, as it requires deep understanding of two drastically different perspectives on a single application [15], [4]. Firstly, one has to understand the application's *problem domain* and the feature specifications it encompasses. Secondly, one has to understand how the feature specifications are represented in the application's *solution domain*, i.e. how they are implemented in the application's source code. Understanding these two domains, as well as the correspondences between them, is a prerequisite to performing functional changes. Thereby, the feature-oriented perspective used by the users in their change requests enforces a feature-oriented perspective on developers during the understanding and modification of source code.

Maximizing the effort-efficiency of feature-oriented changes is of crucial importance. According to several authors, changes concerned with extending, enhancing and correcting application features can amount to 75% of overall cost during software evolution [16], [17], [18]. Interestingly, several works have indicated that more than half of these efforts are spent on identifying and understanding the units of source code prior to their actual modification [19], [20].

The effort required for incorporating a feature-oriented change, and any type of change in general, is determined by the degree to which the change is confined within a small number of logically separable units of source code, also known as *modules* [5]. According to Parnas [5], an appropriate division of source code into modules grants three vital evolutionary qualities:

- "Comprehensibility: it should be possible to study a system one module at a time"
- "Product flexibility: it should be possible to make drastic changes to one module without a need to change others"
- "Managerial: development time should be shortened because separate groups would work on each module with little need for communication"

Hence, a proper modularization of features is a prerequisite to modular comprehension, modification and work division during feature-oriented evolutionary changes. Such a "proper" modularization requires that features are explicitly represented, localized and separated from one another in the module-granularity units of source code such as packages or namespaces.

1.2 RESEARCH PROBLEM

Unfortunately, it is rare that features are properly modularized in packages or namespaces of object-oriented applications.

The boundaries of individual features tend to remain implicit in the structure of object-oriented source code. This is because the object-oriented style of modularizing applications does not consider features as the units of decomposition. Instead, object-oriented applications tend to be structured into layers that aim at modularizing purely technical concerns such as model, view, controller or persistence [21], [22], [23].

While layered separation of technical concerns has its advantages during particular types of evolutionary changes (e.g. adding a caching mechanism for database access is best achieved if the persistence concern is represented by a single self-contained module), it is certainly suboptimal during feature-oriented evolutionary modifications.

Implementing the specification of a user-identifiable feature in source code inherently requires of a mixture of technically diverse classes. In particular, each non-trivial feature encompasses some forms of (1) interfacing classes, which allow a user to activate the feature and see the results of its execution; (2) logic and domain model classes, which contain the essential processing algorithms, and (3) persistence classes, which allow for storing and loading the results of the computation. Hence, features can be viewed as implicit vertical divisions that crosscut the horizontally-layered object-oriented designs. These implicit divisions consist of graphs of collaborating classes that end up *scattered* and *tangled* within individual layers [4], [24]. This situation is schematically depicted in Figure 1.1.

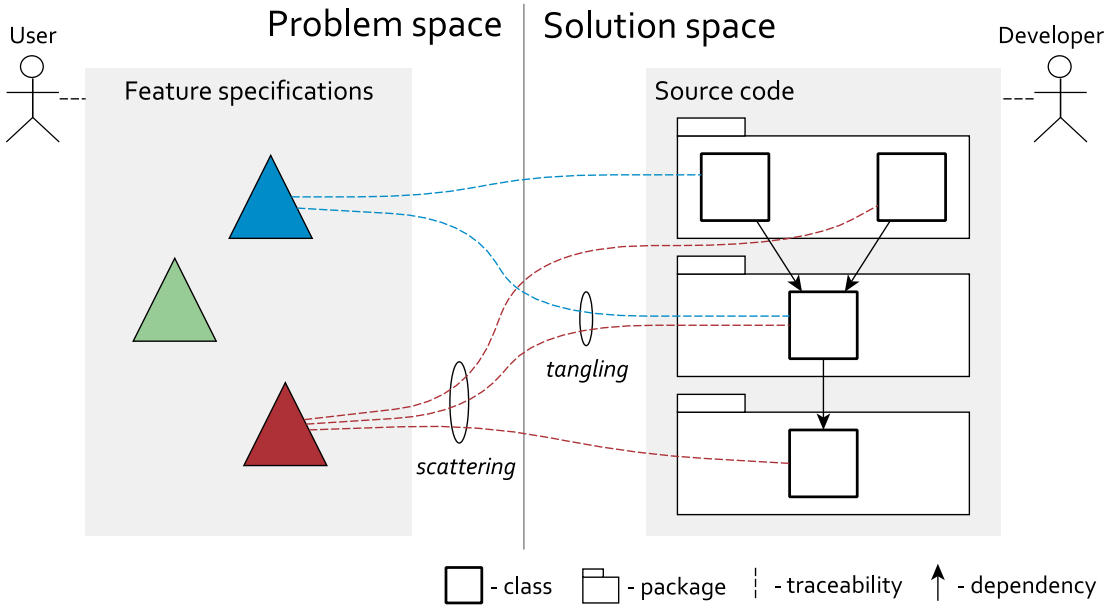


Figure 1.1: Relations between feature specifications and units of source code

The two fundamental properties of the relations between feature specifications and units of source code are defined as follows:

- *Scattering* denotes the delocalization of a feature over several source code units of an application. Hence, scattering describes the situations in which a single

feature is implemented by several units of source code, such as classes or packages [2], [24].

- *Tangling* denotes the overlap of several features on a single source code unit. Hence, tangling describes the situations, in which a single unit of code, such as class or package, contributes to several features [2], [24].

The implicitness and the complexity of the relations between feature specifications and source code units affect software evolution in several ways:

- The implicit boundaries of features and their misalignment with the structural source code units force developers to establish and maintain intricate mental mappings between features and source code [4]. The implicit character of these mappings is recognized as a significant source code comprehension overhead during software evolution [14], [25].

- Scattering of features corresponds to the phenomenon of *delocalized plans*, which occurs when a programming plan (such as a feature) is realized by source code units delocalized over several modules an application's source code [26]. This is known to make it difficult to identify the relevant source code units during change tasks [27], [26].

- Tangling of features hinders comprehension by the means of the *interleaving* phenomenon, which occurs when modules are responsible for accomplishing more than one purpose (such as a feature) [28]. This is known to make it difficult to understand how multiple features relate to each other with respect to code reuse [29].

Apart from software comprehension, the representation gap between features and modules makes it more difficult to modify source code. Due to scattering, a modification of one feature may require understanding and modification of several seemingly unrelated modules. Due to tangling, a modification intended to affect only one feature may cause accidental change propagation to other features that also use a module being modified. These issues hinder the usage of individual features as the units of program comprehension, work division and code modification [30], [31].

Overall, the presented evidence suggests that overcoming the misalignment between features and modules of object-oriented applications should lead to a significant reduction of the costs of software evolution and to shortening the feedback loops between software developers and software users. Developing practical means of overcoming this misalignment is the aim of the research presented in this thesis. This aim is formulated as the following research question:

Research Question

How can features of Java applications be located, analyzed and modularized to support comprehension and modification during software evolution?

By posing this research question, this thesis strives to develop a practical approach to reducing the negative effects of implicitness, scattering and tangling of features in existing Java applications. Although this thesis uses Java as a concrete example of a popular object-oriented programming language, the presented research is not conceptually restricted to only this language.

1.3 RESEARCH METHOD

The research method used in this thesis follows the guidelines of the *constructive research approach* [32]. The aim of constructive research is to produce *innovative constructions* that are intended to solve *problems* faced in the real world, and thereby to contribute to the *theory* of the application discipline.

Based on this, the following methodological steps are applied to address the research question of this thesis:

1. Study existing literature to identify and understand the main challenges involved in the research problem.
2. For each of the individual challenges of the research problem:
    a. Develop new theories and methods for addressing the research challenge.
    b. Develop tools that allow for a practical and reproducible application of the theories and methods to existing Java applications.
    c. Use applicative studies to evaluate the theories and methods by applying the tools to existing Java applications in a structured manner.
    d. Analyze the generality of the obtained results and their contribution.
3. Based on the results, conclude about the theoretical and practical consequences of the approach as a whole. Critically assess the limitations of the approach, and identify open questions and opportunities for further research.

1.4 CONTRIBUTIONS

The research presented in this thesis and in its related publications resulted in a number of contributions to the state-of-the-art of feature location, feature-oriented analysis and feature-oriented remodularization.

At the conceptual level, this thesis has led to the following contributions:

- Defining the notion of feature-entry points and using it as a basis for dynamic feature location [FOSD'09], [SCICO'12].
- Defining a conceptual framework for feature-oriented analysis and using it as a basis for developing a unified method for feature-oriented analysis of Java applications [SEA'10], [AC'10], [CSIS'11].
- Developing a method for manual analysis-driven modularization of features in existing Java applications [CSMR'12].
- Developing a method for automated feature-oriented remodularization of Java applications based on metric-driven multi-objective optimization [SCICO'12], [CSMR'12].
- Evaluating the role of class reconceptualization in feature-oriented remodularization [CSMR'12].
- Developing a heuristic for automated detection of seed methods of features [TOOLS'12].

At the technical level, this thesis resulted in a number of practical tools:

- Featureous Location library – a lightweight annotation-driven approach to tracing execution of features in Java applications [FOSD'09], [SCICO'12].
- Featureous Workbench – an extensible tool for feature-oriented analysis and modification of Java applications build on top of the NetBeans Rich Client Platform [AC'10], [ICPC'10], [WCRE'11b].
- JAwareness – a library for enabling meta-level the awareness of feature execution in Java applications [WCRE'11a].

1.5 THESIS ORGANIZATION

Chapter 2 presents the background literature.

Chapter 3 provides a general overview of the approach.

Chapter 4 develops and evaluates the feature location part of the approach. The chapter is based on the following publications: [SCICO'12], [WCRE'11a], [FOSD'09].

Chapter 5 develops and evaluates the feature-oriented analysis part of the approach. The chapter is based on the following publications: [CSIS'11], [SEA'10], [AC'10].

Chapter 6 develops and evaluates the feature-oriented manual remodularization part of the approach. The chapter is based on the following publications: [CSMR'12], [SCICO'12].

Chapter 7 develops and evaluates the feature-oriented automated remodularization part of the approach. This chapter is based on the following publications: [CSMR'12], [SCICO'12], [FOSD'09].

Chapter 8 develops and evaluates a heuristic for automated detection of seed methods that is aimed at scaling feature-oriented measurement. The chapter is based on the following publication: [TOOLS'12].

Chapter 9 discusses the limitations, open questions and future perspectives on the presented research.

Finally, Chapter 10 concludes the presented research.

2. BACKGROUND

This chapter presents existing literature that this thesis builds upon. The presented works deal with the topics of software modularity, the role of features as unit of modularity, feature location, feature-oriented analysis and feature-oriented remodularization of software.

2.1 MODULARITY OF SOFTWARE

Parnas [5] was one of the firsts to point out that decomposition of software's source code into modules is not merely a means of enabling its separate compilation, but most importantly is a means of shaping its evolutionary properties. Parnas demonstrated that using different decomposition criteria yields radically different changeability characteristics. According to Parnas, a proper modularization of source code should allow one to comprehend, modify and work with source code one module at a time. Since then, the conclusions of Parnas found empirical support in multiple published works.

Benestad et al. [29] conducted a large-scale experiment to investigate the cost drivers of software evolution. They concluded that majority of the effort during evolutionary changes is caused by dispersion of changed code among modules. Dispersed changes were found to make it difficult to comprehend source code and to anticipate the side effects of changes.

Similar conclusions about the negative impact of delocalization of change units were reached in the experiment of Dunsmore et al. [33] who investigated the role of delocalization on defect correction performance.

The organizational importance of modularization is discussed by Brooks [34], who observed that scaling software development teams is difficult because of the exponentially growing need for communication. This need was argued to arise when developers work on heavily interdependent modules. In the same spirit, other researchers pointed out that proper modularization is expected to increase the potential number of collaborators working in parallel [35] and to facilitate autonomous innovation [36] and trial-and-error experimentation within individual modules [37], [38].

2.1.1 Modularization Criteria

Parnas [5] postulated that it is beneficial to design individual modules to encapsulate a system's design decisions that are likely change in the future. This qualitative modularization criterion, known as *information hiding*, was demonstrated to support several types of evolutionary changes in the Key Words In Context (KWIC) system. Thereby, Parnas demonstrated relativity of modularization to the concrete types of changes that a software system has to undergo.

As later observed by Garlan et al. [39], the criterion of information hiding facilitates diverse changes of data representation, but not the changes in a system's functionality. To support also this type of changes, Garlan et al. proposed a decomposition based on the *tool abstraction*. This abstraction is centered around a set of cooperating "toolies" which operate on a shared set of abstract data structures. The individual "toolies" implement individual units of a system's functionality. This decomposition was demonstrated to surpass information hiding in supporting functional changes, but at the same time to have limited support for changes of data representation.

In his other work, Parnas [40] focused on the challenges involved in functional extension and reduction of software systems. This was done by proposing the notion of layered division of responsibilities in software systems based on an acyclic graph of *uses* relations among modules. This approach was applied to an address-processing system, and resulted in a system consisting of two levels of single-purpose routines, where the upper level routines were implemented in terms of the lower level ones. This design was concluded to support functional reduction and extensibility by making it possible to flexibly remove and replace the upper-level routines.

Stevens et al. [41] proposed that *modules should possess the properties of low coupling and high cohesion*. In other words, they postulated that modules should be independent of each other, so that the change propagation among them is minimized; whereas the contents of individual modules should be strongly interconnected, so that

they serve as good units of comprehension and modification. This objective formulation allowed the use of static analysis to perform automated quantification of modularization quality of software systems. Consequently, it led to a number of cohesion and coupling metrics that were proposed over the years by various authors.

A set of qualitative principles dealing with object-oriented class design and package structure design was proposed by Martin [42]. The core of his proposal revolves around the principles of (1) *single responsibility* that states that a class should have only one reason to change, (2) the *common closure principle* that states that classes that change together should be grouped together, (3) the *common reuse principle* that states that classes reused together should be grouped together and (4) the *stable dependency* that states that dependencies in source code should take into account stability of code units.

Dijkstra [43] introduced the notion of *separation of concerns*. This notion expresses the need for focusing one's attention upon a single selected aspect of a system at a time. Focusing on a single viewpoint was argued to facilitate incremental understanding of complex systems. This work of Dijkstra, while not defining any practical means to achieve separation of concerns, is regarded as pioneering in recognizing concerns as first-class units of software.

Kiczales et al. [3] observed that there exist some concerns that neither procedural, nor object-oriented programming techniques are able to represent in a modular fashion. Such concerns, known as *crosscutting concerns*, or *aspects*, were observed to often experience a lack of proper modularization, which was demonstrated to hinder software development and evolution. To address this problem, Kiczales et al. proposed AspectJ – a language-based extension to object-oriented paradigm that supports modularization of the crosscutting concerns in terms of aspects.

Ossher and Tarr [2] proposed a generalized vision of software systems as multi-dimensional concern spaces. These concern spaces were postulated to consist of multiple orthogonal dimensions that encompass different concern types, such as features, caching, distribution, security, etc. Ossher and Tarr emphasize that different dimensions of concern are affected by different evolutionary changes, and therefore the optimality of a given modularization depends on concrete types of changes that a system will undergo. They postulated that modern programming languages support only one dimension of decomposition at a time, causing all other dimensions to be scattered and tangled. This problem was termed as *the tyranny of the dominant decomposition*. Ossher and Tarr postulate that ideally it should be possible to switch between the dominant dimensions of concern of source code decomposition in an on-demand fashion, according to the type of a particular task at hand. To this end, they proposed the notion of *on-demand remodularization* [44] and attempted to realize it by developing a language-based approach called Hyper/J.

2.1.2 Features as Units of Modularity

Despite the diversity of possible change types, software systems are known to follow a set of general high-level trends known as *the laws of software evolution* [8]. Primarily, the laws of software evolution state that software applications embedded in the real world, the so-called E-type systems [9], are bound to change over time, because they have to respond to changes in their problem domains and their requirements. In particular, according to the sixth law of software evolution, applications experience user-driven continuing growth of *functional content* [8]. In the context of the work of Ossher and Tarr [2], this suggests a particular importance of the *feature* dimension of concern.

Turner et al. [4] observed that the units of user-identifiable software functionality, which are referred to as *features*, bridge the gap between the problem and solution domains of software systems. Thereby, the notion of features facilitates the communication between the users, who are concerned with the behavior of software, and software developers, who are concerned with the software's lifecycle artifacts, such as source code and design specification. Furthermore, Turner et al. claim that explicitly representing features in design and implementation of software systems is of crucial importance for component-based systems, where each component should contribute a chunk of overall functionality. Based on this, Turner et al. develop a comprehensive conceptual framework for feature-oriented engineering of evolving of software.

Roehm et al. [45] reported an empirical study of strategies used by professional software developers to comprehend software. Among other findings, the study suggested usefulness of three feature-oriented comprehension strategies. These three strategies are: (1) *interacting with the UI to test expected program behavior and find starting points for further inspection*, (2) *using debuggers to acquire runtime information* and (3) *investigating the way in which the end-users use an application*. Based on these observations it was concluded that developers tend to put themselves into the role of end-users. Namely, the they try to understand program behavior as the first step of program exploration.

Greevy et al. [14] investigated how software developers modify features during software maintenance. The results indicate that in object-oriented systems developers divide work along the structural boundaries of modules at the initial stages of development, whereas they tend to divide the work along boundaries of features during software evolution and maintenance.

Eaddy et al. [27] evaluated the role of feature scattering on the amount of defects in the source code of three small and medium-sized open-source Java applications. The presented empirical results demonstrate that the more scattered a feature is, the more likely it is to have defects. This effect was observed to be independent of the size of

features. The authors interpret the reported results as caused by the comprehension overhead that is associated with the physical delocalization of features.

2.2 FEATURE LOCATION

Feature location is the process of identifying units of source code that contribute to implementing features in a software application [46], [47]. The main two types of approaches to feature location are the approaches based on static and dynamic analyses.

A static approach to feature location based on manual inspection of application's dependence graphs was proposed by Chen and Rajlich [48]. Their approach relies on an expert's judgment about the relevance of individual units of source code to concrete features. Discovery of complete feature-code mappings is performed interactively by participating in a computer-driven search process based on static analysis of source code. The discovered features can then be marked in source code using several methods. One of them was proposed by Kästner et al. [49], who developed the CIDE tool that allows for fine-grained manual marking of features in source code using colors in the code editor of the Eclipse IDE.

Dynamic analysis was exploited in the *software reconnaissance* method proposed by Wilde and Scully [50]. Software reconnaissance associates features with control flow of applications and proposes to trace it at runtime, while the features are being triggered by dedicated test suites. Such test suites are designed to encode appropriate scenarios in which some features are exercised while others are not. By analyzing this information, the approach identifies units of source code that are used exclusively by single features of an application. Thereby, the approach of Wilde and Scully reduces the need for manual inspection of source code.

Similarly, Salah and Mancoridis [51] proposed an approach called *marked traces* that traces execution of features. However, they replace the feature-triggering test suites with user-driven triggering during a standard usage of an application. *Marked traces* require users to explicitly enable runtime tracing before they activate a given feature of interest, and to explicitly disable tracing after the feature finishes executing. This mode of operation reduces the involved manual work, as compared to the approaches of Wilde and Scully [50], as it eliminates the need for implementing dedicated feature-triggering test suites. Furthermore, the approach of Salah and Mancoridis identifies not only the units of code contributing to single features, but also the ones shared among multiple features.

As demonstrated by Greevy and Ducasse [52], marked traces can also be captured by using a GUI automation script instead of the actual users. The automation scripts used by their tool TraceScraper encode the actions needed for activating and tracing

individual features of an application. While preparing automation scripts resembles preparing dedicated test suites of Wilde and Scully, the major advantage of the automation scripts their independence of an application's implementations details.

Apart from static and dynamic feature location approaches, there exist hybrid approaches that combine different location mechanisms and additional information sources. Examples of such approaches include enhancing dynamic analysis with concept analysis and static analysis [53], information retrieval and prune dependency analysis [54], web mining methods [55], change sequences from code repositories and issue tracking tickets [56]. Comparative analyses of these diverse information sources performed by Ratanotayanon et al. [56] and Revelle et al. [55] reveal that the choice of particular mechanisms and information sources influences the precision and recall of feature location.

2.3 FEATURE-ORIENTED ANALYSIS

Feature-oriented analysis, known also as feature-centric analysis, considers features as first-class entities of investigating source code of existing applications [52], [51], [57], [4]. The research efforts on feature-oriented analysis are traditionally focused on the topics of feature-oriented measurement and visualization of source code.

2.3.1 Feature-Oriented Measurement

Brcina and Riebisch [58] propose three metrics for assessing the modularization of features in architectural designs. The first one, *scattering indicator* is designed to quantify the delocalization of features over architectural components of a system. The second metric, *tangling indicator* captures the degree of reuse of architectural components among multiple features. The third one, *cross-usage indicator* quantifies static dependencies among features. Additionally, Brcina and Riebisch provide a list of resolution actions that can be applied to address the problems detected by the proposed metrics.

Ratiu et al. [59] proposed to quantify the quality of how software problem domains are represented in software source code. To address this issue, they proposed several metrics for assessing the *logical modularity* of applications. This correspondence is proposed to be characterized using the degree of *delocalization* and *interleaving* of a dimension of domain knowledge (such a features) in code units (such as packages or classes), *spanning* of a knowledge dimension in a package, *misplaced classes* and the *dominant knowledge dimension* of packages.

Wong et al. [60] defined three metrics for quantifying *closeness between program components and features*. These metrics capture the *disparity* between a program

component and a feature, the *concentration* of a feature in a program component, and the *dedication* of a program component to a feature. To support practical application of their metrics, Ratiu et al. propose a dynamic-analysis approach for establishing traceability links between features and source code using an execution slice-based technique that identifies regions of source code invoked when a particular feature-triggering parameter is supplied.

Eaddy et al. [27] proposed and validated a metrics suite for quantifying the degree to which concerns (such as features) are scattered across components and separated within a component. The defined metrics include *concentration* of a concern in a component, *degree of scattering* of a concern over components, *dedication* of a component to a concern and *degree of focus* of a component. Furthermore, Eaddy et al. provide a set of guidelines for manually identifying concerns in source code.

Sant'Anna et al. [61] proposed a metrics suite for measuring concern-oriented modularity and a technique for documenting concerns in software architectures. The suite consists of the following metrics: *concern diffusion* that quantifies scattering of concerns, *coupling between concerns* that quantifies dependencies among classes implementing concerns, *coupling between components* that quantifies dependencies among components, *component cohesion* that quantifies the tangling of components and *interface complexity* that quantifies the size of components interface. This suite was evaluated on using three case studies.

2.3.2 Feature-Oriented Visualization

There exist several approaches to using visualization to improve comprehension of execution traces and features in source code. The often demonstrated benefits of such approaches are helping software developers to understand where and how a feature is implemented and how it relates to other features in an application.

Mäder and Egyed [62] evaluated the performance of code navigation that uses traceability links during evolutionary changes. In a controlled experiment, they visualized runtime traces using a simple color-based overlay to mark trace-related source files in a file browser window and trace-related methods in a code editor. The results indicate a significant positive effect of traceability on both performance and quality of adoption of evolutionary changes in unfamiliar source code. Moreover, traceability was observed to enhance how the participating developers navigated source code and identified parts that needed to be modified.

Cornelissen et al. [30] evaluated EXTRAVIS, an Eclipse-based tool for visualization of large execution traces. EXTRAVIS offers two views over traces: the *massive sequence view* corresponding to a large-scale UML sequence diagram [63] and the *circular bundle view* that hierarchically projects the software's structural entities and their bundled relations onto a circle. In a controlled experiment involving several evolutionary tasks, Cornelissen et al. quantified the effect of their trace visualization

on software comprehension. The results show a significant decrease of the time needed to finish the comprehension tasks and an increase in the correctness of the gained source code understanding.

Eisenbarth et al. [53] proposed an approach to incremental exploration of features in unfamiliar source code using the techniques of formal concept analysis. The proposed approach uses a mixture of dynamic and static analyses to identify sets of single-feature and multi-feature source code units. This information forms a basis for forming concept lattices (with source code units as *formal objects*, and features as *formal attributes*) that represent traceability links between features and source code. The concept lattices are visualized as directed acyclic graphs. The nodes represent individual formal concepts (which are tuples of related objects and attributes). The edges represent formal relations among concepts. Lattices are demonstrated to support incremental analysis, by allowing to starting with a minimal set of concepts and then incrementally expanding it.

Salah and Mancoridis [51] presented a hierarchy of dynamic views over software execution traces. They define grid and graph-based *feature-interaction* views that depict runtime relations among features. The defined relations include object reuse and producer-consumer dependencies among features. Furthermore, Salah and Mancoridis introduce the *feature-implementation view* that uses graph structure to visualize traceability links between features and code units of an application. Case studies involving two Java systems are used to demonstrate the usage of the proposed views to support software comprehension.

Greevy and Ducasse [52] define a two-sided approach based on summarized traces for analyzing traceability links between features and units of source code. The first proposed perspective is the *feature perspective* that defines two views: *feature characterization* that depicts the distribution of classes in features and the *feature class correlation* that precise correlates classes and features. In contrast, the *code perspectiv*e defines the *class characterization* view that depicts reused of classes by features. By applying their two-sided approach in two case studies, Greevy and Ducasse demonstrated practicality of their views during software comprehension.

Kothari et al. [64] proposed an approach to optimizing feature-oriented analysis of software by choosing only a particular subset of an application's features to focus on. They use three software systems to demonstrate that the implementations of semantically similar features tend to converge over subsequent releases of applications. Thus, selecting and investigating only a few key features is proposed to give a representative overview of all the remaining features. Kothari et al. formulate the identification of such representative features, called *canonical features*, as a search problem.

2.4 SOFTWARE REMODULARIZATION

Software remodularization is a restructuring operation aimed at altering the decomposition into modules of an existing application. Being a restructuring, rather than reengineering operation, remodularization transforms source code in a behavior-preserving fashion [7]. The concrete lower-level source code transformations are usually based on commonly known refactorings [65]; either within the boundaries of the original implementation language, or supported by a module system [66] or by mechanisms for advanced separation of concerns [67].

The existing approaches to remodularization generally differ from one another with respect to: degree of automation, the applied remodularization criteria and the technical mechanisms of applying these criteria to an existing codebase. In the particular case of feature-oriented remodularization, the number of existing manual approach greatly exceeds the number of the automated ones. Nevertheless, it appears that a significant portion of the existing non-feature-oriented automated approaches could in principle be adapted to using feature-oriented remodularization criteria.

2.4.1 Manual Approaches

In their book, Demeyer et al. [68] proposed a comprehensive guide to reengineering existing software systems. They provide a holistic set of reengineering patterns that address topics ranging from understanding, redesigning and refactoring a legacy codebase, to placing the whole process within an organizational and methodological context. Furthermore, they argue for the importance of establishing a culture of continuous reengineering within an organization to enhance flexibility and maintainability of software systems. Demeyer et al. support their proposals with a range of examples taken from their industrial experiences.

Mehta and Heineman [69] discussed their experiences from a real-world manual remodularization of an object-oriented application to a feature-oriented decomposition. They present a method for locating and refactoring features into fine-grained reusable components. In a presented case study, feature location is performed by test-driven gathering of execution traces. Features are then manually analyzed, and manually refactored into components that follow a proposed component model. Mehta and Heineman conclude that the performed remodularization improved the maintainability and reusability of features.

Prehofer [70] proposed a programming paradigm called Feature-Oriented Programming (FOP) (also known as Feature-Oriented Software Development (FOSD) [71]). The paradigm proposes to separate core functionality of classes and methods from their feature-specific fragments, which are referred to as lifters. This is realized in practice by means of inheritance and mixins. Furthermore, the approach proposes

techniques for handling interactions among features at the composition time. A language extension to Java supporting the proposed approach is presented.

The approach of Prehofer was extended by Liu et al. [72], who proposed the feature oriented refactoring (FOR) approach to restructuring legacy applications to feature-oriented decompositions. FOR aims at achieving a complete separation features in source code, as a basis for creating feature-oriented software product-lines. This is done by the means of base modules, which contain classes and so-called introductions, and derivative modules, which contain manually extracted feature-specific fragments of original methods. The approach encompasses a firm algebraic foundation, a tool and a refactoring methodology.

Murphy et al. [67] explored the tradeoffs between three policies of splitting tangled features: a lightweight class-based mechanism, AspectJ and Hyper/J. By manually separating a set of independent features at different levels of granularity, Murphy et al. observed a limited potential for tangling reduction of the lightweight approach. In the cases of AspectJ and Hyper/J, they discovered that using these mechanisms makes certain code fragments difficult to understand in isolation from the others. Furthermore, aspect-oriented techniques were found to be sensitive to the order of composition, which resulted in coupling of features to one another.

The problem of order-sensitivity of feature composition was dealt with by McDirmid et al. [73]. They have used their *open class pattern* to remove the composition order constraints of the *mixin*-based feature-fragments proposed earlier by Prehofer [70]. The open class pattern is based on cycling component linking, which reveals the final static shape of classes being composed to the individual mixin-based features.

2.4.2 Automated and Semi-Automated Approaches

Tzerpos and Holt [74] proposed the ACDC algorithm aimed at clustering source code files for aiding software comprehension. To support software comprehension, they equipped their approach with pattern-driven clustering mechanisms, whose aim was to produce decompositions containing well-named, structurally familiar and reasonably sized modules. To achieve this goal, Tzerpos and Holt formalize seven subsystem patterns commonly occurring in manual decompositions of applications. The algorithm was applied to two large-scale software systems to demonstrate that the focus on comprehension does not have a negative effect on the resulting decompositions. To this end, a comparison of ACDC against authoritative decompositions created by developers was performed.

Mancoridis et al. [75] proposed to use cohesion and coupling metrics as the criteria for optimizing the allocation of classes to modules. The metrics were combined into a single objective function to create one measure for evaluating the allocation quality. Mancoridis et al. report on experiments of using this objective function in conjunction with three optimization mechanisms: hill climbing, simulated annealing and genetic

algorithms. During optimization, the objective function is evaluated based on a simple module dependency graph that represents source code files and relations among them. The approach was implemented as the Bunch clustering tool.

Bauer and Trifu [76] remodularized software using a combination of clustering and pattern-matching techniques. Pattern matching was used to automatically identify structural patterns at the level of architecture. These patterns were then used as an input to a coupling-driven clustering algorithm. To form a single objective function out of six coupling metrics, a weighted sum of the metrics' values was used. Here, Bauer and Trifu recognize that further studies are needed to determine the optimal weights for each coupling metric. The approach was applied to remodularizing the Java AWT library and demonstrated to be superior to a clustering approach oblivious to structural patterns.

O'Keeffe and O'Cinneide [77] defined an approach to optimizing a sequence of refactorings applied to object-oriented source code to improve its adherence to the QMOOD quality model. This is done by formulating the task as a search problem in the space of alternative designs and by automatically solving it using hill climbing and simulated annealing techniques. The search process is driven by a weighted sum of eleven metrics of the QMOOD. In application to two case studies, the approach was demonstrated to successfully improve the quality properties defined by QMOOD.

Seng et al. [78] proposed an approach to search-based decomposition of applications into subsystems by applying grouping genetic algorithms [79]. Using this kind of genetic algorithm was motivated by the observation that remodularization is essentially a grouping optimization problem. This makes it possible to exploit tailored gene representations and operators that help to efficiently explore solution spaces by preserving and promoting successful groups of classes. Seng et al. evaluated this approach by comparing the performance of their grouping operators in a case of remodularizing an open-source Java project. They used a single objective function that aggregated five metrics concerned with: cohesion, coupling, complexity, cycles, bottlenecks.

Harman and Tratt [80] observed it problematic that existing approaches to search-based optimization class grouping require complex fitness functions, where appropriate weights for individual metrics are difficult to identify. They proposed to address this problem by using the concept of Pareto optimality. Thereby, the need for weights can be eliminated and multiple Pareto-optimal solutions proposed to the users. This approach was evaluated in application to several open-source Java applications. Two independent objectives were used: the inter-class coupling and the standard deviation of method count in classes. A similar approach, but based on metrics of cohesion and coupling, was proposed by Bowman et al [81] to address the problem of class responsibility assignment in UML class diagrams. Finally, Praditwong et al. [82] used analogous approach to remodularize existing applications using the move-class refactorings.

Cohen et al. [83] postulated the need for on-demand reversible remodularization of existing applications by means of automated source code transformations. They used existing solutions to the expression problem to demonstrate the insufficiency of single structuring of a program during long-term evolution. The approach is based on formally specifying a set of bi-directional application-tailored code transformations. These transformations are demonstrated to transform source code on-demand, so that the impact of changes of the anticipated types can be minimized.

The problem of the tyranny of the dominant decomposition was attacked by Janzen and De Volder [84] using their approach of effective views. This approach aim at reducing the problem of crosscutting concerns by providing predefined editable textual views that expose two alternative decompositions of a single application. After an application is developed with support to effective views in mind, it become possible to alternate and synchronize between the views, to allow developers to choose the decomposition that better supports a task at hand.

A similar approach was proposed by Desmond et al. [85], who proposed a code presentation plugin for Eclipse IDE called fluid views. By visually inlining contents of several source files into a single editor window, this approach is postulated to enable developers to fluidly shift attention among multiple task-supporting materials without explicitly navigating across multiple files.

2.5 SUMMARY

The existing literature strongly indicates that features are among the important units of software modularity during evolutionary changes. Several authors report that feature-oriented tactics to source code comprehension and modification are applied by software developers. However, in the common case of improper modularization of features in source code, usage of such tactics is reported to lead to comprehension overhead and increasing the number of introduced defects.

Based on the existing works, it appears that an approach to practically addressing these issues should posses the following properties:

- Feature location needs to be both *simple to introduce* to existing large codebases, as well as *highly reproducible*. In particular, it appears relevant to explore the spectrum of tradeoffs between the two extremes: (1) software reconnaissance [50] that requires the initial implementation of dedicated test suites that can be re-executed in an automated fashion, and (2) marked traces [51] that require no such up-front investment, but makes it necessary to manually start and stop the tracing agent for each feature.

- While the existing techniques of feature-oriented analysis are demonstrated to be efficient at facilitating program comprehension, they lack a common conceptual

frame of reference. Such a common frame of reference is needed in order to meaningfully relate the existing techniques to one another, to compose them with one another and to develop new techniques that will cover any currently unaddressed needs of software evolution. Finally, to achieve practicality and reproducibility of feature-oriented analysis, freely available implementations within a modern interactive development environment (IDE) is needed.

- The literature reports on several approaches to manual feature-oriented restructuring of existing applications. The reported experiences point out the tradeoff between the impact of finer-grained separation of concerns on program comprehension and the inability of coarse-grained approaches to completely separate features. Hence, it appears important to further explore the involved tradeoffs, to report on real-world restructuring experiences and to develop tools for guiding such a process. As for automated remodularization, whilst a number of diverse approaches to scalable and reproducible restructurings exist, they have not been applied for the particular case of feature-oriented remodularization.

- Finally, the existing literature does not provide a satisfactory solution to automated feature-oriented measurement of software systems. Improving the automation of feature-oriented measurement is needed to enable feature-oriented quality assessment of large sets applications and their release histories.

3. OVERVIEW OF FEATUREOUS

This chapter provides an overview of the Featureous approach. The chapter firstly presents the conceptual modules of Featureous and the responsibilities of its three constituent layers. The theoretical bases, designs and implementation details of the individual layers follow in chapters 4 to 8. Lastly, this chapter describes the overall design and implementation of Featureous Workbench plugin to the NetBeans IDE.

3.1 THE FEATUREOUS CONCEPTUAL MODEL

The starting point of the Featureous approach is the observation that object-oriented applications are commonly modularized using layered separation of technical concerns. While such modularizations may indeed be beneficial during initial stages of development, evidence exists for their problematic nature during user-originated evolutionary changes. Despite this, a modularization established at early stages of development usually remains the dominant decomposition throughout whole software lifecycle. In particular, the initial modularizations of source code prove difficult to restructure due to significant effort, complexity and risk involved in doing so.

In order to address this problem, Featureous provides an integrated approach to *locating*, *analyzing* and *modularizing* features in existing Java applications. This is done by proposing a single conceptual model that consists of three conceptual layers – each of which addresses separate aspects of the overall problem. As depicted in Figure 3.1, the three layers build upon source code of an existing Java application and incrementally define the parts of Featureous: feature location, feature-oriented

analysis and feature-oriented remodularization. The advantages of this layered structure include dividing the whole approach into components that can be designed, implemented and evaluated independently from one another. The three conceptual layers of Featureous are defined as follows:

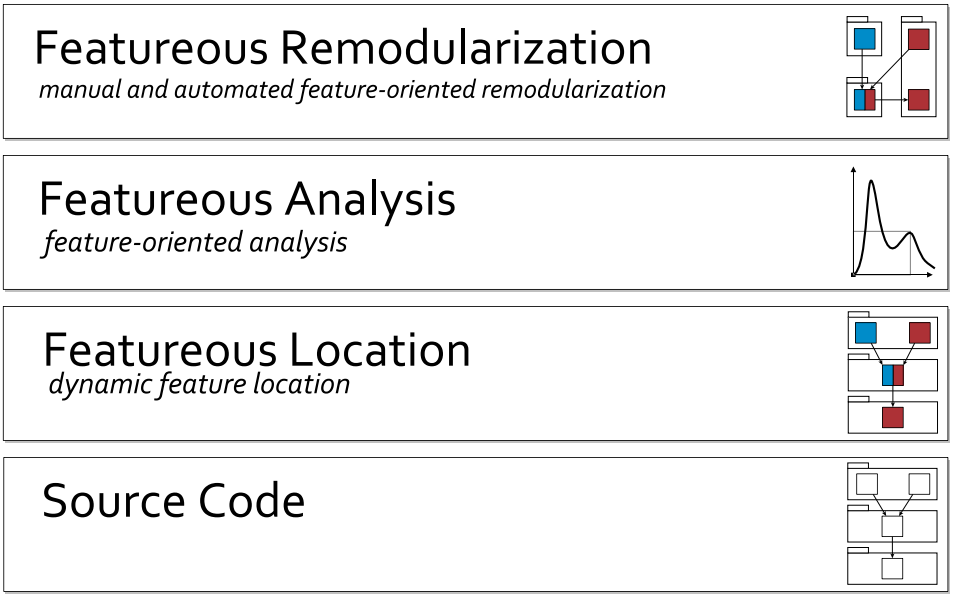


Figure 3.1: Layered conceptual structure of Featureous

- *Feature location* establishes traceability links between features and source code of an existing Java application. This fundamental traceability data becomes a basis for the next two layers of Featureous. Feature location is discussed in detail in Chapter 4.

- *Feature-oriented analysis* provides means of visualizing and measuring feature traceability links through several analytical views. The views provide a theoretically grounded support for comprehension and modification of features. Feature-oriented analysis is discussed in Chapter 5. Furthermore, Chapter 8 presents a method for automated feature-oriented measurement of applications.

- *Feature-oriented remodularization* optimizes the modularization of features in package structures of Java applications. A manual method for performing analysis-driven remodularization is presented in Chapter 6. Based on the manual method, Chapter 7 develops an automated remodularization process by formulating remodularization as a multi-objective design optimization problem based on the notion of Pareto-optimality.

As for the practical application of Featureous, the layered conceptual model also reflects the intended high-level workflow of the approach. Namely, (1) based on the source code of an application, feature location is used to establish traceability links between features and source code; (2) traceability links are used as an input to the feature-oriented analysis of source code; (3) based on traceability information and the

gained feature-oriented understanding, remodularization of an application's source code can be performed.

Through the three conceptual layers, Featureous provides an integrated mixture of *reverse engineering* and *restructuring* methods, as defined by the taxonomy of Chikofsky and Cross [7]. The reverse engineering is supported by the feature location and feature-oriented analysis layers, because they are aimed at creating a feature-oriented representation of an application for the purpose of examination. Restructuring capabilities are provided by the feature-oriented remodularization layer of Featureous that aims at transforming an application's structure from its current form to a feature-oriented one, while preserving the application's externally observable behavior. Through this integrated approach, Featureous sets out to realize the vision of *feature engineering* anticipated by Turner et al., which considers features are first-class entities throughout software life cycle [4].

3.2 THE FEATUREOUS WORKBENCH

The conceptual layers of Featureous were implemented in form of a *workbench*. The workbench not only serves as a convenient vehicle for evaluating various aspects of the conceptual model of Featureous, but it also facilitates practical application of the ideas presented in this thesis in other academic and industrial contexts. The Featureous Workbench is implemented as a plugin to the NetBeans IDE [86] and is tightly integrated with existing support for Java development in the IDE, such as Java project execution, code editing and navigation. Both the binary builds and the source code of Featureous Workbench are freely available [87].

Figure 3.2 shows an overview of the user interface of the Featureous Workbench. Featureous augments the standard NetBeans IDE in the three following ways.

Featureous extends the main toolbar of the NetBeans IDE with two *feature tracing actions* (1). The first action executes a Java project with runtime tracing enabled, whereas the second executes the JUnit test suite of a project with runtime tracing enabled.

The main window of Featureous (2) contains a *trace explorer* frame and a number of actions that open feature-oriented views over the traces provided by the feature location mechanism. The trace explorer frame displays a list of currently loaded trace files and provides actions for loading/unloading and grouping/ungrouping traces.

The individual *views* of Featureous (3) become visual in the editor position of the NetBeans IDE interface, when activated from the main window. The views include a number of chart-based and graph-based feature-oriented analytical views, the

remodularization view, feature-aware source code editor and a simple Java console based on the BeanShell2 library [88].

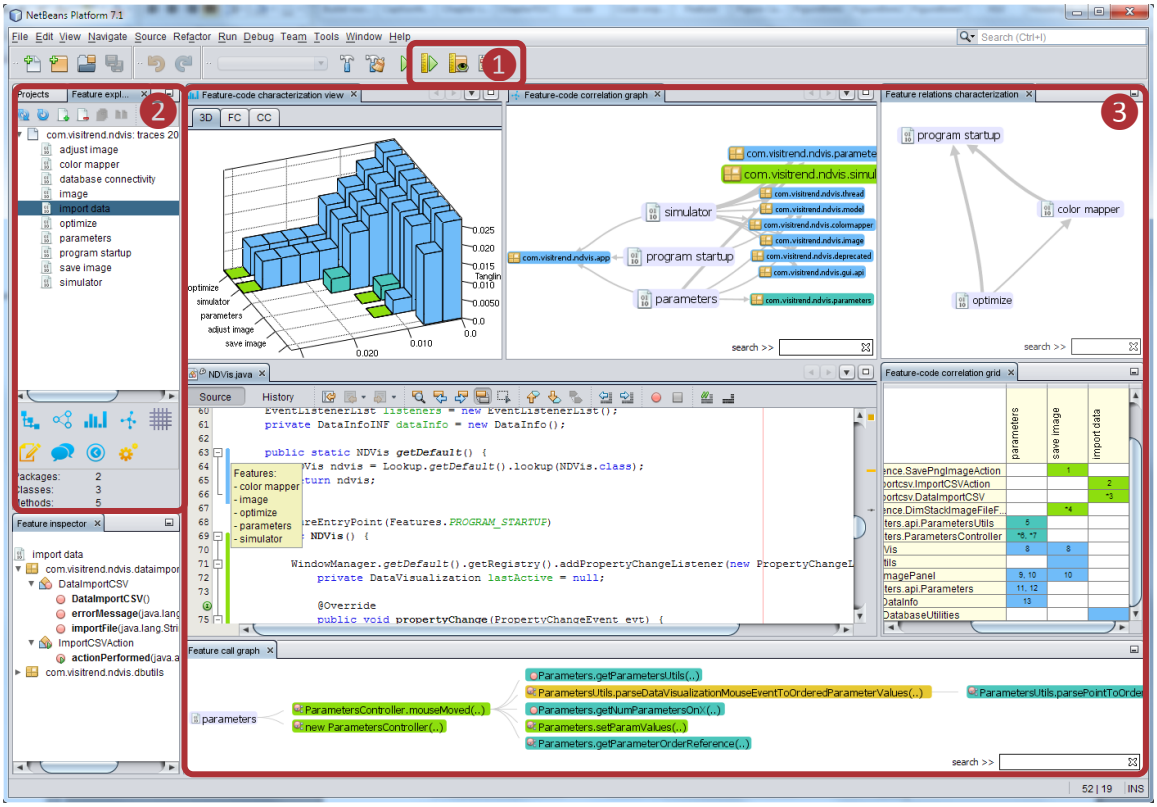


Figure 3.2: User interface of the Featureous Workbench

An introductory video demonstrating a practical usage of these user interface elements is available on the official website of Featureous [87]. The motivations and designs of individual views are discussed in the remaining chapters of this thesis.

3.2.1 Design of the Featureous Workbench

A major design decision in the development of the Featureous Workbench is its tight integration with the NetBeans IDE. This decision was driven by the aim of integrating Featureous with the existing practices and tools used daily by software developers. As a side effect, it became possible to reuse large quantities of existing infrastructures that are provided by these tools. In particular, Featureous builds upon APIs of the NetBeans IDE and NetBeans RCP such as Window System, Java Project Support, Java Source, Execution.

Furthermore, Featureous Workbench takes advantage of the NetBeans Module System and the Lookup mechanism to facilitate plugin-based extensibility of itself. As depicted in Figure 3.3, this results in a separation between the reusable infrastructural modules of the Featureous Workbench and its plugins. This makes it possible to

flexibly add or remove the individual views provided in the Featureous Workbench. Similarly, new third-party views can be added as plugins without affecting the already installed views.

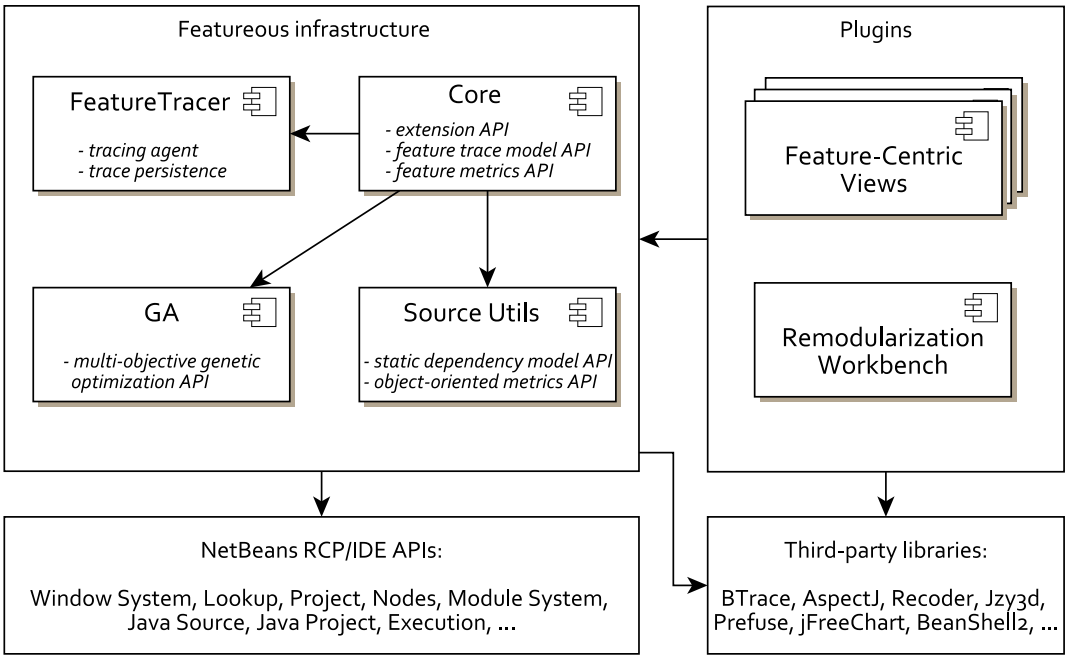


Figure 3.3: Architecture of the Featureous Workbench

The infrastructure of the Featureous Workbench is centered on the feature location mechanism that captures the traceability links between features and source code of Java applications. This data is further exposed through a number of APIs to pluggable view modules. Apart from the *feature trace models* that describe the captured traceability links, Featureous Workbench provides additional services such as: *feature metrics* that measure various properties of the traceability links, a *static dependency model* that represents static structure of a Java applications, and implementations of several *object-oriented metrics*. The details of these APIs are presented in Appendix A1.

3.3 SUMMARY

This chapter presented a high-level overview of the Featureous approach and outlined the responsibilities of its constituting conceptual layers. The detailed descriptions of the individual layers were delegated to subsequent chapters of this thesis. Lastly, this chapter discussed the Featureous Workbench plugin to the NetBeans IDE that implements the Featureous approach.

4. FEATUREOUS LOCATION

This chapter defines the concept of a feature-entry point and uses it as a basis for developing Featureous Location. Featureous Location is a dynamic method for recovering traceability links between features and source code of existing Java applications. The conceptual design, the implementation details, the application experiences and evaluation of effort-efficiency and accuracy of the proposed method are presented.

This chapter is based on the following publications:
[SCICO'12], [WCRE'11a], [FOSD'09]

4.1 OVERVIEW

Feature location is the process of identifying units of source code that contribute to implementing features of an application [46], [47].

The vast body of existing research on feature location, which was discussed in Chapter 2, indicates feasibility of various mechanisms and diverse sources of data. The choice of a particular mechanism determines the set of required artifacts, the amount

of manual effort imposed on a software developer, as well as the precision and recall of feature location.

The requirements in the context of this work include a high degree of automation, simplicity of application to unfamiliar and large codebases, a minimal impact on existing source code and high precision (i.e. minimizing false-positive traceability links).

In order to acquire the mentioned properties, it is necessary to develop a dedicated feature location method that builds upon the selected aspects of the existing methods involving dynamic analysis mechanisms. In particular, it is worthwhile to explore the middle-ground between driving execution tracing by implementing test cases (e.g. software reconnaissance [50]) and making it the end-user's responsibility to manually enable and disable the tracing mechanisms (e.g. marked traces [51]).

The workflow of the Featureous method of feature location, called *Featureous Location* is depicted in Figure 4.1. This workflow and its resulting feature traces will form a basis to other feature-oriented workflows discussed in the remainder of this thesis.

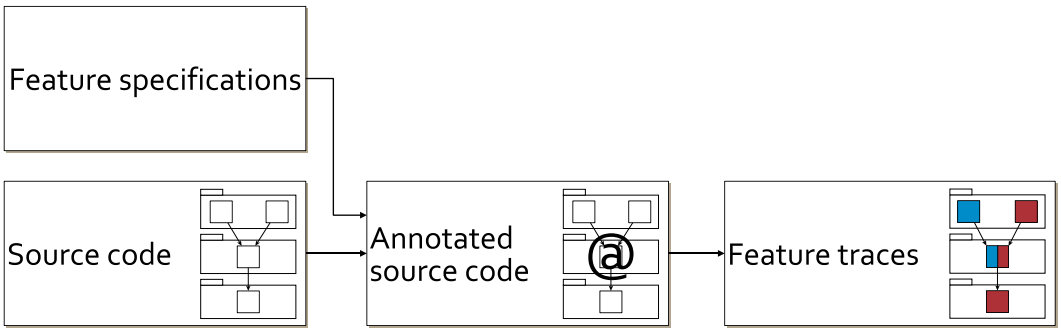


Figure 4.1: Overview of Featureous Location

Featureous Location achieves a middle ground between software reconnaissance and marked traces by storing the knowledge about activation of features in source code in the form of minimal meta-data. This meta-data is used to mark the starting points of method-invocation sequences that occur when a user activates a feature in an application. Featureous Location refers to such special methods being the starting points of features as *feature-entry points*. The mentioned meta-data is used upon tracing to automatically reason about activations and deactivations of individual features.

Featureous Location is evaluated by applying it to several medium and large open-source Java applications. The detailed procedures and outcomes are reported for the JHotDraw SVG and BlueJ applications. An initial assessment of the effort-efficiency, coverage and accuracy of Featureous Location is presented.

4.2 RECOVERING FEATURE SPECIFICATIONS

In order to establish traceability links between features and source code units, both the source code and the specifications of features need to be available. Whereas the source code is usually available, it is seldom the case for the specifications of features.

For legacy systems, whose functionality is documented in terms of *use cases* [89] rather than features, specifications of features can be formed by grouping use cases into semantically coherent groups. This is consistent with the definition of features proposed by Turner et al., who consider features as a concept that organizes the functionality of a system into coherent and identifiable bundles [4]. Given a set of use cases, Featureous Location proposes to group them around the domain entities (nouns), rather than the activities (verbs). The prerequisite is, however, that a common user-identifiable generalization for the activities being grouped can be created. For instance, the following five use cases {OPEN DOCUMENT, CLOSE DOCUMENT, OPEN HELP, LOAD DOCUMENT, SAVE DOCUMENT} can be grouped into three features: DOCUMENT MANAGEMENT = {OPEN DOCUMENT, CLOSE DOCUMENT}, HELP = {OPEN HELP}, DOCUMENT PERSISTENCE = {LOAD DOCUMENT, SAVE DOCUMENT}.

Depending on the needs of a particular application, the granularity of the groups can be adjusted. This can range from treating each use case as a separate feature to grouping all activities of an entity (in the mentioned example, this would merge DOCUMENT MANAGEMENT with DOCUMENT PERSISTENCE into the DOCUMENT group). Regardless of a particular choice being made here, one should strive for consistently applying a single chosen granularity policy for all the use cases of a single application.

When use-case-based documentation is unavailable, one has to understand the application's problem domain and its runtime behavior in order to identify the use cases, which can thereafter be grouped into features. Good candidates for identifying use cases are the elements of a graphical *user interface* allowing for user-triggerable actions (e.g. main menus, contextual menus, buttons, etc.), keyboard shortcuts and command-line commands. Depending on the complexity of the problem domain and the experience of a developer, this process can also be performed using a more formalized approach for requirements recovery, such as AMBOLS [90].

An important consideration when forming feature specifications is the design of scenarios for invoking them in a running application. Such scenarios should encompass the sequences of actions that need to be carried out in the user interface of an application to activate individual features. While providing such scenarios may not be essential for the users who are experienced with using a target application, it certainly helps ensuring consistency and reproducibility of the feature location process. Apart from signalizing the importance of this issue, Featureous Location does not impose any particular method or constraints for specifying execution scenarios.

Lastly, please note that using Featureous Location does not make it necessary to create full feature models, which are usually applied in the context of software product lines [72]. For the purposes of Featureous Location, textual specifications of features are sufficient.

4.3 THE METHOD

This section presents the design of the Featureous Location method. Its corresponding implementation details will be presented in Section 4.4.

Feature-Entry Points

When a list of feature specifications of an application is acquired, it can be used to prepare the source code of an application for runtime tracing. To do so, one has to identify the so-called *feature-entry points*.

Feature-entry point is a method, through which a thread of control enters a particular feature. Such a method is typically the first method being invoked when a user activates a feature in the graphical interface of an application. In the example provided in Figure 4.2, the `actionPerformed()` method of a Swing button constitute an entry point to FEATURE1. Consequently, an execution of a feature stops when the thread of control returns from the feature-entry point method. Every code statement executed between the entrance of and return from a feature-entry point is assumed to belong to the corresponding feature (highlighted in Figure 4.2).

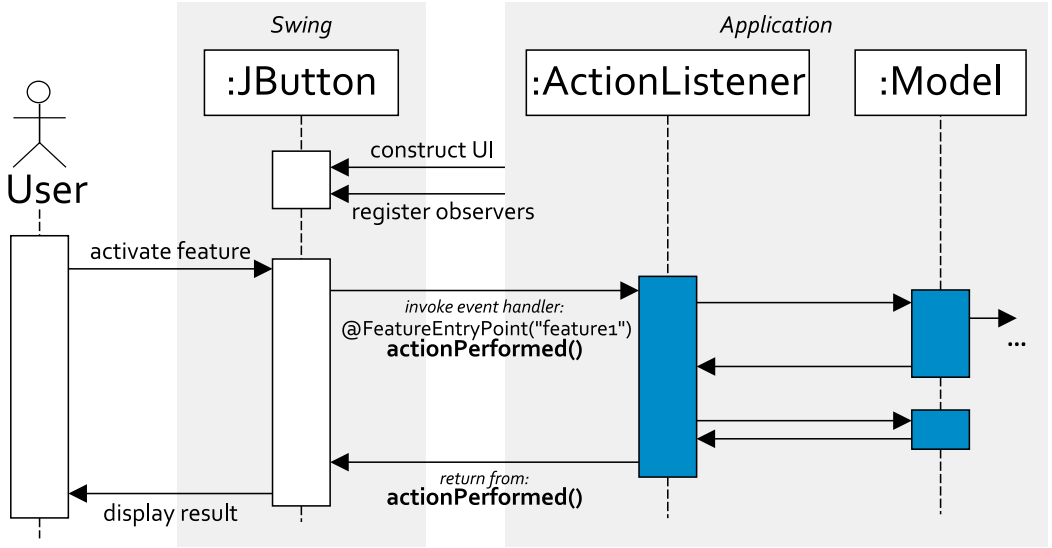


Figure 4.2: Using feature-entry points to locate features

Declaring Feature-Entry Points

Declaring a method as being a feature-entry point is done by manually marking the method declaration with a dedicated Java annotation `@FeatureEntryPoint`. As exemplified in Figure 4.2, the annotation takes one parameter, a string-based identifier of its corresponding feature. The presence of this annotation, as well as the value of its parameter, is retrieved at runtime by an execution-tracing tool that uses this information to recognize a feature-membership of methods being executed. In Featureous Location, a feature is allowed to have more than one feature-entry point annotated in the source code, to cover cases where a feature can be activated in several ways (e.g. when a feature consists of several independently-activated use cases, or when it can be activated either from a menu, or with a keyboard shortcut).

Table 4.1: Functionality-related callback methods in popular interfacing technologies

Technology	Candidate feature-entry point method

JDK	`java.lang.Runnable.run`
	`java.util.concurrent.Callable.call`
	`*.main`

Swing/AWT	`java.awt.event.ActionListener.actionPerformed`
	`javax.swing.event.ChangeListener.stateChanged`
	`java.awt.event.KeyListener.keyTyped`
	`java.awt.event.KeyListener.keyPressed`
	`java.awt.event.MouseListener.mouseClicked`
	`java.awt.event.MouseListener.mousePressed`

SWT	`org.eclipse.swt.widgets.Listener.handleEvent`
	`org.eclipse.swt.events.KeyListener.keyPressed`
	`org.eclipse.swt.events.MouseListener.mouseDown`
	`org.eclipse.swt.events.MouseListener.mouseDoubleClick`
	`org.eclipse.swt.events.SelectionListener.widgetSelected`
	`org.eclipse.swt.events.SelectionListener.widgetDefaultSelected`

JFace	`org.eclipse.jface.action.IAction.run`
	`org.eclipse.jface.action.IAction.runWithEvent`
	`org.ecplise.jface.operation.IRunnableContext.run`

Eclipse RCP	`org.eclipse.core.runtime.IPlatformRunnable.run`
	`org.eclipse.equinox.app.IApplication.start`
	`org.eclipse.core.commands.IHandler.execute`

Android	`android.app.Activity.onCreate`
	`android.app.Activity.onOptionsItemSelected`
	`android.view.View.OnClickListener.onClick`
	`android.view.View.OnLongClickListener.onLongClick`
	`android.view.View.OnKeyListener.onKey`
	`android.view.View.OnFocusChangeListener.onFocusChange`
	`android.view.KeyEvent.Callback.onKeyDown`
	`android.view.View.OnTouchListener.onTouch`
	`android.app.Service.onStartCommand`
	`android.content.Context.startService`

Servlet	`javax.servlet.HttpServlet.doGet` `javax.servlet.HttpServlet.doPost`

Spring	`org.springframework.web.servlet.mvc.Controller.handleRequest`
	`org.springframework.web.servlet.HandlerAdapter.handle`
	`org.springframework.web.servlet.mvc.SimpleFormController.onSubmit`
	`*.start`
	`*.initApplicationContext`

Struts	`com.opensymphony.xwork2.Action.execute`
	`com.opensymphony.xwork2.ActionInvocation.invoke`
	`com.opensymphony.xwork2.interceptor.Interceptor.intercept`

To support identifying candidates for annotation as feature-entry points, Table 4.1 presents a list of method names used by common Java interfacing technologies. The listed methods were identified by reviewing the documentation of their respective libraries. Each of these methods was identified as dedicated by its library authors to take part in functionality-related callback mechanisms. The libraries enforce this by defining interfaces, or abstract classes, that must be implemented by a client application in order for it to register for a particular type of event emitted by a library.

By making it necessary to annotate only the starting methods of features, Featureous Location removes the proportionality between the size of a feature and the number of methods that need to be manually marked in source code. This is aimed at reducing the amount of manual work involved in locating features in unfamiliar applications. This improves over other methods based on marking source code [48], [49]. Moreover, Java annotations do not affect an application's correctness and they remain synchronized with the code during common refactorings like changing method signature, or moving method to another class. Finally, annotations introduce no performance overhead when the application is executed with execution tracing disabled.

Activating and Tracing Features

Activating features in an instrumented application can be done either by creating and executing appropriate test cases, or by interacting with an application's user interface. The first method allows for better repeatability and systematic coverage of multiple execution paths, but at the same time, it requires substantial manual effort, good knowledge of a legacy system at hand, and possibly modifications thereof to additionally expose classes and method implementing features. Additionally, with the popular unit testing frameworks it is problematic to cover graphical-user-interface classes and thus to capture them in feature traces. In comparison, the manual effort required by a user to trigger features in a running application is negligible. However, this effort reduction comes at the potential costs of lowering repeatability and not exercising all execution paths in the investigated application. Repeatability can be improved by using a GUI testing robot as demonstrated by Greevy and Ducasse [52]. GUI testing robots are, however, sensitive to changes in the contents and layouts of the user interface. Therefore, Featureous Location opts for the user-driven triggering method due to its low manual workload and good applicability to unfamiliar source code.

By tracing user-driven execution of features, Featureous Location produces a set of *feature-trace models* that encompass the traceability links between features and source code of an application. The definition of feature-trace models is shown in Figure 4.3. For each of the traced features, a separate trace model is instantiated. The implementation details of the tracing mechanism will be presented in Section 4.4.

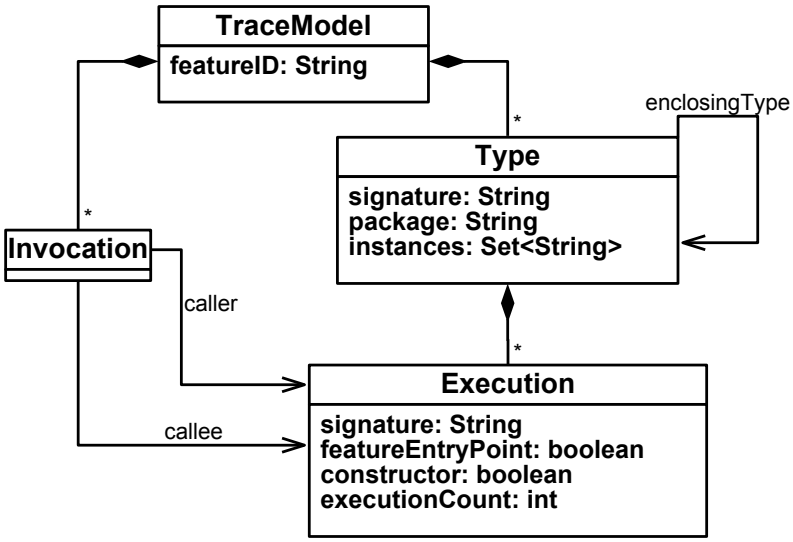


Figure 4.3: Feature-trace model produced by Featureous Location

A feature-trace model captures three types of information about its feature: the methods and constructors (jointly referred to as `Executions`) executed, their enclosing types (i.e. classes, interfaces or enums) and inter-method invocations. An execution is placed in a feature trace only if it was invoked at least once during the feature's execution, but it is placed there only once, regardless of the actual times that its invocation is repeated. An analogous rule applies to types. In the case of reentrance to a single feature-entry point, multiple entry points or parallel executions of a single feature, the new data are aggregated in an already existing model associated with the target feature. Hence, at any point in time a single feature is represented by exactly one, consistent feature-trace model. The collected feature-trace models are automatically serialized to files upon the termination of a traced application.

4.4 IMPLEMENTATION

The proposed method of feature location was implemented as a Java library that exploits load-time instrumentation of classes for tracing runtime execution. The library consists of several parts.

As discussed in Section 4.3, feature-entry points of a target application need to be manually annotated. This has to be done using the `dk.sdu.mmmi.featuretracer.lib.FeatureEntryPoint` annotation type provided by the Featureous Location library. Figure 4.4 shows an example of marking an action-handler `actionPerformed` method of a FEATURE1 feature. This is done by using the mentioned annotation type and parameterizing it with the string-based identifier of the feature. When such an annotated method is executed with tracing enabled, each method called within the flow of control of the `actionPerformed` will be registered as a part of FEATURE1.

```
public class MainWindow {
  public static final String f1 = "feature1";
  ...
  public static void main(String[] args){
    ...
    JButton b = new JButton();
    b.addActionListener(new ActionListener(){

      @FeatureEntryPoint(f1)
      public void actionPerformed(ActionEvent e){ ... }

    });
  }
}
```

Figure 4.4: Annotating feature-entry points in code

After all desired feature-entry points of an application are annotated, the application has to be executed in a traced mode. This is done by declaring the Featureous Location library as a Java agent parameter to the Java Virtual Machine during the invocation of a target application. Doing so transparently instruments the application's JAR files with execution-tracing routines. The task of the inserted tracing routines is to detect the execution of features and to save the collected trace data in the form of feature-trace files upon the application's shutdown.

```
public abstract aspect ExecutionTracer {
  ...
  public abstract pointcut packages();
  public final pointcut anyCall(): packages() && (call(* *.*(..)) || call(*.new(..)));

  before() : anyCall(){
    callerClass.set(thisJoinPointStaticPart.getSourceLocation().getWithinType());
    ...
  }

  Object around() : anyExecution(){
    ...
    enterExecution(callerClass.get(), callerID.get(), callerMethod.get(),
      calleeClass, calleeID, calleeMethod, fepTo, isConstructor);
    try{
      return proceed();
    }finally{
      leaveExecution(calleeClass, calleeID, calleeMethod);
    }
  }
}
```

Figure 4.5: The tracing aspect

The tracing agent provided by Featureous Location is implemented using *AspectJ*, an aspect-oriented programming tool for Java [91]. The AspectJ Load-Time Weaver is used for performing transparent instrumentation of class byte-code at their load-time. The essentials of the implemented aspect are shown in Figure 4.5. The aspect is based on the `before` and `around` advices that match the `call` and `execution` join points for every method and every constructor in a target Java application. The aspect extracts the signatures of executing methods and constructors and identifies the caller-callee pairs of execution among them (this is why both `call` and `execution` pointcuts are needed). Furthermore, the information about the enclosing types of executions and about the

identities of the executing objects is retrieved. The object identity information is based on object hash codes.

To determine the contribution of a currently executing method to the features of an application, Featureous Location library maintains an internal stack-like representation of the feature-entry points entered by each thread of execution. Thanks to this internal representation, the tracer is able to correctly interpret situations where a thread of control leaves the tracing scope and returns to it (e.g. during reflection-based method invocations, or callbacks from external libraries). The tracing agent updates its state upon entering and leaving every execution through the calls to the `enterExecution` and `leaveExecution` methods.

The presented aspect has to be externally configured to specify a concrete root package of a target application that should be traced. This is done by implementing the abstract pointcut `packages` using the standard *aop.xml* configuration file, as supported by the Load-Time Weaver. This externalization of the configuration data enables applying Featureous Location to different applications without recompiling the tracing agent.

4.4.1 Important Design Decisions

When encountering an invocation of an inherited, abstract or overridden method, Featureous Location always registers the signature of the method whose body was actually executed due to the invocation. Because of this decision, interfaces and some abstract classes will not be registered in feature traces due to the abstract nature of their methods. While it remains possible to modify this design decision, Featureous deems it important that the traces contain only classes and methods that actually contribute to the processing logic of feature executions.

Memory usage of any dynamic method of feature location is an important concern, because it may impose limitations on how long a user will be able to interact with a running application. Featureous Location removes the proportionality relation between the tracer's memory usage and the duration of a target application's execution by using a set-based representation of an applications execution. The library increases its memory footprint only when the sets containing identifiers of methods, classes and objects are expanded, which occurs during the execution of elements not previously contained in these sets. However, maintaining only this information comes at the cost of not being able to use feature traces as a basis for reconstructing concrete time-ordered sequences of method invocations.

Finally, the choice of AspectJ as the implementation technology imposes several constraints on the properties of tracing. Firstly, Featureous Location does not keep track of read and write access to fields – while it is technically possible, the incurred performance overhead is significant. Secondly, tracing of granularities finer than whole methods, e.g. individual code blocks or statements, is not possible with AspectJ.

Thirdly, tracing of JDK classes is not possible. While these properties are not essential to Featureous as a whole, the source code Featureous Location contains a prototype reimplementation of the code tracer using the BTrace library [92]. Using this prototype, it is possible to overcome the last two limitations of the AspectJ-based implementation – however, doings so was observed to significantly affect the execution performance of traced applications.

4.4.2 Integration with the Featureous Workbench

Featureous Location library is included as one of the core modules of the Featureous Workbench. Featureous Workbench incorporates Featureous Location as the source of traceability data and integrates its tracing routines with the Java project execution facilities of the NetBeans IDE. Featureous Workbench adds two new actions representing the tracing facilities of Featureous Location: "Run Traced Project" and "Test Traced Project" to the main toolbar of the IDE as shown in Figure 4.6. The first of these two actions represents the default user-driven mode of applying Featureous Location to an existing Java application. Additionally, the second action makes it possible to trace features that are activated by JUnit test suites.

Figure 4.6: Tracing actions within Featureous

One of the benefits of the integration with the NetBeans IDE is the possibility to employ the NetBeans debugger to aid identifying feature-entry point methods. This can be done by adding a breakpoint on the entry to all methods with a given name, such as `actionPerformed` or `keyPressed`, in all the classes contained in an application. The resulting configuration of the breakpoint should be analogous to the one shown in Figure 4.7. Such a breakpoint, if configured properly, halts the debug-execution of the application's code upon the activation of a feature by a user. The line of code that triggered the breakpoint will be automatically displayed in the NetBeans editor. This line will be a candidate for annotation as a feature-entry point. The experiences of the author suggest that this simple technique provides a significant support for incremental annotation of feature-entry points in large and unfamiliar codebases.

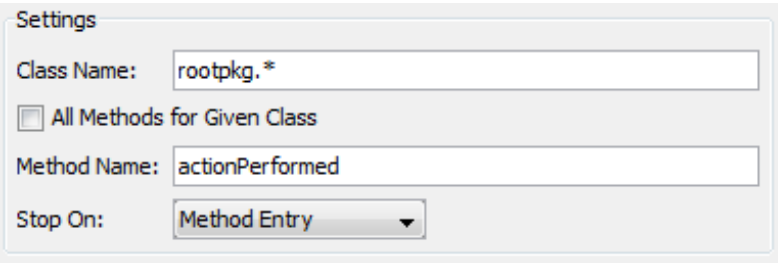


Figure 4.7: Using configurable breakpoints to indentify feature-entry points

Lastly, it is important to mention that despite of the integration with Featureous Workbench, Featureous Location library is a standalone library that can be used outside of Featureous Workbench and of the NetBeans IDE. In practice, this enables packaging an application together with Featureous Location and delivering it to the end-users, to make them produce feature traces without involving them in using the NetBeans IDE.

4.5 EVALUATION

Featureous Location was applied to locating features in several open-source Java applications ranging from 15 KLOC to 80 KLOC. These applications will be used as the target systems in the evaluations of the Featureous approach throughout the remainder of this thesis. Overall, it was found that none of the investigated applications was initially equipped with any form of traceability links between functional requirements and source code. Furthermore, only single applications had their functionality textually documented in a use-case-like fashion.

The remainder of this section presents evidence for viability of Featureous Location by applying it to two applications. The presented results are used as a basis for an initial qualitative evaluation of the characteristics of the method: the manual effort involved and the accuracy and code coverage.

The two applications being the focus of the presented evaluation are BlueJ [93] and JHotDraw SVG [94]. It is worth mentioning that neither the source codes nor the architectures of the two applications were known to the author beforehand.

BlueJ is an open-source interactive programming environment created to help students learn the basics of object-oriented programming in Java. Since BlueJ is a nontrivial application, whose operation involves compilation, dynamic loading, execution and debugging of Java code, it is a particularly interesting target for applying Featureous Location. The version of BlueJ used here is 2.5.2 and it measures 78 KLOC in size.

JHotDraw is an open-source graphical framework created as an example of a well-designed object-oriented application. Due to this property, JHotDraw is widely used in software case studies. The presented investigations focus on the concrete application of the framework called SVG. SVG is an example application built on top of the framework that is distributed together with JHotDraw. The version of JHotDraw SVG used here is 7.2 and it measures 62 KLOC in size.

4.5.1 Applying Featureous Location

In order to evaluate Featureous Location, the method was applied in a systematic manner to both BlueJ and JHotDraw SVG.

Identifying Use Cases and Features

In the case of BlueJ, it was possible to infer a vast majority feature specifications from the available user documentation. The documentation of BlueJ is arranged into a list of usage scenarios, which correspond to the application's use cases. Thereby, 127 use cases were identified, which were then grouped into 41 coherent features. The list of features identified for BlueJ, as well as for JHotDraw SVG can be found in Appendix A2.

In the case of JHotDraw SVG no documentation of functionality was available. Therefore, the author relied solely on the contents of the application's user interface, i.e. its main menu, contextual menus and toolbars. In the course of inspecting these elements and the behaviors they represent, 90 use cases were identified, which were then grouped into 31 coherent features.

The recovery of use cases and grouping them into features was performed using the guidelines proposed earlier in Section 4.2.

Annotating Feature-Entry Points

After identifying the features of both applications, the author investigated their source code to annotate feature-entry points.

The major groups of methods annotated in both applications were the `actionPerformed` methods that handle events produced by the Swing GUI framework when a user activates a GUI widget, e.g. presses a button or chooses a menu item. Such annotations accounted for 14% of all annotations in BlueJ and 24% of all annotations in JHotDraw SVG.

Here, it was observed that the applications convert some of the more general Swing events, such as `mouseDragged` or `mousePressed` to their custom application-specific event types that are then dispatched to custom event handlers. This indirection made it necessary to annotate not only the Swing-based event handlers, but also the application-specific ones. Hence, methods such as `addObject`, `message` or `load` were annotated in BlueJ, and methods such as `handleMouseClick`, `createDisclosedComponent` and `doIt` were annotated in JHotDraw SVG.

Lastly, the number of identified use cases was found to correspond to the number of placed annotations in the case of JHotDraw SVG. Namely, for the 90 recognized use cases, 91 feature-entry point methods were annotated (which corresponds to approximately three entry points per feature). In contrast, the 127 use cases recovered from the documentation of BlueJ were found to frequently encompass alternative

execution scenarios (e.g. triggering a feature from a toolbar or from a menu) and therefore they required annotating 228 feature-entry points (which corresponds to nearly six entry points per feature). This observation confirms the need for supporting multiple entry points for a single feature, as proposed in Featureous Location.

Tracing the Execution Of Features

To trace the execution of features, the tracing agent of Featureous Location was used on the two annotated applications. By using GUIs to activate features and their contained use cases, two sets of feature traces were obtained. During the tracing process, no significant overhead of execution performance was observed.

Interestingly, it was found impossible to correctly activate two features in each of the applications. This was caused by runtime errors that prevented the features from finishing their executions. In all these cases, it was confirmed that the applications exhibited the same behavior regardless of the presence of the tracing agent.

4.5.2 Results: Manual Effort

Table 4.2 summarizes the registered effort of applying Featureous Location to BlueJ and JHotDraw SVG. Listed are the times required to identify the use cases and features, to place the feature-entry-point annotations on appropriate methods and to activate features at runtime.

Table 4.2: Summary of manual effort required by Featureous Location

Property	BlueJ	JHotDraw SVG
Application size	78 KLOC	62 KLOC
Number of identified use cases	127	90
Number of identified features	41	31
Time of indentifying features and use cases	2 person-hours	1 person-hour
Number of activated features	39	29
Number of feature-entry points	228	91
Time of annotating and activating features	6 person-hours	4 person-hours
Total time	8 person-hours	5 person-hours

As can be seen, the total time required to apply Featureous Location to the unfamiliar codebases of the BlueJ and JHotDraw SVG was eight person-hours and five person-hours respectively. For the used applications, the required time appears proportional to the number of features and to the KLOC size of the codebases. Based on this, the amount of work required by the manual part of Featureous Location is evaluated as relatively low – especially if one takes into account that the investigated applications were unfamiliar to the author. Qualitatively, the manual work involved was found to be uncomplicated and not to require extensive investigations of the codebases.

Featureous Location method significantly improves over manual feature location approaches. In a study of Eaddy [95], manual feature location was observed to progresses at rates between 0.7 KLOC per hour in dbviz (12.7 KLOC) and 0.4 KLOC per hour in Mylyn-Bugzilla (13.7 KLOC). In comparison with these results, Featureous Location appears to offer an order of magnitude work rate improvement. This result is, however, an expected consequence of the dynamic semi-automated nature of the method.

While a rigorous comparison of the work-intensity of Featureous Location with other dynamic approaches would be desired, it remains difficult to perform in practice. This is because of a lack of appropriate external effort-benchmarking data and the complexity of an empirical experiment that would be needed to faithfully construct such data. Hence, at this stage the presented results provide only an initial evidence for a satisfactory work-efficiency of the Featureous Location method.

4.5.3 Results: Coverage and Accuracy

Table 4.3 presents a summary of code coverage results achieved by Featureous Location. The table distinguishes four ways of how a class can participate in implementing features. *Single-feature classes* are the classes that participate in only one feature. *Multi-feature classes* participate in implementing more than one feature. *Non-covered classes and interfaces* are the units that were not used at runtime by features. Finally, "dead code" consists of the classes that are not statically referenced from the applications' main codebases, despite being physically present.

The "dead code" classes were detected using static analysis of source code. For BlueJ, dead code accounts for 34 out of the 234 non-covered units. These types were found to implement parsing of code tokens of several programming languages and to be located in package `org.syntax.jedit.tokenmarker`. As for JHotDraw SVG, "dead code" accounts for 161 out of the 282 non-covered types. The "dead code" was found to consist of various JHotDraw framework's classes that were not used by the SVG application.

Table 4.3: Summary coverage results

Type category	BlueJ	JHotDraw SVG
Single-feature	63	59
Multi-feature	238	152
Non-covered classes + interfaces	155+45	72+49
Dead code	34	161
Total types	535	493

Overall, it can be seen that after exclusion of "dead code" Featureous Location covered 301 out of 501 types (60%) of BlueJ. Furthermore, 45 of the non-covered types are identified as interfaces, which by design are not captured by Featureous Location, as

was discussed earlier in Section 4.4.1. For JHotDraw, Featureous Location covers 211 out of 332 non-dead-code types (64%). Furthermore, 49 of the 121 non-covered types are interfaces.

The reported levels of code coverage are considered sufficiently high to be representative for the complete investigated applications. The obstacles to achieving a higher coverage with Featureous Location, as well as with all other dynamic methods for feature location, are not only by the completeness of the recovered feature specifications, but also the difficult task of exercising all control branches of source code at runtime. An additional factor that might have affected the code coverage was the earlier-reported unsuccessful activation of two features in both applications.

Lastly, similarly with other dynamic approaches and given the assumption that source code was annotated correctly, Featureous Location exhibits complete *precision* (i.e. no false positives) and incomplete *recall* (i.e. existence of false negatives). Precision is maximized, because dynamic analysis registers the actual runtime resolutions of polymorphism and conditional statements. Without using dynamic analysis mechanisms, this information remains problematic to compute [96].

Nevertheless, dynamic analysis tends to suffer from incomplete recall, because of the difficulties associated with exercising alternative control flows at runtime. These difficulties are connected with the need for gathering implementation-specific knowledge in order to identify detailed conditions for activating individual conditional statements in source code (such as `if`, `switch` or `catch` statements). Such an implementation-specific knowledge cannot be assumed as available to the end-users who drive the runtime tracing process. In the context of Featureous, lack of false positives, for the cost of incomplete recall, is a reasonable tradeoff. This is preferred over the situation of having results that are more complete, but which cannot be fully trusted due to the presence of false positives.

4.6 APPLICATION TO RUNTIME FEATURE AWARENESS

In a publication related to this thesis [WCRE'11a], the author applied the tracing library of Featureous Location for a purpose radically different from the traditional purposes of feature location. There, it was proposed to expose the feature trace models to the executing applications being traced, to enable them to observe and act upon the activations of their own features. This method was termed as *runtime feature awareness*.

This idea was implemented as *JAwareness*, a library-based meta-level architecture that exposes the traceability links being created by Featureous Location to the base-level application through an API. The thereby established metaobject protocol [97] enables the base-level application to obtain the identifiers of currently active features.

Furthermore, the base-level application can register observers, in order to be notified about activations and deactivations of features occurring at runtime.

The feasibility of this idea was demonstrated by proposing and developing three proofs-of-concept applications of runtime feature awareness. Firstly, it was shown how to make error reports printed by a crashing application easier to interpret by equipping them with information about features active during the occurrence of the error. Secondly, it was shown how an application could collect and report statistics on how often its features are activated by the users. The information on popularity of individual features could be then used to inform the prioritization of evolutionary enhancements and error corrections. Lastly, it was discussed how to create feature-aware logic to help users to learn the usage of an application, i.e. how to create a command recommendation system [98] based on a feature-oriented metaobject protocol.

4.7 RELATED WORK

Dynamic analysis was exploited in the *software reconnaissance* method proposed by Wilde and Scully [50]. Software reconnaissance associates features with the control flow of applications and proposes to trace features at runtime, as they are being triggered by dedicated test suites. The test suites have to encode appropriate scenarios in which some features are exercised while others are not. By analyzing this information, the approach identifies units of source code that are used exclusively by single features of an application. Thereby, the approach of Wilde and Scully reduces the need for manual identification of whole features in source code.

Similarly, Salah and Mancoridis [51] proposed to trace runtime execution of applications to identify features in their approach called *marked traces*. However, they replace the feature-triggering test suites with user-driven triggering that is performed during a standard usage of an application. *Marked traces* require users to explicitly enable runtime tracing before they activate a given feature of interest, and to explicitly disable tracing after the feature finishes executing. This mode of operation reduces the involved manual work, as compared to the approaches of Wilde and Scully [50], as it eliminates the need for implementing dedicated feature-triggering test suites. Furthermore, the approach of Salah and Mancoridis identifies not only the units of code contributing to single features, but also the ones shared among multiple features.

As demonstrated by Greevy and Ducasse [52], marked traces can be also captured by using GUI automation scripts instead of real users. The automation scripts used by their tool TraceScraper encode the actions needed for activating and tracing individual features of an application. While preparing automation scripts resembles preparing dedicated test suites of Wilde and Scully, the major advantage of automation scripts is

requiring no knowledge of an application's implementations details, which is certainly needed when implementing test cases.

In contrast, Featureous Location assumes manual marking of the feature-entry points directly in the source code of an application, whereas the remaining boundaries of features are discovered using dynamic analysis. As for annotating source code with feature-entry points, Featureous Location follows the principles of manual inspection and marking [49], [48], but requires marking only single methods and not whole features.

4.8 SUMMARY

Locating features in source code of existing applications is a prerequisite to code comprehension and modification during feature-oriented changes. In order to be practically applicable, a method for doing so needs to be simple to introduce to unfamiliar large-scale applications, independent of artifacts other than the source code and highly automated.

This chapter presented Featureous Location, a method for locating features in source code of Java applications that aims at providing these properties.

The proposed method defined the concept of feature-entry points and composed it with code annotation and execution tracing. The method was implemented using AspectJ and was applied to several open-source Java applications. Initial evidence for the effort-efficiency, coverage and accuracy of the method was presented. Additionally, Featureous Location was used for providing Java applications with runtime feature awareness.

5. FEATUREOUS ANALYSIS

This chapter uses the traceability links provided by Featureous Location as a basis for proposing a method for feature-oriented analysis of Java applications. At the core of this method, called Featureous Analysis, lies a firm conceptual framework for defining views over the correspondences between features and Java source code. The framework is instantiated in a form of feature-oriented views that are motivated by concrete needs of software evolution. Finally, the chapter presents four studies of feature-oriented comprehension and evolutionary modification of open-source Java applications that evaluate individual aspects of Featureous Analysis.

This chapter is based on the following publications:
[CSIS'11], [SEA'10], [AC'10]

5.1 OVERVIEW

The inherent grounding of features in the problem domains of applications is what makes them particularly important in comparison with other dimensions of concern, such as such as security, persistence, caching, etc. Nevertheless, the high-level designs of object-oriented applications rarely modularize and represent features explicitly. Hence, in order to support adoption of user-originated evolutionary changes, software

developers need to perceive source code from a perspective different than the one offered by the static structures of their applications.

This alternative perspective is provided by so-called feature-oriented analysis. *Feature-oriented analysis* of software, also known as *feature-centric analysis*, is the process of investigating the source code by considering features as first-class analysis entities [52], [51], [57], [4]. In other words, feature-oriented analysis is the process of projecting the feature dimension of concern onto the dominant dimension of decomposition [2] of an application.

This can be done by providing direct visualizations of the traceability links between features and source code of an application. A number of such visualizations were demonstrated to help developers to identify classes implementing a given feature and to identify features being implemented by a given class in unfamiliar applications [31], [99], [100]. Additionally, the results achieved by several non-feature-oriented approaches suggest that feature-oriented analysis could benefit from incorporating new types of visual metaphors and new analysis techniques [101], [102], [30].

However, defining a complete and evolutionary-grounded method for feature-oriented analysis remains difficult. This is because the existing works in the field are based on different conceptual bases and on different technical assumptions. This divergence makes it difficult to meaningfully compare the existing methods to one another, to combine them during evolutionary activities and to use them as bases for defining new methods.

This chapter proposes a comprehensive method of feature-oriented analysis of Java applications called *Featureous Analysis*. The goal of Featureous Analysis is to support comprehension and modification of features during evolutionary changes. This goal is schematically depicted in Figure 5.1.

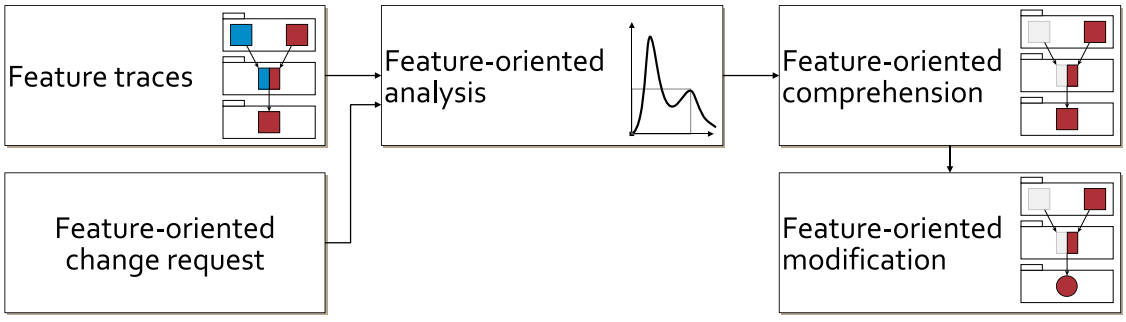


Figure 5.1: Overview of Featureous Analysis

Featureous Analysis proposes a unifying analytical framework that encompasses: (1) three complementary perspectives on traceability relations between features and code units, (2) three levels of code granularity and (3) three levels of analytical abstraction. Based on this framework, Featureous Analysis proposes a number of feature-oriented views and implements them within the Featureous Workbench tool. The individual

views are motivated by analyzing the needs of the evolutionary change mini-cycle [103].

The set of feature-oriented views proposed by the Featureous Analysis method is evaluated in four studies. Firstly, the method is used to perform a feature-oriented comparison of four modularization alternatives of the KWIC system [5]. Secondly, the method is applied to support comprehension of features in JHotDraw SVG [94]. Thirdly, the example of JHotDraw SVG is used to demonstrate applicability of Featureous Analysis to guiding source code comprehension and modification during adoption of an evolutionary change request. Lastly, the support of Featureous Location for several comprehension strategies is analytically evaluated based on the framework of Storey et al. [104].

5.2 THE METHOD

This section introduces the conceptual framework of Featureous Analysis and analyzes the need for its concrete configurations during the change mini-cycle.

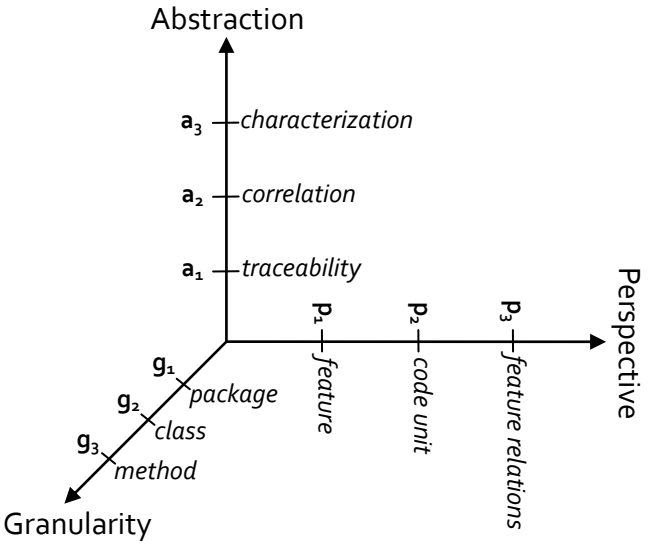


Figure 5.2: Three-dimensional conceptual framework of Featureous Analysis

5.2.1 Structuring Feature-Oriented Analysis

In order to structure and unify feature-oriented analysis, Featureous Analysis defines the conceptual framework depicted in Figure 5.2. This framework represents feature-oriented views as points in a three-dimensional space of *granularity*, *perspective* and *abstraction*. Using this framework, feature-oriented views can be specified as points in the three-dimensional configuration space. As it will be demonstrated, the individual

views have different properties and purposes during feature-oriented comprehension and modification of evolving applications.

Granularity

Firstly, the correspondences between features and code units can be investigated at different granularities. Based on the feature traces produced by Featureous Location, it is possible to obtain the following three granularities of traceability information:

- *Package granularity* g_1 depicts traceability between features and packages of a Java application.
- *Class granularity* g_2 depicts traceability between features and classes.
- *Method granularity* g_3 depicts traceability between features and methods.

These three levels of granularity are necessary for investigating applications with respect to both architectural and implementation-specific aspects, depending on the particular needs of a developer. In particular, adjustable levels of granularity are necessary for applying both the top-down and bottom-up comprehension strategies, where a developer starts by choosing a particular granularity level to focus on and incrementally refines or generalizes the scope of investigations.

Perspective

The second dimension of the proposed conceptual framework is the *perspective* dimension. As observed by Greevy and Ducasse in their two-sided approach [52], there exist two complementary perspectives on the traceability links between features and code units. The first of them characterizes features in terms of code units, while the second characterizes code units in terms of features. Apart from these two, a third feature-oriented perspective is possible – one that captures relations among features, or in other words characterizes features in terms of other features. These three perspectives are depicted in Figure 5.3.

As further depicted in Figure 5.3, Featureous Analysis associates the notion of scattering [2] with the feature perspective, and the notion of tangling [2] with the code unit perspective. Overall, the three perspectives are defined as follows:

- *Feature perspective* p_1 focuses on how features are implemented. In particular, it describes features in terms of their scattering over an application's code units.
- *Code unit perspective* p_2 shows how code units, such as packages, classes and methods, participate in implementing features. In particular, it captures the tangling of features in individual code units.
- *Feature relations perspective* p_3 focuses on inter-feature relations that are caused by the conjunction of scattering and tangling.

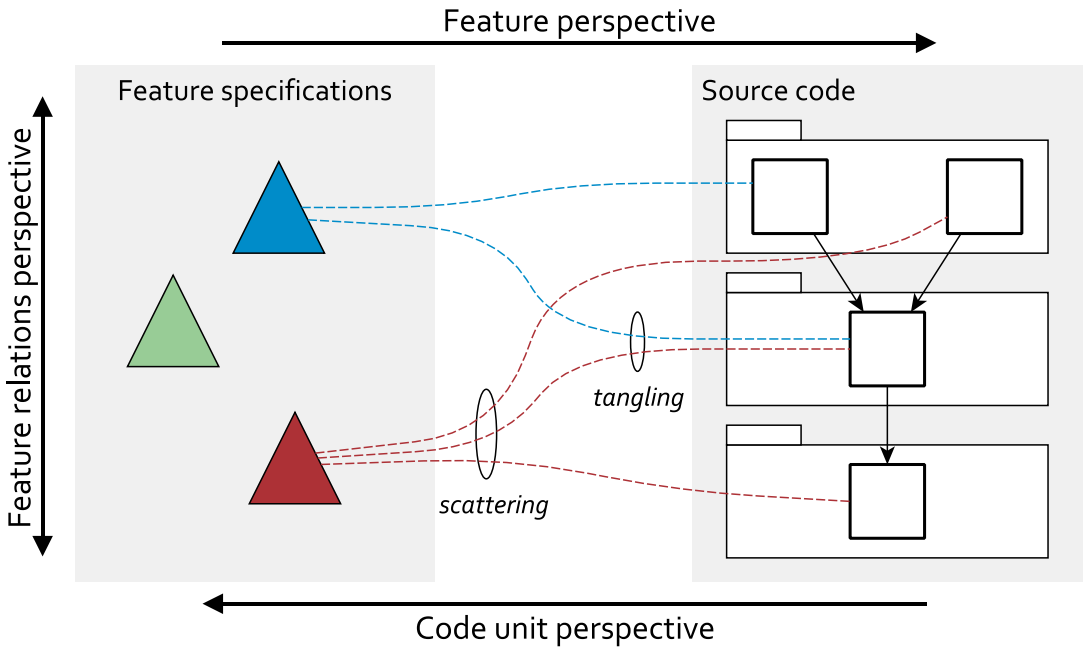


Figure 5.3: Three perspectives on feature-code traceability (adopted after [52])

These three perspectives match the needs of a number of evolutionary scenarios. For instance, the feature perspective could be used to aid identifying classes being the targets of an error correction request. After the error is corrected, the code unit perspective could be used to display the tangling of the modified class, and thereby to aid anticipating the impact of the modification on the correctness other features. A detailed analysis of the associated usage scenarios of this and the two other dimensions of the conceptual framework will be presented in Section 5.2.2.

Abstraction

The third dimension of the conceptual framework is the *abstraction* dimension. The purpose of the stratified levels of abstraction is to allow for limiting the amount of information simultaneously presented to the analyst and for investigating the complexity of feature-code traceability in an incremental fashion. The method distinguishes three levels of abstraction:

- *Traceability level* a_1 provides navigable traceability links between features and concrete source code units.
- *Correlation level* a_2 provides correlations between individual features and symbolically represented source code units.
- *Characterization level* a_3 shows high-level summaries of the overall complexity of feature-code mappings.

The three levels of abstraction make room for employing several, radically different visual metaphors to represent feature-code traceability. At the traceability level of

abstraction, display of the source code could be augmented with feature-oriented information. The correlation level could then use graph-based or matrix-based visualizations that would allow for direct correlation of features and source code units, without presenting the actual contents of source code files. Finally, the characterization level could be used to provide metrics and plots of the overall quality of modularization of features in an application.

Three-Dimensional Configuration Space

Together, the three dimensions of granularity, perspective and abstraction define a common representation scheme that allows Featureous to be equipped with a multitude of analytical views. Each of the possible feature-oriented analytical views can be objectively characterized using the three coordinates of the configuration space, regardless of the particular visual metaphor involved. For instance, view $\{g_2, p_1, a_3\}$ stands for a characterization of features in terms of classes, whereas view $\{g_1, p_1, a_2\}$ represents a correlation view from features to packages. Finally yet importantly, this uniform representation creates a basis for objectively comparing existing views and for explicitly identifying their roles during software evolution.

5.2.2 Feature-Oriented Views during Change-Mini Cycle

While the three-dimensional view configuration space of the conceptual framework allows for twenty-seven different views, not all of the possible configurations would be equally useful during software evolution. Therefore, this section focused on identifying particular configurations of views that can address the needs of comprehension and modification activities during software evolution.

In order to assess the evolutionary importance of feature-oriented analysis techniques, the *change mini-cycle* is used. Change mini-cycle is a detailed model of stages of change adoption during software evolution [103]. In the following, the change mini-cycle is analyzed from the perspective of a software developer who was given a feature-oriented modification task from a user. Therefore, the developer needs to comprehend and modify source code in a feature-oriented fashion, despite the lack of modularization of features in the target application. In this context, it is discussed how concrete view configurations from the three dimensions of the conceptual framework, can be used to support individual phases of the change mini-cycle. This perspective on the change mini-cycle serves as basis for designing concrete feature-oriented visualizations that will be presented in Section 5.3.

Table 5.1 presents the summary results of the analysis, being a correlation between the individual phases of the change mini-cycle and the levels of granularity, perspective and abstraction applicable to them. The details of analyzing the individual phases are discussed below.

Table 5.1: View configurations during change mini-cycle

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

Request for Change

Software users initiate the change mini-cycle by communicating their new or changed expectations in the form of a change request. During this process, they often use vocabulary related to features of applications, since users perceive applications through its identifiable functionality [4]. Henceforth, such a feature-oriented change request, which expresses the need for adding, enhancing or correcting features, becomes the basis for feature-oriented modification.

Planning Phase

The goal of the planning phase is to evaluate the feasibility of a requested change by understanding its technical properties and its potential impact on the application. This is not only a prerequisite for later implementation of the change, but can also be used for informed prioritization and scheduling of change implementation during multi-objective release planning [105].

During the process of *software comprehension* within the planning phase, a developer investigates an application in order to locate and understand the source code units implementing the 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. Narrowing-down the search space facilitates discovery of relevant initial focus points, which can be built upon in a feature-oriented fashion by navigating along the static relations between single-feature units, to gain understanding of a whole feature-sub-graph [106]. 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 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 code units (feature perspective) and

how modifying shared code units will affect the remaining features (code 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 an application's source code according to a given change request. This phase first prepares for accommodating a change by restructuring the source code, and then propagates the necessary changes among the source code units.

Restructuring involves 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 code units that have to be visited and modified, and reduces the impact on other features, so that features can evolve independently from each other. During restructuring, the feature perspective is used to identify boundaries of a feature under restructuring, and the code unit perspective is used to identify portions of code shared by features, which become candidates to splitting. Since restructuring focuses on working with 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 potentially be investigated at all granularities (i.e. package, class, method), because of the different possible granularities of feature tangling and scattering that has to be addressed.

Change propagation deals with implementing a change while ensuring syntactical consistency of the resulting source code. In order to implement a feature-oriented change, a developer needs to use the feature perspective to identify the concrete fine-grained (i.e. classes and methods) parts of a feature that need to be modified, as well as the parts of other features that can be reused. This involves applying the views at the levels of abstraction closest to source code (i.e. correlation and traceability). On the other hand, the code units perspective is essential to facilitate feature-oriented navigation over source code, to make feature-boundaries explicit and to give early indication of the consequences of modifying feature-shared code (doing so may cause inter-feature change propagation) and of reusing single-feature code (doing so may the increase evolutionary coupling among features and can be a sign of design erosion).

Verification and Validation

The goal of verification and validation phase is to ensure that the newly modified feature is consistent with its specification and that the remaining features of the

application were not affected. By investigating the fine-grained (i.e. class and method granularity) changes made to the source code from the code unit perspective, it is possible to determine which features were altered during change implementation. This is done at the traceability level of abstraction, where detailed traceability links from source code units to features are available. Each feature that was modified has to be considered as a candidate for validation. Using this method to identify the minimal set of features to be tested can be used to conserve project resources in the situations where the validation process is effort-intensive (e.g. manual testing, field-testing of control or monitoring systems).

Re-documentation

The re-documentation phase aims at updating the existing end-user's and developer's documentation of the application to reflect the performed changes. The goal of updating the developer's documentation is to capture information that can reduce the need for software comprehension during forthcoming evolutionary modifications. Featureous Analysis has a potential for reducing the need for manually maintaining the documentation of how features are implemented, because it can be reverse engineered on-demand in the form of various visualizations within the Featureous Workbench.

Summary

In total, the summary results in Table 5.1 show that out of 27 possible view configurations of the three-dimensional framework, 22 were identified to be applicable during the change mini-cycle. However, the degrees of their applicability differ. The distribution of the applicability degrees of individual configurations is depicted in Figure 5.4 as node sizes for each view in the three-dimensional configuration space.

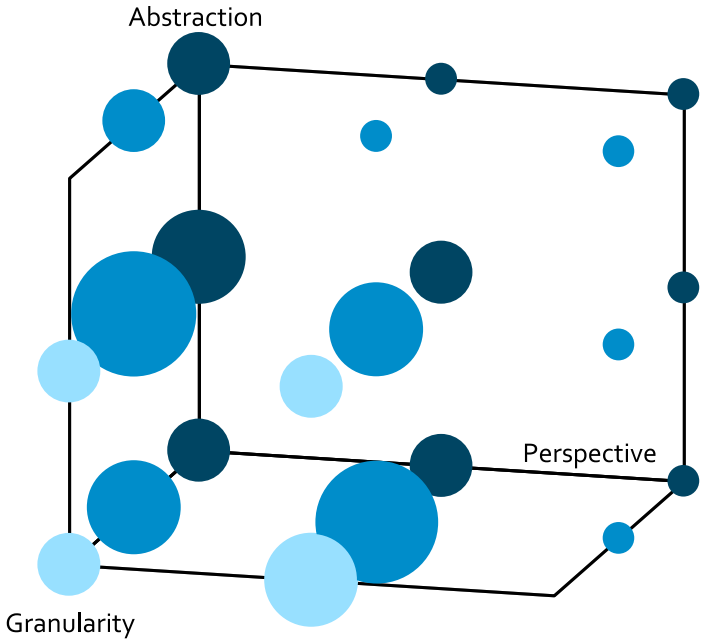


Figure 5.4: Applicability of view configurations within the change mini-cycle

The figure shows that two view configurations {g_2, p_1, a_2,} and {g_2, p_2, a_1} are applicable during four activities of the change mini-cycle. At same time, many of the views in the perspective p_3 and the abstraction level a_3 are dedicated to single change adoption activities. Finally, there exist several configurations that have not been mapped to any activities of the change mini-cycle – all of them located at the granularity g_3, i.e. {g_3, p_1, a_3}, {g_3, p_2, a_3}, {g_3, p_3, a_3}, {g_3, p_3, a_2} and {g_3, p_3, a_1}.

5.3 POPULATING FEATUREOUS ANALYSIS

Based on the applicability of individual view configurations during software evolution, Featureous Analysis proposes seven feature-oriented visualizations. The proposed views are arranged to cover the identified essential areas of the three-dimensional configuration space and to separate analytical concerns of the individual steps of the change mini-cycle. Thereby, the Featureous Analysis method aims at providing a means of guiding comprehension and modification efforts in a systematic and structured fashion. The importance of these two properties was demonstrated by Robillard et al. [107].

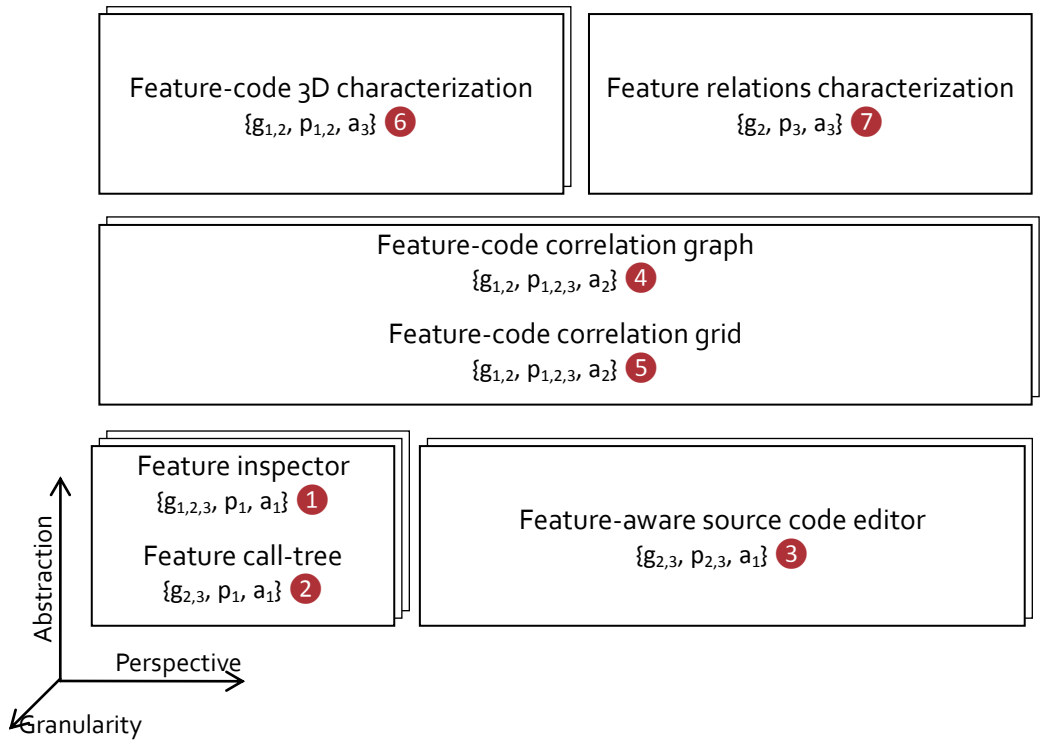


Figure 5.5: Unifying framework for feature-oriented analysis

Figure 5.5 presents the designed set of seven views. Most of the views make it possible for a user to switch them on-demand among multiple levels of granularity. An overview of the corresponding visualizations implemented in Featureous Workbench

is shown in Figure 5.6. The remainder of this section discusses the intents, visual designs and properties of each of the proposed feature-oriented views.

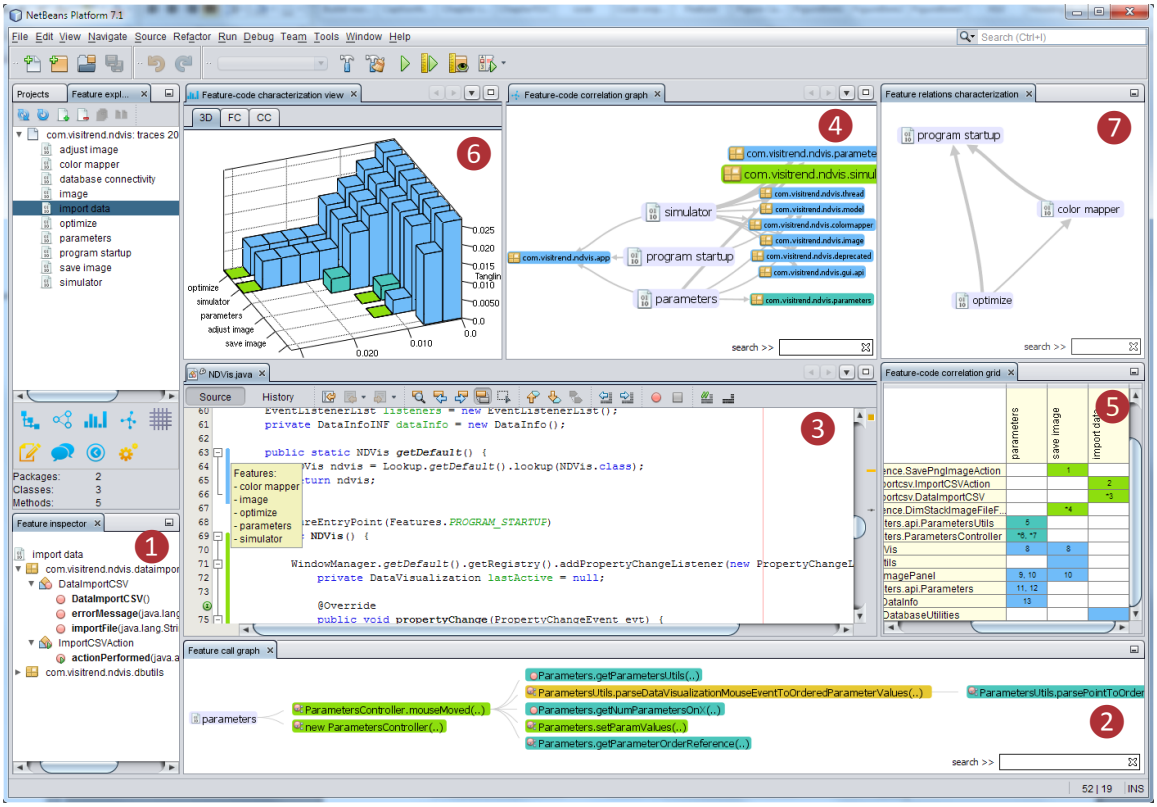


Figure 5.6: Feature-oriented views in the interface of Featureous Workbench

Feature Inspector

Feature inspector ❶ $\{g_{1,2,3}, p_1, a_1\}$ provides detailed navigable traceability links from features to concrete fragments of an application's source code. This view is presented in Figure 5.7.

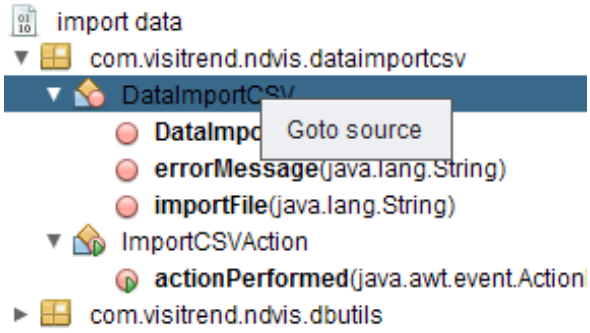


Figure 5.7: Feature inspector view

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 units in the NetBeans editor. The methods annotated as feature-entry points are marked in the hierarchy tree by a green overlay icon, due to their special role in features.

Feature Call-Tree

Feature call-tree (2) $\{g_{2,3}, p_1, a_1\}$ provides a tree-based visualization of the runtime call graphs of methods implementing features. Hence, this view provides an execution-based alternative to the hierarchical fashion of browsing features supported of the mentioned feature inspector view. Feature call-tree view is presented in Figure 5.8.

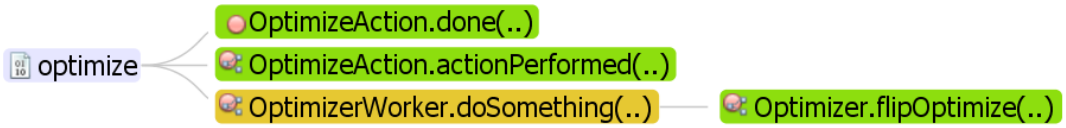


Figure 5.8: Feature call-tree view

The feature call-tree view is a graph containing a hierarchy of nodes that symbolize class-name-qualified methods. The edges among method-nodes stand for inter-method *call*-relations registered at runtime. Individual nodes of the graph can be selected to unfold their outgoing call relations and to automatically fold all control paths not involving a currently selected method. This allows one to selectively and incrementally explore individual control flows of a feature, starting from its feature-entry points. From this view, it is also possible to automatically navigate the NetBeans code editor to the source code units corresponding to a concrete node.

Because a tree-based visual representation of method call graphs is used, the provided view only approximates the actual control flows graph. In particular, the tree-based view does not preserve the information about cycles in the method call graphs. The projection from the potentially cyclic call graphs to the tree structure is handled by the Prefuse library [108] by filtering all call edges that would result in violating the tree structure.

Please note that the coloring scheme of this view, as well as of the remaining views of Featureous Analysis is designed to reflect how source code units participate in implementing features. Overall, the scheme assigns green to single-feature source code units and shades of blue to multi-feature code units. The details of the global coloring scheme of Featureous Analysis are discussed in Section 5.3.1.

Feature-Aware Source Code Editor

Feature-aware source code editor (3) $\{g_{2,3}, p_{2,3}, a_1\}$ extends the default source code editor of the NetBeans IDE with automated feature-oriented code folding and a colored sidebar visualizing traceability from code to features. This view is shown in Figure 5.9.

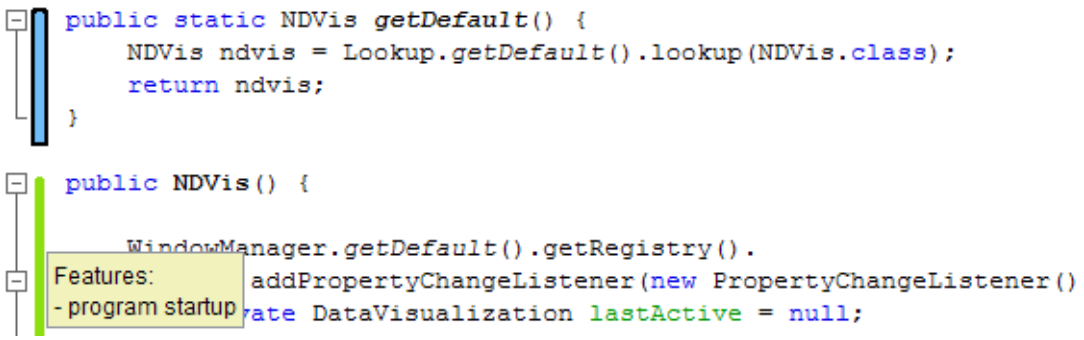


Figure 5.9: Feature-aware source code editor view

Feature-aware source code editor enhances the standard NetBeans IDE's editor with fine-grained traceability from individual methods and classes to features that they implement. This is done by using colored sidebars that symbolize participation of a method or a class in implementing features. For each of such sidebars, not only the color, but also the names of the features associated with a source code unit are available. Namely, the names of the associated features can be displayed on-demand in the form of a sidebar's tooltip.

Furthermore, the view alters the default folding mechanism of the editor to provide automatic visual folding of methods that do not contribute to a selected feature of interest. Thereby, it is possible to hide irrelevant code during comprehension and modification of a single feature. This is aimed at reducing the search space during feature-oriented tasks and at simplifying the reasoning about the possible impact of modifications on features other than the feature of interest.

Feature-Code Correlation Graph and Grid

Feature-code correlation graph 4 $\{g_{1,2}, p_{1,2,3}, a_2\}$ and *feature-code correlation grid* 5 $\{g_{1,2}, p_{1,2,3}, a_2\}$ enable detailed investigations of the correspondences between source code units and features. These views are presented in Figure 5.10.

The feature-code correlation graph is a graph-based visualization of traceability links (edges) between features (nodes) and classes or packages (nodes). This view builds upon the feature-implementation graph proposed by Salah and Mancoridis [51]. The original formulation is extended with the coloring scheme defined in Section 5.3.1, a layer-based layout algorithm and edge-thickness-based indication of the number of contained source code units that participate in the traceability relations with individual features. Furthermore, the heights of the source code unit nodes reflect the numbers of their contained finer-grained units (e.g. the height of a package node reflects the number of the classes that it contains).

As demonstrated by the two examples of the feature-code correlation graph view in Figure 5.10, the layout algorithm arranges nodes in layers according to their relations to a selected node (which is the leftmost node of the graph). This algorithm is

selection-driven, so that it is possible to dynamically switch between the code unit and feature perspectives by focusing the graph on a particular node type of interest.

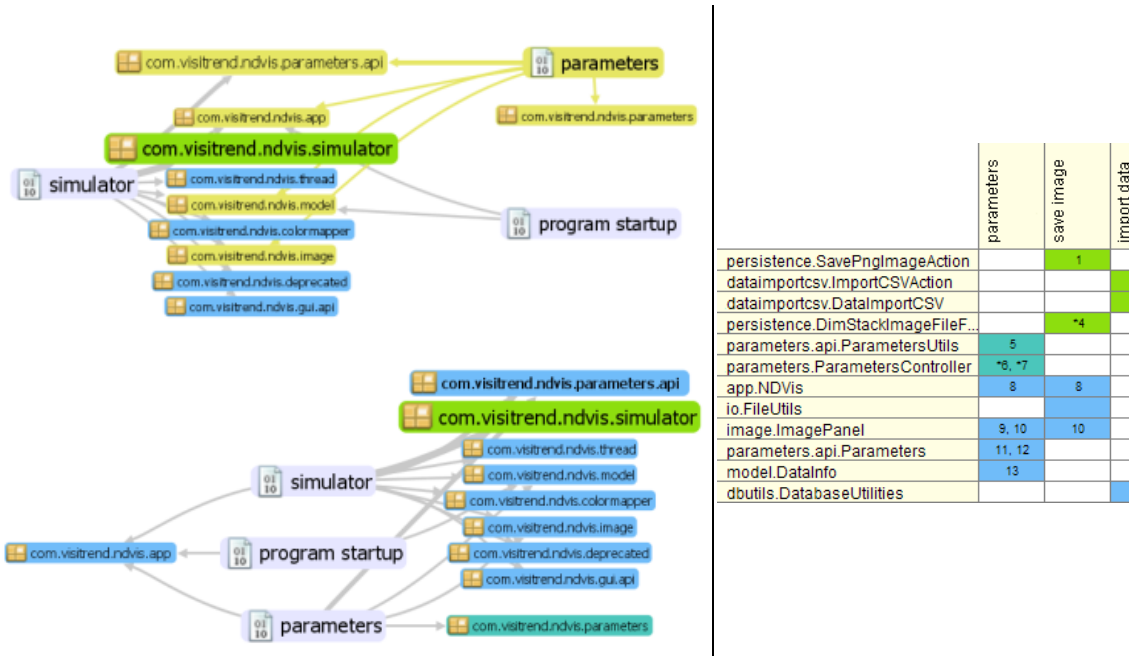


Figure 5.10: Feature-code correlation graph and grid views

By enabling selective bi-directional investigation of traceability links, this view can be used to depict scattering and tangling of features. The source code units that tangle two features of interest can be identified by combining the properties of the layout and a mouse hover-driven highlighting mechanism. The highlighting mechanism marks all the nodes directly related to a node being hovered over in yellow. This usage can be seen in the first example of the view presented in Figure 5.10, where it is used to identify four packages common to the SIMULATOR and PARAMETERS features.

As can be further seen in Figure 5.10, the traceability information is also available in the form of a correlation matrix. This view correlates features with packages or classes using a matrix-based representation. Additionally, in the case of classes, the matrix displays the concrete identifiers of the objects of a given class used by features. The sorting of the matrix's rows and columns is done according to the scattering of features and tangling of code units.

Feature-Code 3D Characterization

Feature-code 3D characterization 6 $\{g_{1,2}, p_{1,2}, a_3\}$ serves as an abstract summary of the complexity of feature-code relations. This view is presented in Figure 5.11.

The feature-code 3D characterization view visualizes features using a three-dimensional bar chart, where each feature is depicted as a series of bars. In turn, each of the bars represents a single package or class used by a feature. The names of the source code units that the bars represent can be displayed on-demand as tooltips.

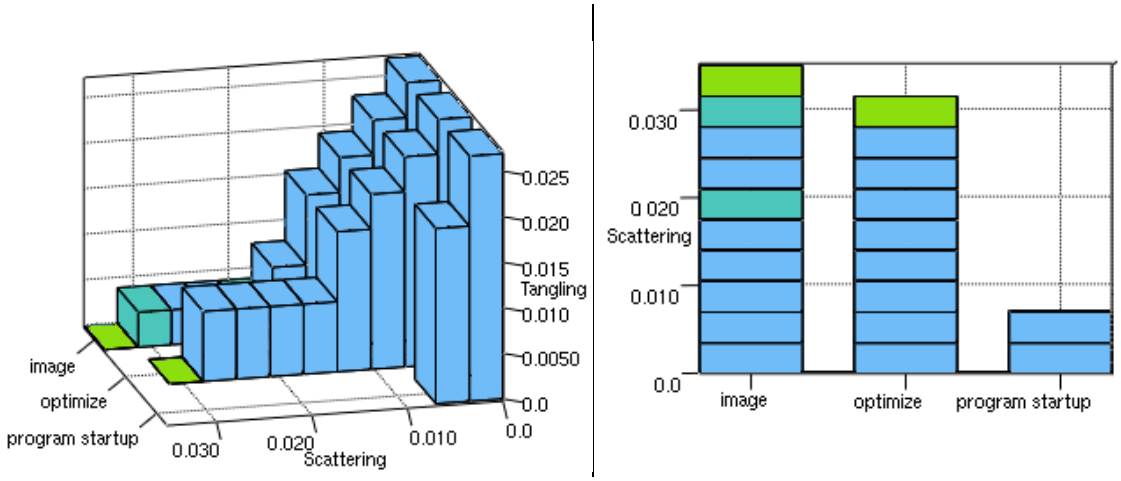


Figure 5.11: Two rotations of the feature-code 3D characterization view

The number of bars displayed for each feature reflects the scattering of its implementation measured by the scattering metric proposed by Brcina and Riebisch [58]. As introduced in Chapter 1, scattering corresponds to the size of the search space during a feature-oriented change adoption and is associated with the phenomenon of delocalized plans [26].

The height of the individual bars represents tangling of their corresponding code units, which is measured with the tangling metric proposed by Brcina and Riebisch [58]. Tangling indicates the potential inter-feature change propagation and is associated with the phenomenon of interleaving [28].

The view is aimed at depicting two general characteristics of the distribution shapes representing features: the regularity of how features are implemented and the sharpness of the division between single-feature and multi-feature code units in an application. Furthermore, the view can be flexibly rotated to focus on the characterization of feature scattering.

Feature Relations Characterization

Feature relations characterization 7 $\{g_2, p_3, a_3\}$ relates features to each other with respect to how they depend on the objects created by other features and how they exchange data by sharing these objects with one another. This view is presented in Figure 5.12.

The feature relations characterization view is a graph-based representation of dynamic relations that builds upon the *feature-interaction graph* of Salah and Mancoridis [51]. Features, denoted as vertices, are connected by dashed edges in case of common usage of objects at runtime. A directed, solid edge is drawn if a pair of features is in a producer-consumer 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.

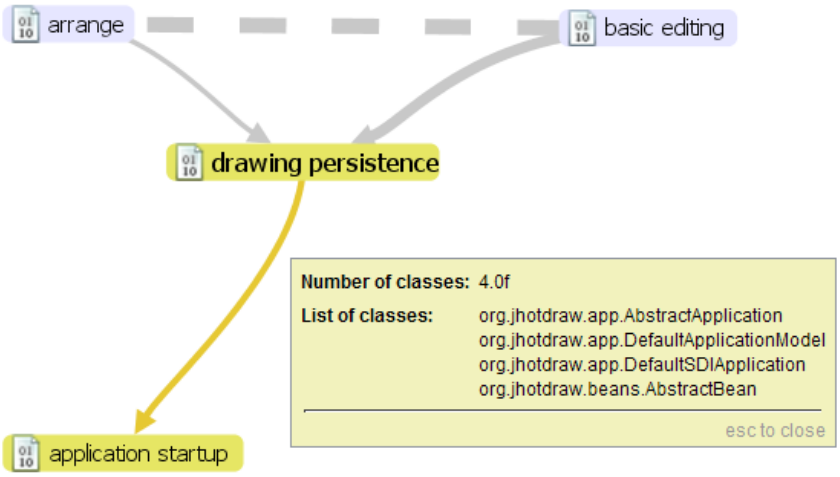


Figure 5.12: Feature relations characterization view

The formulation of Salah and Mancoridis is extended in several ways. The thickness of edges reflect the number of classes whose objects are being shared between features – the more classes whose objects are shared, the thicker the edges. Furthermore, the view makes it possible to automatically re-layout the graph visualization around a selected feature of interest. Lastly, individual classes participating in a relation between two features can be displayed on-demand as tooltips over the graph edges.

5.3.1 Affinity Coloring

The proposed feature-oriented analytical views are equipped with a common coloring scheme called *affinity coloring*. As originally proposed by Greevy and Ducasse [52], the purpose of an affinity coloring is to indicate the type of feature-oriented reuse that source code units experience. They proposed to realize this vision by assigning colors to code units by applying predefined thresholds to distinguish between several extents of tangling.

Featureous Analysis takes a different approach to defining affinity categories. The affinity scheme proposed by Featureous Analysis is based on identification of so-called *canonical features* [109], [64] that serve as centroids for grouping features with maximally similar implementations. At the same time, the individual canonical features are chosen so that they are maximally dissimilar to one another. As demonstrated by Kothari et al., this can be used to reduce the number of features that need to be understood during feature-oriented analysis, because the canonical features are representative for the features contained in their canonical groups. To automatically compute canonical groups of features, Featureous adopts the original algorithm proposed by Kothari et al.

By employing the concept of canonical groups, the affinity scheme of Featureous Analysis is defined as consisting of two top-level categories: *multi-feature source code*

units and *single-feature source code units*. The multi-feature source code units are further divided into three categories as follows:

Multi-feature source code units

Core units ■{R=35, G=126, B=217} are the source code units that are reused by all canonical feature-groups. The high degree of reuse of such units among feature-groups indicates their membership in the very core infrastructure of an application.

Inter-group units ■{R=112, G=188, B=249} are the code units that are reused among multiple, but not all, canonical feature-groups. Hence, this category contains classes whose reuse is neither localized to a single canonical group, nor extensive enough to be categorized as *core*. From the evolutionary point of view, it would be migrate such classes towards either being universally reusable, or towards confining their usage to single feature-groups.

Single-group units ■{R=75, G=198, B=187} are the code units that are used by multiple features, exclusively within a single canonical feature-group. Such units form a local reusable core of a canonical group of functionally similar features.

Single-feature units ■{R=141, G=222, B=14} are the code units that participate in implementing of only one feature in an application.

5.4 EVALUATION

This section demonstrates applicability of Featureous Analysis to supporting feature-oriented comprehension and modification of existing applications. This is done by applying Featureous Analysis to three evolutionary scenarios that are designed to cover the majority of the phases involved in the change mini-cycle. The scenarios include comparative analysis of feature modularity, feature-oriented comprehension and feature-oriented modification of source code. In addition, an analytical evaluation of the method's support for program comprehension is presented. Hence, the contents of this section are structured as follows:

- Section 5.4.1 applies Featureous Analysis to comparative analysis of four modularization alternatives of the classical example of KWIC system [5].
- Section 5.4.2 applies Featureous Analysis to feature-oriented comprehension of the unfamiliar codebase of the JHotDraw SVG application.
- Section 5.4.3 applies Featureous Analysis to feature-oriented modification of the JHotDraw SVG application.

- Section 5.4.4 uses the evaluation framework proposed by Storey et al. [104] to analytically evaluate the support for diverse comprehension modes in Featureous Analysis.

Please note that apart from the evaluations presented in this chapter, Featureous Analysis will be applied in Chapter 6 in extensive analysis-driven restructuring of an open-source application.

5.4.1 Feature-Oriented Modularity Assessment

This section presents an application of Featureous Analysis to four modularization alternatives of the KWIC system.

KWIC (being an abbreviation for Key Word In Context) is defined as an index system that accepts an ordered set of lines, consisting of words and characters, and outputs a listing of all "circular shifts" of all lines in alphabetical order. A set of circular shifts of a line is produced by repeatedly removing the first word and appending it at the end of line.

This simple system was used by Parnas [5] to compare evolutionary properties of two modularization alternatives: *shared data* modularization and *abstract data type* modularization based on information hiding. The comparison was done by considering the locality of changes during several change scenarios, such as changes in the algorithms or in the data representation. The investigations of Parnas were later extended with additional modularization alternatives, such as implicit invocation, and additional change scenarios, such as change in functionality, by Garlan et al. [39] and Garlan and Shaw [23]. In particular, Garlan and Shaw formalize all the proposed modularization alternatives in the form of architectural diagrams and summarize their support for the existing change scenarios.

In the context of the KWIC problem, the goal of this study is to demonstrate that Featureous Analysis can be used to discriminate between the existing four modularization alternatives with respect to how they modularize features. Furthermore, it is hypothesized that the conclusions of Garlan and Shaw about the utility of different modularizations during changes to functionality can be reproduced using Featureous Analysis.

Please note that for convenience of the reader, the architectural diagrams and the descriptions of the four modularization alternatives of KWIC are reproduced after Garlan and Shaw [23] in Appendix A3.

Implementing Four Modularization Alternatives of KWIC

Based on the architectural diagrams provided by Garlan and Shaw, the four known modularization alternatives of KWIC were implemented. This was done by representing the modules of the designs as Java classes and by ensuring the

dependencies among the created classes correspond to the dependencies depicted by Garlan and Shaw.

Identifying and Locating Features

Prior to using Featureous Analysis, Featureous Location was performed on KWIC to produce the necessary traceability links. This was done by inspecting the original functional specification of the KWIC system. Based on this, four features were identified: READ_INPUT, SHIFT_CRICULAR, ALPHABETIZE and WRITE_OUTPUT. After annotating the entry points of these features for all four modularizations and tracing the runtime execution of KWIC, four sets of feature traces were obtained.

Results

The four modularization alternatives of KWIC were investigated using two views of Featureous Analysis: the *feature-code correlation graph* view and the *feature-code 3D characterization* view. These two views were applied at the granularity of classes to investigate the phenomena of feature scattering and tangling in the investigated applications. Figure 5.13 presents the obtained results.

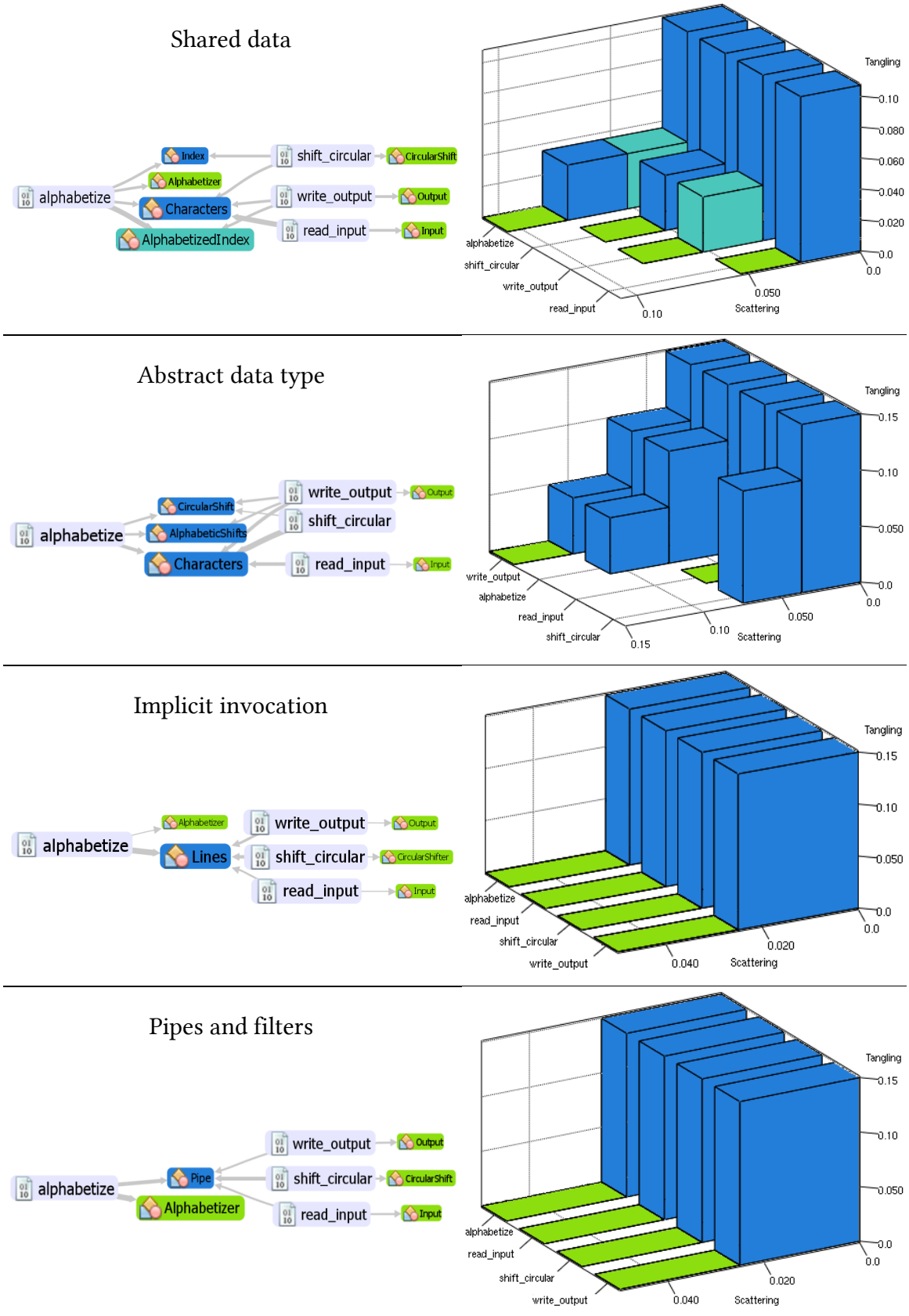


Figure 5.13: Feature-oriented views of the four modularizations of KWIC

At a first glance, Figure 5.13 indicates that none of the KWIC modularization alternatives completely separates and localizes features. Given that features in an application have to communicate by sharing data, such a result should be expected. More interesting are the detailed differences between how the individual modularizations of KWIC implement features.

Firstly, Figure 5.13 reveals that the *shared data* and the *abstract data type* modularizations implement features significantly differently than the *implicit invocation* and *pipes and filters* modularizations. The former two involve visibly more modules in implementing features, thus contributing to feature scattering, which increases the number of modules potentially affected during feature-oriented changes. Moreover, the former two modularizations contain more multi-feature core modules. Such tangled modules are a potential source code of inter-feature change propagation during feature-oriented changes.

Secondly, the *implicit invocation* and *pipes and filters* modularizations exhibit the same feature-oriented characteristics despite of relying on radically different modularization criteria. While *implicit invocation* makes the processing algorithms of individual features work independently on a common line-based representation of data, the *pipes and filters* modularization arranges features as processing steps connected by data-flow pipes, where data exchange format is specific to each pipe and the pair of features on both ends. As a result, both these modularizations exhibit high regularity of feature implementations; they make features consist of one single-feature module that depends on one multi-feature core module.

Thirdly, Figure 5.13 shows that the *abstract data type* modularization completely tangles two of its features. This tangling appears is particularly problematic, as it affects the features that provide the essential algorithms of KWIC, namely alphabetizing and circular shifting. This might suggest that any additional processing feature (e.g. word filtering) that would be added to this modularization would experience complete tangling with the remaining features.

In order to quantify the presented observations, the phenomena of scattering and tangling of the four modularization of KWIC were measured with the metrics proposed by Brcina and Riebisch [58]. By doing so it is possible to compare the modularization alternatives against one another. Moreover, such measurement makes it possible to objectively compare the outcomes of Featureous Analysis with the assessment of Garlan and Shaw [23] about the support for functional changes in the individual modularizations of KWIC. These results are summarized in Table 5.2.

Table 5.2: Feature-oriented properties of the four KWIC modularizations

Property	Shared data	Abstract data type	Implicit invocation	Pipes and filters
Scattering	0.29	0.35	0.20	0.20
Tangling	0.18	0.30	0.15	0.15
Change in function [23]	+	-	+	+

As shown in Table 5.2, the *abstract data type* modularization exhibit the highest extent of feature scattering and tangling. This modularization is also concluded by Garlan and Shaw not to support functional changes. The second-worst modularization according to the metrics is the *shared data* modularization, which was earlier found to significantly tangle features despite having one single-feature module for each feature. While Garlan and Shaw evaluate this modularization positively with respect to supporting functional changes, they hint at some problems associated with it: *"(...) changes in the overall processing algorithm and enhancements to system function are not easily accommodated"* [23]. Finally, as recognized both by Featureous Analysis and by Garlan and Shaw, the two last modularizations are particularly well suited to accommodating changes to functionality.

Summary

This study applied Featureous Analysis to four classic modularization alternatives of the KWIC system. The presented results demonstrate that Featureous Analysis provides means for differentiating alternative modularizations of a single application with respect to how well they modularize features. Furthermore, the obtained analytical results were quantified using the metrics of scattering and tangling and compared to analytical assessments of Parnas, Garlan et al. and Garlan and Shaw. A close correspondence between these two set of assessments was observed.

5.4.2 Feature-Oriented Comprehension

This section applies Featureous Analysis to guiding feature-oriented comprehension of an unfamiliar codebase. The subject is the SVG vector graphics editor application built on top of the JHotDraw 7.2 framework [94]. JHotDraw SVG consists of 62K lines of code and contains a significantly high number of features to be considered a realistic application scenario. Prior to conducting this study, the author had no significant exposure to the design and the implementation details of the application.

The goal of the presented feature-oriented analysis is to identify and understand the nature of "hot-spots" in the correspondences between features and source code. Such hot-spots include highly tangled packages and classes, as well as highly scattered features. This investigation is motivated by evolutionary repercussions of these two phenomena.

The presented feature-oriented analysis of JHotDraw SVG is performed in a top-down fashion. Namely, the study starts on the highest level of abstraction and concretizes investigations. In order to keep the scope of the study manageable, the analysis focuses on selected examples of hot-spot features, classes and inter-feature relations.

Identifying and Locating Features

As a prerequisite to using Featureous Analysis, the Featureous Location method was applied. Since appropriate documentation of the application's functionality was not available, it was necessary to identify the features. This was done by inspecting user-triggerable functionality of the running application that is available through the graphical user interface elements like the main menu, contextual menus and toolbars. As a result, 29 features were identified, whose 91 feature-entry point methods were then annotated in the application's source code (e.g. the BASIC EDITING feature can be triggered by both invoking *copy* and *paste* commands from JHotDraw SVG's main menu). A detailed list of all identified features can be found in Appendix A2.

A set of feature traces was obtained by manually triggering each identified feature at runtime in the instrumented application. These traces formed a basis for feature-oriented analysis of JHotDraw SVG.

Feature-Code 3D Characterization

The feature-oriented analysis of JHotDraw SVG was initiated by investigating a high-level summary of how classes participate in implementing features. This is done by applying the *feature-code 3D characterization* view on the granularity of classes. The result of doing so is presented in Figure 5.14.

Figure 5.14 reveals that there exists a high degree of both scattering and tangling among the features of JHotDraw SVG at the granularity of classes. The three most scattered hot-spot features are MANAGE DRAWINGS, SELECTION TOOL and BASIC EDITING. The three most tangled hot-spot classes that are reused by nearly all features and multiple canonical groups are `org.jhotdraw.draw.AbstractFigure`, `org.jhotdraw.draw.AttributeKey` and `org.jhotdraw.draw.AbstractAttributedFigure`.

As can further be observed in Figure 5.14, features generally contain relatively small amounts of single-feature classes. In particular, 15 out of all 29 features contain no single-feature classes. Such purely crosscutting features (e.g. PATH EDITING and CANVAS) are either composed of fragments of other features or are themselves completely reused in execution of other features. Most of the code sharing among features occurs over inter-group classes, rather than the core classes. The nearly continuous shapes of the profiles of individual features depict the diversity of reuse patterns of the involved classes. Overall, the high degree of scattering and tangling among features indicates that JHotDraw SVG was not designed with modularization of features in mind.

At this stage, the focus of further investigations was decided to center on SELECTION TOOL hot-spot feature that deals with selecting figures, and BASIC EDITING hot-spot feature that deals with basic editing actions such as copying and pasting figures. As for classes, the further investigations focus on the `org.jhotdraw.draw.AbstractFigure` and `org.jhotdraw.draw.AbstractAttributedFigure` hot-spot classes.

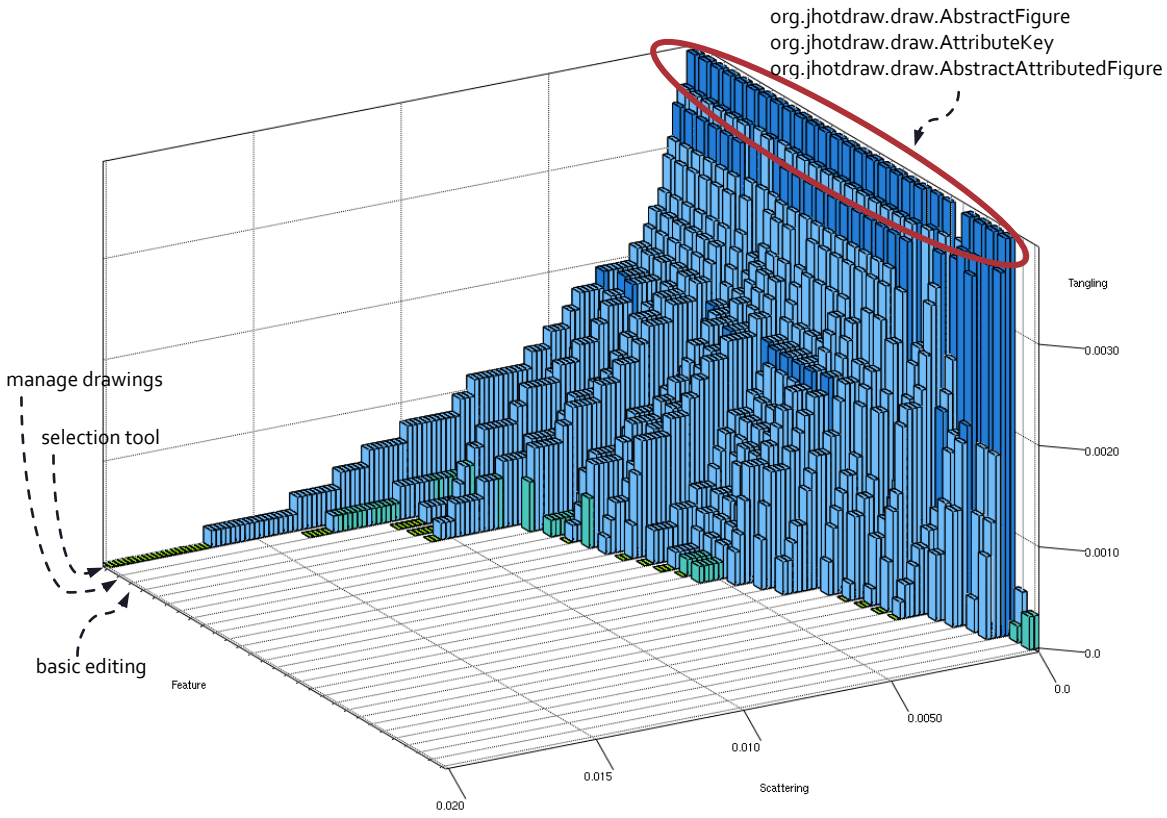


Figure 5.14: Feature-package 3D characterization of JHotDraw SVG

Feature-Relations Characterization

The feature relations characterization view was used to investigate dynamic relations between the two features of interest. The results of applying this view are displayed in Figure 5.15. There, all the remaining features of JHotDraw SVG are represented by the [OTHER FEATURES] group.

Figure 5.15 shows that there exist bi-directional dynamic dependencies between SELECTION TOOL and BASIC EDITING, as well as between these two features and other features of SVG. A closer inspection of the dependency of SELECTION TOOL on BASIC EDITING (highlighted in Figure 5.15) revealed that BASIC EDITING instantiates both the `AbstractFigure` and `AbstractAttributedFigure` objects for usage in SELECTION TOOL. This reinforces a possible intuitive interpretation of these classes' names. Namely, these two classes appear to be the programmatic representations of the graphical figures in JHotDraw SVG. The objects of these classes can be created by basic editing actions, such as copying and pasting. The thereby created figures can then be selected by

SELECTION TOOL in order to be further manipulated. Hence, an unanticipated dynamic dependency between the two features emerges.

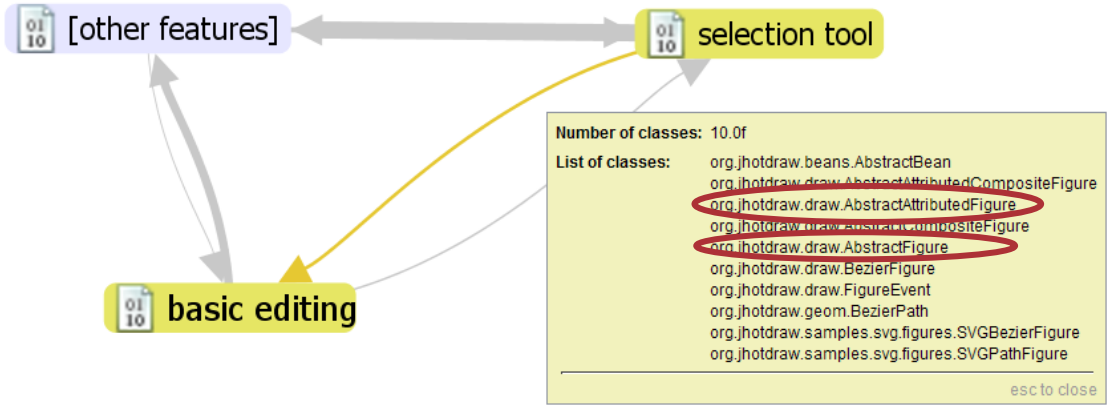


Figure 5.15: Feature relations characterization of JHotDraw SVG

Correlation

The *feature-code correlation graph* was used to identify tangled hot-spot packages of JHotDraw SVG.

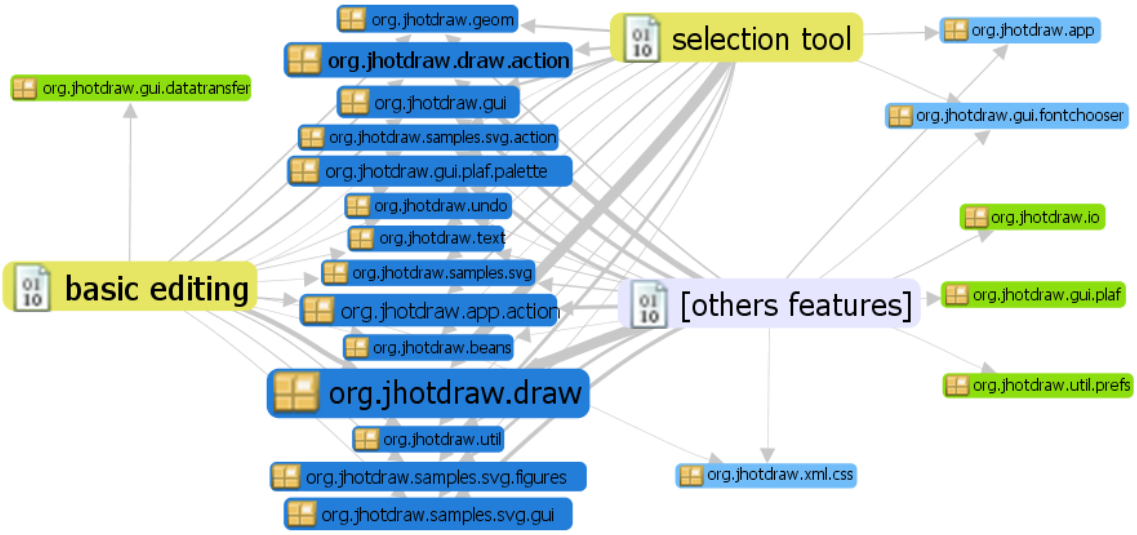


Figure 5.16: Feature-package correlation graph of JHotDraw SVG

The feature-package correlation graph shown in Figure 5.16 indicates that out of the two features of interest, only BASIC EDITING has a single-feature package, named `org.jhotdraw.gui.datatransfer`. Furthermore, it can be seen that the package containing the highest number of tangled classes is the `org.jhotdraw.draw` package. This complex package encloses both the `AbstractFigure` and `AbstractAttributedFigure` classes, as well as 100 other classes that represent a wide spectrum of concepts provided by the JHotDraw framework (e.g. `ArrowTip`, `BezierTool`, `Layouter`, `FigureEvent`). The size of this

package, as well as its tangling and the diversity of contained classes deem it to be a major structural hot-spot of JHotDraw SVG.

Traceability

Finally, the *feature-aware source code editor* view was used to investigate how the implementations of the `AbstractFigure` and `AbstractAttributedFigure` core classes contribute to implementing SELECTION TOOL on BASIC EDITING features.

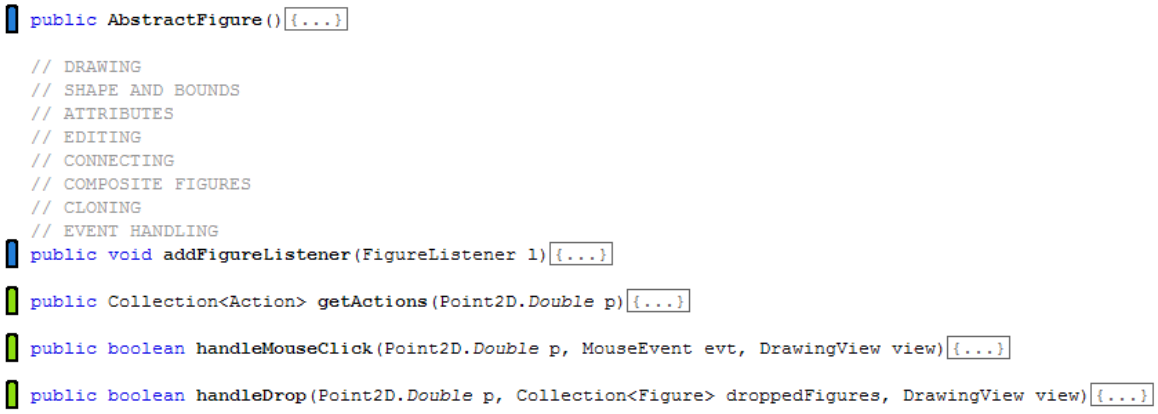


Figure 5.17: Feature-aware source code view of the `AbstractFigure` class

The feature-oriented investigation of the `AbstractFigure` class, an excerpt of which is presented in Figure 5.17, revealed a number of interesting details. `AbstractFigure`, being a class that provides base implementation of figures in JHotDraw SVG, is used by most of the application's features. Yet, there exist methods that are used exclusively by the SELECTION TOOL feature. These methods were found to be concerned with selection-specific functionalities, like handling mouse events, dragging figures, getting a list of available figure's actions, etc. The presence of such selection-specific methods in a base class suggests that the selection mechanism is one of the core non-removable functionalities offered by the JHotDraw framework.

Furthermore, the inspection revealed that the `AbstractAttributedFigure` class inherits from the `AbstractFigure` class to extend it with a map of attributes, whose keys are represented by the mentioned earlier `org.jhotdraw.draw.AttributeKey` class. With the help of colored sidebars of setter and getter methods that encapsulate the access to the attribute maps, it was determined that several features can read the attributes, but only a subset of them is allowed to modify the attributes. Namely, the read-write access to attributes is granted only to tool features (e.g. SELECTION TOOL, LINE TOOL), editing features (e.g. BASIC EDITING, ATTRIBUTE EDITING), DRAWING PERSISTENCE and MANAGE DRAWINGS. In addition to these features, read-only access is granted to palette features (e.g. STROKE PALETTE, ALIGN PALETTE), UNDO REDO and VIEW SOURCE.

Finally, yet importantly, the source code of the `AbstractFigure` class shown in Figure 5.17 was observed to contain an interesting source comment preceding the `addFigureListener` method. This comment resembles a list of names of SVG's

functionalities, such as: “drawing”, “shape and bounds”, “attributes”, “editing”, “connecting”, “composite figures”, “cloning”, “event handling”. These names closely correspond to the names of SVG’s features identified in this study. This comment is an important evidence of a developer’s effort to maintaining the traceability to features that can observe figures. This sharply demonstrates the need for presence of explicit traceability links between features and source code.

Summary

This section applied Featureous Analysis to identify and understand several hot-spot correspondences between features and source code in the JHotDraw SVG application. Application of feature-oriented analysis, as supported by Featureous Analysis, was found to provide a useful perspective on unfamiliar source code. Featureous Analysis was demonstrated to aid identification of complexities in features and to guide their incremental understanding.

5.4.3 Feature-Oriented Change Adoption

This section applies Featureous Analysis to adopting an evolutionary change to JHotDraw SVG. The structure of the presented study focuses on the phases of the change mini-cycle that deal with modification of source code.

Request for Change

To initiate change adoption, a *request for change* was formulated. The proposed request is to modify the EXPORT feature of JHotDraw SVG to automatically include a simple *watermark text* in the drawings being exported. Similar functionality is commonly found, for instance, in trial or demo versions of applications. The watermark should be included uniformly for all possible export formats.

Planning Phase

The intent of the *planning phase* is to get an impression of the effort needed to perform a modification task. This was done by using the *feature-code 3D characterization* view to investigate the target EXPORT feature that is responsible for exporting drawings from the application’s canvas to various file formats.

Figure 5.18 shows the resulting feature-code 3D characterization at the granularity of class for EXPORT and two other features of SVG. There, it can be seen that the EXPORT feature is scattered over only 15 classes. These 15 classes constitute the maximum scope that will need to be investigated to carry out the requested task. Unfortunately, this feature contains only one single-feature class, which indicates a high chance for any modifications made to EXPORT to propagate to other features.

Another interesting feature shown in Figure 5.18 is the TEXT TOOL feature. Since this feature is responsible for drawing text-based shapes on the editor’s canvas, it may contain classes that need to be reused to programmatically draw a textual watermark

on a drawing being exported. The relatively low scattering of this feature signals a relatively small scope of the forthcoming search for the classes that have to be reused.

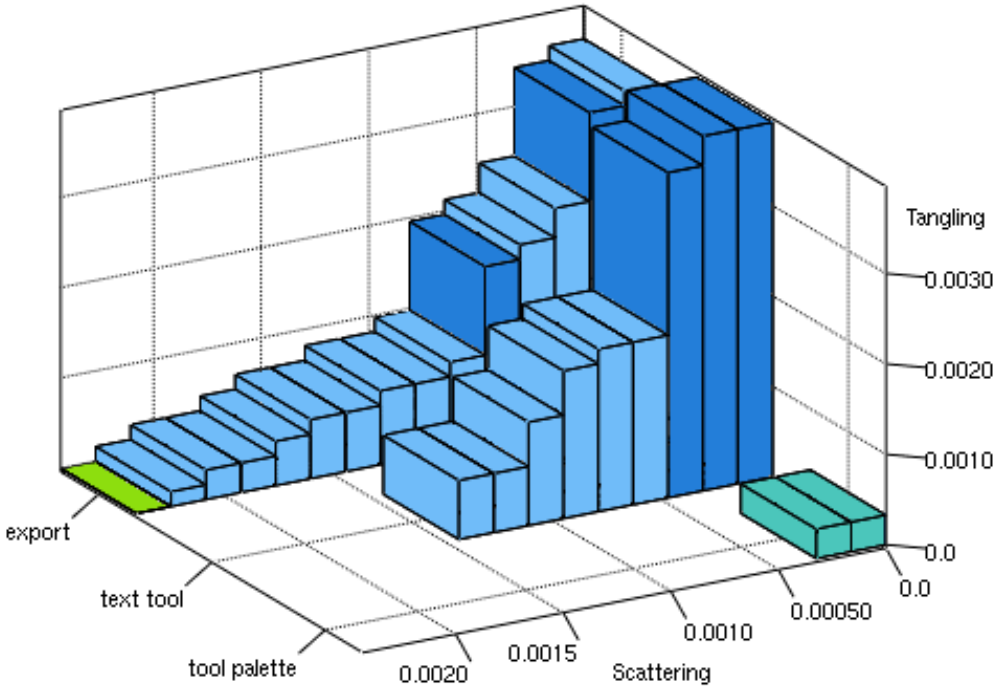


Figure 5.18: Feature-class 3D characterization of three features of JHotDraw SVG

Change Implementation

The aim of change implementation was to inspect the EXPORT and TEXT TOOL features in search of the concrete classes that needed to be modified and reused to implement the change request.

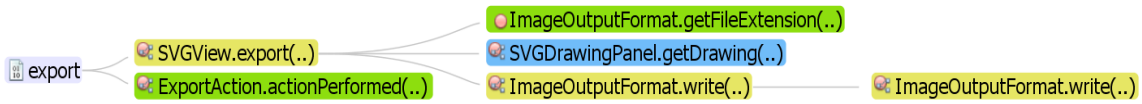


Figure 5.19: Call-tree view of the export feature in JHotDraw SVG

The feature call-tree view on the EXPORT feature was used to identify the classes that has the responsibility of accepting a drawing and saving it into a file. As shown in Figure 5.19, the export feature consists of a chain of inter-class method invocations, most of which appear to be good candidates for inserting a watermark into a drawing, before it is saved to a file. An important advantage of this perspective on method calls is not only narrowing the search space, but also visual removal of the polymorphism-based layers of indirection that separate the codebase of the JHotDraw framework from the SVG application. This way it was possible to easily identify the concrete framework classes that carry out the export functionality, despite of them being a part of polymorphic indirection-based variation points of JHotDraw.

In order to confine the impact of change, the `SVGView` class was chosen for modification. This class handles rendering the internal models of drawings onto the screen and it the `export` to initiate a chain of invocations that saves the underlying models to a file. Hence, the `export` method can be used to implement the requested change, by making it insert a watermark into the model before the drawing is being saved, and to remove it from the model afterwards.

After locating the to-be-modified method in the EXPORT feature, the TEXT TOOL feature was investigated to identify a 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, the candidate class was identified. The identified class is named `SVGTextFigure` and it is responsible for representing textual figures in JHotDraw.

```
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);
```

Features:
- export

Figure 5.20: Feature-aware source code editor for `SVGView` class

Based on the gained understanding, the change request was implemented. The resulting source code is shown in Figure 5.20. The performed modifications consisted of adding a simple watermark to the drawing's model (lines 359, 360) before it is written to the output file by `format.write(...)`. Thereafter, the watermark is removed (line 362), so that the original state of the drawing is restored for display in the UI of JHotDraw SVG.

Validation and Verification

After implementing the change, it became necessary to *validate* that the modification was correct and that it did not affect the correctness of other features in the application. Firstly, it was confirmed that a watermark is correctly added when exporting images by triggering the updated EXPORT feature in the running application. Secondly, it was investigated whether the correctness of any other features of SVG might have been affected by the performed modification. This was done by inspecting the affinity of the modified EXPORT method in the feature-aware code editor, as shown in Figure 5.20. This revealed that even though four other features are reusing the `SVGView` class, the modified method is used exclusively by the EXPORT feature. This

indicated that no other features could have been directly affected by the implemented change. This hypothesis was confirmed by interacting with the running applications.

Summary

This study applied Featureous Analysis to the scenario of supporting adoption of a feature-oriented change request. It was demonstrated how several views provided by Featureous Analysis can be used aid estimating the effort involved in a forthcoming change task, identifying classes that need to be modified or reused and assessing the impact of performed changes on correctness of features.

5.4.4 Support for Software Comprehension

This section evaluates the support of Featureous Analysis for feature-oriented comprehension. This is done based on the analytical framework for evaluating cognitive design elements of software visualizations developed by Storey et al. [104].

Storey et al. formulate their taxonomy of cognitive design elements as a hierarchy. At the top-most level, the hierarchy divides the cognitive design elements into two categories: improving software comprehension and reducing the maintainer's cognitive overhead. The presented evaluation focuses on the cognitive design elements contained in the first of these categories. The summary results of the performed evaluations are shown in Table 5.3. The detailed discussions of the individual elements follow.

Table 5.3: Support for cognitive design elements in Featureous Analysis

Cognitive design elements		Support in Featureous
Enhance bottom-up comprehension	Indicate syntactic and semantic relations between software objects (1)	Traceability element
	Reduce the effect of delocalized plans (2)	Traceability element
	Provide abstraction mechanisms (3)	Three levels of granularity
Enhance top-down comprehension	Support goal-directed, hypothesis-driven comprehension (4)	Not addressed
	Provide an adequate overview of the system architecture, at various levels of abstraction (5)	Three levels of abstraction
Integrate bottom-up and top-down approaches	Support the construction of multiple mental models (domain, situation, program) (6)	Three perspectives
	Cross-reference mental models (7)	Correlation element Affinity coloring Identifiers

Bottom-up comprehension involves investigating an application's source code in order to incrementally build high-level abstractions (being either structural abstractions, or functional abstractions), until understanding is achieved. The traceability element of Featureous Analysis plays an important role in this comprehension strategy as it allows for expressing concrete source code units in the context of features as well as features in terms of source code units (1). Reduction of delocalized plans (2) 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, an application's source code 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 (3). Featureous Analysis supports this by providing adjustable granularity of source code units and the possibility to group features.

Top-down comprehension requires that domain knowledge is used to formulate goals and hypotheses that will drive the comprehension process. 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 explicitly managing and documenting goals and hypotheses (4) that an analyst may develop. This, however, does not prevent goal-driven and hypothesis-driven usages of Featureous Analysis. As was demonstrated in the study of feature-oriented comprehension of JHotDraw SVG in Section 5.4.2, top-down analysis is feasible as long as the analyst can take care of managing analytical aims and hypotheses. The support of Featureous Analysis consists of providing multiple abstraction levels (5) to facilitate gradual refinement and concretization of hypotheses. The three abstraction levels make help addressing situations where a hypothesis cannot be evaluated on a given level of abstraction and has to be decomposed into a set of more concrete sub-hypotheses.

Finally, the integration *of bottom-up and top-down approaches* is motivated by the observation that software developers tend to alternate between these two comprehension strategies in an on-demand fashion. This is due to a need for creating multiple mental models (6) that can be switched during software comprehension. Featureous Analysis provides support for multiple mental models through the code-unit perspective (which corresponds to the program model of Storey et al.), the feature perspective (the situation model of Storey et al.) and the feature relations perspectives (the situation model of Storey et al.). In terms of individual perspectives, multiple mental models are supported by viewing source code in terms of features, features in terms of source code, viewing runtime information about usage of class instances by features, feature metrics and affinity coloring. Interestingly, Featureous Analysis in some sense also supports usage of the domain mental model. This is because of the rooting of features in the problem domains of application. Hence, feature-oriented analysis allows for understanding source code in terms of an application's problem domain. To aid alternation between the supported mental models (7), Featureous

Analysis provides three mechanisms for model cross-referencing. These are the correlation element, affinity coloring and identifiers of features and code units. The correlation element supports relating the three perspectives to each other. Affinity coloring allows for relating all analysis elements to the characterization of source code units. Identifiers of feature and code units cross-reference the three abstraction levels as well as the three perspectives of Featureous Analysis.

Overall, the presented discussion indicates that Featureous Analysis provides a wide support for core cognitive design elements required by the bottom-up, top-down and mixed comprehension strategies. This is an important property with respect to the applicability of the method during software evolution and maintenance, since diverse nature of program-modification tasks creates a need for diverse comprehension strategies [110].

5.5 RELATED WORK

Myers et al. [111] proposed the Diver tool for exploration of large execution traces of Java applications. The core of the approach is a trace compacting algorithm that reduces the number of messages that are displayed to a user. The tool is implemented as a plugin to the Eclipse IDE and is equipped with several visualizations based on UML sequence diagrams. The individual invocations depicted by the visualizations can be interactively unfolded and inspected for the concrete runtime values of their parameters. Thereby, Diver aims at supporting debugging and comprehension of individual runtime executions of applications. The main contrast with Featureous Analysis is that Featureous associates feature-oriented semantics with execution traces and focuses on their modularization in source code, rather than on the runtime events occurring during their execution.

Simmons et al. [112] reported a study performed at Motorola that evaluated an industrial feature location tool CodeTEST in conjunction with the visualization tools TraceGraph and inSight. Despite reported interoperability problem, this tool combination was demonstrated to effectively support locating, understanding and documenting features. Based on investigating four evolutionary scenarios, Simmons et al. reported that this approach significantly helped identifying units of source code to be investigated by the participants. The total effort needed to integrate the three tools and to introduce them to a project was quantified as 180 person-hours. Featureous Analysis makes it possible to avoid this initial overhead because of its integration with the Featureous Location method and with the NetBeans IDE.

Apel and Beyer [113] proposed an approach to visual assessment of feature cohesion in software product lines. This was done through FeatureVisu – a graph-based visualization of methods, fields and their relations within features. The visualization uses feature-oriented metrics and a clustering-based layout algorithm to depict the

cohesion of features. The viability of the approach was demonstrated in an exploratory case study involving forty feature-oriented product lines. While the approach was applied to product-lines annotated with dedicated preprocessor directives, the approach displays potential for usage with another feature location mechanism and in another application context. Thus, the visualization of Apel and Beyer is considered an interesting candidate for eventual inclusion in Featureous Analysis.

Röthlisberger, et al. [31] presented a feature-centric development environment for Smalltalk applications and demonstrated its usefulness in comprehension and maintenance activities. The proposed development environment combines interactive visual representations of features with a source code browser that filters the classes and methods participating in a feature under investigation. The visual representations used are: the *compact feature overview*, which depicts the methods contributing to a feature and indicates the level of their tangling using colors; the *feature tree*, which presents the call-tree of methods implementing a feature and the *feature artifact browser*, which displays the classes and methods used in a feature.

The overall idea of decorating source code with feature-oriented information colored sidebars bears similarities to Spotlight [114] and CIDE [49]. However, Featureous Analysis puts the primary emphasis on the level of sharing of features (indicated by colors on the bars) rather than on identifying the concrete features that use the source code units (displayed on-demand as tooltips).

5.6 SUMMARY

Feature-oriented comprehension and modification is difficult to achieve without fundamentally changing the perspective on the existing object-oriented applications. Such a feature-oriented perspective needs a structured and evolutionary-motivated set of analytical views over the feature-code traceability links. Only then, it can properly support software comprehension and modification effort of software developers, and guide them in a systematic fashion.

This chapter presented the Featureous Analysis method for feature-oriented analysis of Java applications.

The method defines a three-dimensional conceptual framework as an objective frame of reference for constructing feature-oriented analytical views. Based on an analysis of the needs of the change mini-cycle, essential configurations of feature-oriented views were identified. These configurations were then realized by several visualizations. The method was evaluated in a series of studies involving: (1) comparative assessment of four modularization alternatives of KWIC, (2) feature-oriented source code comprehension of JHotDraw SVG and (3) its feature-oriented modification. Finally,

third-party criteria were used to analytically evaluate Featureous Analysis with respect to its support for three types of program comprehension strategies. Together, these evaluations deliver broad initial evidence for applicability of Featureous Analysis to supporting feature-oriented program comprehension and modification of Java applications.

6. FEATUREOUS MANUAL REMODULARIZATION

This chapter uses Featureous Analysis developed in Chapter 5 as a basis for performing manual remodularization of existing Java applications. The proposed method, called Featureous Manual Remodularization, aims at improving modularization of features in source code of applications. The chapter contrasts restructurings based on relocation and reconceptualization of source code units, and captures their involved tradeoffs as the *fragile decomposition problem*. Based on this, a target design for feature-oriented remodularization of Java applications is proposed. The method is applied to a study of analysis-driven remodularization of a neurological application called NDVis, and the obtained results are evaluated both qualitatively and quantitatively.

This chapter is based on the following publications:
[CSMR'12], [SCICO'12]

6.1 OVERVIEW

While feature-oriented analysis can be used to aid comprehension and guide feature-oriented changes, it does not physically reduce the scattering and tangling of features

in source code. Hence, a feature-oriented perspective is only the first step towards the last two qualities of modularity postulated by Parnas [5]: product flexibility that should support making drastic changes to a single feature without changing others, and managerial flexibility that should enable separate groups of software developers to work on separate features with little need for communication.

In order to acquire these two qualities it is necessary to reduce scattering and tangling of features. This can be done by the means of behavior-preserving manipulation of the static structure of source code, also known as restructuring [7]. Hence, a restructuring process should result in an improvement of structure-related non-functional properties of an application, while remaining completely transparent to application users.

The particular kind of restructuring aimed at improving modularization of features is referred to as being *feature-oriented*. Furthermore, since such a restructuring fundamentally aims at changing the criteria of how an application is divided into modules, it is referred to as *remodularization*. Thereby, *feature-oriented remodularization* is a restructuring that aims at introducing the modularization of features as the primary decomposition criterion of an application's source code.

In order to conduct a feature-oriented remodularization of an existing application, three main challenges have to be tackled. Firstly, traceability links between features and source code units have to be established. Secondly, a new feature-oriented structure of an application has to be designed. Thirdly, the source code of an application has to undergo a transition from its current structure to the newly designed feature-oriented structure.

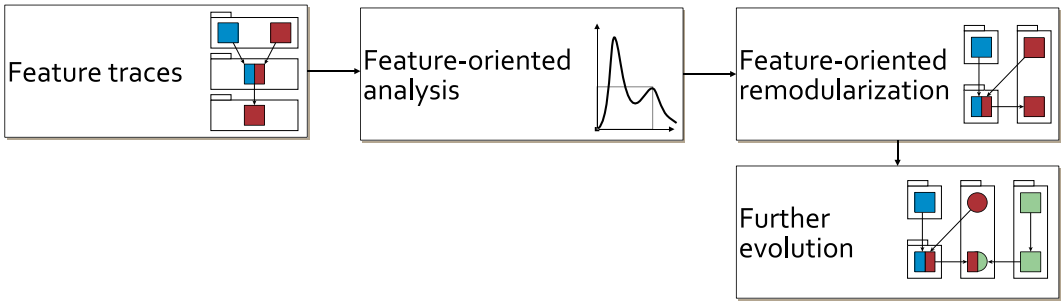


Figure 6.1: Overview of Featureous Manual Remodularization

This chapter presents a method for designing and performing a manual analysis-driven feature-oriented remodularization of an existing application. This method is called *Featureous Manual Remodularization*. The proposed process for conducting this restructuring is schematically depicted in Figure 6.1. This process uses feature traces of Featureous Location introduced in Chapter 4 and the Featureous Analysis method introduced in Chapter 5 to support comprehension of features and to guide their incremental restructuring. The method uses common low-level refactoring techniques such as move class, move method and rename class [65]. After remodularization is

completed, the established feature-oriented design forms a basis for further evolution of the target application.

The proposed vision of feature-oriented remodularization is applied in a study of migrating a neurological application NDVis [115] to the NetBeans module system [86]. This study demonstrates the feasibility of Featureous Manual Remodularization and discusses the advantages of feature-oriented division of applications in the context of migrations to module systems.

6.2 THE METHOD

This section presents the Featureous Manual Remodularization method. The prerequisites to Featureous Manual Remodularization are (1) establishing traceability links between features and source code and (2) gaining an initial feature-oriented understanding of a target application.

Having the means of feature location and feature-oriented analysis already discussed in the form of the Featureous Location and Featureous Analysis methods; this section focuses on the topic of source code restructuring. Firstly, two main restructuring types applicable to modularizing features are identified and their characteristics discussed. Based these on this, Featureous Manual Remodularization proposes a target high-level design that existing applications should be restructured towards through the identified restructuring types to improve modularization of features.

6.2.1 Remodularization through Relocation and Reconceptualization

Two primary types of restructurings can be used to separate and localize features in source code of Java applications. Namely, this can be done through so-called *reconceptualization* and *relocation* of source code units at various levels of granularity.

For the needs of the forthcoming discussion, the following assumptions are made. Four types of Java code units are distinguished and ordered according to their granularity: packages, classes, methods and instructions. If *granularity N* is the granularity level corresponding to a given code unit, then *granularity N+1* corresponds to the following *finer* level of granularity than N; that is methods (N+1) are the following finer granularity than classes (N). Similarly, *granularity N-1* stands for the following *coarser* level of code unit granularity.

A source code unit is referred to as *single-feature unit* when it is used exclusively by a single feature. Similarly, a code unit is referred to as *multi-feature unit* when it is used by more than one feature. Hence, the general term multi-feature unit represents uniformly the core units, inter-group units and single-group units that were discussed earlier in Section 5.3.1.

Relocation at granularity N is the process of moving a code unit at granularity N from its containing code unit at granularity N-1 to another code unit at granularity N-1. As relocation does not involve a modification of the code unit being relocated, it preserves its original semantics. An example of relocation at the granularity of classes is the move class refactoring [65], which moves a class from one package to another.

Reconceptualization at granularity N is the process of modifying a code unit at granularity N by changing or relocating one or more of its contained code units at granularity N+1. Because of modifying a code unit, reconceptualization is likely to modify its semantics as well. An example of reconceptualization at the granularity of classes is to use the move method refactoring [65], or the split class refactoring; that is, extracting a new class from an existing one.

In order to separate and localize features at granularity N, relocation has to be performed at least at granularity N+1 and reconceptualization has to be performed at least at granularity N. That is, if one wishes to separate and localize features at the granularity of packages, one has to relocate classes among packages. Accordingly, if one wishes to separate and localize features at the granularity of classes, one has to move methods and member fields among classes.

Hence, when *separating features* in terms of classes (N=class), existing classes (N) have to be reconceptualized. This is done by splitting classes through relocating all their single-feature methods (N+1) to new single-feature classes. If a class contains any multi-feature methods, then a complete separation cannot be achieved at the granularity of classes (N), and the investigation has to be refined to the granularity of methods (N+1). This can be done by reconceptualizing methods by relocating their single-feature instructions (N+2) to new single-feature methods (N+1). Because instructions are the lowest level of granularity, any further reconceptualization is not possible and the remaining multi-feature instructions can only be separated among features by creating a duplicate for each feature involved. While this might be an acceptable tradeoff in the case of handling method bodies, it becomes more problematic when applied to fields and variables, because introducing state duplication requires providing additional means of synchronizing state among the duplicates.

When *localizing features* in terms of packages (N=package), existing packages (N) have to be reconceptualized. This is done by creating one single-feature package for each feature and relocating to it all its single-feature classes (N+1). If there exist multi-feature classes, then a complete localization cannot be achieved at granularity of packages (N) and separation needs to be performed at granularity of classes (N+1). By following the process that was discussed in the previous paragraph, it is possible to split multi-feature classes into several single-feature classes, which can then be relocated to their appropriate single-feature packages.

From the presented discussion, it follows that *relocation at granularity N manifests itself as reconceptualization at granularity N-1.* Furthermore, it can be seen that the applicability of relocation to separating and localizing features at granularity N is determined by the degree of separation of features present at granularity N+1. Hence, reconceptualization at finer granularities acts as an enabler of relocation at coarser granularities. For instance, a class can be cleanly separated among features using relocation of methods only if the features are already separated within the method bodies.

6.2.2 Problematic Reconceptualization of Classes

The mentioned need for refining the granularity of relocation and reconceptualization during modularization of features is problematic for several reasons. Firstly, refining the granularity of syntactic separation often creates a need for more advanced mechanisms for separation of concerns (e.g. aspects [67] or derivatives [72]) than the ones available in the mainstream object-oriented programming languages. Secondly, the finer the granularity of restructurings; the more complex, laborious and potentially error-prone the restructuring process becomes. Thirdly, reconceptualization of the responsibilities of legacy classes into several new single-feature classes forces one to invent new abstractions that will match the semantics of the new class. In the case of classes representing domain entities, this results in delocalizing of the implementation of domain concepts.

Object-oriented design of applications is traditionally based on networks of collaborating classes [21]. The *classes* and their interrelations are often arranged to reflect entities found in the problem domains of applications These entities are typically identified during requirements analysis and usually captured in the form of a domain model [116], [21]. Thereby, domain models establish a valuable one-to-one correspondence between source code and entities in software's problem domain.

As it was discussed in Chapter 1 and empirically demonstrated in Chapter 5, classes and methods in object-oriented applications often implement multiple features. Hence, a single domain entity typically participates in several functionalities of an application. In order to separate features that are tangled at these granularities, the involved classes and methods would have to be divided into parts. These parts are referred to as *fragments,* to use a technology-neutral term. As demonstrated in literature fragments of classes and methods can be implemented using various mechanisms, e.g. aspects, mixins, etc. [66].

An example of a fine-grained restructuring of an existing class into fragments is shown in Figure 6.2. In the first part of the example, the `Car` class that represents the problem-domain concept of a car participates in two features: OVERTAKE and PARK. After performing a fine-grained division of `Car` into fragments by means of reconceptualization, one arrives at two separate class-fragments jointly representing

the car domain concept. Each of the obtained single-feature fragments encompasses only a subset of the responsibilities of the original `Car` class.

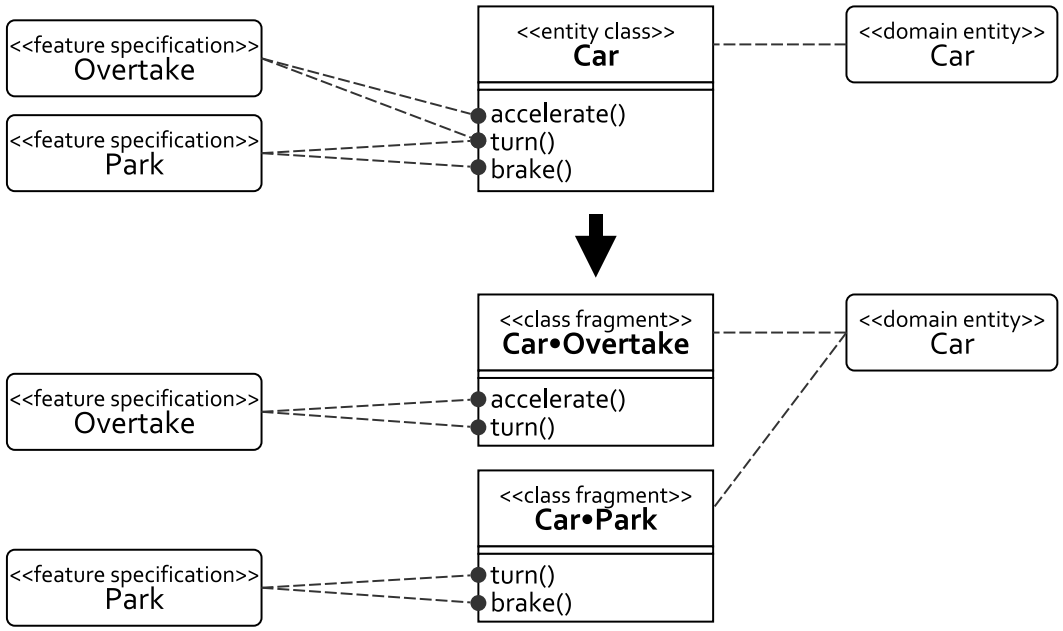


Figure 6.2: Reconceptualization of entity classes into class fragments

On the technical level, it is easy to discover that this example division leads to a number of implementation issues. These are the need to share the same object identity among the two fragments and the duplication of the `turn` method definition (which in principle could be avoided by introducing yet another class-fragment or by introducing a dependency between the fragments – however, both the solutions are problematic in their own ways). Nevertheless, it is the following conceptual issue that is considered the most problematic aspect of this reconceptualization.

As can be observed, the division of the original `Car` class results in *delocalization of the car concept*. As a result, understanding or modifying the concept of the car requires in principle visiting both the class-fragments. Moreover, any modification made to one of the fragments can have unforeseen consequences on the other fragment. This is because the division of a single coherent class into a number of class-fragments inevitably leads to a strong coupling between the fragments. This coupling is not only a semantic one that deals with delocalized assumptions of what a car is, but often also a syntactic one, since one fragment may depend on methods defined by another fragment. As discussed by Murphy et al. [67] such dependencies are evolutionary-problematic.

Most importantly, it becomes difficult to equip the created class-fragments with semantics that finds it reflection in the application's problem domain. In the example, it is questionable whether a entity that is only capable of turning and braking, without

being able to accelerate (i.e. the `Car•Park` fragment) can be meaningfully referred to as car. Because in this case there exists no real-world domain entity that consists purely of these two capabilities, a developer is forced to invent a new concept to represent such a fragment. It is argued that intuitive understanding of such invented concepts is likely difficult for other developers. This is because of the uncertainty about the precise part of the problem domain that such a custom concept represents, and about the boundaries of responsibilities that it accumulates. This situation is an instance of the so-called fragile decomposition problem.

6.2.3 The Fragile Decomposition Problem

This thesis defines the *fragile decomposition problem* as the situation in which there exist dependencies on a particular shape of an application's decomposition. This problem is further divided into the cases of *semantic* and *syntactic* fragility. As will be argued, the fragile decomposition problem has its repercussions on software comprehension and changeability.

The first case of dependency on a particular shape of an application's decomposition was introduced already in the previous section. It is the reliance of developers on a direct correspondence between the decomposition of an application's problem domain and its solution domain. This type of decomposition fragility is referred to as *semantic fragility*. The key observation here is that not all decompositions of applications are equally easy to understand. As discussed by Rugaber [117], Rajlich and Wilde [118] and Ratiu et al. [59], the implementation abstractions that can be easily related to concepts in the application's problem domain prove easier to understand for developers.

In the context of the previous example involving the `Car` class, a developer who knows what a car is in the real world would be able to apply this knowledge to anticipate the meaning and possible responsibilities of an unfamiliar class named *Car*. Based on this premise, it is important that during remodularization a particular attention is paid to preserving the human-understandable classes that correspond to the problem-domain entities of applications.

A similar case of semantic fragility of decomposition occurs when a developer becomes familiarized with the semantics and relations among particular classes. As soon as these patterns are learned, developers become dependent on the gained understanding. Given the assumption that familiarity of source code helps developers to reduce the scope of comprehension activities, it is crucial that any restructuring efforts preserve how the existing classes are shaped and how they interact with one another. Thus, substantial reconceptualization of classes may lead to invalidating existing familiarity of source code of the developers, and may force them to spend additional effort to re-learn the shape of classes and the patterns of their interactions.

The second facet of the fragile decomposition problem is referred to as *syntactic fragility*. Syntactic fragility is induced by inbound static dependencies on source code units. This includes cases where a unit of source code is used, e.g. as a library, by other independently developed applications. In such a case, restructuring of a library's source code, without actually modifying its offered capabilities, may affect all the dependent parties. As a result, all the dependent parties would need to undergo syntactical modifications (e.g. a dependency on particular names of packages, classes, etc.) as well as semantical modifications (e.g. a dependency on semantics of classes in the context of the problem domain, the protocols of interaction with them, etc.). Thereby, the syntactic facet of decomposition fragility constrains the possibilities for changing the existing structural decompositions of evolving software, when third-party dependencies are in effect.

6.2.4 Feature-Oriented Structure of Applications

Motivated by the characteristics of relocation and reconceptualization and the presence of the fragile decomposition problem, this section proposes the structural design shown in Figure 6.3 as the target of feature-oriented remodularization.

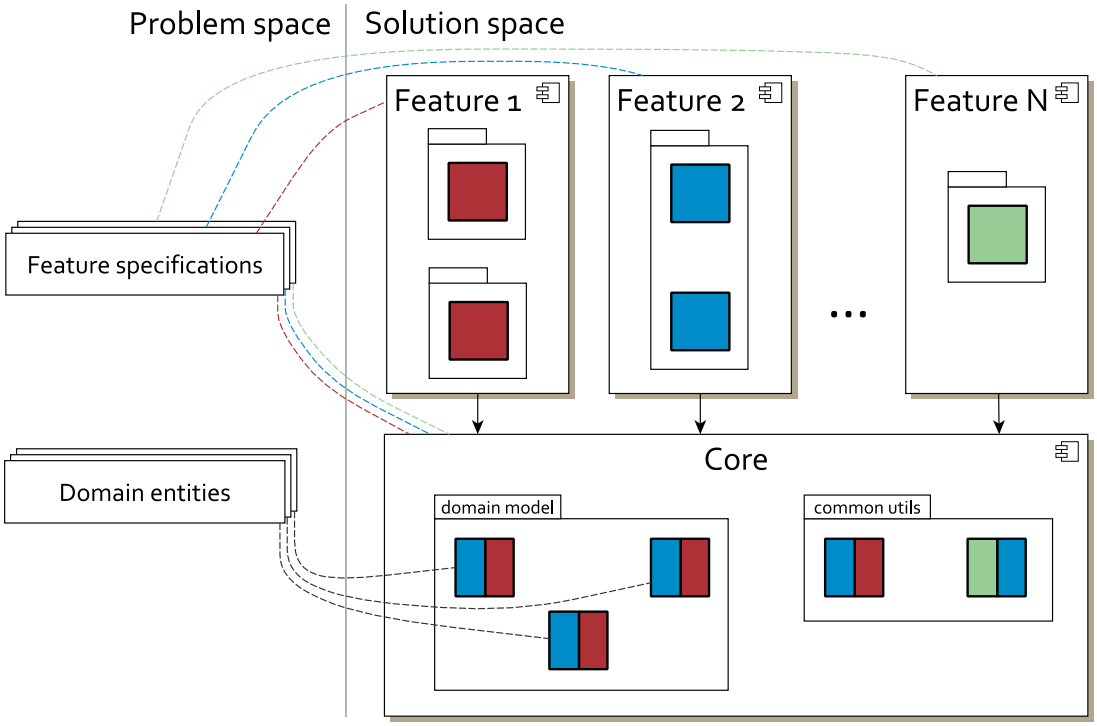


Figure 6.3: Target feature-oriented design of applications

The proposed design consists of one or more multi-feature modules that constitute the application's core, and a set of single-feature modules. Each of the modules can physically encompass one or more Java packages.

The single-feature modules provide explicit and localized representations of feature specifications from the application's problem space. In principle, these modules should be statically independent from one another, unless logical dependencies among their corresponding feature specifications exist in the problem domain. This arrangement of single-feature source code units into explicit and isolated modules is aimed at reducing the tangling among features.

All the individual single-feature modules depend by default on a set of underlying multi-feature modules. The responsibility of the multi-feature modules is to encompass the essential domain models that the features operate on, as well as other reusable utility-classes. Thereby, the core modules are designed to contain direct representations of domain entities and their relations, as found in the application's problem space. In consequence, Featureous Manual Remodularization postulates that the tangled domain-model classes should not be reconceptualized into feature-specific fragments. Finally, while the core modules are allowed to freely depend on one another, they should not depend on any of the single-feature modules, to reduce the possibilities for inbound propagation of feature-oriented changes.

There exist a number of interesting properties of the proposed feature-oriented structuring of source code. Firstly, single-feature modules and their enclosed packages are good starting points for comprehending individual features in isolation from others. Secondly, if modifications to a feature are confined to classes within a single module, then they will have no effect on classes belonging to other features. This corresponds to the *common closure principle* [42], which states that classes that change for the same reason should be grouped together. In situations where the scope of modification cannot be confined to single-feature modules, the relevant classes in multi-feature modules can be identified by following the static dependencies from their referencing single-feature classes. Thus, the multi-feature modules are expected to follow the *common reuse principle* [42], which proposes to that classes reused together should be grouped together. Lastly, thanks to the multi-feature nature of core modules, performing a feature-oriented remodularization requires only minimal, if any at all, extent of class reconceptualization.

Summing up, Featureous Manual Remodularization proposes the discussed design as the means of making features explicit and reducing their scattering and tangling in the structural organizations of Java applications. Apart from that, the presented feature-oriented design remains compatible with traditional object-oriented design principles, such as high cohesion and low coupling among packages and modules. This compatibility is particularly important to guide handling any classes not covered by the feature traces produced by Featureous Location, because such classes cannot be interpreted in the terms of scattering and tangling.

Inspecting Feature Representations

Grouping of single-feature classes into single-feature modules provides developers with an explicit correspondence between a feature specification and its single-feature

classes. However, such a correspondence is not provided by the grouping of multi-feature classes into multi-feature modules.

To aid developers in identifying the boundaries of individual features within multi-feature modules, the *feature-aware source code editor* view $\{g_{2,3}, p_{2,3}, a_1\}$ of Featureous Analysis can be used. As it was discussed in detail in Chapter 5, this view extends the source code editor of the NetBeans IDE with feature-oriented code folding and colored sidebars that decorate the source code of classes and methods with detailed traceability information.

The explicit marking of single-feature methods helps to identify single-feature parts of multi-feature classes, whereas marking of tangled methods makes the relations among features and the reuse level of classes among features explicit to a developer. This should be sufficient to support a developer in identifying the reusable code and in qualified estimation of the effort required to modify individual multi-feature classes.

6.3 EVALUATION

This section reports on an action research study of applying Featureous Manual Remodularization to an open-source neurological application called NDVis.

NDVis is a 6.6 KLOC Java-based tool for visualization and analysis of large multi-dimensional neurological datasets that was developed by VisiTrend [115]. Figure 6.4 depicts how NDVis uses its graphical interface to visualize neurological data.

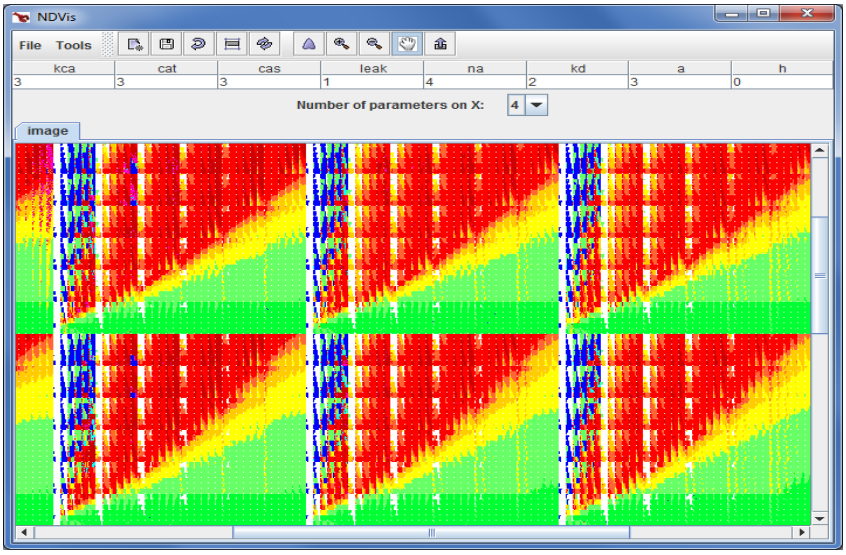


Figure 6.4: Graphical user interface of NDVis

After completing the initial development of the tool, the VisiTrend company decided to migrate the monolithic application to the NetBeans Rich Client Platform and

modularize it using the NetBeans Module System [86]. The major aim was to facilitate independent evolution and deployment of functionalities, and to facilitate code reuse across multiple project branches, as well as within a larger portfolio of applications being developed. The secondary aims included a better management of source code complexity and preparing the application for adoption of the upcoming Java Module System technology [119].

Having initially no knowledge of the design, the implementation or the problem domain of NDVis, the author joined the planned migration efforts as an external contributor. This context proved appropriate for demonstrating that Featureous Manual Remodularization can successfully support the feature-oriented migration of the project to a module system. The following sections report on the efforts and outcomes of feature-oriented remodularization of NDVis.

6.3.1 Analysis-Driven Modularization of Features

Similarly to plain JAR files, module systems are the means of dividing applications into separate units of development, compilation and deployment. In addition, module systems make it possible to explicitly version modules, to separate exported packages from private ones and to establish explicit dependencies among modules. Moreover, most module systems make it possible to load and unload modules at runtime and provide mechanisms for dynamic discovery of new services being added to an application at runtime.

However, if performed improperly, the additional separation of code provided by module systems can result in new kinds of evolutionary pressures [120]. In particular, the presence of a module system can amplify the negative consequences of improper modularization of evolutionary-important concerns. Hence, migration of a monolithic application to a module system generally requires careful structural considerations.

In the case of NDVis, it turned out that the planned migration to a module system had to involve restructuring. It was found that the evolutionary needs of NDVis anticipated by the development team closely resembled the existing evidence for evolutionary importance of features. Hence, the author proposed to design and carry out an incremental feature-oriented remodularization of source code as a part of the migration efforts. Here, using Featureous Manual Remodularization to establish single-feature modules was intended to facilitate independent evolution and deployment of features, while explicit multi-feature modules were envisioned to promote reuse of essential domain models and utilities. This proposal was accepted by the development team of NDVis.

The NetBeans Module System and the standard specification of the Java language were the available technical means. In particular, syntactical mechanisms of advanced separation of concerns and other module systems were excluded by the project owners.

The remainder of this section reports on the performed analysis-driven remodularization of NDVis. The examples of three features OPTIMIZE, PROGRAM STARTUP and PARAMETERS are used as a basis for discussing the most important challenges, design decisions and analytical methods involved.

Establishing Traceability Links

A set of feature specifications of NDVis was recovered by interviewing the lead developer and by inspecting the functionality provided in the GUI of the application. Overall, 27 use cases were identified, which were then grouped into 10 coherent features. Table 6.1 lists and briefly describes the identified features.

Table 6.1: Features of NDVis

Feature	Estimated Size (KLOC)	Summary description
Adjust image	0.93	Zooming, panning, enlarging and resetting image.
Color mapper	4.92	Mapping data to colors on the image using SQL queries, managing, editing and bookmarking queries.
Database connectivity	0.26	Configuring and connecting to a database.
Image	3.28	Creating and rendering an image-based representation of data, hover-sensitive preview of data values.
Import data	0.99	Importing data from a CSV file.
Optimize	3.76	Optimizing how multiple data parameters are being displayed as two-dimensional image.
Parameters	2.52	Displaying and manually reordering data parameters.
Program startup	4.05	Initializing the program.
Save image	0.79	Persisting created images.
Simulator	3.08	Simulating the behavior of neurons based on data.

The recovered specifications of features made an input for applying the Featureous Location method defined in Chapter 4. In the case of the NDVis application, in which features are triggered through GUI elements, feature-entry points were most often associated with the `actionPerformed` methods of event-handling classes. In total, 39 methods were annotated.

Subsequently, a set of feature traces was obtained through manual activation of features in the GUI of NDVis. Overall, 30 out of all 91 types of NDVis were identified as single-feature classes and 22 as multi-feature classes. The remaining 14 interfaces and 25 classes were not covered by feature traces.

Initial Feature-Oriented Assessment

Firstly, a top-down investigation of confinement of features in design of NDVis was performed to estimate the extent of the forthcoming restructurings and to identify potential challenges.

The analytical view used for this purpose was the *feature-package 3D characterization view* of Featureous Analysis. This view was applied to investigating the scattering and tangling of NDVis features in terms of packages, as presented in Figure 6.5.

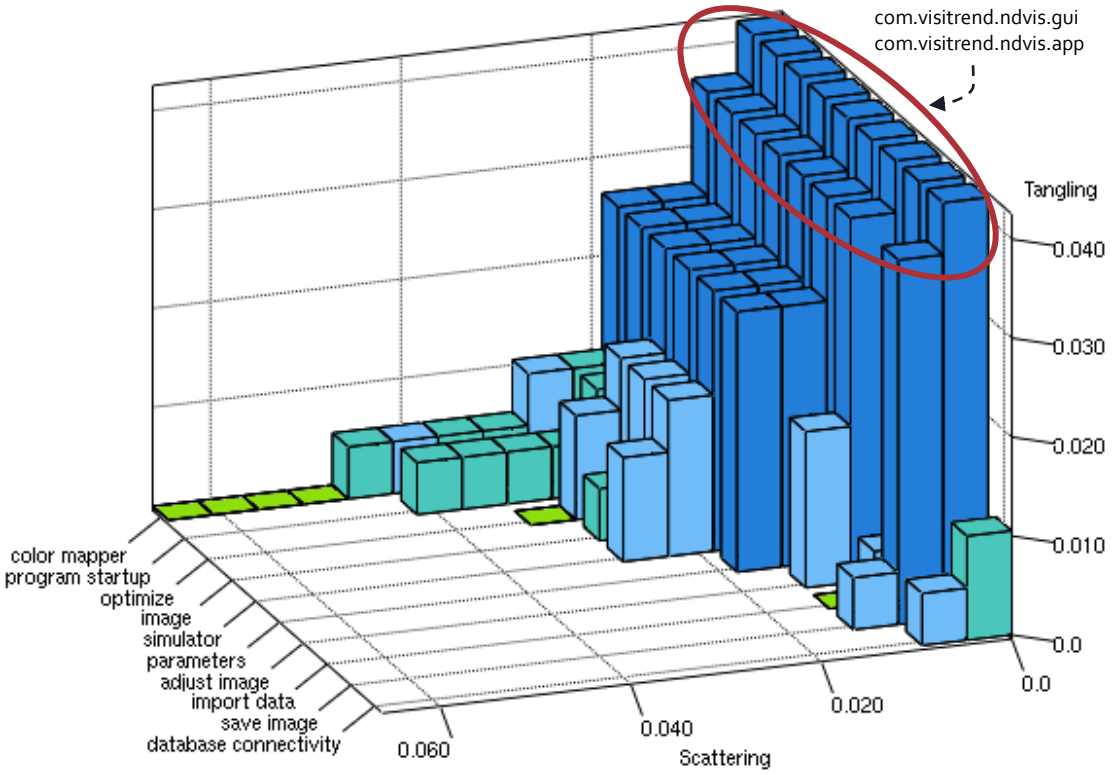


Figure 6.5: Feature-package 3D characterization of NDVis

The information on feature scattering revealed that an average feature of NDVis is using 6 out of 19 packages. The scattering profiles indicated the lack single-feature packages for the majority of features.

The analysis of the two most tangled packages, namely `com.visitrend.ndvis.gui` and `com.visitrend.ndvis.app`, revealed at a strong difference in the ways they were reused among features. While `gui` consisted of a set of simple single-feature classes, `app` contained only one, but highly tangled class `NDVis`. As it will be discussed later, this class turned out to be a major complexity point that needed to be reconceptualized.

Identifying core domain concepts as the most tangled classes. By refining the granularity of the presented view, it was possible to identify the classes that participate in most of the features. It was observed that these classes (with the exception of the `NDVis` class) constitute the implementations of the central domain concepts of the application. These classes were: `ImagePanel`, `DataInfo`, `Parameters`, `ParametersUtils` and `ColorEngine`. This observation helped to recognize and understand the essential domain model of NDVis early in the process. Moreover, the recognized domain model classes created an initial set of candidates for forming multi-feature core modules.

Modularizing Features by Relocation – Optimize

One of the features modularized by the means of class relocation was OPTIMIZE. This feature is concerned with optimizing how multiple dimensions (`Parameters`) of data (`DataInfo`) are used to form a two-dimension representation of the dataset.

The way OPTIMIZE shares classes with two other example features is presented in Figure 6.6. There, it can be seen that OPTIMIZE is implemented by five single-feature classes and a number of multi-feature classes identified earlier.

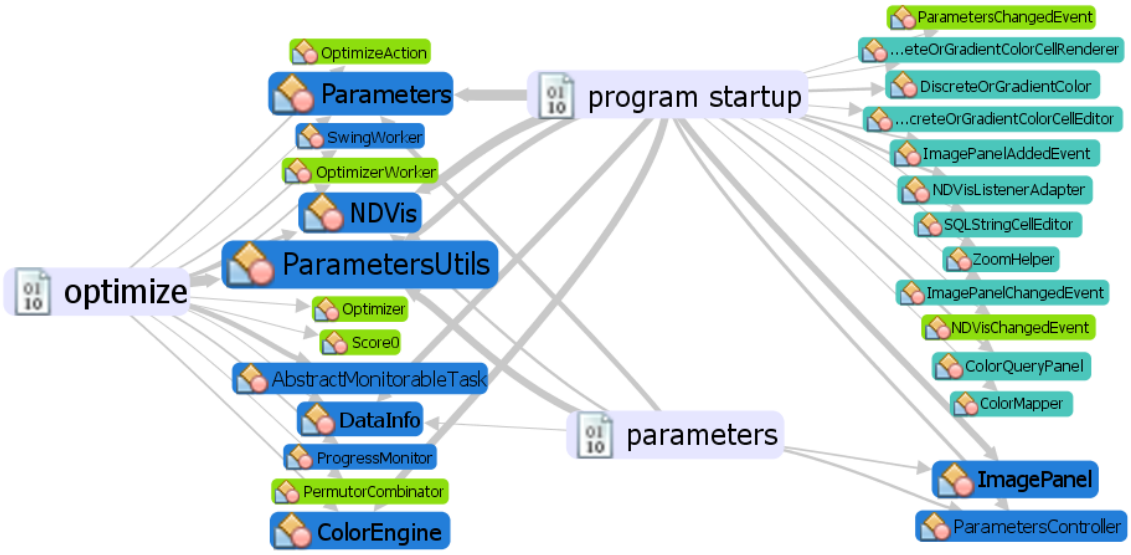


Figure 6.6: Correlation of the three example features with classes

The fairly high separation of OPTIMIZE from other features in classes made it possible to avoid extensive reconceptualization of classes. This feature was modularized mostly by relocating its single-feature classes (the green classes in the second column in the Figure 6.6) to its dedicated single-feature module. The remaining highly tangled classes of OPTIMIZE were used for forming multi-feature core modules, as will be discussed later.

Identifying classes exhibiting under-reuse. Interestingly, it was observed that one of the single-feature classes of OPTIMIZE, called `PermutorCombinator`, was originally placed in the `utils` package of NDVis. This indicates that this class was designed for reuse among multiple features. However, it can be seen that this class ended up not being used by features other than OPTIMIZE. This sharp disparity between the initial design intent of the class stated by placing it in the `utils` package and its actual usage was interpreted as a manifestation of design erosion.

Identifying classes exhibiting over-reuse. A situation may occur, where there exist static dependencies between seemingly unrelated features. Such dependencies are excessive, as long as they do not correspond to logical relations among feature specifications in the problem domain. Hence, in the case of OPTIMIZE it was ensured that its resulting module neither exposes any API classes, nor depends on any other single-feature

modules. Doing so allowed for independent exclusion and inclusion of the OPTIMIZE feature from the application through simply (un)installing its dedicated single-feature module.

Decentralizing Initialization by Reconceptualization – Program Startup

PROGRAM STARTUP is the single feature responsible for initializing the NDVis application. This centralized initialization model was found to be a major obstacle to achieving runtime independence of features.

The runtime importance of PROGRAM STARTUP is depicted in Figure 6.7 in the form of the *feature relations characterization* graph. As can be seen, all other features of NDVis use objects instantiated by PROGRAM STARTUP feature.

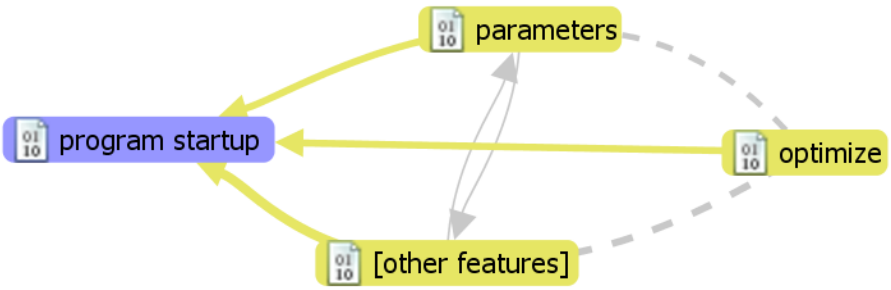


Figure 6.7: Dynamic dependencies among features

In order to precisely understand how PROGRAM STARTUP initialized the individual features of NDVis, the investigation focused on the main hot-spot of the feature – the `NDVis` class. It was discovered that the `NDVis` class was the central point of instantiation and composition of essential parts of most features of the application. In the context of software evolution, this meant that the `NDVis` class had to be modified every time a feature was added or removed from NDVis. Such evolutionary coupling hinders functional extensibility and customizability.

To address this issue, an extensive reconceptualization of the `NDVis` class was performed. The reconceptualization was aimed at redistributing the initializations of individual features among the features themselves. Thereby, the centralized model of feature initialization and composition was replaced with a decentralized model. In this new model, plugability of features was facilitated by making each feature fully control its own initialization and integration with the rest of the application.

As a tangible outcome of the extensive refactorings performed, 330 out of 490 lines of code were removed from the `NDVis` class. The reconceptualized `NDVis` class, being the only remain of PROGRAM STARTUP, was then placed in one of the application's core modules.

Consolidating the Reusable Core – Parameters

The PARAMETERS feature is one of the mandatory features of NDVis. It provides the concept of dimensional parameters (`Parameters`) of data that form a conceptual basis for multiple other features. Furthermore, this feature takes care of table-based visualization of dimensional parameters of data in the GUI of NDVis.

During modularization of PARAMETERS, the discussed earlier reconceptualization of the `NDVis` class was found helpful. It was found to allow for isolating the classes belonging to the PARAMETERS feature from the rest of the program (these classes include `Parameters`, `ParametersController` and `ParametersUtils`). The isolated classes were then relocated into a dedicated PARAMETERS module. However, this module was identified to be reused by several other features of NDVis. As a result, this module was decided to be incorporated into the set of the reusable core modules of NDVis. A closer investigation of the problem domain confirmed the mandatory nature of PARAMETERS as a basis for other features of the application.

In order to promote reusability of the created PARAMETERS module, its lack of dependency on single-feature modules was furthermore ensured. Firstly, the lack of such a dependency is required by the NetBeans Module System, which prohibits circular dependencies among modules. Given that all single-feature modules have to depend on core modules in the first place, no opposite dependencies can be established. Secondly, in the context of potential reuse in other applications, the PARAMETERS module could be reused independently of any NDVis-specific features.

6.3.2 Evaluation Criteria

In addition to manually inspecting the results of Featureous Manual Remodularization of NDVis, this evaluation employs quantitative criteria. These criteria consist of four third-party package-granularity software metrics. The definitions of these metrics are shown in Figure 6.8.

Based on the formulation proposed in [58], total scattering of features among packages FSCA measures the average number of packages P_F that contribute to implementing application features F (i.e. packages that fulfill the ⇝ "*implemented by*" relation with features). This value is furthermore normalized according to the number of non-insulated packages P_F found in the application (i.e. packages that contribute to implementing at least one feature).

Based on the formulation proposed in [58], total tangling of features in packages FTANG is a measure complementary to FSCA, as it quantifies the average number of features F that use packages P.

The definitions of average package cohesion PCOH and total coupling of packages PCOUP are based on cohesion and coupling measures proposed in and [121]. They are based on the notions of interactions between data declarations (*DD-interactions*) and

interactions between data declarations and methods (*DM-interactions*) and the ⇒ operator for specifying the containment relations between classes and packages. This formulation of package coupling corresponds to a sum of the ACAIC, OCAIC, ACMIC, and OCMIC coupling measures defined in [121], whereas this formulation of cohesion is the package-level version of the RCI metric proposed in [122].

$$\boldsymbol{FSCA}(F) = \sum_{f \in F} fsca(f)\,,\ \ where{:}\ \boldsymbol{fsca}(f) = \frac{|\{p \in P_F : f \rightsquigarrow p\}| - 1}{|F| \cdot |P_F|}$$

$$\boldsymbol{FTANG}(P_F) = \sum_{p \in P} ftang(p)\,,\ \ where{:}\ \boldsymbol{ftang}(p) = \frac{|\{f \in F : f \rightsquigarrow p\}| - 1}{|F| \cdot |P_F|}$$

$$\boldsymbol{PCOH}(P) = \frac{\sum_{p \in P} pcoh(p,T)}{|P|}\,,$$
$$where{:}\ \boldsymbol{pcoh}(p,T) = \frac{\sum_{t1 \in T} \sum_{t1 \Rightarrow p} \sum_{t2 \in T} \sum_{t2 \Rightarrow p} \left|DD_{t1,t2} \cup DM_{t1,t2}\right|}{\sum_{t1 \in T} \sum_{t1 \Rightarrow p} \sum_{t2 \in T} \sum_{t2 \Rightarrow p} \left|MaxDD_{t1,t2} \cup MaxDM_{t1,t2}\right|}$$

$$\boldsymbol{PCOUP}(P) = \sum_{\substack{p \in P \\ T \Rightarrow p}} pcoup(p,T)\,,$$
$$where{:}\ \boldsymbol{pcoup}(p,T) = \sum_{\substack{t1 \in T \\ t1 \Rightarrow p}} \sum_{\substack{t2 \in T \\ t2 \nRightarrow p}} \left|DD_{t1,t2} \cup DM_{t1,t2}\right|$$

Figure 6.8: Metrics for evaluation of Featureous Manual Remodularization

In order to aggregate these two package-scope metrics to the application-level measurement, the mean cohesion value (due to the *normalization* property of this measure – see [123] for an extensive discussion) and summary coupling is computed. It can be demonstrated that these formulations fulfill all the properties of software measurement framework of Briand et al. that are required for cohesion and coupling metrics [123].

6.3.3 Results

The iterative modularization of features into independent single-feature modules and reusable multi-feature core modules resulted in a new package and module structure of NDVis. In total, achieving this result required 35 person-hours of work and 20 intermediate commits. The majority of the restructurings performed were class relocations – one notable exception was the `NDVis` class, for which extensive reconceptualization was required. The obtained feature-oriented modularization is schematically depicted in Figure 6.9.

As shown in Figure 6.9, each of the top single-feature modules depends only on the underlying multi-feature core modules. By ensuring lack of dependencies among single-feature modules and by employing the service discovery mechanism of the NetBeans Module System it was possible to make NDVis functionally customizable.

Namely, the established modularization enables exclusion and inclusion of individual single-feature modules without the need to change a single line of code in the rest of the application. Moreover, the established set of core modules can be easily reused across a larger portfolio of applications, because the core modules do not depend on any single-feature modules. Hence, the qualitative outcomes of manual feature-oriented remodularization of NDVis are considered to fulfill the original goals of the migration process.

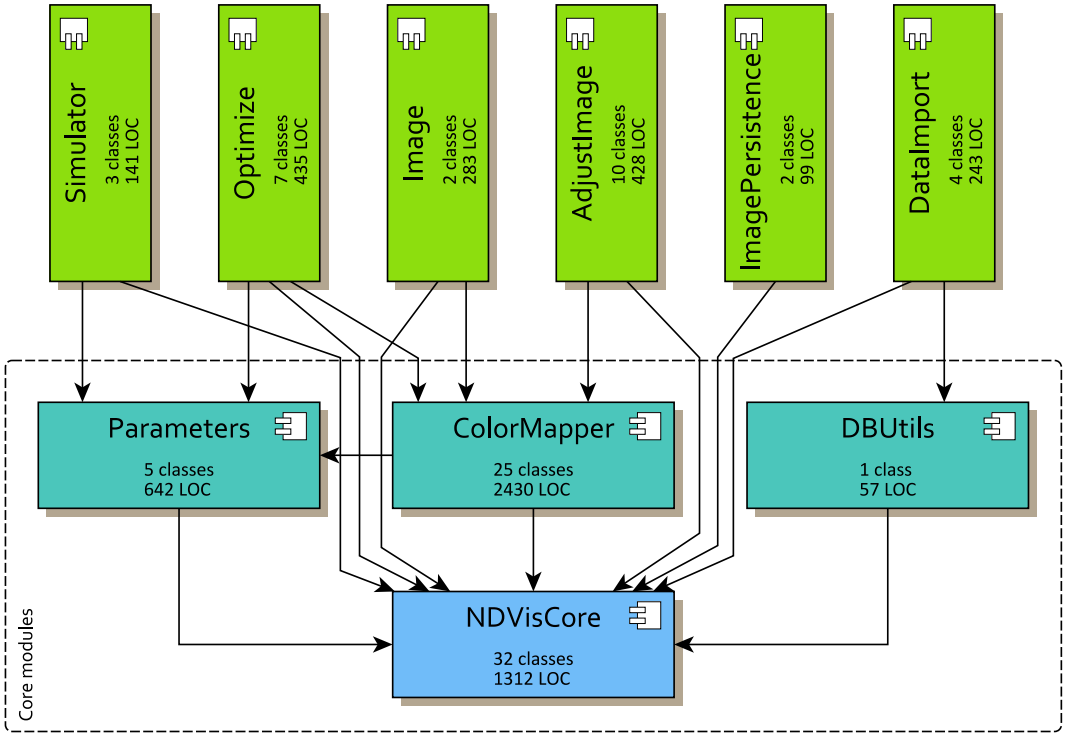


Figure 6.9: Feature-oriented architectural module structure of remodularized NDVis

In order to quantify the impact of Featureous Manual Remodularization on NDVis, the metric introduced in Section 6.3.2 were used. These metrics were applied to both the original design (referred to as “Original”) and the remodularized one (referred to as “Manual”). The summary of the results is presented in Table 6.2, whereas the detailed effects on scattering of features and tangling of packages are shown in Figure 6.10.

Table 6.2: Effect of remodularization on quality of NDVis

Metric	Original	Manual	Δ%
KLOC	6.58	6.68	+2%
FSCA	0.247	0.208	-16%
FTANG	0.200	0.146	-27%
PCOH	0.208	0.273	+31%
PCOUP	424.0	445.0	+5%

The summary results presented in Table 6.2 indicate that the conducted manual feature-oriented remodularization significantly improved three out of four metrics used as evaluation criteria.

In particular, the *Manual* modularization of NDVis reduces the values of the metrics of feature scattering (FSCA) and tangling (FTANG) in packages. These values were reduced by 16% and 27% respectively. The difference in the observed reductions suggests that the performed restructurings were more efficient at separating features from one another than at localizing their implementations to a small number of packages. Overall, these results remain consistent with the qualitative assessment presented earlier, and they reinforce the conclusion about the positive impact of Featureous Manual Remodularization on the feature-oriented modularity of NDVis.

In the case of the two metrics that represent the general properties of cohesion (PCOH) and coupling (PCOUP), the results vary. While it can be seen that the method significantly improved the cohesion of packages, it can also be seen that the amount of coupling was increased. Nevertheless, the extent of this increase is considered relatively low, especially if compared to the improvement of cohesion. Hence, the overall outcome with respect to these metrics is evaluated as positive.

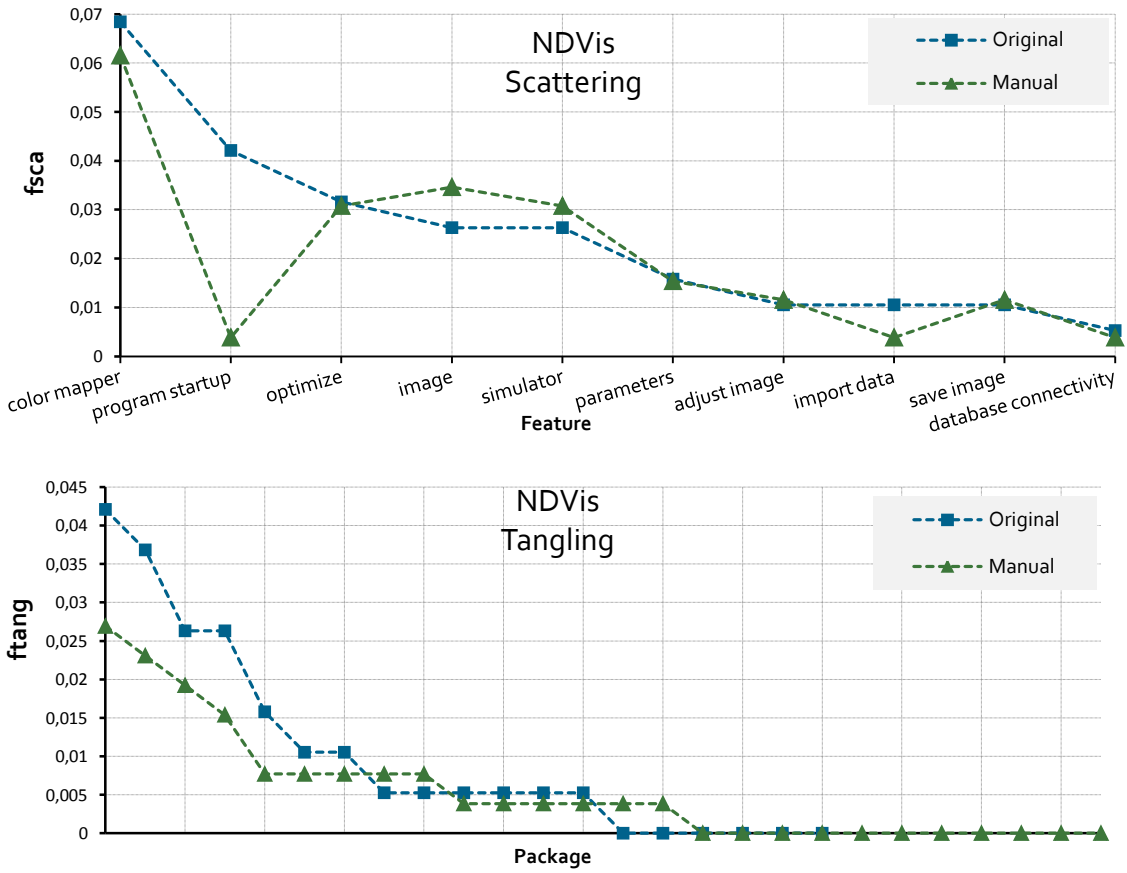


Figure 6.10: Detailed impact of manual remodularization on scattering and tangling in NDVis

The detailed distributions of the scattering and tangling values presented in Figure 6.10 reveal additional insight into the impact of Featureous Manual Remodularization on NDVis. Scattering is presented for each feature separately to allow for direct comparison of individual features between the *Original* and the *Manual* modularizations of the application. As for the distribution of tangling, please note that the individual package names are omitted because the Original and Manual designs consist of two different sets of packages. Since packages present in one of the modularizations are not necessarily present in the other, per-package comparisons are infeasible – instead, only the overall shapes of the distributions of package tangling are investigated.

The scattering plot reveals that manual remodularization succeeded at improving locality of several features, but failed to do so in some cases. The biggest reductions of scattering occurred for the PROGRAM STARTUP feature, followed by IMPORT DATA and COLOR MAPPER. This reveals that the effects of the performed restructurings on reducing feature scattering were highly selective. On the other hand, IMAGE and SIMULATOR became more scattered because of remodularization. A closer investigation revealed that that this negative impact was caused by moving the multi-feature parts of these features to the newly created core modules. These parts were originally grouped together with the single-feature classes of these features. Overall, this result suggests a tension between optimizing the locality of features and facilitating reusability of domain models by forming explicit multi-feature core modules.

The tangling plot displays a significant positive effect of Featureous Manual Remodularization on the tangling distribution of packages in NDVis. Apart from reducing the overall tangling profile, the method resulted in increasing the number of single-feature packages – while in the original application 33% of all packages were single-feature, the performed remodularization increased their proportion to 44%. As it can be observed, seven new packages were added to NDVis in the process.

6.3.4 Summary

In the presented study, the Featureous Manual Remodularization method was applied to conducting feature-oriented restructuring of the NDVis neurological application during its migration to the NetBeans Module System. The industrial collaboration involved, the externally defined aims and the initial unfamiliarity with the target application make the presented study a realistic example of feature-oriented remodularization of an existing application.

Overall, the obtained results indicate a twofold success of the performed restructurings. Firstly, the qualitative properties envisioned for the new modular structure of NDVis by its owners were achieved. Secondly, quantifiable reduction of scattering and tangling of features in source code was observed. Additionally, the presented reports granted several new insights into the challenges involved in feature-oriented restructuring of existing applications. While the generalizability of the results

of this single study is certainly limited, it provides a solid initial evidence for feasibility of the proposed method in application to unfamiliar codebases. These results are considered satisfactory at this stage.

6.4 RELATED WORK

Conceptually, Featureous Manual Remodularization postulates that migrations to module systems should result in a set of modules that allow separating and flexibly composing different chunks of an application's functionality. Therefore, feature-oriented modularization is perceived as an important factor in ensuring evolutionary benefits of applying module systems. A similar conclusion is reached by Turner et al. [4] in their general vision of feature-oriented engineering of software.

The approach that relates the most to Featureous Manual Remodularization is the approach of Mehta and Heineman [69]. They present a method for locating and refactoring features into fine-grained reusable components. Feature location is done by test-driven gathering of execution traces. Features are then manually analyzed with the help of standard object-oriented tools, and manually refactored into components that follow a predefined component model. The aim of this approach is to improve reusability of the features. The main differences between Featureous Manual Remodularization and the approach of Mehta and Heineman are: (1) the method of feature location, (2) the target architectural design and (3) the usage of feature-oriented analysis techniques during the process.

Marin et al. [124] proposed a systematic strategy for migrating crosscutting concerns in existing software to the aspect-oriented paradigm. The proposed approach consists of steps related to that of Featureous Manual Remodularization, namely: concern mining, exploration, documentation and refactoring. To support exploration and documentation of concerns, Marin et al. categorize common types of crosscutting phenomena into several concern sorts. In contrast with Featureous, Marin et al. do not focus on features, but on concerns in general, and focus on extensive reconceptualization of existing classes and methods to covert them to aspects. Marin et al. report on manual aspect-oriented migration of JHotDraw, where, among others, they discuss the case of the *undo* feature. Finally, Marin et al. presented a template-driven tool for semi-automated aspect-oriented refactoring aimed at reducing the complexity of the restructuring process for developers. An exploratory application of the tool to JHotDraw is reported and qualitatively contrasted with the manual results.

Liu et al. [72] proposed the *feature oriented refactoring* (*FOR*) approach to restructuring legacy applications to feature-oriented decompositions. The aim of FOR is to completely separate features in the source code, for creating feature-oriented software product-lines. This is done by reconceptualization centered on *base modules*, which contain classes and introductions, and *derivative modules*, which contain method

fragments. The approach encompasses a firm algebraic foundation, a tool and a refactoring methodology. A case study of manual feature location and refactoring of a 2KLOC application is presented. In contrast with FOR, Featureous Manual Remodularization postulates that relocation of legacy classes should be favored over their reconceptualization, due to the fragile decomposition problem. Therefore, Featureous relies on the Featureous Analysis views to visually explore the contents of multi-feature modules.

Lopez-Herrejon et al. [125] reported their experience with manually refactoring features of two applications to form feature-oriented software product-lines. They identified eight refactoring patterns that describe how to extract the elements of features into separate code units. The proposed patterns constitute relocation at granularities of instructions, methods and classes, and reconceptualization at granularities of methods, classes and packages. Furthermore, Lopez-Herrejon et al. recognize the problem of feature tangling at the granularity of methods and use heuristic-based disambiguation to advise a feature to whose implementation such methods should be relocated to.

Kästner et al. [49] compared their annotative approach based on C/C++-like preprocessor with *AHEAD* – a *step-wise refinement* approach of Batory et al. [126] that is based on composition of template-encoded *class refinements*. The observed drawbacks of the template-based compositional mechanisms include limited granularity and the inability to represent extensions to statements, expressions and method signatures. On the other hand, the annotative mechanisms were found to textually obfuscate the source code and to lack modularity mechanisms. Kästner et al. addressed the code obfuscation problem by employing a colored source code editor to better visualize features.

6.5 SUMMARY

In order to fully support feature-oriented evolutionary changes requested by software users, it needs to be possible to divide work on source code along the boundaries of features, so that modifications to one feature have minimal impact on other features. These properties cannot be achieved without a proper modularization of features in source code. To achieve this, existing applications that do not modularize features can be restructured. Unfortunately, performing such restructurings is often a complex and laborious undertaking, especially in absence of appropriate methods and analysis techniques and tools.

This chapter presented Featureous Manual Remodularization – a method for analysis driven restructuring of Java applications aimed at improving modularization of their features.

The method divided possible restructurings into the ones based on relocation and the ones based on reconceptualization. The fragile decomposition problem was introduced and related to reconceptualization of existing classes. Based on this, Featureous Manual Remodularization developed a feature-oriented design of source code centered on single-feature and multi-feature modules. Featureous Manual Remodularization was applied to the open-source neurological application called NDVis. A number of restructuring experiences were reported and the obtained results were evaluated both qualitatively and quantitatively.

7. FEATUREOUS AUTOMATED REMODULARIZATION

This chapter presents an automated counterpart to the manual remodularization method presented in Chapter 6. The proposed automated method, called Featureous Automated Remodularization, formulates remodularization as a multi-objective optimization problem and addresses it using a multi-objective grouping genetic algorithm and automated source code transformations. Featureous Automated Remodularization is evaluated through application to several open-source Java systems. Furthermore, the method is used to assess the impact of manual reconceptualization of classes on scattering and tangling of features in the NDVis application.

This chapter is based on the following publications:
[CSMR'12], [SCICO'12], [FOSD'09]

7.1 OVERVIEW

This chapter presents *Featureous Automated Remodularization*, a restructuring method that automates feature-oriented remodularization of Java applications. The method supports both *forward remodularization* and *reverse remodularization* of source code.

The forward remodularization is based on formulating the remodularization task as a multi-objective optimization problem. The optimization problem is driven by five metrics that constitute a mixture of feature-oriented and traditional modularization quality indicators. The need for each of the included metrics is discussed. This is used as a basis for developing a multi-objective grouping genetic algorithm and using it to automatically optimize relocation of classes among packages in existing Java applications. The thereby proposed package structures can be reviewed and flexibly adjusted by a developer in the provided UML-like visualization of the *Featureous Remodularization View*. Finally, the resulting feature-oriented package structure is physically established using automated source code transformations.

Reverse remodularization supported by Featureous Automated Remodularization aims at reducing the impact of the problem of syntactic fragility defined in Chapter 6. Namely, after any feature-oriented modifications of the source code requested by the users are performed, reverse remodularization can be used to recover the original modularization of the application. Jointly, the forward feature-oriented remodularization and its corresponding reverse remodularization support on-demand bidirectional remodularization. The goal of this is to enable developers to flexibly select the decomposition best suited for accomplishing a concrete task at hand, whether it modifying a feature or modifying the persistence layer of the application. This on-demand usage of Featureous Automated Remodularization is schematically depicted in Figure 7.1.

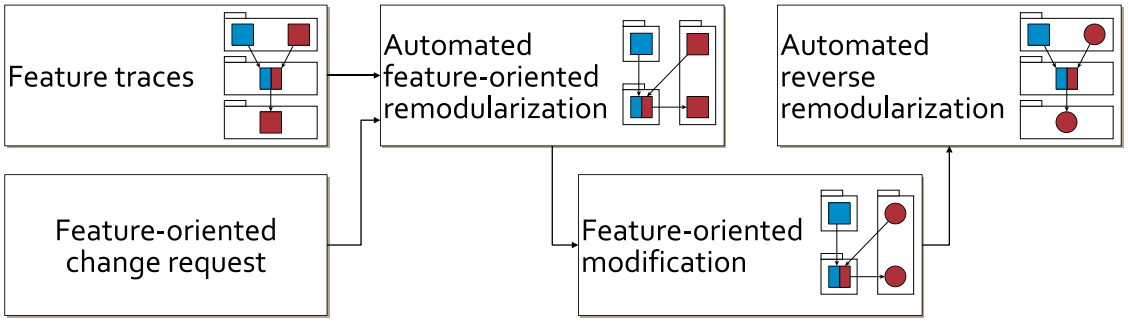


Figure 7.1: Overview of Featureous Automated Remodularization

The Featureous Automated Remodularization method is evaluated using on two case studies. The first one reports in detail on application of the method to remodularizing JHotDraw SVG, and then uses four other medium and large open-source Java systems to generalize the obtained results. The second case study revisits the manual

restructuring to the NDVis from Chapter 6 to compare the manual method with the automated one, and to assess the role of class reconceptualization during the manual remodularization.

7.2 FORWARD REMODULARIZATION

The basis for automated forward feature-oriented remodularization of existing applications rests upon the following three observations:

1. Existing software metrics provide objective means of evaluating the adherence of an application to various design principles, such as high cohesion, low coupling, low scattering and low tangling.

2. Based on existing metrics, and given a set of permitted restructuring operators, it is possible to explore the space of alternative application modularizations to search for the ones that optimize the values of the metrics.

3. Automated source code transformation can be applied to physically transform the original application into an identified optimized modularization.

As pointed out by Clarke et al. [127], the diversity of existing software metrics defined in literature can be used for purposes different from comparing *multiple applications* against each other. Metrics can also be used to compare multiple modularization alternatives of a *single application*. Thus, assuming that all modularization alternatives of an application can somehow be enumerated, software metrics make it possible to choose the one among them that performs best according to a set of desired criteria.

However, in practice enumerating all modularization alternatives quickly becomes infeasible as applications grow in size and complexity. This makes it necessary to adopt a sensible way of traversing the space of promising modularizations. As demonstrated by Harman and Clark [128] and by O'Keeffe and O'Cinneide [77], a number of algorithms, such as hill climbing, simulated annealing and genetic algorithms can be used to find desirable solutions in the space of modularization alternatives. The two prerequisites to practical application of such approaches are to define an appropriate objective function and to develop a set of restructuring operators capable of transforming one modularization alternative into another.

The forward remodularization mode of Featureous Automated Remodularization builds upon the mentioned proposals of search-based software engineering. Having this general conceptual frame as the starting point, Featureous Automated Remodularization makes several design decisions to practically apply it in the context of bi-directional feature-oriented remodularization.

The Aim of Featureous Automated Remodularization

Featureous Automated Remodularization aims at establishing explicit representations of features in package structures of Java applications to facilitate feature-oriented comprehension, modification and work division during software evolution. Hence, the desired package structures of the proposed method resemble the one presented in Figure 7.2, which was originally defined and demonstrated as feasible in Chapter 6.

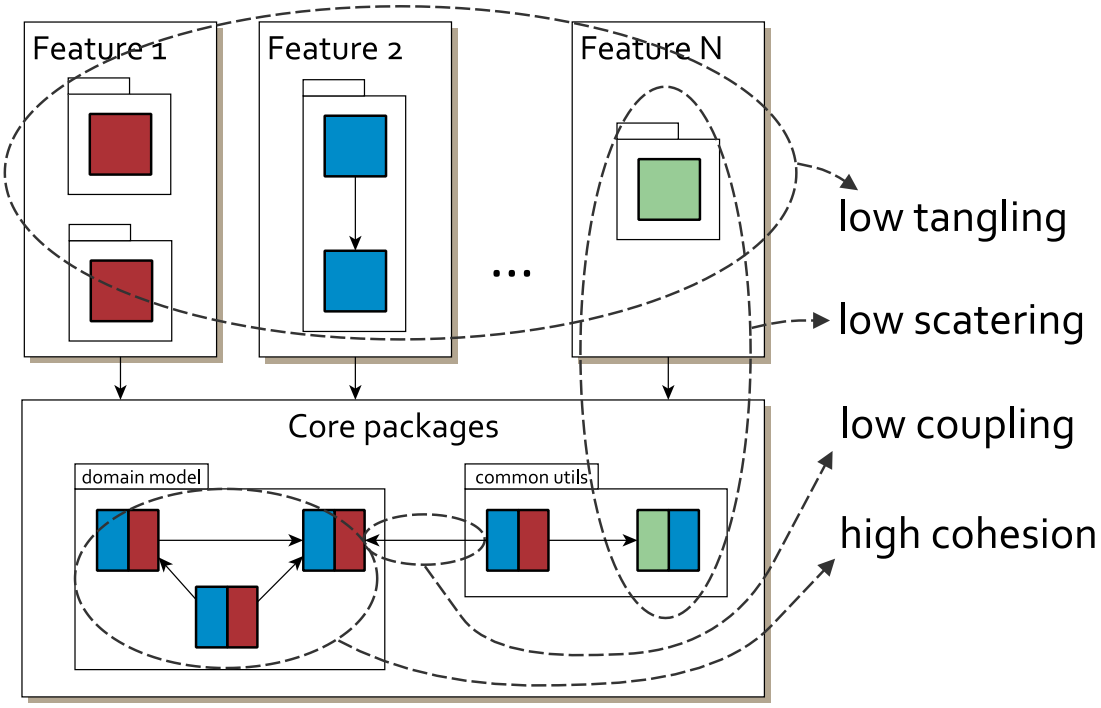


Figure 7.2: Feature-oriented design based on four design principles

As discussed in Chapter 6, the feature-oriented modularization in Figure 7.2 consists of a set of *single-feature packages* that explicitly represent and separate features from one another. Each such single-feature package depends on a number of *multi-feature packages* that enclose reusable multi-feature domain models and utilities. Hence, this modularization can be generally characterized as striving for minimizing *scattering* and *tangling* of features and at ensuring low *coupling* and high *cohesion* of the packages.

Since it is possible to quantify the four mentioned qualities using readily available software metrics, it becomes possible to formulate the search for the proposed feature-oriented design as a multi-objective optimization problem. This formulation is discussed in Section 7.2.1. Subsequently, Section 7.2.2 discusses how to automatically solve such a problem using a multi-objective grouping genetic algorithm.

On the technical side, Featureous Automated Remodularization employs the standard packages of the Java language as the syntactical modularization mechanism. Packages possess the core properties necessary for this purpose: the ability to group classes, to

separate the groups from one another, and to control the access to classes being grouped. Moreover, the package construct is readily available and commonly adopted, since it is a part of the Java language specification.

Remodularization through Relocation of Classes

Motivated by the semantic variant of the fragile decomposition problem discussed in Section 6.2.3, Featureous Automated Remodularization is designed to preserve the original definitions of classes without splitting them into fragments. As discussed previously, this is aimed at preserving the direct correspondence between classes and human-understandable domain concepts. The consequence for the proposed method is that the search space is thereby constrained to modularizations that can be achieved by sole relocation of classes. Apart from reducing the size of the search space, this assumption makes the involved source code transformation significantly simpler.

Nevertheless, constraining the restructuring operators to only relocation of classes comes at the cost of the inability to separate features tangled at the granularities of classes, methods and individual statements. Hence, any further finer-grained reconceptualization of classes and methods is left up to the decisions and manual efforts of a developer. As demonstrated in Chapter 6, human expertise is essential to decide on how to split existing classes and methods into meaningful fragments and what semantics to equip them with.

7.2.1 Remodularization as a Multi-Objective Optimization Problem

Featureous Automated Remodularization formulates feature-oriented remodularization as a multi-objective optimization of grouping classes in terms of packages. This formulation encompasses three feature-oriented objectives and two traditional object-oriented ones.

The optimization objectives used by Featureous Automated Remodularization are encoded as independent software metrics. The value of each of these metrics has to be either minimized or maximized in course of optimization. Four of these metrics (i.e. FSCA, FTANG, PCOH and PCOUP) were already discussed in detail in Chapter 6. In addition, one new feature-oriented metric called FNCR is included. The definitions of all the used metrics together with their optimization goals (either maximization or minimization) listed in Figure 7.3.

The newly proposed FNCR metric stands for *feature-oriented non-common reuse of packages*. Conceptually, this metric constitutes a feature-oriented counterpart to the *common reuse principle* of Martin [42] that postulates that classes reused together should be packaged together. In the feature-oriented terms, this becomes *classes reused by the same set of features should be packaged together*. Hence, FNCR is designed to detect cases of incomplete reuse of classes by features within packages. This is done according to the definition presented in Figure 7.3, i.e. for each package it is determined how many features use at least one class contained in the package, but do

not used all of the classes contained in it. Hence, the higher the value of *fncr* for a given package, the more there is features that use only a subset of its classes. The obtained value is additionally normalized with the number of all features related to the package of interest. A sum of the values for individual packages is used to aggregated the metrics to application level. During optimization, the value of FNCR has to be minimized to identify modularization alternatives that avoid feature-oriented non-common reuse of packages.

$$\boldsymbol{FSCA}(F) = \sum_{f \in F} fsca(f)\,,\ \ where: \boldsymbol{fsca}(f) = \frac{|\{p \in P_F : f \rightsquigarrow p\}| - 1}{|F| \cdot |P_F|} \qquad \rightarrow \textbf{min}$$

$$\boldsymbol{FTANG}(P_F) = \sum_{p \in P_F} ftang(p)\,,\ \ where: \boldsymbol{ftang}(p) = \frac{|\{f \in F : f \rightsquigarrow p\}| - 1}{|F| \cdot |P_F|} \qquad \rightarrow \textbf{min}$$

$$\boldsymbol{FNCR}(P_F) = \sum_{p \in P_F} fncr(p),\quad where: \boldsymbol{fncr}(p) = \frac{\left|\{f \in \{f \in F : f \rightsquigarrow p\} : \exists_{t \Rightarrow p} \neg (f \rightsquigarrow t)\}\right|}{|\{f \in F : f \rightsquigarrow p\}|} \qquad \rightarrow \textbf{min}$$

$$\boldsymbol{PCOH}(P) = \frac{\sum_{p \in P} \underset{T \Rightarrow p}{pcoh(p,T)}}{|P|}\,,\quad where: \boldsymbol{pcoh}(p,T) = \frac{\sum_{t1 \in T} \underset{t1 \Rightarrow p}{} \sum_{t2 \in T} \underset{t2 \Rightarrow p}{} \left|DD_{t1,t2} \cup DM_{t1,t2}\right|}{\sum_{t1 \in T} \underset{t1 \Rightarrow p}{} \sum_{t2 \in T} \underset{t2 \Rightarrow p}{} \left|MaxDD_{t1,t2} \cup MaxDM_{t1,t2}\right|} \qquad \rightarrow \textbf{max}$$

$$\boldsymbol{PCOUP}(P) = \sum_{\substack{p \in P \\ T \Rightarrow p}} pcoup(p,T)\,,\quad where: \boldsymbol{pcoup}(p,T) = \sum_{\substack{t1 \in T \\ t1 \Rightarrow p}} \sum_{\substack{t2 \in T \\ t2 \nRightarrow p}} \left|DD_{t1,t2} \cup DM_{t1,t2}\right| \qquad \rightarrow \textbf{min}$$

Figure 7.3: Multi-objective formulation of feature-oriented remodularization

The remainder of this section motivates the need for each of the incorporated optimization objectives.

The Role of Minimizing Scattering and Tangling

The purpose of the pair of feature-oriented metrics FSCA and FTANG is to confine individual features in terms of a small number of packages, while separating them from each other.

It is simple to demonstrate that both these metrics are need, as exclusion of any of them leads to undesirable extreme decompositions. This desired conflicting nature of these metrics forces the optimization process to seek tradeoff solutions that will not sacrifice any of the properties at a cost of the other. The two first modularizations presented in Figure 7.4 are examples of extreme decompositions created by optimizing for each of the metrics in isolation. The last modularization represents a balanced tradeoff between them.

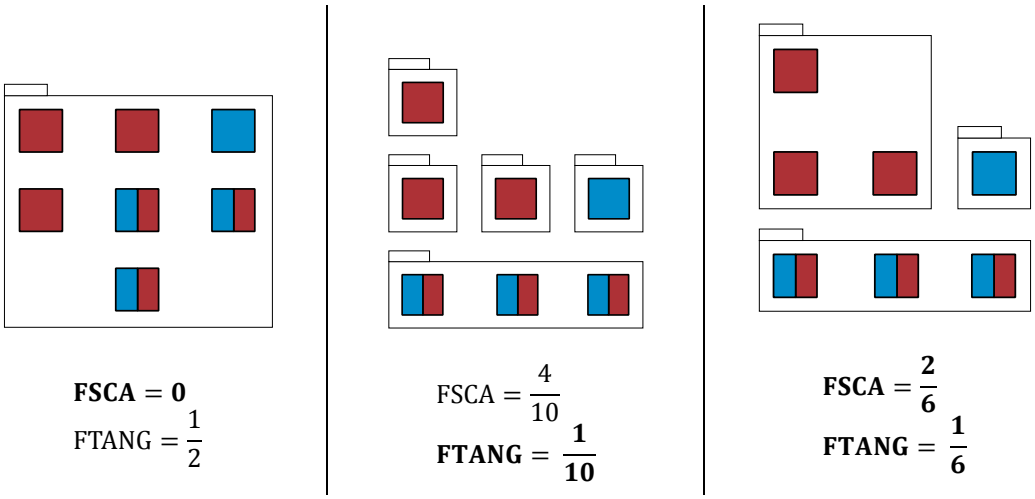


Figure 7.4: The complementary nature of scattering and tangling

The first modularization consisting of a single package that encloses all classes – regardless of the features they implement. As can be seen, such a design is optimal with respect to the objective of reducing feature scattering, as all the features are localized in a single package. However, such a design can hardly be called a reasonable modularization.

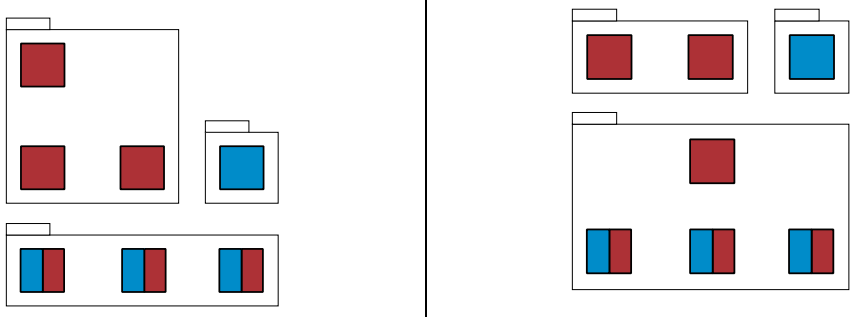
On the other hand, optimizing solely for reduction of tangling promotes another extreme. As shown the second modularization in Figure 7.4, the metric of tangling is optimized by moving each single-feature class to its own package and keeping all multi-feature classes grouped within a single package. While in the presented toy example the thereby introduced delocalization of features may not seem particularly severe, it will surely not be desired in applications containing hundreds or thousands of single-feature classes.

To avoid the extreme characteristics of the mention modularizations, a tradeoff between reducing scattering and reducing tangling needs to be found. This can be done by optimizing the modularization for both FSCA and FTANG metrics simultaneously. One example of such a tradeoff modularization is included in Figure 7.4. The presented tradeoff modularization is Pareto-optimal with respect to both of the metrics.

The demonstrated complementary nature of FSCA and FTANG clearly demonstrates the need for multi-objective optimization in Featureous Automated Remodularization.

The Role of Minimizing Feature-Oriented Non-Common Reuse

Based on the outcome of the previous example it is possible to demonstrate the need for the FNCR metric. The purpose of FNCR is to minimize the incomplete usage of packages by features. The difference between violation and adherence to FNCR is depicted using the two modularizations in Figure 7.5.



$$\text{FSCA} = \frac{2}{6}$$
$$\text{FTANG} = \frac{1}{6}$$
$$\textbf{FNCR} = \mathbf{0}$$

$$\text{FSCA} = \frac{2}{6}$$
$$\text{FTANG} = \frac{1}{6}$$
$$\textbf{FNCR} = \frac{\mathbf{1}}{\mathbf{2}}$$

Figure 7.5: The need for the feature-oriented non-common reuse criterion

As can be seen, the first of the presented modularizations correctly groups all single-feature classes within their respective single-feature packages. In contrast, the second modularization misplaces one single-feature class by including it in a multi-feature package. Interestingly, this difference remains undetected by the FSCA and FTANG metrics, because the relocation of the single-feature class did not expand the original set of features associated with the affected multi-feature package.

In order to account for such obliviousness of FSCA and FTANG, the FNCR metric is used. As can be seen in Figure 7.5, the undesired relocation of the single-feature class to the multi-feature package is successfully detected by FNCR. The increase in the FNCR value is obtained for the second design as a result of detecting incomplete usage of the set of classes contained in the multi-feature package by the feature marked in blue.

The Role of Maximizing Cohesion and Minimizing Coupling

In addition to the three purely feature-oriented criteria, two object-oriented metrics PCOH and PCOUP are used. While the feature-oriented metrics are agnostic to static dependencies among units of source code, the metrics of cohesion and coupling ensure that such dependencies are taken into account. Thus, the general responsibility of PCOH and PCOUP is to assure that the created packages are cohesive and loosely coupled.

The second, less obvious purpose of including the criteria of cohesion and coupling is ensuring that Featureous Automated Remodularization finds appropriate packages for classes that are not covered by captured feature traces of an application. As non-covered classes are ignored by feature-oriented metrics, using non-feature-oriented metrics, such as cohesion and coupling, is the only way to meaningfully handle such classes during remodularization. In general, non-covered classes are an outcome of the

difficult nature of maximizing coverage of dynamic feature location mechanisms, as was discussed in detail in Chapter 4.

Figure 7.6 demonstrates how relying on cohesion and coupling allows to meaningfully handle a non-covered class by assigning it to a package to which it is are related the most. The first modularization shown in the figure extends the earlier examples by considering static dependencies among classes and presence of non-covered classes.

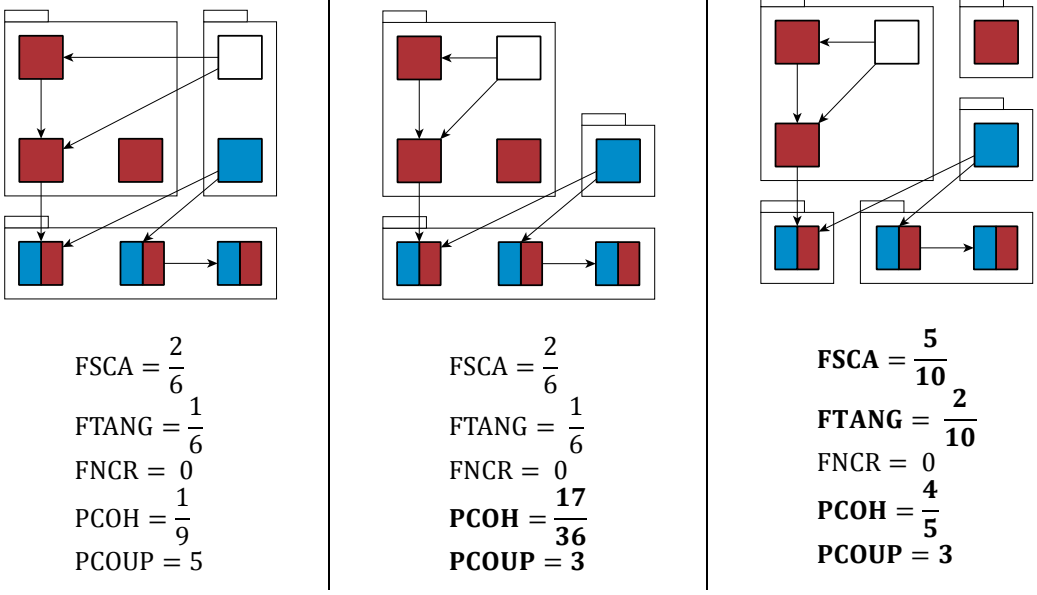


Figure 7.6: The role of cohesion and coupling

As can be seen in the second modularization in Figure 7.6, by relocating the single non-covered class to its related package it is possible to reduce coupling PCOUP and increase cohesion PCOUP, while preserving the original values of scattering and tangling.

Finally, including the criteria of cohesion and coupling ensures appropriate granularity of the packages created by Featureous Automated Remodularization. By aiming at a balance between cohesion and coupling, the method will promote splitting overly large non-cohesive packages and merging too small and extensively coupled packages. Guiding the optimization process to seek appropriate granularity is particularly important in the case of multi-feature packages, as the metrics of scattering and tangling by themselves will tend to enclose all multi-feature classes in a single package, as demonstrated earlier. Secondly, the criteria of cohesion and coupling are considered as means to establishing meaningful division of sets of single-feature classes among several packages. Example outcomes of such effects of cohesion and coupling are presented in the last design in Figure 7.6.

Metrics As Fitness Functions

To form a firm basis for using software metrics as means of driving optimization algorithms, Harman and Clark [128] proposed an approach called *Metrics As Fitness Functions* (MAFF). MAFF defines a set of four properties for a metric to be useful as a fitness function: *large search space*, *low computational complexity*, *approximate continuity*, *absence of known optimal solutions*. As will be demonstrated, the five metrics used in the proposed formulation of feature-oriented modularization posses these properties.

The five used metrics can distinguish individuals in *large search spaces*. Because the problem of interest is that of distributing N classes over M packages, the total size of the search space grows exponentially with code size and equals in general case to M^N. Hence, for any non-trivial application an exhaustive exploration of its search space is infeasible.

The five used metrics have relatively *low computational complexity*. Not only is the algorithmic complexity of all the metrics lower or equal to $O(n^2)$, but the actual calculation of the metrics is done on a simplified representation of an application. This representation consists of feature-trace models, which were defined in Chapter 4, and simple static dependency models of the actual classes and static relations among them. These simple static dependency models are discussed in Appendix A1. Based on these two kinds of models, a population of alternative modularizations of an application can be represented and efficiently measured, without the need for physically transformation the source code first. Instead, during the optimization process only simple transformations of the two models are applied.

The five used metrics are equipped with the property of *approximate continuity* by their definitions. Hence, given an appropriate measurement target, the metrics can be made to return any value from their value ranges, being <0;1> in the cases of FSCA, FTANG, PCOH, and <0;+inf> in the cases of FNCR and PCOUP.

Finally, while the individual metrics have known optimal solutions (e.g. one package per application minimizes scattering) their joint usage in a multi-modal objective function does not have a general *known optimal solution*. As it was discussed in the previous subsections, this was achieved by adopting pairs of complementary metrics that encode contrasting optimization objectives. As was furthermore demonstrated, the adopted pairs of metrics are not trivially correlated; .i.e. optimization of scattering and tangling does not automatically guarantee an improvement of cohesion and coupling, and vice versa.

7.2.2 Multi-Objective Grouping Genetic Algorithm

For solving the described optimization problem Featureous Automated Remodularization relies on a particular type of genetic algorithm. Based on existing works in the field of genetic computation, a formulation is proposed that merges the

multi-objective optimization based on the concept of *Pareto-optimality* with a set of *genetic operators tailored to solving grouping problems* [79]. The feasibility of the first approach was of demonstrated by Harman and Tratt [80], while the feasibility of the second was demonstrated by Seng et al. [78]. Featureous Automated Remodularization combines these two formulations to leverage their individual advantages. This joint formulation is referred to as *multi-objective grouping genetic algorithm (MOGGA)*. MOGGA is applied by Featureous Automated Remodularization in the previously unexplored context of automated feature-oriented remodularization.

Package Structure Representation

On the most abstract level, MOGGA follows the traditional formulation of a genetic algorithm [129]. In short, this formulation assumes evolving a population of individuals by means of *selection*, *reproduction* and *mutation* driven by the score of the individuals with respect to a fitness function. Analogously to the proposal of Seng et al. [78], each individual in a population being evolved by MOGGA is made to represent a particular distribution of classes among packages. Physically, each individual is represented as an array of numbers. Classes are represented as indexes of the arrays, whereas their assignment to packages is represented by the values of the corresponding array cells. The used representation scheme is exemplified in Figure 7.7 using three classes and two packages.

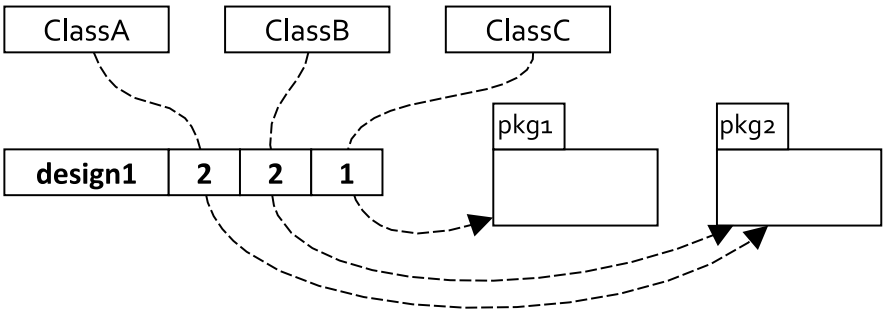


Figure 7.7: The representation of grouping classes into packages

Genetic Operators

Since the optimization of the assignment of classes to packages is an instance of a grouping problem [79], MOGGA adapts two genetic operators to this class of problems. As demonstrated by Seng et al. [78], doing so significantly improves the efficiency of traversing the search space of alternative modularizations.

Firstly, the crossover operator that forms two children from two parents is made to preserve packages as the building blocks of modularizations. Namely, the crossover operator makes individual modularizations exchange whole packages, rather than individual classes. The pairs of input individuals are chosen randomly, while prioritizing the individuals proportionally to their fitness. The usage of the crossover

operator is schematically depicted in Figure 7.8, where the complete package *2* from `design1` is inserted into `design2`.

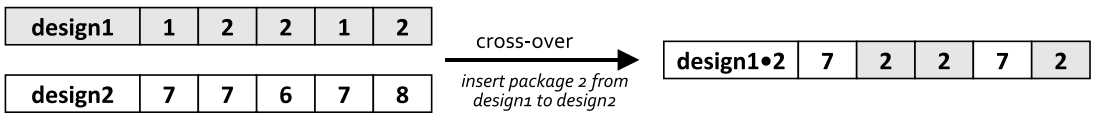


Figure 7.8: The grouping crossover operator

Secondly, a mutation operator is defined to randomly perform one of the three actions: merge two packages with the smallest number of classes, split the largest package into two packages of adopt an *orphan class* [74] being alone in a package into another package. Example application of the individual variants of this operator is depicted in Figure 7.9.

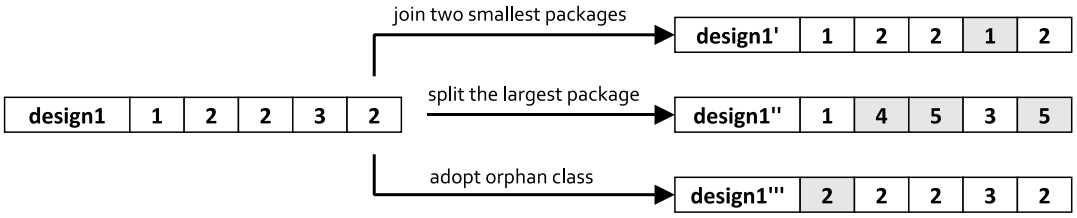


Figure 7.9: The grouping mutation operator

Multi-Objective Evaluation of Fitness

Evaluation of the fitness of the individual modularization alternatives is done by computing the five metrics discussed in the previous section. Assigning the values of these metrics to each individual creates a basis for assessing and selecting the individuals representing the best modularizations of an application.

A simplistic approach to doing so would be to define an aggregate function that sums the values of the five involved metrics to represent them jointly as a single value. However, this approach has a number of drawbacks. Firstly, on the numerical level, the metrics yield different value ranges. This may lead to dominance of one metric that has unbounded maximum value, such as PCOUP, over metrics that operate on limited ranges of values, such as FSCA, which returns values in range <0; 1>. Resolving this situation is not possible without modifying the original formulations or introducing dedicated scaling factors for each of the metrics. Secondly, even if the value ranges of metrics can be faithfully aligned, their aggregation inherently masks the distribution of individual values. Hence, the value of 10 obtained through aggregation of two metrics may represent the distribution {5, 5}, as well as the distribution {0,10}. As demonstrated by Zhang et al. [130], this leads to promoting extreme solutions that optimize one of the objectives at the expense of others.

In order to appropriately represent the regions of the five-dimensional search space that the individual modularizations in the population occupy, MOGGA adopts the concept of Pareto-optimality. Thanks to this, the unaltered value of each metric is preserved, and hence the fitness of each individual becomes a tuple consisting of five independent elements. Such a multi-modal fitness can be used as a criterion for comparing individuals thanks to the Pareto-dominance relation, which states that one out of two individuals is better than the other individual, if all of its fitness values are not worse, and at least one of the values is better. Thereby, it becomes possible to partially order individuals according to their quality and to determine the set of non-dominated individuals in a population, the so-called Pareto-front.

Initialization and Choosing a Final Solution

MOGGA uses the following four groups of modularizations to create the initial population being evolved. This ensures availability of several diverse starting points for evolutionary optimization. For a given size of the population:

- 2% of individuals in the initial population are made to correspond to the original modularization of the application being remodularized.
- 2% of individuals are made to correspond to a package structure that relocates all single-feature classes from their original packages to explicit single-feature packages.
- 2% of individuals are created by reusing randomly-selected solutions from a Pareto front obtained in the previous run of MOGGA (if available)
- All the remaining individuals of the initial population are created as randomized assignments of classes to packages.

Starting with the initial population, a predefined number of evolutionary iterations is executed. Then the last Pareto-front is used to select a single individual being the optimization result. This is done by ranking the individuals in the obtained five-dimensional Pareto front with respect to each metric separately, and then choosing the individual that is ranked best on average. Please note that while the this method is used due to its simplicity, the existing literature defines a number of other methods for choosing a single solution out of a Pareto-front, e.g. [131], [132].

7.2.3 Transformation of Source Code

The final solution identified by MOGGA represents an optimized assignment of classes to a new set of packages. This optimized package structure is then physically established in the source code of an application by the means of automated transformation.

The Recoder code transformation library [133] is used to relocate classes among packages, rename packages, create new packages and remove empty packages. These

transformations are implemented on top of the Recoder's code model that hides the low-level details of the actual abstract syntax trees of Java source code. The implemented move-class transformation not only renames the package declaration in class declarations, but also synchronizes the dependent classes accordingly, and ensures the correctness of import statements. All the implemented transformations are made to preserve the source code comments and formatting.

As explained in Chapter 6, relocation of classes is equal to reconceptualization of their corresponding packages. One of the challenges of such reconceptualization is the need for choosing meaningful names for the newly created or modified packages. In general, doing so requires human expertise, as it involves establishing human-understandable abstractions that will capture the intent behind particular groupings of classes. Featureous Automated Remodularization uses a simple heuristic to partially address this problem. This is done as follows:

- Name all the resulting single-feature packages using the names of their corresponding features. In the case of multiple single-feature packages per feature, append a unique numerical identifier (e.g. `save_image`, `save_image2`).
- Among the remaining multi-feature packages, identify all packages whose contained class set "is similar" to a set of classes contained in one of the original packages of an application. For the sake of simplicity, the mentioned "is similar" relation is defined as the values of the Jaccard similarity coefficient greater or equal to 0.6. All the identified packages are then named after their similar package from the original application.
- Name all the remaining packages using the indexed `pkg` identifier (e.g. `pkg1`, `pkg2`). Designing meaningful names for these packages becomes then the responsibility of a developer, who can do this directly from the Featureous Remodularization View that will be presented in Section 7.3.

7.2.4 Handling Class Access Modifiers

As Featureous Automated Remodularization alters the existing package structure of an application, it also invalidates the original criteria for assigning access control at the boundaries of packages. Hence, the access control needs to be adjusted for the newly created packages. The two particular concerns addressed with respect to access modifiers are syntactic consistency of the resulting applications, and further tightening of access restrictions for improving encapsulation of features.

To start with, the compile-time syntactical consistency of the application is ensured by handling the *default* and *protected* scope modifiers. If they are present in the original application, they constrain the remodularization process, since too restrictive usage of them can cause compilation errors in remodularized applications. This occurs when, for instance, a class in the original application decomposition has the *default* scope and is used only by other classes within the same packages. If such a class is

moved out of its package during remodularization, a compile-time error will be produced, unless all its dependent classes will be moved accordingly or unless its access restrictions are relaxed. This issue is dealt with by replacing *default* and *protected* access declarations with the *public* modifier, which removes the access restrictions.

After any required relaxation of access restrictions, it becomes important to consider the opposite process – i.e. using access modifiers to improve encapsulation of features in code. This is done by reducing the visibility of types and methods that are used exclusively in terms of single-feature packages from public to package-scoped. It is worth noting that a more sophisticated encapsulation of features (e.g. using access modifiers proposed in [134]) would have been possible if compatibility with Java language specification were not a concern to Featureous Automated Remodularization.

7.3 FEATUREOUS REMODULARIZATION VIEW

Featureous Automated Remodularization is implemented by the *Featureous Remodularization View* plugin to the Featureous Workbench. In addition to implementing the proposed method, the view facilitates its application in several ways.

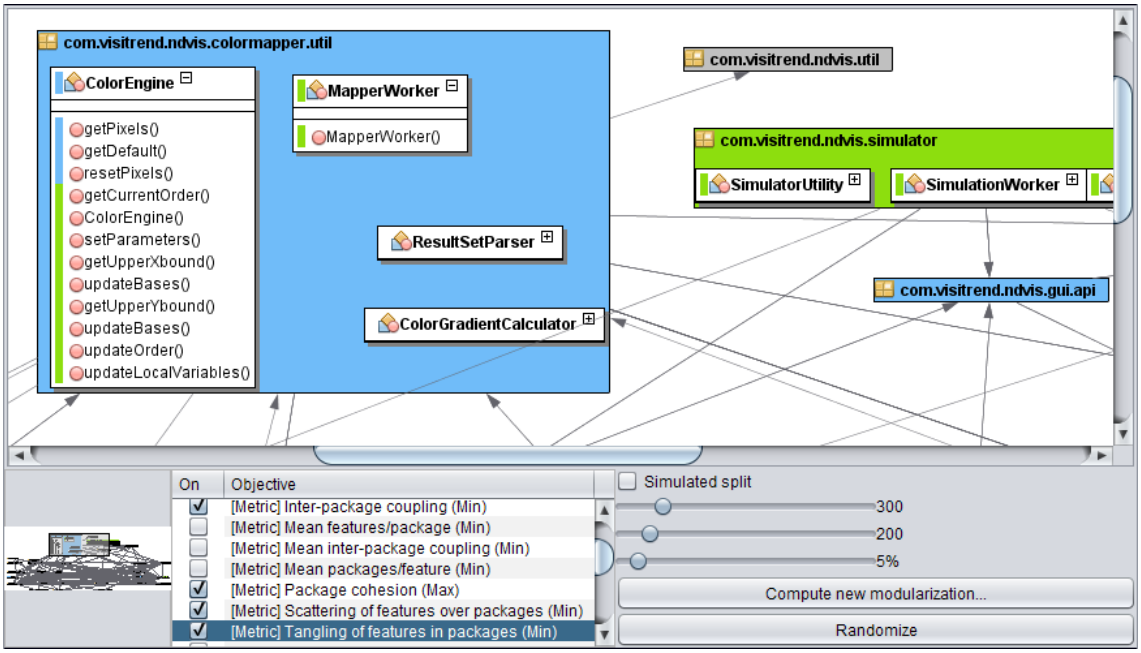


Figure 7.10: Featureous Remodularization view plugin to Featureous

The user interface of Featureous Remodularization view is centered on a graph-based representation of the package structure of an application. As shown in Figure 7.10,

this representation depicts affinity-colored packages (nodes) and static dependencies among them (edges). The individual packages can be unfolded to reveal their enclosed affinity-colored classes and to refine the mapping of static dependencies accordingly. Furthermore, each class can be unfolded to reveal a UML-like representation that lists all the affinity-colored method declarations. Lastly, as shown in Figure 7.11, the view makes it possible to zoom in on individual unfolded classes to directly display their source code in a full-fledged NetBeans source code editor. When source-level inspection is no longer necessary, the code editor is dismissed by zooming out.

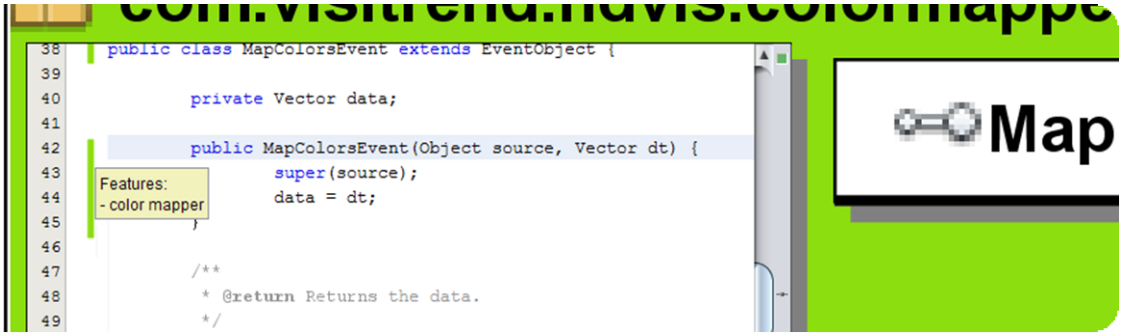


Figure 7.11: Source code editor within the package structure view

The diagrammatic representation is used to visualize the package structures of both an original application through a *pre-remodularization* tab and of a remodularized application through a *post-remodularization* tab.

Apart from the visual representation of package structure, the pre-remodularization tab allows for selection of optimization objectives and for configuration of the parameters of MOGGA. The configurable parameters are (1) the number of iterations to be executed, (2) the size of evolved population and (3) the probability of mutation. Apart from the five metrics that are a part of the Featureous Automated Remodularization method, the view readily implements several other metrics that can be used as additional remodularization objectives in future studies.

After the remodularization process is invoked and finished, Featureous Remodularization View displays the result in a dedicated post-remodularization tab. As shown in Figure 7.12, this view consists of the discussed diagrammatic visualization and a report window that summarizes the difference between the original and the remodularized application using a number of metrics.

In addition to visually inspecting the results, the view makes it possible for developers to manually adjust a proposed package structure before it is being physically established in the source code. This can be done by renaming packages and by dragging classes between packages to relocate them. During the relocation of classes, the values of metrics displayed below the structural visualization are updated accordingly to immediately reflect the effects of performed adjustments.

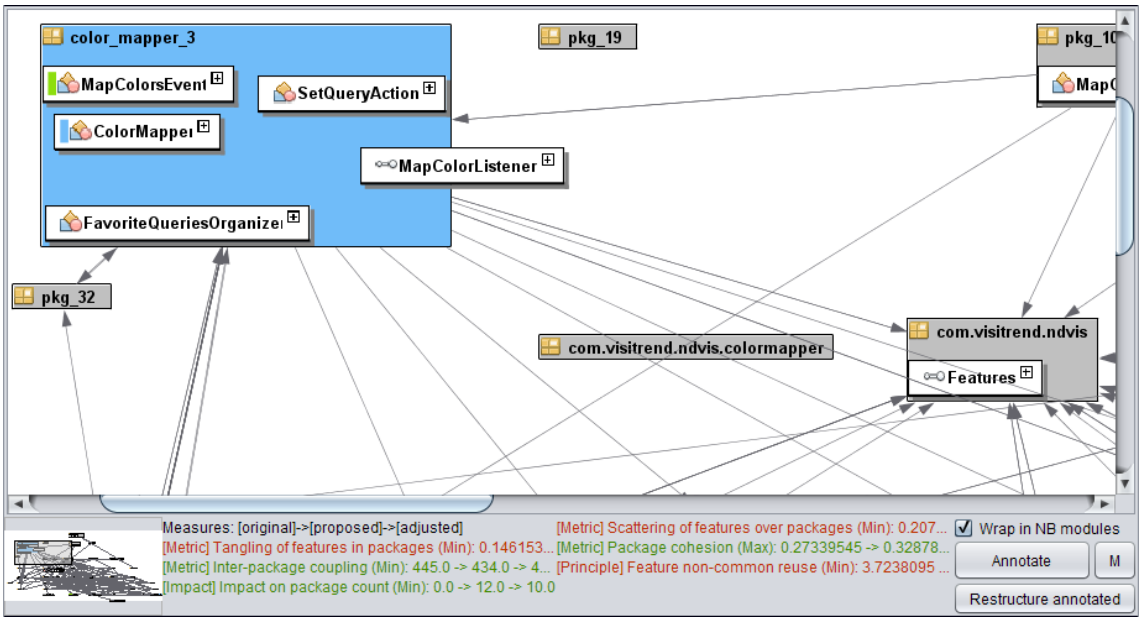


Figure 7.12: Post-remodularization tab of Featureous Remodularization View

7.4 REVERSE REMODULARIZATION

The reverse restructuring aims at re-establishing the original decomposition of an application from its remodularized version. This is done by automatically re-creating the original package structure of an application and automatically re-assigning classes to their original packages. This is performed based on two inputs, as visualized in Figure 7.13.

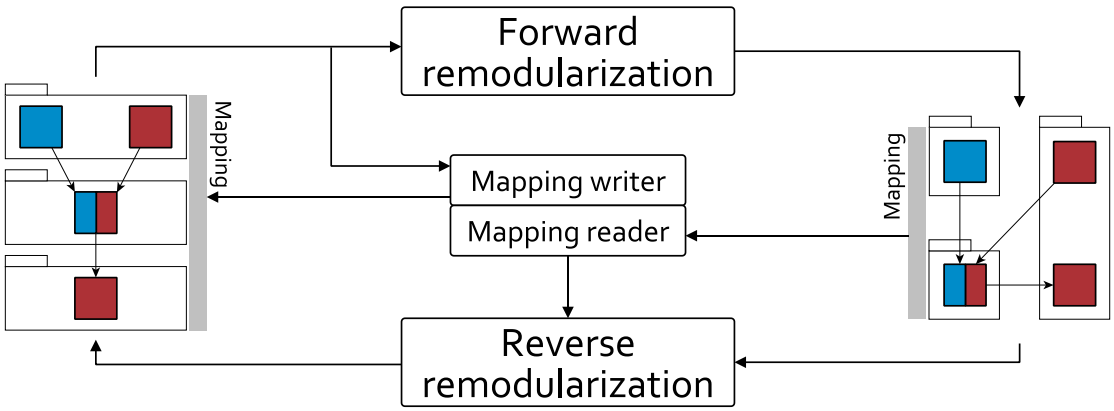


Figure 7.13: Bidirectional remodularization

The first input required by reverse remodularization is the source code of the feature-oriented application decomposition. This is decomposition is created by the forward remodularization, as discussed in Section 7.2.

The second piece of information needed by reverse remodularization is the mapping between classes of the feature-oriented application decomposition and the packages in the original application decomposition. Creation of this mapping is done by pre-processing the source code of the original application decomposition prior to forward remodularization. The information about the names of the original packages that individual classes reside in is extracted by a *mapping writer* and then stored as `@OrgPkg` Java annotations on class declarations in the source code. The annotations are parameterized by the string-based qualified name of the original package of the class. After the forward remodularization and any required modifications of source code are performed, the persisted package-class mapping is read by a *mapping reader* and passed as one of the parameters to the *reverse remodularization* routine. The original package structure is then restored, based on the read mapping of classes to original packages and on the source code of the feature-oriented application decomposition.

Since presence of these annotations is idempotent to the forward remodularization, the process of creating class-package mappings as well as reverse remodularization can be flexibly attached or detached from the forward remodularization process to support either the forward- or the bidirectional remodularization scenarios – depending on the needs of a software developer.

In the bidirectional remodularization scenario, where the feature-oriented decomposition is introduced to support subsequent feature-oriented modifications, it is necessary to consider the impact of these modifications on the reverse remodularization. All possible modifications made by a developer to a remodularized application, such as changing the names of classes or adding new classes, have to be handled in a meaningful fashion during reverse remodularization. The rename operation is supported by attaching the package-class mappings in form of annotations to class definitions. Since annotations are a part of the Java language's syntax, they are part of the source code itself and thus they will remain syntactically attached to its class despite of changing the name of the class or modifying its implementation.

The behavior of reverse remodularization is, however, unspecified for the cases when a developer creates a new class or modifies the semantics of an existing class (e.g. re-purposing a business logic class into a utility class). To correctly resolve such situations, the method relies on developers to decide on a package of the original decomposition to which such a class belongs and declare this decision by manually annotating the target class. This policy is sufficient for the scope of the current investigations, yet it remains possible to replace it in the future with a more sophisticated one (e.g. clustering based on static cohesion and coupling, co-change, etc.).

Similarly, as in the case of forward remodularization, the reverse remodularization method relies on automated source code transformations implemented using the Recoder library.

7.4.1 Recovering Original Access Modifiers

Due to the manipulation of the access modifiers in the code during forward remodularization, it becomes necessary to include a mechanism for recovering the original access restrictions in the code. This is done by capturing the original access-modifier information, together with the package-class mappings, during the pre-processing phase. This information is captured using another annotation called `@OrgAccess`. This annotation is placed on the method- and class-declarations of the original application by the same routine that places the `@OrgPkg` annotations. It is parameterized by a string value that corresponds to the declaration's access modifier. Consequently, during the code transformation from the feature-oriented to the original decomposition this data is retrieved and used to re-establish the original access control. In the case of introduction of new methods or classes in the feature-oriented decomposition, it is again up to the developer to supply appropriately parameterized annotations.

7.5 EVALUATION

This section reports on a case study of applying Featureous Automated Remodularization to a medium-sized open-source application JHotDraw SVG. After presenting the detailed results of the JHotDraw SVG study, this section generalized the presented finding by applying Featureous Automated Remodularization to four other medium and large open-source Java applications.

7.5.1 Recovering Traceability Links

Applying Featureous Location to recover traceability links between features and source code units was the only manual step required by Featureous Automated Remodularization. The precise actions taken during this process and their outcomes were reported earlier in Section 4.5. In summary, 211 out of 332 classes of JHotDraw SVG were covered by feature traces. 59 of them were identified as single-feature and 152 as multi-feature.

As discussed in Section 7.2.1, the set of optimization objectives incorporated in Featureous Automated Remodularization is designed to compensate for incomplete coverage of dynamic analysis. This is done through the object-oriented objectives of high cohesion and low coupling that promote placement of the non-covered classes together with their mostly related covered classes. Doing so addresses the issue of incomplete coverage of feature location.

7.5.2 Remodularization Results

Prior to applying automated remodularization, the original package structure of JHotDraw SVG was measured using the FSCA, FTANG, FNCR, PCOH and PCOUP metrics. Analogously, a similar measurement was performed after applying remodularization. Featureous Automated Remodularization was applied by repetitive execution of MOGGA on a population of 300 individuals over 500 evolutionary iterations with mutation probability of 5%. A comparative summary of the obtained results is shown in Table 7.1, whereas the detailed distributions of scattering and tangling values are presented in Figure 7.14.

Table 7.1: Summary impact on feature representation's quality

	JHotDraw SVG		
Metric	Original	Automated	Δ%
FSCA	0.361	0.162	-55%
FTANG	0.374	0.148	-60%
FNCR	14.81	11.12	-25%
PCOH	0.230	0.367	+60%
PCOUP	3583	3548	-1%

The presented results reveal that automated remodularization has reduced the scattering of features by 55% and the tangling of packages by 60%. Together with reduction of feature-oriented non-local reuse of packages (FNCR) by 25%, these results indicate that Featureous Automated Remodularization has significantly improved the modularization of features in JHotDraw SVG. The automated remodularization, similarly to the manual remodularization investigated in Chapter 6, appears more efficient at reducing the phenomenon of tangling than at reducing the phenomenon of scattering.

At the same time, it can be observed that the cohesion of packages was improved by 60% and the coupling between them was reduced insignificantly, only by 1% of the original value. These outcomes suggest that Featureous Automated Remodularization is significantly more efficient at increasing cohesion of packages than at reducing inter-package coupling. Interestingly, Chapter 6 reports a similar observation in the context of manual remodularization. Based on the obtained results, the overall impact of Featureous Automated Remodularization on both feature-oriented and object-oriented metrics is evaluated as significant and positive.

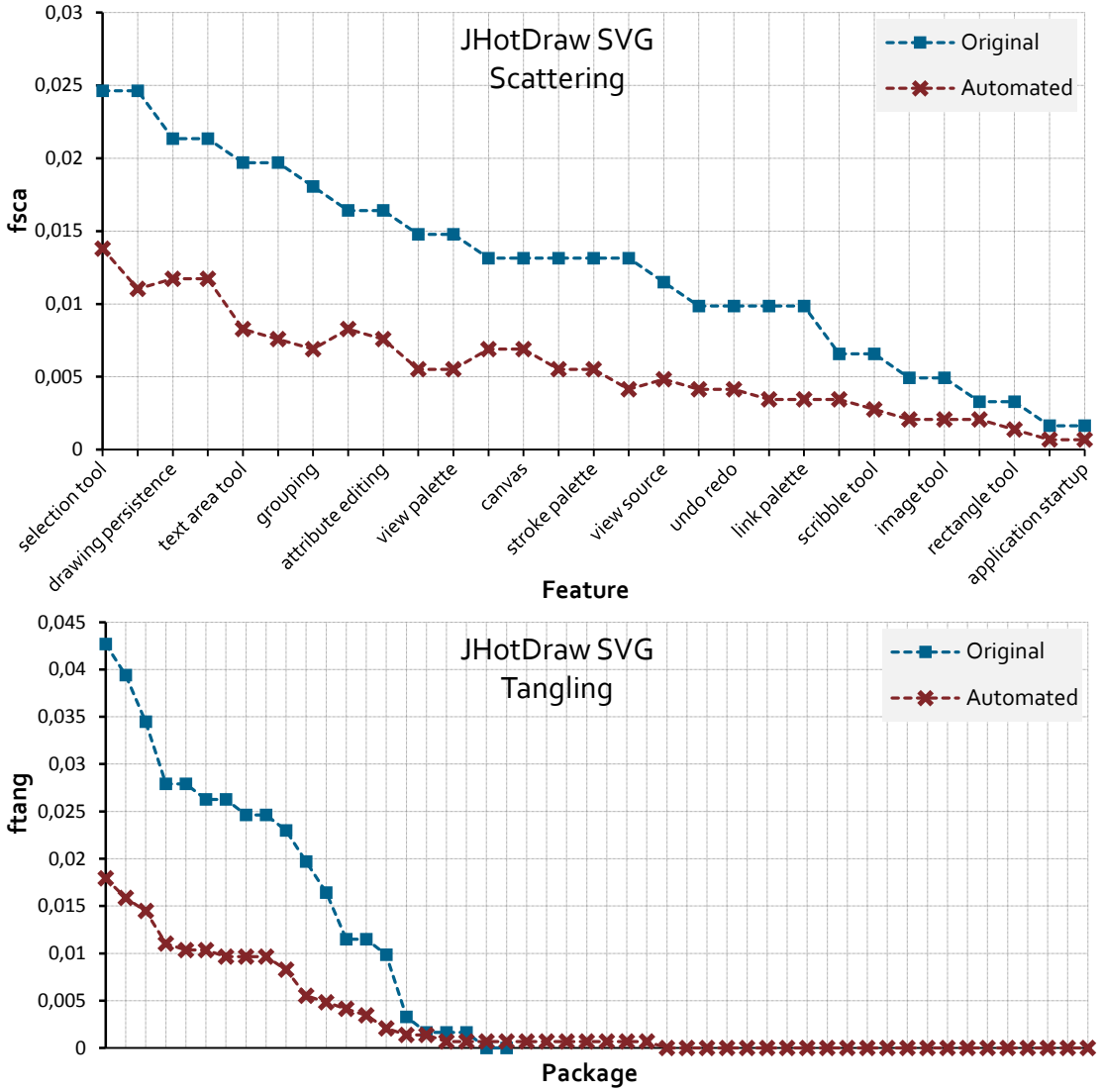


Figure 7.14: Scattering values per feature and tangling values per package in JHotDraw SVG

The detailed distributions of scattering values among features and tangling values among packages reveal consistent reductions achieved by Featureous Automated Remodularization. In the case of scattering, each feature became more localized in terms of packages, although the degrees of improvements vary to a certain extent from feature to feature. In the case of tangling, it can be observed that automated remodularization reduced the overall extent of tangling in the distribution and established a significant number of untangled single-feature packages. Furthermore, it can be seen that Featureous Automated Remodularization increased the total number of packages in the application, similarly as it was observed during the manual remodularization of NDVis presented in Chapter 6.

7.5.3 Inspecting the Obtained Modularization

A closer inspection of the remodularized application reveals a number of interesting details about the nature of the created feature-oriented decomposition. Figure 7.15 shows obtained single-feature packages of three features of JHotDraw SVG: EXPORT, BASIC EDITING and MANAGE DRAWINGS. The static relations of these packages to other multi-feature packages established by Featureous Automated Remodularization are depicted as edges.

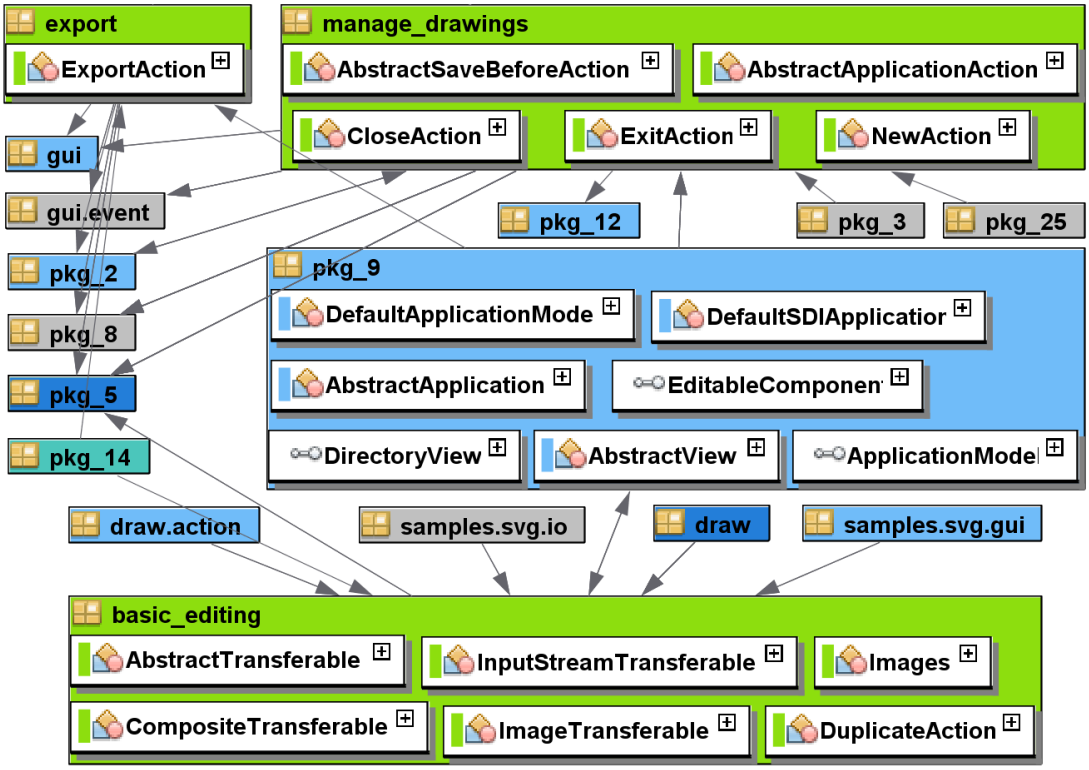


Figure 7.15: Three single-feature packages and their related multi-feature packages in remodularized JHotDraw SVG

The EXPORT feature is responsible for exporting SVG drawings to various formats. It has only one single-feature class in its package `export`. The remaining eight of its classes are distributed over multi-feature packages, since they are being shared with other features (in particular with a feature called DRAWING PERSISTENCE responsible for loading and saving the drawings to the disk). The shape of the established decomposition indicates that it would be a non-trivial task to modify one of these features in isolation from another, and in order to increase their separation from one another manual class reconceptualization would have to be performed.

The second feature shown in Figure 7.15, MANAGE DRAWINGS, is interesting because of the relations among the single-feature classes included in its single-feature package. There, it was observed that base classes defined by the JHotDraw framework and their concrete subclasses used by the SVG application were grouped. This localization of

inheritance hierarchies is expected to aid feature-oriented comprehension of SVG, as it directly links its classes with the classes of the JHotDraw framework Since frameworks often make use of the dependency-inversion principle and polymorphism, it is argued that the grouping of classes established by Featureous Automated Remodularization makes it easier to comprehend how the source code of a framework realizes the user-identifiable functionality of the concrete framework applications.

The third feature depicted in Figure 7.15 called BASIC EDITING is concerned with general editing of graphical figures on the canvas of SVG; the use cases encompassed by this feature are {*copy, cut, paste, delete, duplicate*} *figure*. By taking into account only the semantics of the feature, it is not obvious that classes such as `AbstractTransferable`, `Images` or `InputStreamTransferable` should be associated with the functionality of BASIC EDITING. This clearly demonstrates the value of automated techniques for feature-oriented remodularization, since the relevance of these conceptually unrelated classes would most likely be difficult to discover during a traditional source code inspection.

Apart from single-feature modules, Figure 7.15 shows an example of a multi-feature package created during remodularization. This package, algorithmically named as `pkg_9`, contains several utility classes shared by the features of the application. The seven classes grouped in it constitute a subset of the set of twelve classes contained in the `org.jhordaw.app` package of the original decomposition of the application.

Finally, it can be seen that there exists a diversity of static dependencies among the created packages. While dependencies among single-feature packages were observed to be seldom, it can be seen that there exist numerous examples of dependencies of multi-feature packages on single-feature packages. While such dependencies are undesirable, as was discussed in detail in Chapter 6, at this point their automated avoidance and removal is considered beyond the scope of Featureous Automated Remodularization. A partial solution to this problem could be to encode the rule about dependency direction as an additional objective for MOGGA. Ultimately, an automated code transformation for removing or inverting dependencies would be necessary. However, as discussed by Melton and Tempero [135], a practical automated strategy of doing so in a meaningful fashion is difficult to develop.

Addressing Decomposition Fragility

Because of the fragile decomposition problem and when remodularizing framework-based applications such as JHotDraw SVG, it is advisable to use reverse remodularization to recover the application's original decomposition after performing any required modification tasks. This has to be done to preserve the compliance with the original framework's API, and thus to make it possible to upgrade SVG's code to any future revisions of the JHotDraw framework. Another motivating scenario is committing the performed modifications to a shared version control repository.

Nevertheless, reverse remodularization is not always required. In particular, the established feature-oriented decomposition of a standalone application could be used as a basis for further development, or as a starting point for further manual migration of the application to a software product-line.

7.5.4 Generalizing the Presented Findings

In order to generalize the presented findings, the exact experimental procedure of the JHotDraw SVG case study was replicated with four other open-source applications. This section presents and discusses the obtained results.

Apart from BlueJ, which was already introduced in Chapter 4, the following three applications were used:

- FreeMind is an open-source application for creating graphical mind-map diagrams [136]. The release of FreeMind used in this case study is 7.1 that consists of 14 KLOC.
- RText is a text editor application providing programming-related facilities such as code coloring and macros [137]. The release of RText used in this case study is 0.9.8 that consists of 55 KLOC.
- JHotDraw Pert is a network diagramming application built on top of the JHotDraw framework and distributed together with it [94]. The release of JHotDraw Pert used in this case study is 7.2 that consists of 72 KLOC.

Recovering Traceability Links

Featureous Location was applied to all four applications in order to establish sets of traceability links needed during automated remodularization. The following numbers of features were traced for each application: BlueJ – 39, FreeMind – 24, RText – 20, JHotDraw Pert – 20. The details of performing this process for BlueJ were discussed earlier in Chapter 4. The lists of all recovered features can be found in Appendix A2.

Remodularization Results

Table 7.2 compares the values of the five metrics prior and after performing Featureous Automated Remodularization for each of the applications. The detailed distributions of scattering and tangling for individual applications are shown in Figures 7.16 – 7.19.

The summary results presented in Table 7.2 display consistent reductions of scattering, tangling and feature-oriented non-common reuse. These results remain consistent with the results presented earlier for JHotDraw SVG. The average changes are 54% for FSCA, 61% for FTANG and 35% for FNCR. Similarly, the presented data confirms the earlier-hypothesized emphasis of Featureous Automated Remodularization on reduction of tangling.

Table 7.2: Results of applying automated remodularization to four applications

Metric		BlueJ	FreeMind	RText	JHotDraw Pert
FSCA	Original	0.305	0.604	0.447	0.353
	Automated	0.164	0.264	0.182	0.166
	Δ%	-46%	-56%	-59%	-53%
FTANG	Original	0.309	0.687	0.459	0.355
	Automated	0.127	0.278	0.156	0.147
	Δ%	-59%	-60%	-66%	-59%
FNCR	Original	27.26	5.7	9.08	10.89
	Automated	17.46	4.02	5.80	6.83
	Δ%	-36%	-29%	-36%	-37%
PCOH	Original	0.346	0.514	0.314	0.203
	Automated	0.414	0.614	0.387	0.291
	Δ%	+20%	+19%	+23%	+43%
PCOUP	Original	5723	688	2615	7175
	Automated	5834	702	2590	6994
	Δ%	+2%	+2%	0%	-3%
% of single-feature classes in all covered classes		21%	28%	37%	13%

The impact of automated remodularization on the properties of cohesion and coupling reflects largely the one observed in the case of JHotDraw SVG. The average changes are 26% for PCOH and 0% for PCOUP. The lack of significant improvement of the PCOUP values suggests a limited efficiency of Featureous Automated Remodularization in reducing inter-package coupling. At this stage, the significantly higher improvement of cohesion in the case of JHotDraw SVG, than in the case of the four remaining applications appears to be caused by the relatively low initial cohesion of JHotDraw SVG.

Finally, the obtained improvements appear to be influenced by the percentages of single-feature classes found in the applications. Despite this initial observation and the intuitive appeal of such a dependency, a larger sample of applications would be needed to decisively confirm or refute such a conclusion.

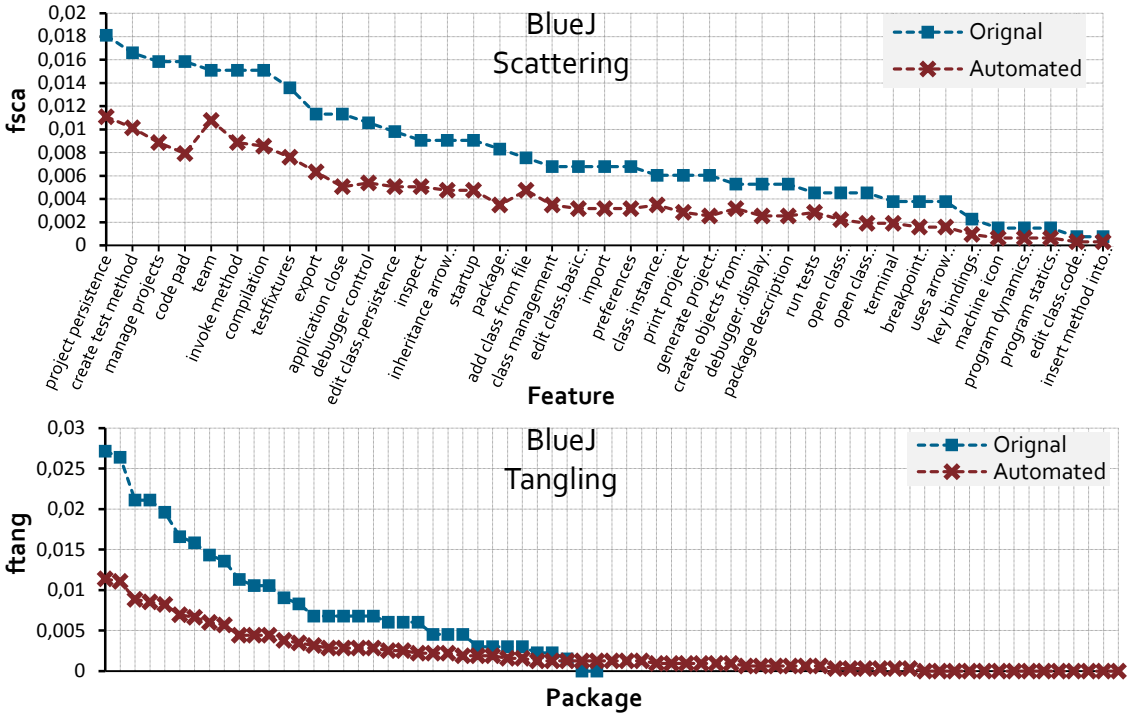


Figure 7.16: Detailed scattering and tangling values for BlueJ

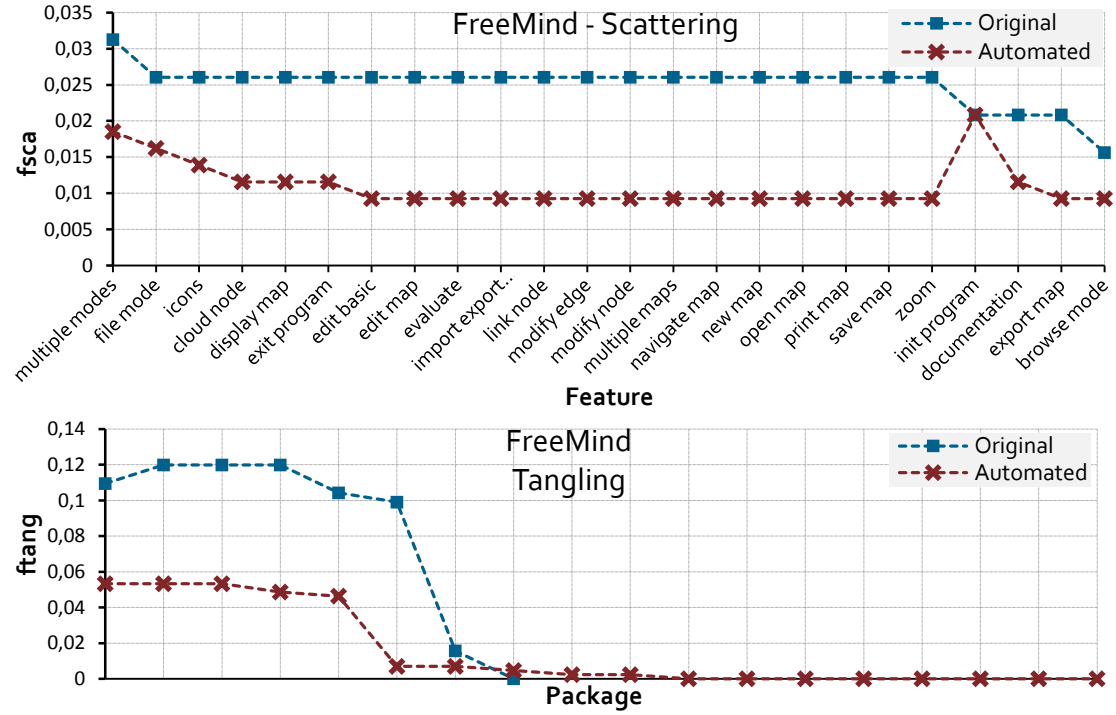


Figure 7.17: Detailed scattering and tangling values for FreeMind

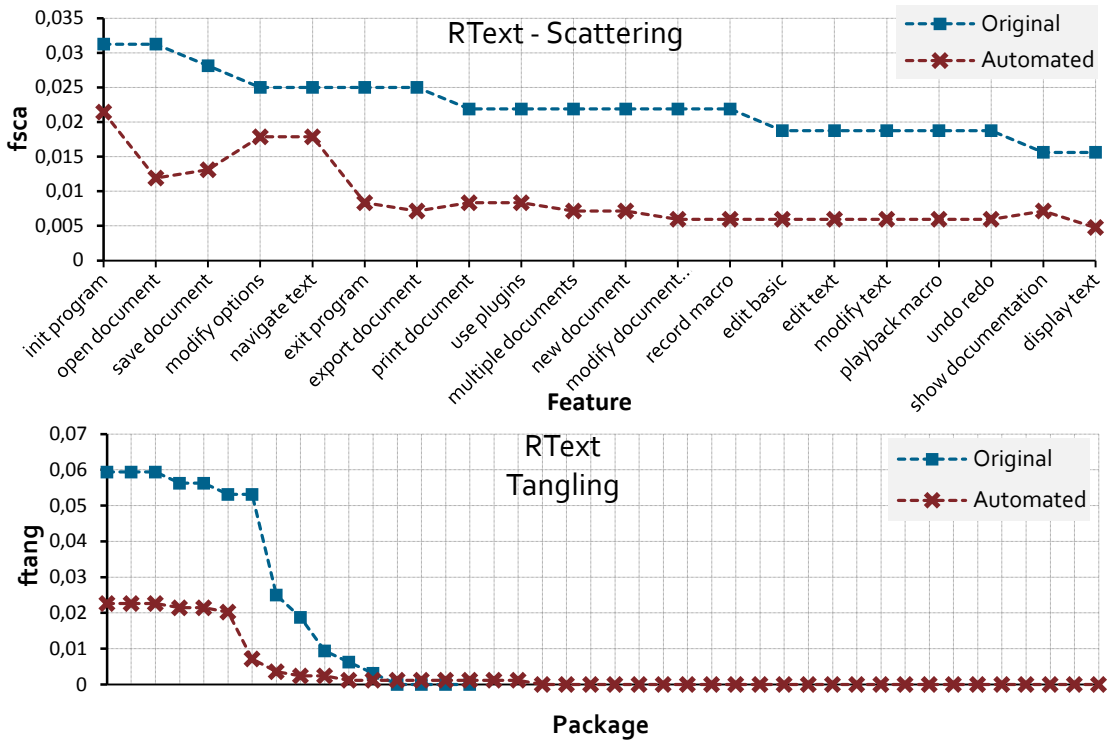


Figure 7.18: Detailed scattering and tangling values for RText

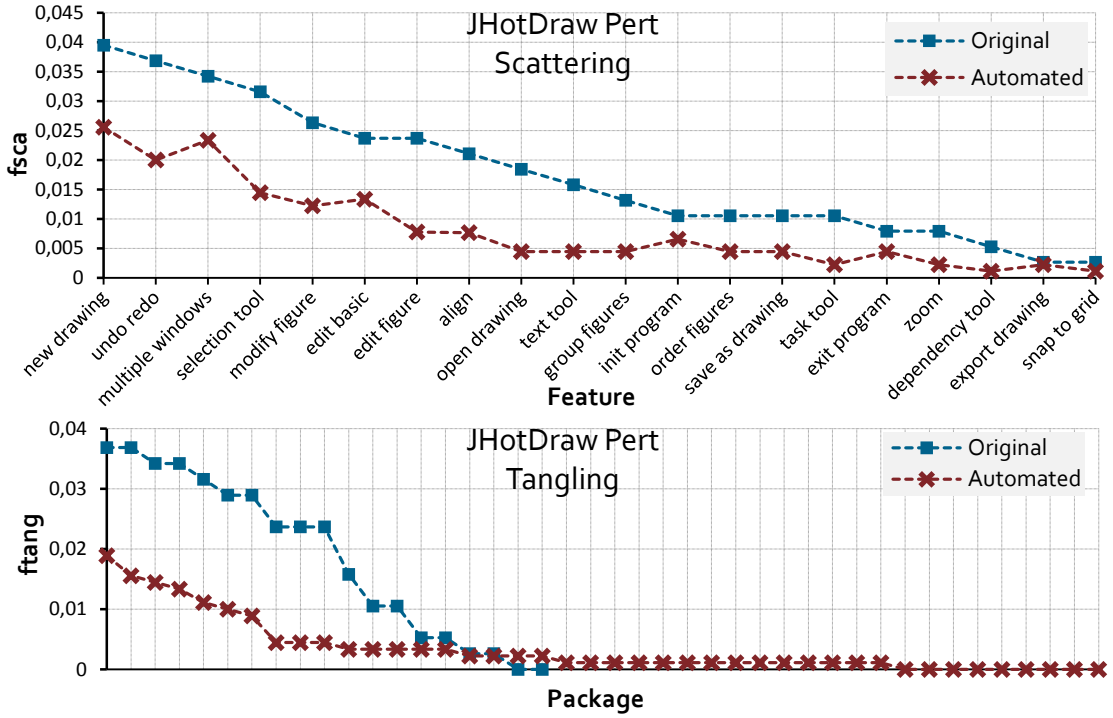


Figure 7.19: Detailed scattering and tangling values for JHotDraw Pert

Finally, the detailed distributions of scattering and tangling values for the four applications presented in Figures 7.16 – 7.19 exhibit a high degree of similarity. In all of these distributions, consistent improvements of both scattering and tangling values can be observed. Similarly, all of these distributions create new packages to improve modularization of features. Overall, the presented results closely correspond to the results presented earlier for JHotDraw SVG, and therefore strongly indicate generality of the results obtained for JHotDraw SVG.

7.5.5 Discussion and Threats to Validity

There exist several possibilities for further experimentation with Featureous Automated Remodularization.

While the optimality of the MOGGA parameters used in the presented case studies were not rigorously evaluated, preliminary observations grounded in multiple executions of the algorithm do not display symptoms of inadequacy of the currently used settings. Determining the optimal parameter values for MOGGA is considered a topic for future research. It is expected that one of the significant challenges involved would be to determine whether a single optimal set of parameters exists, or whether the optimality of parameters is specific to the characteristics of a target application.

Threats to Construct Validity

The presented studies assumed that reducing the values of FSCA and FTANG metrics corresponds to improving modularization of features and in consequence reduces the phenomena of delocalized plans and interleaving of features in packages. Thus, the validity of the study construct depends on the validity of the metrics used. The conclusions on the impact on software comprehension depend on the proposed associations between delocalization and scattering, and interleaving and tangling.

Threats to Internal Validity

The main threat to validity of the conclusions drawn from the presented results is concerned with a certain degree of arbitrariness in partitioning of application requirements into feature specifications. While the presented study followed a set of explicit criteria of doing so, as defined by the Featureous Location method presented in Chapter 4, choosing a different set of criteria is likely to produce a different set of features specifications and thus to affect the pre- and post-remodularization results.

Threats to External Validity

The sample of applications in the presented studies is considered sufficiently large to demonstrate feasibility of Featureous Automated Remodularization and generalize the results to a satisfactory extent. Nevertheless, a need for additional case studies is recognized, especially involving other types of applications and other object-oriented languages. The particular type of applications used in the studies was a user-driven GUI-intensive Java application, therefore at present no basis is provided for reasoning

about efficiency of Featureous Automated Remodularization in usage with embedded systems, source code compilers, database engines, etc.

Opportunities for Additional Experimentation

Additional characteristics of MOGGA can be investigated, such as performance and average results of genetic optimization. The initial performance observations are that the time required by MOGGA for each of the case studies was in the order of magnitude of minutes, rather than seconds or hours. It would be worthwhile to investigate possible optimizations and rigorously evaluate the performance of MOGGA, to assess whether this algorithm can be efficiently used during interactive development sessions. Secondly, the presented case studies aimed at identifying the best achievable results of applying MOGGA in order to evaluate its maximum remodularization potential. However, having the context of real-world usage in mind, it would be worthwhile to determine the average results of the algorithm, as well as their dependency on the algorithm parameters.

While not exploited in any of the presented case studies, it is possible to apply Featureous Automated Remodularization only to a subset to all features in an application. Doing so would result in improving modularization of only the selected features. This mode of operation could be applied to optimize an application's structure to evolutionary iterations with predefined scopes focusing on only subsets of all application's features. In an extreme case, Featureous Automated Remodularization can be used with only one input feature trace to automatically separate this single feature as a whole from the rest of the application's source code.

7.6 REVISITING THE CASE OF NDVIS

This section revisits the case of manual remodularization of NDVis discussed in Chapter 6. The aim of doing so is twofold. Firstly, the results obtained by the manual remodularization of NDVis are compared to the results of its automated remodularization. Secondly, the role and extent of class reconceptualization performed during manual remodularization is assessed.

7.6.1 Remodularization of NDVis: Manual vs. Automated

By applying Featureous Automated Remodularization to the original NDVis application, it is possible to compare the results of automated remodularization with the results of manual remodularization discussed in Chapter 6.

The summary results of applying manual and automated remodularization are compared in Table 7.3. Furthermore, comparison plots of scattering and tangling distributions for both remodularization methods are presented in Figure 7.20.

Table 7.3: Summary results of manual and automated remodularizations of NDVis

		Manual		Automated	
Metric	Original	Value	Δ%	Value	Δ%
FSCA	0.247	0.208	-16%	0.127	-49%
FTANG	0.200	0.146	-27%	0.073	-64%
FNCR	5.333	3.724	-30%	1.875	-65%
PCOH	0.208	0.273	+31%	0.432	+108%
PCOUP	424.0	445.0	+5%	401	+5%

By comparing the *Original* NDVis with its *Manual* and *Automated* counterparts, it can be seen that both remodularizations significantly reduced scattering and tangling of features. However, the package structure constructed by automatic remodularization achieved significantly higher improvements (49% and 64% respectively) than the package structure created by manual method (16% and 27% respectively). An analogous conclusion applies to the three remaining metrics: FNCR, PCOH and PCOUP. Overall, the relative improvements obtained by Featureous Automated Remodularization exceed the ones obtained by manually by two to three times.

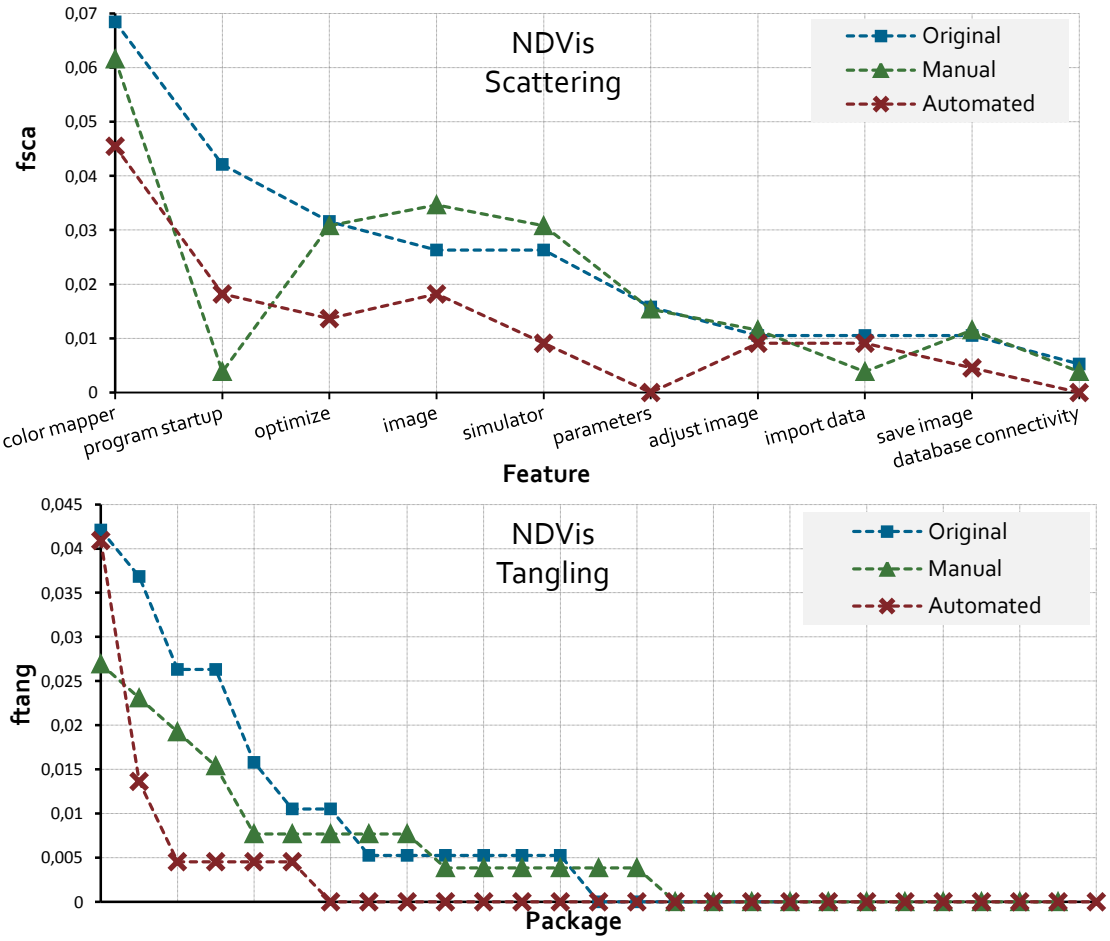


Figure 7.20: Detailed results of manual and automated remodularizations of NDVis

The detailed distributions of scattering and tangling presented in Figure 7.20 reveal interesting insights into the nature of the manual remodularization. While the automated method is shown to better reduce scattering for majority of features, it can be seen that the manual remodularization is significantly more efficient in the cases of PROGRAM STARTUP and IMPORT DATA. As it was discussed earlier in Chapter 6, manual remodularization of PROGRAM STARTUP involved extensive reconceptualization of existing classes. This is the major difference between the two compared remodularization methods, since no class reconceptualization is performed by automated remodularization. This example indicates high efficiency, but also high selectivity of manual remodularization efforts. Finally, the distribution of tangling displays a clear improvement over the manual remodularization – both in localizing multi-feature classes and in creating single-feature packages.

Summary

Overall, the presented results suggest that is worthwhile to combine Featureous Automated Remodularization with Featureous Manual Remodularization in real-world restructuring processes. In this context, automated remodularization could be used to create an initial feature-oriented modularization that could form an input to further selective manual restructurings.

7.6.2 Assessing the Role of Class Reconceptualization

Having the results of manual remodularization of NDVis that involved both relocation and reconceptualization of classes, it becomes possible to assess the role of the performed reconceptualization.

Assessing the role of class reconceptualization on reduction of scattering and tangling during real-world restructurings is necessary to consciously balance the tradeoffs of class relocation. Namely, it is necessary to uncover how much improvement of feature-oriented modularity can be obtained by dividing classes into fragments for the price of triggering the fragile decomposition problem. Furthermore, it is important to assess to which extent Featureous Automated Remodularization is affected by the design decision to avoid reconceptualization of classes.

Experimental Design

In order to assess the role of class reconceptualization it is necessary to identify two impacts of class reconceptualization depicted in Figure 7.21. Firstly, it is necessary to isolate the *impact of manual reconceptualization* on scattering and tangling from the overall impact of manual remodularization. Secondly, it is necessary to assess the hypothetical *maximum impact of reconceptualization* to explore the degree to which the reconceptualization potential was exploited during manual remodularization.

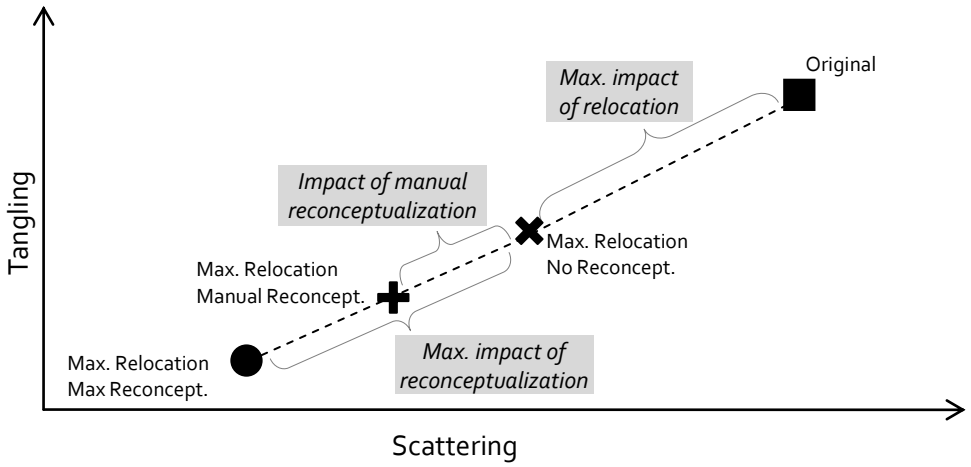


Figure 7.21: Assessing the role of class reconceptualization

Hence, the presented investigations are formulated as two experimental questions:

Q1: What is the absolute impact of the manually performed reconceptualization on the scattering and tangling of features?

Q2: What is the relative degree of the manually performed reconceptualization?

In order to address these questions, two additional modularizations of NDVis are constructed based on the three modularizations discussed earlier (i.e. *Original*, *Manual* and *Automated*). These two new modularizations, summarized in Table 7.4, are made to exhibit varying degrees of relocation and reconceptualization at the granularity of classes, to allow for isolating the individual contribution of reconceptualization.

Table 7.4: Investigated modularizations of NDVis

	Class Refactoring		Related Question	
Modularization	Relocation	Reconceptualization	Q1	Q2
Original	None	None		
Manual	*Manual*	*Manual*		
Automated	Maximum	None	✓	
Manual+Automated	Maximum	*Manual*	✓	✓
Automated split	Maximum	Maximum		✓

The M*anual+Automated* modularization is created by applying Featureous Automated Remodularization to the *Manual* modularization. Thereby, the resulting modularization exhibits maximum degree of relocation and the degree of reconceptualization of the *Manual* modularization. Hence, by comparing this modularization with the *Automated* modularization it is possible to isolate the impact of manual reconceptualization of classes on scattering and tangling.

The *Automated split* modularization is obtained by *simulating* the maximum reconceptualization of classes in the *Manual* modularization and then applying Featureous Automated Remodularization to achieve maximum relocation. The maximum reconceptualization is simulated as follows. For each multi-feature class, all its single-feature methods are identified. Then, the traceability links between these methods and their corresponding features are removed. From the point of view of the FSCA and FTANG metrics, doing so corresponds to the effects of the *move method* refactoring that would be used by a developer to relocate such a method to another class related to its corresponding feature. In the case of multi-feature classes whose methods are used disjointly by multiple features, applying this operation simulates the process of splitting the class into multiple single-feature class-fragments and relocating them to their respective single-feature packages.

Results

By applying the procedure of simulated maximum class reconceptualization to create the *Automatic split* modularization, it was possible to completely detach a feature from a class only in six cases. At this stage, this suggests a relatively low potential of NDVis for untangling features at the granularity of classes.

The two experimental questions are addressed by measuring the obtained modularizations of NDVis using the FSCA scattering metric and the FTANG tangling metric. The results of these measurements, being a basis for addressing the two experimental questions, are listed in Figure 7.22.

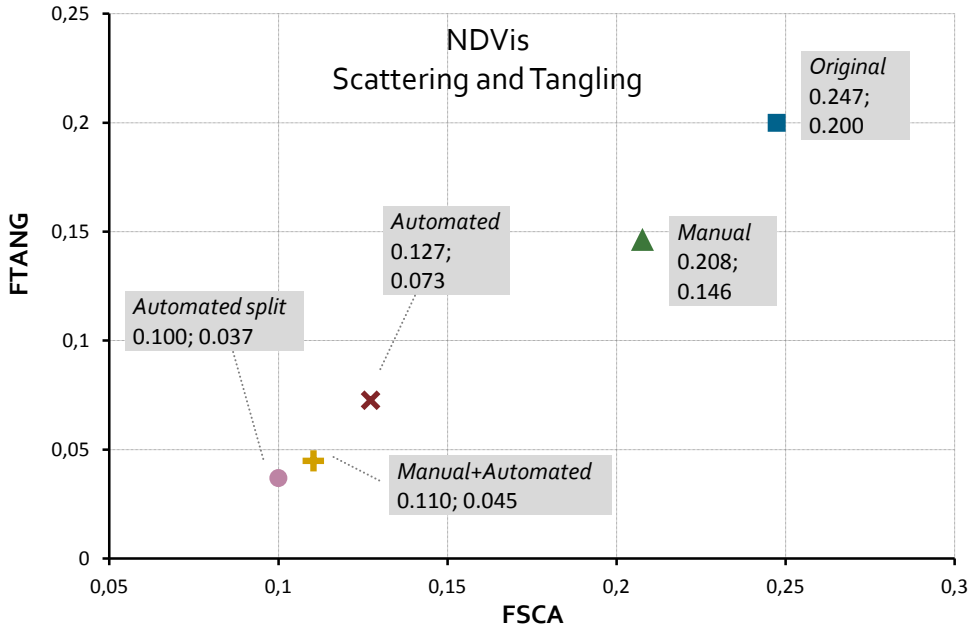


Figure 7.22: Impact of investigated modularizations on feature tangling and scattering

Q1: What is the absolute impact of the manually performed reconceptualization on the scattering and tangling of features?

To assess the impact of the manually performed reconceptualization on scattering and tangling of features, the *Automated* modularization based on optimized class relocation needs to be compared with the *Manual+Automatic* modularization based on manual reconceptualization and optimized relocation. As can be seen from the obtained results, the additional reductions of scattering and tangling introduced by manual reconceptualization (i.e. the distance from *Automated* to *Manual+Automated*) surpass the results of pure relocation (i.e. the distance from *Original* to *Automated*) by only 7% and 14% respectively.

This result suggests that the reconceptualization performed during the manual restructuring had only a minor effect on the overall reduction of scattering and tangling of features.

Q2: What is the relative degree of the manually performed reconceptualization?

To establish a scale against which the degree of reconceptualization performed in the *Manual+Automatic* modularization can be assessed, two modularizations are used. These are: the *Automatic* modularization exhibiting no reconceptualization and the *Automated split* modularization exhibiting maximum reconceptualization. By doing so, it is possible to observe that the manual reconceptualization of the *Manual+Automatic* modularization exploits most of the achievable potential for reconceptualization at the granularity of classes.

This indicates that the manual remodularization of NDVis achieved a high degree of reconceptualization of classes. Furthermore, it can be seen that the maximum achievable improvement offered by reconceptualization remains significantly smaller than the one offered by automated relocation. A closer inspection revealed that this was caused by a relatively high tangling of methods in NDVis, which limited the effects of reconceptualization at the granularity of classes and created the need for further reconceptualization at the granularity of methods.

Jointly, the two reported sets of results depict only a minor influence of the actual and the idealized class reconceptualizations. Hence, the investigated case emphasizes the importance of class relocation over class reconceptualization.

7.6.3 Discussion and Threats to Validity

The presented case study relied on several assumptions that could be refined in the future replications to potentially grant additional insights into the presented findings.

Firstly, one could focus on differentiating between multiple kinds of class-level reconceptualization, such as move method, pull up method, push down method and

move field refactorings [65]. While the presented study treated them uniformly, it is expected that different refactorings have different impact on scattering and tangling of features. Similarly, the impact of reconceptualization at lower granularities (i.e. methods and instructions) would be an interesting topic for further investigation.

Construct Validity

Secondly, the fact that the author performed the manual remodularization of NDVis has to be seen as a threat to internal validity of the presented results. However, this is argued to have a negligible impact on the results, since the manual remodularization was originally performed as a part of separate work, months before the idea of using this data for investigating the role of relocation and reconceptualization was born.

External Validity

Finally, replicating the study on a larger population of applications is necessary to generalize the reported findings. Hence, at this stage, the presented insights should be treated as initial indicators. It is expected that subsequent replications of the presented experimental procedure will lead to reinforcing the presented conclusions.

7.7 RELATED WORK

Featureous Automated Remodularization follows the idea of *on-demand remodularization* proposed by Ossher and Tarr [44]. The conceptual framework behind Featureous is deeply influenced by their observations that a software system can be decomposed only according to one decomposition dimension at a time, thus making the other possible dimensions under-represented. Moreover, they recognize that throughout the lifetime of a software project it is not possible to foresee all future concerns that will have to be modularized. To address this, Ossher and Tarr propose that it should be possible to remodularize applications on-demand, accordingly to the needs that arise throughout software's evolution process.

Murphy et al. [67] explored tradeoffs between three policies of splitting tangled features: a lightweight class-based mechanism, AspectJ and Hyper/J. By manually separating a set of independent features at different levels of granularity, they confirm the limited potential of the lightweight approach for reducing tangling. In the case of AspectJ and Hyper/J, they have discovered that usage of these mechanisms makes certain code fragments difficult to understand in isolation from one another. Furthermore, aspect-oriented techniques were found to be sensitive to the order of composition and to result in coupling of features to each other.

In contrast, Featureous Automated Remodularization is not affected by constraints of feature composition order, since it does not reconceptualize legacy classes into fragments. For the same reason, Featureous Automated Remodularization avoids the

problems of non-trivial semantics and a tight coupling between feature-fragments observed by Murphy et al.

Räihä et al. [138] proposed an approach to automated generation of software architecture using a genetic algorithm. The approach takes use cases as inputs, which, after being refined to sequence diagrams, form a basis for deriving an initial functional decomposition of an application. This decomposition is then evolved using an objective function being a weighted sum of a number of object-oriented metrics. Some of these metrics are used to promote creation of certain architectural styles and patterns. In contrast to the approach of Räihä et al., Featureous Automated Remodularization uses feature-oriented remodularization criteria and a non-weighted multi-objective formulation of the optimization problem.

Bodhuin et al. [139] presented an approach to optimization-based re-packaging of JAR files of Java applications for minimizing their download times. The approach uses execution traces of an application to identify features that can be lazily loaded by the Java Web Start technology to save network bandwidth. The approach relies on a single-objective genetic algorithm to cluster classes that are used together by different sets of features. An empirical study of automatically repackaging three medium-sized Java applications conducted by Bodhuin et al. displays a significant reduction of initial download times of all applications.

Janzen and De Volder [84] proposed the notion of *effective views* to reduce the problem of crosscutting concerns by providing editable textual views on two alternative decompositions of a single application. Fluid alternation and synchronization of the views allows developers to choose the decomposition that better supports a task at hand. Even though a feature-based view is not provided in their prototype tool Decal, it seems that the tool could be extended to support it. In Decal, it is possible to view both the alternative decompositions simultaneously, whereas in Featureous Automated Remodularization a code transformation has to be performed for each transition. It appears that the simultaneous co-existence and cross mapping of multiple views could serve as a way of overcoming the fragile decomposition problem. However, the approach presented in [84] is not readily applicable to legacy applications, and it appears that providing such a mode of operation would be difficult to achieve.

7.8 SUMMARY

The significant complexity, effort and risk involved in the process of manual feature-oriented remodularization makes it difficult to practically apply it to large applications. In order to scale feature-oriented remodularization, automation is required.

This chapter presented Featureous Automated Remodularization – a method for automated search-driven remodularization of Java applications.

The method formulated feature-oriented remodularization as a multi-objective problem of optimizing class relocation among packages. This problem was further specified by motivating the use of five metrics as optimization objectives. The method proposed a multi-objective grouping genetic algorithm as the means of automatically solving the five-objective optimization problem. To allow for automated enforcement of calculated modularizations, a set of source code transformations was developed. Moreover, a reverse source code transformation was developed to enable on-demand recovery of the original modularizations of source code. The method was demonstrated to significantly and consistently improve modularization of features in a case study involving several medium and large open-source Java applications. Finally, revisiting the results of the manual remodularization of NDVis demonstrated significantly higher efficiency of the automated method, and revealed an overall minor effect of class reconceptualization on scattering and tangling of NDVis.

8. TOWARDS LARGE-SCALE MEASUREMENT OF FEATURES

This chapter proposes a method to automating feature-oriented measurement of source code. This is done by proposing the concept of seed methods and developing a heuristic for their automated detection. Seed methods are then used as starting points for automated static call-graph slicing. Using 14 medium and large open source Java projects, it is demonstrated that this method achieves a comparable level of code coverage to a manual approach. Moreover, this method is applied to tracking the evolution of scattering and tangling in long-term release history of Checkstyle.

This chapter is based on the following publications:
[TOOLS'12]

8.1 OVERVIEW

The evolutionary significance of features makes it important to incorporate feature-oriented measurement into the practices of software quality assessment. However,

automated large-scale computation of feature-oriented metrics remains difficult to achieve.

Feasibility of large-scale feature-oriented measurement is constrained by incomplete automation of feature location approaches. This is, however, not caused by deficiencies of the existing mechanisms, but by the very fundamental assumptions of feature location. Traditionally, feature location aims at associating *semantically-concrete feature specifications* with their corresponding units of source code. This makes it necessary to consider the semantics of feature specifications, prior to associating them with appropriate test cases, feature-entry points or elements of an application's UI.

However, for large-scale quality assessment procedures that aim at measuring tens or hundreds of software systems, the information about individual features is not essential. Instead, summarizing systems-level indicators are required. Therefore, if it would be possible to calculate such a system-level results directly, without deriving it from individual per-feature results, then interpreting semantics of individual features could be avoided. As a result, automation of feature-oriented system-level measurement could be acquired.

This chapter defines a method to large-scale quantification of feature-oriented modularity based on automatic detection of so-called *seed methods* in source code. This method is schematically depicted in Figure 8.1.

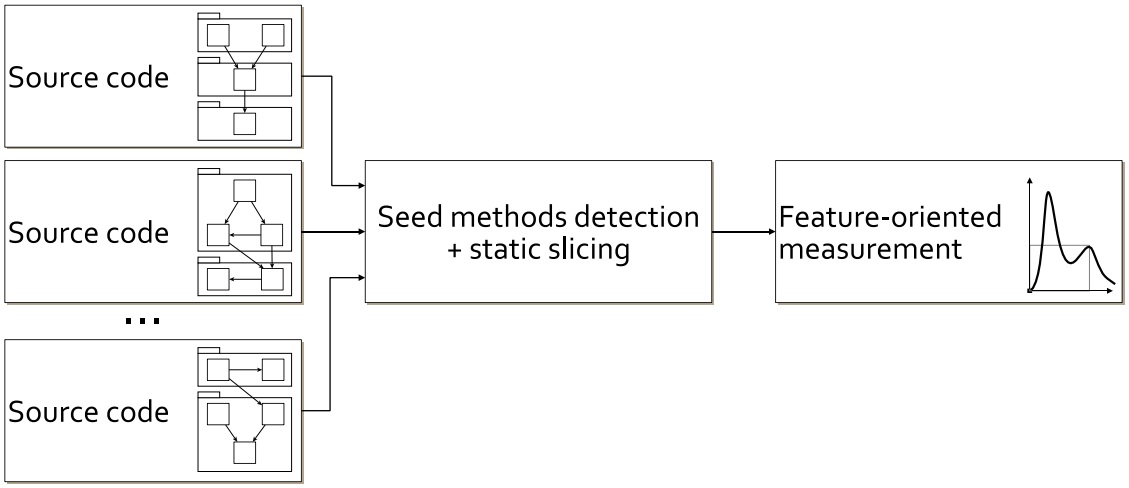


Figure 8.1: Overview of automated detection of seed methods

Seed methods constitute the "anonymous counterpart" of feature-entry points – while they can mark a method as a starting points of *a feature*, they do not associate with any particular feature specification. It will be demonstrated that this property allows for automated detection of seed methods and for using them as a basis for automated slicing of static call graphs extracted from source code. Reliance on seed methods and static slicing removes the need for the two manual steps associated with feature-entry

points: manual annotation of source code and user-driven activation of features at runtime.

The proposed method for automated detection of seed methods is evaluated using a group of 14 diverse open-source systems. This is done by comparing the slices produced by the proposed method with manually constructed ground truth, with respect to their coverage of source code. Furthermore, the method is applied to automatic measurement of 27 revisions of an open-source project released over a period of 10 years.

8.2 THE METHOD

Seed methods are the "starting points" of features in source code – they are the methods through which the control flow enters the implementations of features in software's source code. The notion of seed methods is a generalization of the notion of feature-entry points defined in Chapter 4. While each feature-entry point is a starting point of *a concrete feature* of an application, all that is known about a seed method is that it is a starting point of *a feature* of an application, without determining which concrete feature it is. Hence, seed methods are oblivious to concrete semantics of features that they represent.

To automatically detect seed methods, a heuristic is proposed that allows for filtering them from all methods presents in an application's source code. The proposed filtering method is explained using the example in Figure 8.2, which represents a call-graph of the methods in a small Java system. Please note that while the provided example uses Java, the proposed method is not limited to only this language.

An initial heuristic for filtering the methods of the example system could be to keep only those methods, which are not called by other methods. Such methods could be assumed as called by UI interfacing libraries through a callback interaction. Within Figure 8.2, this would lead to identifying the two `actionPerformed`-methods as the seed methods.

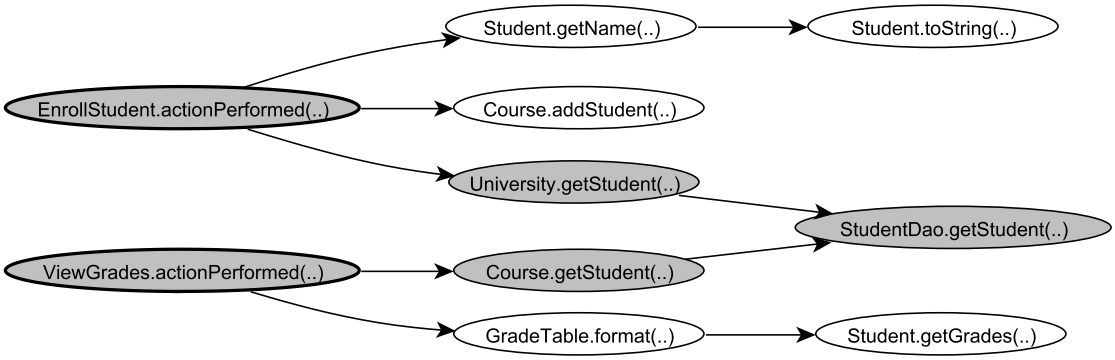


Figure 8.2: Call graph of an example program

Unfortunately, there are cases in which this simplistic tactic does not perform well. For programs defined with a command-line interface the only method that is called only from outside is the static `main` method that starts the program. Identifying this single method would not be appropriate, as a common pattern of command-line applications is to use the main method as a dispatcher that selectively dispatches control to other features based on the supplied command-line parameters. Generally, such an issue can occurs not only in command-line applications, but also in all applications that contain an internal event dispatching mechanism that calls starting methods of features based on the types of received events.

A second heuristic for filtering is counting the names of all methods within a system and identify names which occur proportionally more often than other names. Note that in this situation the short name of methods (i.e. `toString` or `getStudent`) instead of their full name (i.e. `Student.toString` or `Course.getStudent`) should be counted since the latter name uniquely identifies a method within a system, and thus the number of methods with this name is always one.

The assumption behind this heuristic is that the regularity in method names is caused by usage of polymorphism and conventions that are enforced by interfacing libraries. In consequence, methods with the same name but a different implementation implement are expected to have a high likelihood of implementing user-triggerable functionalities. Given the example in Figure 8.2, the method `actionPerformed` is implemented multiple times since this method is enforced by a generic interface provided by the Swing GUI framework to handle actions taken by the user. Similarly, the `getStudent` method is implemented, because of either polymorphism or a convention, by both `Course` and `University` classes.

Unfortunately, a straight-forward application of this heuristic is problematic because this heuristic also identifies those methods which offer generic functionality for objects, such as the `toString` method, as well as getters and setters for common properties such as names and id's. This last category of methods should not be considered as seed methods, since getters and setters typically do not implement complete features.

To filter out such uninteresting methods the number of methods needed to implement a specific method is taken into account. This is done by counting the number of distinct methods called by the specific method, and then recursively counting the distinct methods used by those methods. The assumption is that a higher number of methods used in the implementation of a method correspond to a method that implements more sophisticated functionality.

Therefore, the hypothesis becomes that by only keeping those methods, which are (1) implemented proportionally more often and (2) which use many other methods in their implementation, it is expected to discover the seed methods of a system.

8.2.1 Heuristic Formalization

For the formalization of the heuristic a software system S is modeled as a directed graph $D = (V,E)$. The set of vertexes V are methods defined in the software product, and the set of edges E are calls modeled as a pair (x, y) from one method x (the source) to another method y (the destination). Let FN and SN be the sets of full names and short names, a vertex $v \in V$ is a record containing a full name and a short name, i.e. $v=(fn, sn)$ where $fn \in FN$ and $sn \in SN$.

For the first part of the heuristic the sets of vertexes that have the same short-name need to be defined. Using the function $shortname((fn,sn)) = sn$, which retrieves the short name component (sn) from a given vertex $v \in V$. The set of vertexes V_{sn} is the set of vertexes $v \in V$ that have sn as short name, defined as $V_{sn} = \{v \mid shortname(v) = sn\}$.

For the second part of the heuristic, it is needed to compute the vertexes that are transitively connected to a given vertex. For this purpose, two functions are defined. First a function $connected$: $V \times V \mapsto 2$ which distinguishes the vertexes that are directly connected by a given edge $e \in E$. For two vertexes $v_1,v_2 \in V$, $connected$ will yield $True$ if $\exists_{e \in E}$ such that $e = (v_1,v_2)$ and $False$ in all other cases. Secondly, a function $connected^+$: $V \times V \mapsto 2$ is defined as the transitive closure of function $connected$. Given these functions, the set V_v consisting of vertexes that are transitively connected to vertex $v \in V$ can be defined as $V_v=\{v \mid connected^+(v)\}$.

Given this formalization, the heuristic can be defined in three functions. First, a function to calculate the normalized frequency of methods with a certain short name:

Definition 1. $freq(sn) = \frac{|V_{sn}|}{|V|}$

Second, a function to calculate the average number of methods needed to implement the methods with a given short name:

Definition 2. $depth(sn) = \sum_{v \in V_{sn}} \frac{|V_v|}{|V|}$

Note that the results of both of these functions fall into the range [0,1], which ensures that values calculated from different systems can be compared if desired.

Lastly, to calculate the score for each short name the values of the two functions need to be combined. Ideally, the aggregation function prevents compensation, i.e. high value on one function should not overly compensate a low value on the other function. Given this property, two simple aggregation functions can be chosen: the minimum and the product. In the heuristic the product is used to ensure a higher level of discriminative power, the total score for a given short name thus becomes:

Definition 3. $score(sn) = freq(sn) \times depth(sn)$

Applying the heuristic to the example in Figure 8.2 provides the scores in Table 8.1; note that the scores are normalized against the total number of methods defined within the system. The `actionPerformed` methods receive the highest score because these methods occur twice in the system and the average number of methods needed to implement them is 4.5. The methods called `getStudent` are second in rank, because they occurring three times in the system and rely on average on 0.66 methods.

Table 8.1: Normalized scores for the methods as shown in Figure 8.2

ShortName	freq	depth	score
actionPerformed	0.20	0.45	0.09
getStudent	0.30	0.06	0.02
getName	0.10	0.10	0.01
format	0.10	0.10	0.01
addStudent	0.10	0.00	0.00
toString	0.10	0.00	0.00
getGrades	0.10	0.00	0.00

8.2.2 Automated Quantification of Feature Modularity

Calculation of feature modularity metrics requires two steps; identification of seed methods and identification of those parts of the system that are executed when a seed method is executed.

For the first step the score function, as defined above, can be used to identify the δ most interesting methods. For practical reasons this work treats the δ=10 best methods as seed methods. Nevertheless, the optimality of this value and the potential context-dependency of the δ parameter needs to be investigated further and is considered as future research.

The second step required for measuring feature modularity is to identify which parts of the application are executed when a seed method is executed. This is done by statically slicing the call-graph of the system under review. Using the terminology defined above, the method $connected^{+}$ is executed for a seed method to obtain a set of methods in return. This set, which is referred to as a *static trace*, represents a group of functionally related code units.

8.3 EVALUATION

To validate the heuristic for identifying seed methods the following steps are taken. First, a set of subject programs is chosen (Section 8.3.1). For each of the programs, the author manually identified a ground-truth set of seed methods enforced by the

respective interfacing technologies and libraries being used (Section 8.3.2). This data is then used to compute static traces, whose aggregated source code coverage allows to reason about the completeness of the constructed ground-truth. Then, the aggregated coverage of ground-truth slices is compared against aggregated coverage of traces generated by the heuristic (Section 8.3.3). Based on this, it is possible to evaluate whether the proposed method covers similar amounts and similar regions of source code as the manually established ground truth.

Please note, that the design of this validation experiment deviates from the traditional designs of evaluating the accuracy of feature location methods. There, false positives and false negatives are usually computed by comparing results of a method to ground truth on *per-feature* basis. Such a method is valid for assessing the accuracy of locating features associated with particular semantics, but unfortunately it is inapplicable in this case, since the proposed method aims at system-level application and identifies groups of functionally related code units without attaching semantics.

8.3.1 Subject Systems

The evaluation experiment is performed on a set of 14 open-source Java programs summarized in Table 8.2. The chosen population is intentionally diversified in order to observe how the method deals with discovering seed methods in not only stand-alone applications but also libraries, frameworks, web applications and application containers. Thereby, the aim is to validate the ability of the method to detect seed methods that are triggered not only by GUI events, but also by command-line parameters, calls to API methods, HTTP requests, etc.

Table 8.2: Subject systems used in the study

Program	Version	KLOC	Type
ArgoUML	0.32.2	40	Application
Checkstyle	5.3	60	Library
GanttProject	2.0.10	50	Application
Gephi	0.8	120	Application
JHotDraw	7.6	80	Framework
k9mail	3.9	40	Mobile application
Mylyn	3.5.1	185	Application
NetBeans RCP	7.0	400	Framework
OpenMeetings	1.6.2	400	Web application
Roller	5.0	60	Web application
Log4J	1.2.16	20	Library
Spring	2.5.6	100	Framework/Container
Hibernate	3.3.2	105	Library
Glassfish	2.1	1110	Container

8.3.2 Ground Truth

The ground truth in the experiment is formed by manually identifying seed methods in the subject programs. In order to make the classification of methods objective and consistent across all experimental subjects, the following procedure is used that is based on the observation that libraries and frameworks, which are used for interfacing with an environment, tend to enforce a reactive mode of implementing functionality and standardize the names of methods for doing so.

For instance, the Swing Java GUI framework defines a set of methods, such as `actionPerformed`, `onClick`, etc., that are meant to be implemented by a client application and are called by Swing upon the reception of a given event from a user. Such methods, exhibiting individual features in response to external events, are used as ground-truth seed methods in the presented experiment. Such chosen methods also appear as appropriate candidates for execution by *software reconnaissance's* test cases [50], annotating as feature-entry-points or marking starting points for static analysis [140].

Table 8.3: Correlation of subjects with technologies and their ground-truth seed methods

Technology	Seed method	ArgoUML	CheckStyle	GanttProject	Gephi	JHotDraw	K9mail	Mylyn	NetBeans RCP	OpenMeetings	Roller	Log4J	Spring	Hibernate	Glassfish
JDK	run, call, main	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Swing	actionPerformed, stateChanged, keyTyped, keyPressed, mouseClicked, mousePressed	✓	✓	✓	✓	✓			✓						
Eclipse/SWT	handleEvent, keyPressed, mouseDown, mouseDoubleClick, widgetSelected, widgetDefaultSelected, runWithEvent, run, start, execute							✓							
Servlet	doGet, doPost										✓				
Android	onCreate, onOptionItemSelected, onClick, onLongClick, onKey, onKeyDown, onTouch, onStartCommand, startService						✓								
Spring	handle, handleRequest, onSubmit, start, initApplicationContext									✓			✓		
Struts	execute, invoke, intercept										✓				
Log4J	getLogger, log											✓			
Hibernate	buildSessionFactory, openSession, update, save, delete, createQuery, load, beginTransaction, commit, rollback													✓	
Glassfish	start, execute, load														✓

Based on the mentioned observation, manually identified are the interfacing technologies used by the subject programs. This was done based on static dependencies found in source code. For each of the discovered technologies, indentified were the methods that are intended to be implemented/overridden by client programs in order to provide a client's functionality. This was done by

surveying the available official documentation of the investigated libraries. The summary results of this process are listed in Table 8.3.

8.3.3 Results

For each of the subject programs, the seed methods of its interfacing technologies served as a starting point for static call-graph slices. Their *aggregated coverage*, being the union of these slices, was used as the ground-truth. The aggregated coverage percentages of both the ground truth and the proposed heuristic are shown in Table 8.4. In the "Ground truth" column, the percentage of code covered by the static-slices originating from the ground truth is shown. The "Method" column shows the percentage of code covered by the static-slices originating from the methods found by the proposed heuristic. In the "Intersection" column, the percentage of code covered by intersection of both result-sets is shown.

Table 8.4: Percentage of LOC covered for both methods

Program	Ground truth	Intersection	Method
ArgoUML	81.5 %	79.3 %	82.2 %
Checkstyle	51.8 %	48.6 %	73.6 %
GanttProject	93.9 %	93.6 %	96.1 %
Gephi	90.7 %	89.2 %	92.0 %
JHotDraw	87.3 %	85.9 %	88.9 %
k9mail	97.1 %	97.0 %	97.0 %
Mylyn	80.7 %	78.1 %	81.9 %
NetBeans RCP	81.5 %	79.9 %	89.0 %
OpenMeetings	75.9 %	73.6 %	79.5 %
Roller	80.7 %	79.8 %	83.6 %
Log4J	90.1 %	86.3 %	88.6 %
Spring	69.3 %	66.5 %	76.9 %
Hibernate	84.1 %	82.1 %	84.9 %
Glassfish	71.4 %	70.5 %	78.8 %

It can be observed that for most systems the ground-truth coverages remain over 75% of the LOC, which suggests a high degree of completeness of the established ground truth. The only exceptions here are Checkstyle, Spring and Glassfish that are covered in 51.8%, 69.3% and 71.4% respectively. The result of Checkstyle seems to suggest incompleteness of the used ground truth. However, a closer look reveals that there exist four other systems that achieved over 75% coverage, based on the same seed methods as Checkstyle. As will be discussed later, this particular result of Checkstyle had a different cause.

Comparison of columns one and three indicates that aggregated coverage generated by the proposed method surpasses that of the ground truth for all the systems but k9mail. While the differences for most of the systems appear negligible (below 5% LOC), there are four notable exceptions, namely Checkstyle with difference of 21.8%, Spring with 7.6%, NetBeans RCP with 7.5% and Glassfish with 7.4%. Interestingly, three of these systems are also the ones that exhibit the lowest ground-truth coverage.

A closer investigation of the reasons for the difference of 21.8% for Checkstyle revealed that the results generated by the method contained a number of methods that can be categorized as non-technology-originated seed methods. For instance, the methods `process`, `processFiltered`, `verify`, `traverse` and `createChecker`, were found to yield slices containing the highest numbers of classes. These methods constitute important domain-specific abstractions that were established by Checkstyle's developers for implementing functionality, instead of relying on the general-purpose abstractions provided by the JDK or by Swing. Similarly, a similar pattern in other subjects was observed, i.e. `afterPropertiesSet`, `invoke`, `postProcessBeforeInitialization` and `find` in Spring, or `execute`, `addNotify` and `propertyChange` in the NetBeans RCP.

Comparison of the columns one and two shows that the proposed heuristic manages to cover most of the regions of source code covered by the manually extracted ground truth, with the average loss of only 2.5% LOC. While this result expected for the highest sets of coverages (e.g. for the intersection of two result-sets achieving 95% coverage, the maximum possible loss is 5%), it is especially important in the context of the lowest-scoring ground-truth values, i.e. Checkstyle (for which the maximum possible loss is 26.4%) and Spring (for which the maximum possible loss is 23.1%). This indicates that the proposed method not only covers as much source code as the manually-established ground truth, but that it also identifies largely the same regions of source code, thus providing analogous input to measuring feature modularity.

Lastly, the aggregated coverage obtained by the proposed method does not appear to be influenced by size or type of systems. Nevertheless, a sample larger than the one used in the experiment would be needed to confirming the lack of such causalities at a satisfactory level of statistical significance.

8.4 EVOLUTIONARY APPLICATION

In this section, the proposed seed detection method is applied to evaluating the quality of features modularity in an evolving program. This is done by automatically measuring long-term evolutionary trends of modularity metrics in the release history of Checkstyle [141], a library for detecting violations of coding style in Java source code. The units of functionality in Checkstyle library, and whose historical quality will be assessed, are the *detectors* for various types of style violations, as well as the core

infrastructure of the library responsible of parsing source code, reporting results, etc. This study measures 27 major releases of the library since version 1.0 until version 5.4.

8.4.1 Measuring Feature Modularity

The existing literature proposes and demonstrates the usefulness of a number of diverse metrics for measuring feature modularity, e.g. [60], [27], [58]. The presented study relies on the two most fundamental of known metrics: scattering and tangling.

Because the actual number of features cannot be determined based on seed methods, the formulations of scattering and tangling used in this study are boiled down to simply counting the number of related classes or features. Thus, they will be measured as follows:

- Scattering will be measured for each seed method as the total number of classes that appear in its static trace.
- Tangling will be measured for each class as the number of seed methods in whose static traces a given class appears.

The metrics chosen to quantify feature modularity are calculated for each static trace (scattering) and each class (tangling). They are then aggregated to the system level and normalized with respect to size of the system or the number of static traces. The rationale therefore is to obtain a rating that can be used as an objective criterion for comparing multiple applications, or tracking evolution of a single application over time.

8.4.2 Aggregation of Metrics

To aggregate measured values of scattering and tangling, the benchmarking-based aggregation strategy of Alves et al. [142] is used. This strategy relies on a repository of systems to derive thresholds for categorizing unit of measurement in system (i.e. class or the trace) into one of four categories. By summing up the size of all entities in the four categories a system-level profile is calculated, which in turn is used to derive a system-level rating [143].

The resulting rating, normally on a scale of one to five, indicates how the profile of a specific system compares to the profiles of the systems within the *benchmark* used for calibrating the profile rating. For example, a rating of 1 indicates that almost all systems in the benchmark have a better profile, while a rating of 4 means that most systems in the benchmark have a lower profile.

The repository used to calibrate the rating for both scattering and tangling consists of industry software systems previously analyzed by the Software Improvement Group (SIG), an independent advisory firm that employs a standardized process for

evaluating software systems of their clients [144]. These industry systems were supplemented by open-source software systems previously analyzed by SIG's research department.

The repository consists of 55 Java systems, of which 11 systems are open-source. These systems differ greatly in application domain (banking, logistics, development tools, applications) and cover a wide range of sizes, starting from 2 KLOC up until almost 950 KLOC (median 63 KLOC).

8.4.3 Results

Figure 8.3 shows a plot of the measured evolutionary trends of Checkstyle. The figure shows the values of KLOC metrics and the *ranking values* of scattering and tangling for each release. Please note that because of benchmarking, the quality rankings have to be interpreted inversely to the metrics they originate from – e.g. a high quality rank of scattering means low scattering of features.

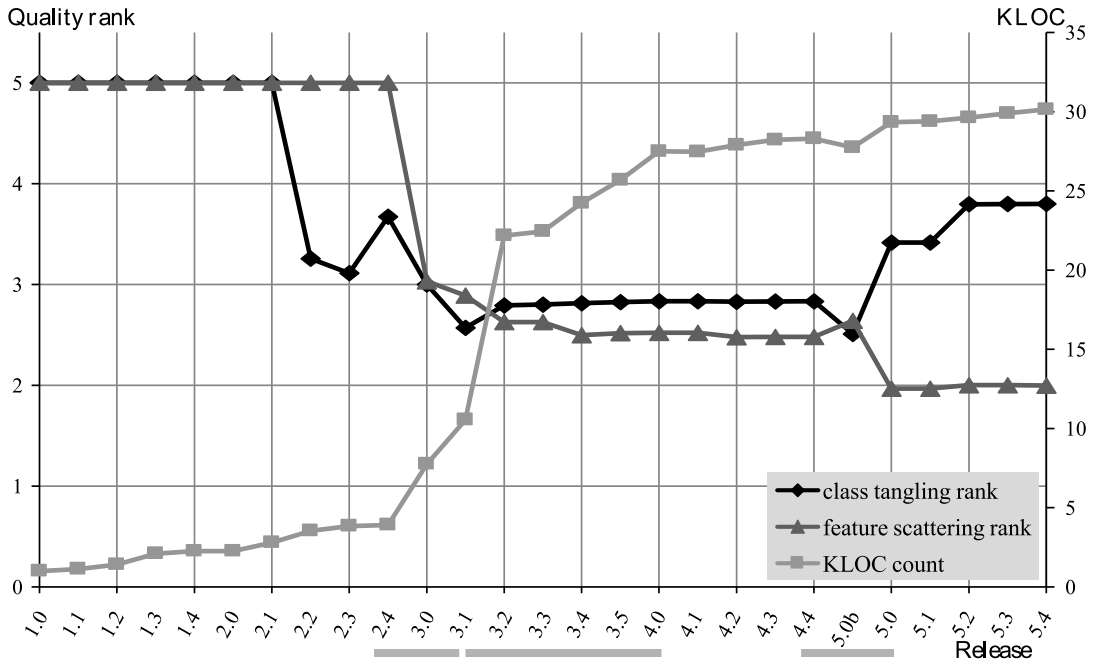


Figure 8.3: Evolution of Checkstyle

The evolutionary trends plotted in Figure 8.3 indicate that feature-oriented quality, represented by ranks of scattering and tangling tends to degrade over time. In the following, three periods marked in Figure 8.3 are investigated, as they exhibit particularly interesting changes of the measured ranks.

Versions 2.4 – 3.1: a significant degradation of both scattering and tangling quality ranks is observed. The observed degradation was initiated by changes done in release *3.0*, where one of the major changes was a restructuring to a "*completely new*

architecture based around pluggable module". This restructuring involved factoring-out a common infrastructure from existing detectors. Doing so was bound to increase the number of classes that features are scattered over, and create a number of infrastructural classes to be reused by multiple features, thus contributing to tangling.

Further degradation continued in release *3.1*. According to Checkstyle's change log, the crosscutting introduction of severity levels to all detectors forces all of the detectors to depend on an additional class. This significantly contributes to the increase of tangling and scattering of features because before this introduction most of the detectors were confined to a single class.

Versions 3.1 – 4.0: a rapid extension of Checkstyle's size is observed. In contrast with the previous period, the feature-oriented quality of the program remains stable. Version *3.2* is the version in which the program's size doubled, while the tangling rank slightly improved and the scattering rank declined. Based on the change log, this is caused by the addition of multiple fairly well separated J2EE-rule detectors. As discussed later, this hypothesis is supported by observed reverse changes in tangling and scattering ranks in version *5.0b*, where these detectors are removed.

One of the most interesting characteristics of the *3.1 – 4.0* period is the observed preservation of feature-oriented quality despite a nearly twofold growth of the program's size. This suggests that the new architecture established in *3.0* and adjusted in *3.1* proved appropriate for modularizing the forthcoming feature-oriented extensions. The established underlying infrastructure appears to provide all the services needed by features and the new features are made confined to approximately the same number of classes as the already-existing features.

Versions 4.4 – 5.0: An interesting shift in feature-oriented quality is observed in this period. Firstly, a slight improvement of the scattering rank and a degradation of the tangling rank are observed in the release *5.0b*. Together with the decrease of program's size, these changes suggest a removal of a number of separated features. The program's change log supports this hypothesis, as it reports removal of all the J2EE-rule detectors. It needs to be noted that the observed magnitude of degradation of the tangling rank and improvement of scattering rank is approximately equal to their respective changes in the release *3.2*, where the J2EE-rule detectors were originally added.

Secondly, a significant improvement of the tangling rank and a significant degradation of the scattering rank are observed in release *5.0*. According to the change log, the most likely reason is the "*Major change to* `FileSetCheck` *architecture to move the functionality of open/reporting of files into Checker*", which "*reduces the logic required in each implementation of* `FileSetCheck`". In other words, by freeing individual detectors from explicitly calling "*open/reporting*", the developers managed to reduce the tangling among them. At the same time, the ten newly introduced complex detectors caused a visible degradation of the scattering rank.

8.5 DISCUSSION

The results presented in Section 8.3 show that seed methods automatically identified by the proposed method yield static slices that capture largely the same regions of source code as a manually established ground truth. Moreover, the heuristic improves on the ground-truth coverage results by identifying non-technology-originated seed methods that reflect important domain-specific functional abstractions. Given these observations, it can be concluded that the seed methods computed by the method are adequate substitutes to manually identified seed methods for the purpose of system-level quantification of feature modularity.

The application of the method presented in Section 8.4 shows that the automated measurement of the evolution of scattering and tangling properties provides a valuable perspective on the evolution of an existing software system. It was demonstrated how to interpret these metrics in the context of Checkstyle's change log by generating informed hypotheses about the impact of the individual changes on the feature-oriented quality of the program. While the generated hypotheses need additional validation, they provide a sound and firm starting point for evaluating the evolutionary quality of feature modularity.

Algorithm Parameters

As explained in Section 8.2.2, the parameter δ is used to limit the number of best-ranked methods to be chosen as seed methods. Theoretically, such a value should preserve all the methods that contribute significantly to aggregated program coverage, whereas all the remaining methods should be filtered out. Even though the chosen δ seems to be adequate for the presented case studies (i.e. adding more methods to the list of seed methods did not increase the program coverage substantially), more work is needed to determine the optimal value of the δ parameter. Additionally, it is important to investigate whether a single optimal value of δ can be found for a portfolio of programs, or whether each program needs an individually chosen δ value.

Limitations

One of the limitations of the performed experiments is the lack of a direct comparison against outputs of existing feature location methods. Ideally, a correlation study of system-level scattering and tangling metrics contrasting the proposed method with the existing ones could be conducted. However, such a study requires a significant number of data points, being software systems, to achieve a satisfactory level of statistical confidence. While in the case of the proposed method this data can be generated automatically, no sufficiently large data sets exist for existing feature location methods.

In the presented evolutionary investigation, the differences among the sets of identified seed methods for subsequent versions of Checkstyle could have influenced the results. It was observed that this behavior occurs when new types of detectors

using new seed methods were added. While such a flux of the sets of seed methods reflects the evolution of how features change over time, it may turn out problematic with respect to comparability of measurements across versions. As a means of addressing this threat to validity, the metric aggregation discussed earlier was used. Additionally, it was confirmed that even though the set of seed methods changed over time the coverage remained between 68% and 75%.

Lastly, because only open-source Java systems where used in the evaluations, the results cannot be generalized to systems with different characteristics (i.e. using a different programming paradigm). However, since the heuristic is largely technology agnostic, it remains possible to validate the method using a more diverse set of systems.

8.6 RELATED WORK

The problem of feature location can be seen as an instance of the more general problem of concern location. In this context, Marin et al. [145] have proposed a semi-automatic method to identify crosscutting concerns in existing source code, based on analysis of call relations between methods. This is done by identifying the methods with the highest fan-in values, filtering them and using the results as candidate seed methods of concerns. These candidate seeds are then manually inspected to confirm their usefulness and associate them with the semantics of a particular concern they implement.

Similarly, the several approaches to feature location associate features with control flows in source code. One of the first such works the software reconnaissance approach of Wilde and Scully [50]. Their approach is a dynamic feature location technique that uses runtime tracing of test execution. Wilde et al. require that feature specifications are investigated in order to define a set of feature-exhibiting and non-exhibiting execution scenarios. Individual execution scenarios are implemented as a suite of dedicated test cases that, when executed on an instrumented program, produce a set of traceability links between features and source code.

Salah and Mancoridis [51] proposed a different approach to encoding the feature-triggering scenarios. Their approach, called marked traces, requires one to manually exercise features through a program's user interface. Prior to executing a feature-triggering scenario, a dedicated execution-tracing agent is to be manually enabled and supplied with a name of the feature being exercised. Effectively, this approach removes the need for identifying starting-point methods in source code and the need for the up-front effort of implementing appropriate feature-triggering test cases. However, this is achieved at the price of manual scenario execution.

Feasibility of using designated methods as guiding points for static, as opposed to dynamic, analysis was demonstrated by Walkinshaw et al. [140]. They developed a feature location technique based on slicing of static call-graphs according to user-supplied landmarks and barriers. There, landmarks are manually identified as the "methods that contribute to the feature that is under consideration and will be invoked in each execution", whereas barriers are the methods know to be irrelevant to a feature. These two types of methods serve as starting points and constraints for a static slicing algorithm. This static mode of operation improves the overall level of automation by removing the need for designing and executing feature-exhibiting scenarios. Unfortunately, the lists of suitable landmarks and barriers have to be established manually.

8.7 SUMMARY

Cost-efficiency of applying feature-oriented measurement is determined by the extent of its automation. As a result of incomplete automation of feature-oriented measurement, it remains difficult to assess quality of features, control it over time and to validate new feature-oriented metrics.

This chapter proposed a method for automated measurement of system-level feature modularity, based automated discovery of seed method for slicing of static call-graphs of applications.

The method introduced the concept of seed methods. A heuristic was proposed to automatically detect seed methods, based on popularity of method names and size of the static call-graph slices they yield. The heuristic was evaluated on 14 software systems by comparing the coverage of static slices produced by the proposed method against a manually constructed ground truth. Finally, practical applicability of the proposed method was demonstrated in the scenario of measuring the evolution of feature-oriented modularity in the release history of Checkstyle.

9. DISCUSSION

This chapter discusses the applicability, limitations, open questions and perspectives on the Featureous approach.

9.1 LIMITATIONS AND OPEN QUESTIONS

There exist several limitations and open questions associated with the presented research. This section discusses the most important of them.

Used Metrics

The presented research rests upon the feature-oriented concepts of scattering and tangling. Why the definitions of these concepts are generally consistent across the literature, there exist multiple proposals for how to quantify them. Therefore, the particular set of metrics used throughout this thesis, which was defined by Brcina and Riebisch [58], is only one possible frame of reference. However, surveying existing literature on the topic, as done in Chapter 2, suggests a high degree of adherence of the existing metrics to the common conceptual definitions of scattering and tangling. Namely, all the existing scattering metrics are based on counting the numbers of code units that a feature is delocalized over, whereas tangling metrics are based on counting the number of features that reuse a single code unit.

Thus, while using a different formulation of scattering and tangling metrics would lead to different numerical results; it is not expected to significantly affect the presented conclusions.

Other Optimization Objectives

The optimization objectives used by Featureous Automated Remodularization can be flexibly replaced by other software metrics. In particular, it is possible to add metrics that represent other design principles, such as elimination of cyclic dependencies among packages or avoidance of dependencies of multi-feature packages on single-feature packages. Another possibility would be to include custom architectural constraints (e.g. GUI classes that depend on Swing library should not be grouped together with persistence classes that depend on JPA) by encoding them as function returning numerical fitness values. Any metrics used for such purposes should be formulated according to the MAFF guidelines of Harman and Clark [128].

Applicability to Other Programming Languages

At present, Java is the only programming language that the Featureous Workbench tool is compatible with.

On the technical side, adapting Featureous Workbench to a different programming language would require re-implementing three mechanisms. They are: the execution tracer defined by Featureous Location, the infrastructure for static analysis of source code and the code transformations used by Featureous Automated Remodularization.

On the conceptual side, the Featureous approach remains fairly general, and could easily be adapted to other object-oriented languages, such as Smalltalk, C++ or C#. As for application to non-object-oriented languages, the conceptual foundation of Featureous would have to undergo major changes, in an extent dependent on the properties of a target language.

Replacing the Feature Location Mechanism

Featureous makes it simple to replace the Featureous Location method. The only requirement imposed on a candidate replacement method is being able to correctly instantiate the feature-trace model defined in Chapter 4. This model is the only interface between Featureous Location and the other parts of Featureous.

Additionally, it is preferred that a replacement method emphasizes avoidance of false positives, even for the cost of an increased number of false negatives. Thereby, it is expected to deliver a minimal correct set of traceability links that other parts of Featureous, such as Featureous Automated Remodularization, can rely on.

Lastly, the non-dynamic candidate replacements can safely ignore providing the identifiers of the objects used by features. Doing so will not affect feature-oriented views other than the *feature relations characterization* view.

Porting to Other IDEs

Featureous Workbench plugin is integrated with the NetBeans IDE mostly at the level of user interface. The data models used by Featureous Workbench remain separate from the ones offered by the NetBeans infrastructure. In particular, the Workbench

uses its own representation of source code as a basis for calculating metrics. This representation is instantiated using the IDE-independent Recoder library [133]. Similarly, most of the analytical views would need only minor, if any at all, modifications to execute within another IDE.

While overall Featureous Workbench retains a relatively high degree of portability, there exist individual aspect that would need a major rework. These are re-implementing the code editor coloring, reintegrating with project management and with project execution infrastructures of a new IDE.

9.2 FEATUREOUS WITHIN SOFTWARE LIFE CYCLE

Practical applicability of Featureous depends not only on how well it can be applied, but also on whether the results of its application form an appropriate basis for further evolution of applications. Therefore, it is necessary to discuss the methodological perspective on the post-remodularization handling of software systems. In particular, it has to be discussed how the feature-oriented arrangement of source code can be propagated to software life-cycle concerns such as iteration planning, effort estimation, architectural design, work division, testing and configuration management.

Currently, there exists only one software development methodology that is related to feature-orientation. It is called Feature-Driven Development (FDD) [146]. Furthermore, there exists a number of compatible practices advocated by other methodologies, such as Extreme Programming (XP) [147] and Crystal Clear [148]. The remainder of this section presents a high-level perspective on how individual phases of software life cycle can use these practices to embrace the notion of feature-orientation.

Requirements Engineering and Domain Analysis

As it was discussed in Chapter 4, the popular approach to requirements analysis based on *use cases* [89] is largely compatible with the feature-oriented point of view. This is because a feature can be mapped to one or more functional requirements represented by use cases. Similarly, a feature can be mapped to one or more *user stories* [147], which are the means of specifying functional requirements in XP. FDD proposes limiting granularity of feature specifications to ensure that each feature specification can be implemented in under two weeks [146].

Furthermore, FDD encourages an performing an explicit domain analysis and motivates it the domain models as being common bases for features to build upon. This proposal finds its reflection in Featureous. As argued earlier in Chapter 6, Featureous postulates preserving original domain models and using them as important building blocks for features.

Planning, Scoping and Effort Estimation

FDD proposes to plan and scope development iterations on per feature basis. Hence, each iteration has the goal of implementing a set of feature specifications. Thereby, the iterations result in delivering new usable units of applications functionality to the end-users. This is argued to facilitate gathering early feedback about the functionalities being developed.

The per-feature iteration scoping resembles the practice of *walking skeleton* proposed by Crystal Clear [148]. A walking skeleton is a proof-of-concept implementation of an application that performs a small end-to-end user-function and that connects the main architectural components. Hence, walking skeleton is essentially a functional slice of a desired application.

Feature-oriented effort estimation can be performed using a range of existing effort estimation techniques. One of them is the *function points* method [149]. As reported by Capers Jones [150], the Java language requires 53 LOC per function point, which in turn corresponds to productivity of between 10 and 20 function points per person-month of work. Using Featureous Location it remains possible to estimate the LOC-sizes of existing features, and in consequence their accumulated number of function points. Such a measurement can be practically exploited during feature-oriented effort estimation by seeking analogies between already existing features and planned features.

A lightweight effort estimation technique is proposed by XP in the form of *planning poker* [147]. In this practice, the time required to implement a user story is firstly estimated by each developer in secrecy, and then revealed to become an input to discussing a consensus estimate. As user stories map well to features, this technique appears applicable to feature-oriented effort estimation.

Finally, a per-feature tactic can also be applied to project accounting, as discussed by Cockburn [151]. Namely, a development contract can specify a price per delivered feature, in addition to a (lowered) per hour pricing. Cockburn argues that such a model better aligns the financial incentives of developers and clients. This model, however, is difficult to realize in practice if appropriate facilities for feature location and feature-oriented analysis are not available.

Architecture and High-Level Design

FDD proposed the notion of *iterative design by feature* [146]. In this proposal, developers initiate each iteration by designing the only the features found in the iteration's scope. This is done by analyzing how the planned features use the existing domain models and developing detailed sequence diagrams depicting this usage.

This proposal remains consistent with Featureous. In addition, Featureous emphasizes the need for recognizing and addressing: (1) accidental introduction of tangling on single-feature classes, and (2) insufficient reuse of classes contained in shared multi-

feature modules. Occurrence of these phenomena indicates a need for relocation of classes to reflect their updated degree of reuse among features.

Implementation

As a consequence of per feature planning and per feature design, implementation efforts have to also proceed on a per feature basis. Hence, as proposed by FDD, developers need to form *feature teams* that concentrate on end-to-end development of individual features. The practice of forming feature teams is reported by Cusumano and Selby [152] to be successfully applied in several major projects at Microsoft. In addition, Cusumano and Selby describe a method of dividing the work on features into three development milestones, as follows: (1) first 1/3 of features (the most critical features and shared components), (2) second 1/3 of features and (3) final 1/3 of features (least critical features).

In addition to these considerations, the experiences from applying Featureous make it a relevant question whether a part of a development team should be permanently assigned to the multi-feature core modules of an application. Because such modules should enclose code domain models and reusable assets, a particular attention should be paid to preventing their erosion and to maintaining their exposed APIs.

Testing

When evolving an application modularized according to feature-oriented criteria, several testing practices become necessary. Firstly, having explicit multi-feature core modules that expose APIs to single-feature modules, it is important that the APIs are well tested to prevent introduction of accidental errors. Secondly, test suites have to be made feature-oriented, i.e. they should treat features as the units of testing effort. This can be achieved by developing integration tests that explicitly map to features of an application. Thirdly, combinations of features should be considered as test candidates, in order to detect undesired effects of *feature interactions* [51].

Additionally, automated integration tests can be used to automatically regenerate feature traces, whenever any change to the source code is committed to a version control system. This can be done by making the integration tests run with the Featureous Location execution tracer and by ensuring that they execute feature-entry points of the application. By doing so, feature traces can continuously be kept synchronized with the evolving source code.

Configuration Management

Finally, feature-oriented modularization of source code solves some of the issues of configuration management. Fowler [6] discussed a technique called *feature branching* that proposes that developers create per feature branches in their version control systems and work on separate features in parallel. While this approach helps feature-teams to work in isolation, Fowler points out that it artificially postpones and thereby complicates the integration of changes. The integrations become complex due to

textual merge conflicts over commonly modified files, which is especially problematic if refactorings is involved.

Featureous resolves these problems by removing the fundamental motivation of feature branching – having features modularized in source code, multi-team parallel development can be performed on a single branch of version control. The only modules where conflicting changes may occur are the multi-feature core modules, which by definition should be more stable than individual single-feature modules.

10. CONCLUSIONS

This chapter concludes this thesis by summarizing the claimed contributions and lessons learned during the research on Featureous. Furthermore, consequences of Featureous for the practice of software engineering and opportunities for further research are discussed.

10.1 SUMMARY OF CONTRIBUTIONS

The ability to change is both a blessing and a burden of software systems. On the one hand, it allows them to adapt their features to changing requirements of their users. On the other hand, changing existing features is oftentimes an overwhelmingly challenging task.

This thesis focused on the core challenge of feature-oriented changes – the lack of proper modularization of features in source code of Java applications. In this context, the thesis developed practical means for supporting comprehension and modification of features. These means were feature location, feature-oriented analysis and feature-oriented remodularization. This thesis developed and integrated these methods in terms of a layered conceptual model. A schematic summary of how this layered conceptual model was realized is presented in Figure 10.1.

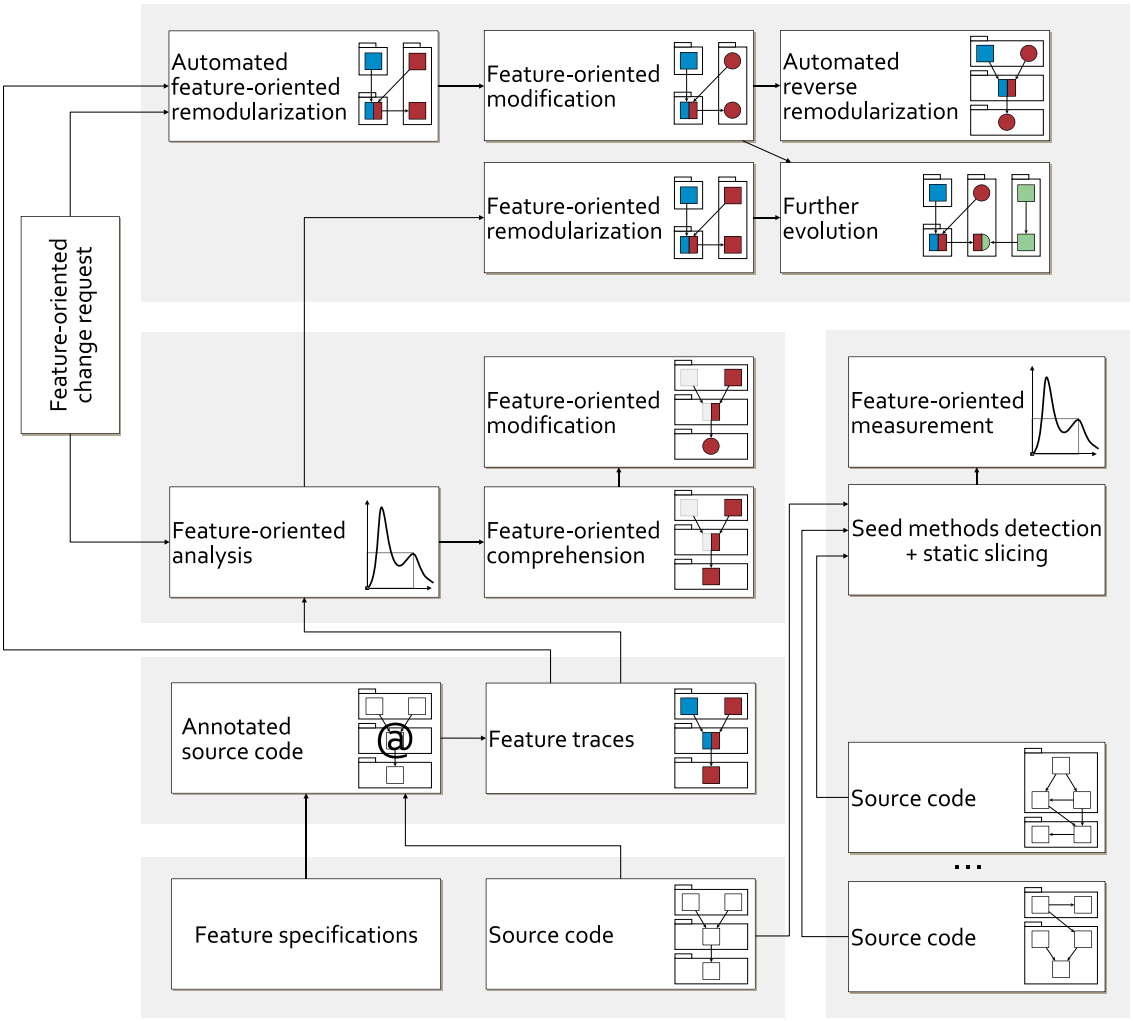


Figure 10.1: Realization of the conceptual model of Featureous

The realization of the individual layers of the conceptual model of Featureous resulted in the following contributions:

- *Feature location*: This thesis proposed the Featureous Location method for establishing traceability between features and unfamiliar source code. The method relies on the notion of feature-entry points that have to be annotated in source code to guide dynamic analysis. Featureous Location was implemented using AspectJ and was successfully applied to several open-source Java systems. Initial evaluations of effort-efficiency, coverage and accuracy of the method were presented. Finally, Featureous Location was used as a basis for developing the concept of runtime feature awareness.

- *Feature-oriented analysis and measurement*: This thesis proposed the Featureous Analysis method for feature-oriented analysis of existing Java applications. Featureous Analysis proposes a three-dimensional conceptual framework for representing and relating feature-oriented analytical views. Based on recognizing the needs of the change mini-cycle, the method identifies the most important view configurations and realizes them using several feature-oriented

visualizations. Featureous Analysis was evaluated analytically, as well as through a series of application studies concerned with assessing feature-oriented modularity, supporting comprehension and guiding change adoption.

Furthermore, this thesis proposed a method for large-scale quantification of feature modularization quality. The method is based on static slicing techniques and on automated heuristic-driven discovery of seed methods, being a generalization of the notion of feature-entry points. The method was evaluated by comparison to a manually identified ground truth. Subsequently, applicability of the method to quantifying evolutionary changes in feature modularization quality was demonstrated.

- *Feature-oriented remodularization*: This thesis proposed the Featureous Manual Remodularization method for restructuring existing Java applications to improve modularization of their features. The method is based on iterative relocation and reconceptualization of classes and is driven by feature-oriented analysis. Featureous Manual Remodularization identifies and addresses the class-level fragile decomposition problem by defining a feature-oriented modular structure that consists of both single-feature and multi-feature modules. The feasibility of the method was demonstrated in a study of feature-oriented remodularization of an open-source neurological application. Apart from qualitative and quantitative results, several interesting experiences were reported.

  Finally, this thesis presented the Featureous Automated Remodularization method aimed at automating the process of feature-oriented restructuring. Featureous Automated Remodularization formulates the task of remodularization as a multi-objective grouping optimization problem. The method realizes this formulation using five metrics to guide a generic algorithm in optimization of package structures of existing Java applications. Apart from the automated forward remodularization, the method supports also an on-demand reverse remodularization. Featureous Automated Remodularization was evaluated using two case studies, and was used as means of assessing the role of class reconceptualization during feature-oriented remodularization.

To facilitate practicality and reproducibility of the obtained results, the aforementioned contributions were implemented as the Featureous Workbench plugin to the NetBeans IDE. This implementation is freely available, both as binary and as source code.

All the mentioned contributions, jointly known as the Featureous approach, constitute the results of addressing the research question that was defined at the beginning of this thesis: *How can features of Java applications be located, analyzed and modularized to support comprehension and modification during software evolution?*

10.2 CONSEQUENCES FOR PRACTICE OF SOFTWARE ENGINEERING

The presented work results in two important consequences for the current practice of software engineering.

Firstly, Featureous delivers practical means of performing feature-oriented analysis of existing Java applications. As demonstrated, Featureous Location and Featureous Analysis can be introduced to existing unfamiliar and large Java codebases in a matter of hours. Apart from the low effort investment, these methods do not introduce new technological risks, as their only impact on existing source code is the insertion of feature-entry point annotations.

Secondly, the Featureous Manual Remodularization and Featureous Automated Remodularization methods demonstrate that it is possible to significantly improve modularity of features in existing Java applications. While the manual method demonstrates feasibility of feature-oriented remodularization during migration to module systems, the automated method improves scalability of such restructurings. Thereby, Featureous makes enables existing Java applications to migrate to feature-oriented modularization criteria.

10.3 OPPORTUNITIES FOR FUTURE RESEARCH

The presented work opens several opportunities for future research. These opportunities include additional validation of Featureous, adapting the approach to the scenario of continuous application and applying it to conduct basic research on the modularity of features in evolving software systems.

10.3.1 Further Empirical Evaluations

While the presented evaluation studies are satisfactory at the current stage, additional experimentation would be needed to grant additional insights into the precise characteristics of the approach. Particularly desirable would be large-scale controlled experiments involving professional developers and industrial applications. Based on such studies, it would be possible to conclude about the effort-efficiency and accuracy of Featureous Location, and about the support for comprehension of Featureous Analysis in the context of industrial software engineering practices.

Moreover, it is deemed relevant to gather more experiences from performing feature-oriented remodularizations. Such experiences, preferably reported by third parties, would allow not only for precise evaluation of the role of Featureous in the process,

but also for a deeper understanding of the effects of such restructurings on long-term evolution of object-oriented applications.

Finally, it is important to evaluate the remodularization methods proposed by Featureous using other software metrics and quality models, to reinforce the results reported in this thesis.

10.3.2 Continuous Analysis and Remodularization

While the presented work applied Featureous to single revisions of applications, continuous application of Featureous remains an interesting topic for further research.

One particularly interesting direction of such investigations would be to explore the concept of *continuous remodularization*. In such a scenario, automated remodularization of a codebase could be performed after every change to source code is committed. Doing so has potential for reducing the erosion of software architectures over time, as the desired architectural design could be automatically re-enforced after each change. Realizing this vision is expected to require an additional optimization goal – a goal that promotes similarity of the evolved package structures to an externally specified architectural rationale.

More generally, additional experiences with applying Featureous to support evolution of applications are needed. Challenges of doing so involve: (1) evaluating Featureous Location in the scenario of automated feature location driven by integration tests and (2) developing an efficient and transparent method for continuous updating of feature traces during development sessions, to ensure constant synchronization between feature traces and the source code under modification.

Finally, it remains relevant to seek further integration of Featureous with existing development tools and IDEs to develop mechanisms such as feature-oriented code navigation, feature-oriented code completion or feature-oriented interfaces. A promising direction for experimenting with such mechanisms would be to implement a feature-trace importer plugin to the Mylyn task-based environment [153].

10.3.3 Basic Research on Feature-Orientation

Finally yet importantly, the Featureous Workbench tool can be used by researchers as a vehicle for conducting basic research on the topic of feature-orientation. Its free availability and its plugin-based architecture are hoped to encourage such a use. Promising research topics include: investigating the applicability of feature-oriented analysis during the initial development phase of software life-cycle (as opposed to the evolution phase), investigating of erosion of feature-oriented modularity over subsequent software releases, or investigating the relation of feature-oriented metrics to other concern-oriented and general-purpose software metrics.

10.4 CLOSING REMARKS

This thesis developed Featureous, an integrated approach to location, analysis and modularization of features in Java applications.

The presented work advances the state of the art in supporting feature-oriented comprehension and modification of existing object-oriented applications. This was achieved by proposing a set of novel conceptual and technical means for performing feature location, feature-oriented analysis and feature-oriented remodularization.

Support for comprehension of features was provided at several levels. Firstly, Featureous Location supports efficient location of features in the source code of existing applications. Secondly, Featureous Analysis provides several views that guide feature-oriented exploration of source code and that focus it on concrete granularities, perspectives and levels of visual abstraction. To automate and scale feature-oriented assessment of applications, a heuristic for detection of seed methods was proposed. Finally, Featureous Manual and Automated Remodularization made it possible to physically reduce scattering and tangling of features in packages, and thereby to reduce their associated comprehension-hindering phenomena of delocalized plans and interleaving.

Featureous provided support for evolutionary modification of features in the following ways. Firstly, Featureous Location made it possible to efficiently establish traceability mappings between the evolving specifications of functional requirements and the source code of an application. Secondly, Featureous Analysis provided views that help identifying source code of a feature under modification, anticipating change propagation among features and consciously managing source code reuse among features. Finally, Featureous Manual and Automated Remodularization supported improving physical separation and localization of features in packages, to confine evolutionary modifications and reduce the risk of accidental inter-feature change propagations.

BIBLIOGRAPHY

[1] IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990.

[2] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. M. (1999). N degrees of separation: multi-dimensional separation of concerns. In ICSE'99: *Proceedings of the 21st international conference on Software engineering*, 107-119.

[3] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., M., Lopes, C., V., Maeda, C. and Mendhekar, A. (1996). Aspect-oriented programming. *ACM Computing Surveys*, vol. 28.

[4] Turner, C. R., Fuggetta, A., Lavazza, L. and Wolf, A. L. (1999). A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1), 3-15.

[5] Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058.

[6] Fowler, M. (2009). Feature Branch. *Blog post.* http://martinfowler.com/bliki/FeatureBranch.html

[7] Chikofsky, E. J. and Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), 13-17.

[8] Lehman, M.M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1060-1076.

[9] Cook, S., Harrison, R, Lehman, M. M. and Wernick, P. (2006). Evolution in Software Systems: Foundations of the SPE Classification Scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1), 1–35.

[10] Port, O. (1988). The software trap – automate or else. *Business Week* 3051 (9), 142-154.

[11] Moad, J. (1990). Maintaining the competitive edge. *Datamation,* 64, 61-62.

[12] Eastwood, A. (1993). Firm fires shots at legacy systems. *Computing Canada* 19 (2), 17.

[13] Erlikh, L. (2000). Leveraging legacy system dollars for E-business. *(IEEE) IT Pro*, May/June 2000, 17-23.

[14] Greevy, O., Girba, T. and Ducasse S. (2007). How developers develop features. In CSMR'07: *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 265-274.

[15] Shaft, T.M. and Vessey, I. (2006). The role of cognitive fit in the relationship between software comprehension and modification. *MIS Quarterly*, 30(1), 29-55.

[16] Nosek, J. and Palvia, P. (1990). Software maintenance management: changes in the last decade. *Journal of Software Maintenance: Research and Practice,* 2(3), 157-174.

[17] Bennett, K. H. (1991). Automated support of software maintenance. *Information and Software Technology*, 33(1), 74-85.

[18] Van Vliet, H. (2000). Software Engineering: Principles and Practice. Wiley.

[19] Fjeldstad, R. and Hamlen, W. (1983). Application program maintenance-report to our respondents. *Tutorial on Software Maintenance*, 13-27.

[20] Standish, T. (1984). An essay on software reuse. *IEEE Transactions on Software Engineering* SE-10 (5), 494-497.

[21] Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2), 211-221.

[22] Krasner, G.E. and Pope, S.T. (1988). A Cookbook for Using the Model- View-Controller User Interface Paradigm in Smalltalk-80. *Joop Journal Of Object Oriented Programming*, 1(3), 26-49.

[23] Garlan, D. and Shaw M. (1994). An Introduction to Software Architecture. Technical Report. Carnegie Mellon Univ., Pittsburgh, PA, USA.

[24] Van Den Berg, K., Conejero, J. M. and Hernández, J. (2006). Analysis of crosscutting across software development phases based on traceability. In EA'06: *Proceedings of the 2006 international workshop on Early aspects at ICSE*, 43–50.

[25] Von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. (D. L. Carver, Ed.)*Computer*, 28(8), 44-55.

[26] Letovsky, S. and Soloway, E. (1986). Delocalized Plans and Program Comprehension. *IEEE Software*, 3(3), 41-49.

[27] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N. and Aho, A. V. (2008). Do crosscutting concerns cause defects?. *IEEE Transactions on Software Engineering*, 34, 497-515.

[28] Rugaber, S., Stirewalt, K. and Wills, L. M. (1995). The interleaving problem in program understanding. In WCRE'95: *Proceedings of the 2nd Working Conference on Reverse Engineering*, 166-175.

[29] Benestad, H. C., Anda, B. and Arisholm, E. (2009). Understanding cost drivers of software evolution: a quantitative and qualitative investigation of change effort in two evolving software systems. *Empirical Software Engineering*, 15(2), 166-203.

[30] Cornelissen, B., Zaidman, A., Van Deursen, A. and Van Rompaey, B. (2009). Trace visualization for program comprehension: A controlled experiment. In ICPC'09: *Proceedings of the 17th International Conference on Program Comprehension*, 100-109.

[31] Röthlisberger, D., Greevy, O. and Nierstrasz, O. (2007). Feature driven browsing. In ICDL'07: *Proceedings of the 2007 international conference on Dynamic languages*, 79-100.

[32] Lukka, K. (2003). The Constructive Research Approach. *Case study research in logistics*, 83-101.

[33] Dunsmore, A., Roper, M. and Wood, M. (2000). Object-oriented inspection in the face of delocalisation. In ICSE'00: *Proceedings of the 22nd international conference on Software engineering*, 467-476.

[34] Brooks, F. P. (1995). *The Mythical Man-Month*. Addison Wesley.

[35] Langlois, R. N. and Robertson, P. L. (1992). Networks and innovation in a modular system: Lessons from the microcomputer and stereo component industries. *Research Policy*, 21(4), 297-313.

[36] Langlois, R. N. and Garzarelli, G. (2008). Of Hackers and Hairdressers: Modularity and the Organizational Economics of Open-source Collaboration. *Industry Innovation*, 15(2), 125-143.

[37] Nelson, R. R. and Winter, S. G. (1977). In search of useful theory of innovation. *Research Policy*, 6(1), 36–76.

[38] Baldwin, C. Y. and Clark, K. B. (2000). *Design Rules: The Power of Modularity*. MIT Press.

[39] Garlan, D., Kaiser, G. E. and Notkin, D. (1992). Using Tool Abstraction to Compose Systems. *Computer*, 25(6), 30-38.

[40] Parnas, D.L. (1979). Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2), 128-138.

[41] Stevens, W. P., Myers, G. J. and Constantine, L. L. (1974). Structured Design. (E. Yourdon, Ed.)*IBM Systems Journal*, 13(2), 115-139.

[42] Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.

[43] Dijkstra, E. W. (1974). On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, 60-66.

[44] Ossher, H. and Tarr, P. (2000). On the Need for On-Demand Remodularization. In ECOOP'2000 workshop on Aspects and Separation of Concerns.

[45] Roehm, T., Tiarks, T., Koschke, R. and Maalej, W. (2012). How do professional developers comprehend software?. In ICSE'12: *Proceedings of the 2012 International Conference on Software Engineering*, 255-265.

[46] Wilde, N., Gomez, J.A., Gust, T. and Strasburg, D. (1992). Locating user functionality in old code. In ICSM'92: *Proceedings of International Conference on Software Maintenance*, 200-205.

[47] Biggerstaff, T. J., Mitbander, B. G. and Webster, D. (1993). The concept assignment problem in program understanding. In ICSE'93: *Proceedings of 1993 15th International Conference on Software Engineering*, 482-498.

[48] Chen, K. and Rajlich, V. (2000). Case Study of Feature Location Using Dependence Graph. In IWPC '00: *Proceedings of the 8th International Workshop on Program Comprehension*. 241.

[49] Kästner, C., Apel, S. and Kuhlemann, M. (2008). Granularity in software product lines. In ICSE'08: *Proceedings of the 13th international conference on Software engineering*, 311-320.

[50] Wilde, N. and Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. *Journal Of Software Maintenance Research And Practice*, 7(1), 49-62.

[51] Salah, M. and Mancoridis, S. (2004). A Hierarchy of Dynamic Software Views: From Object-Interactions to Feature-Interactions. In ICSM '04: *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 72–81.

[52] Greevy, O. and Ducasse, S. (2005). Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach. In CSMR '05: *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, 314–323.

[53] Eisenbarth, T., Koschke, R. and Simon, D. (2003). Locating Features in Source Code, *IEEE Transactions on Software Engineering*, 210-224.

[54] Eaddy, M., Aho, A. V., Antoniol, G. and Guéhéneuc, Y. G. (2008). Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In ICPC '08: *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 53-62.

[55] Revelle, M., Dit, B. and Poshyvanyk, D. (2010). Using Data Fusion and Web Mining to Support Feature Location in Software. In ICPC'10: *Proceedings of 18th International Conference on Program Comprehension*, 14-23.

[56] Ratanotayanon, S., Choi, H. J. and Sim, S. E. (2010). My Repository Runneth Over: An Empirical Study on Diversifying Data Sources to Improve Feature Search. In ICPC'10: *2010 IEEE 18th International Conference on Program Comprehension*, 206-215.

[57] Li, S., Chen, F., Liang, Z. and Yang, H. (2005). Using Feature-Oriented Analysis to Recover Legacy Software Design for Software Evolution. In SEKE'05: *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, 336-341.

[58] Brcina, R. and Riebisch, M. (2008). Architecting for evolvability by means of traceability and features. In ASE'08 Workshops: *Proceedings of 23rd IEEEACM International Conference on Automated Software Engineering*, 72-81.

[59] Ratiu, D., Marinescu, R. and Jurjens, J. (2009). The Logical Modularity of Programs. In WCRE'09: *Proceedings of the Working Conference on Reverse Engineering*, 123-127.

[60] Wong, W. E., Gokhale, S. S. and Horgan, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2), 87-98.

[61] Sant'Anna, C., Figueiredo, E., Garcia, A. and Lucena, C. (2007). On the Modularity of Software Architectures: A Concern-Driven Measurement Framework. In ECSA'07: *European Conference on Software Architecture ECSA*, 207-224.

[62] Mäder, P. and Egyed, A. (2011). Do software engineers benefit from source code navigation with traceability? - An experiment in software change management. In ASE'11: *Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering*, 444-447.

[63] Booch, G., Rumbaugh, J. and Jacobson, I. (1999). The Unified Modeling Language User Guide. Addison-Wesley.

[64] Kothari, J., Denton, T., Shokoufandeh, A. and Mancoridis, S. (2007). Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence. In ICPC'07: *Proceedings of the 15th IEEE International Conference on Program Comprehension*, 17-26.

[65] Fowler, M. (1999). Refactoring: Improving the Design of Existing Programs. Addison-Wesley.

[66] Bergel, A., Ducasse, S. and Nierstrasz, O. (2005). Analyzing Module Diversity, *Journal of Universal Computer Science*, 11, 1613–1644.

[67] Murphy, G. C., Lai, A., Walker, R. J. and Robillard, M. P. (2001). Separating features in source code: an exploratory study. In ICSE'01: *Proceedings of the 23rd International Conference on Software Engineering*, 275-284.

[68] Demeyer, S., Ducasse, S. and Nierstrasz. O. (2009). Object-Oriented Reengineering Patterns, Square Bracket Associates, 1-338.

[69] Mehta, A. and Heineman, G. T. (2002). Evolving legacy system features into fine-grained components. In ICSE'02: *Proceedings of the 24th International Conference on Software Engineering*, 417-427.

[70] Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. In ECOOP'97: *Proceedings of the European Conference on Object-Oriented Programming*, 419-443.

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

[72] Liu, J., Batory, D. and Lengauer, C. (2006). Feature oriented refactoring of legacy applications. In ICSE '06: *Proceedings of the 28th international conference on Software engineering*, 112-121.

[73] McDirmid, S., Flatt, M. and Hsieh, W. C. (2001). Jiazzi: new-age components for old-fashioned Java. In OOPSLA'01: *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 211-222.

[74] Tzerpos, V. and Holt, R. C. (2000). ACDC: An Algorithm for Comprehension-Driven Clustering. In WCRE'00: *Proceedings of Seventh Working Conference on Reverse Engineering*, 258-267.

[75] Mancoridis, S., Mitchell, B. S., Chen, Y. and Gansner, E. R. (1999). Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In ICSM'99: *Proceedings of the IEEE international Conference on Software Maintenance*, 50.

[76] Bauer, M. and Trifu, M. (2004). Architecture-aware adaptive clustering of oo systems. In CSMR'04: *Proceedings of European Conference on Software Maintenance and Reengineering*, 3–14.

[77] O'Keeffe, M. and O'Cinneide, M. (2008). Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4), 502-516.

[78] Seng, O., Bauer, M., Biehl, M. and Pache, G. (2005). Search-based improvement of subsystem decompositions. In GECCO'05: *Proceedings of the 2005 conference on Genetic and evolutionary computation*, 1045-1051.

[79] Falkenauer, E. (1998). Genetic algorithms and grouping problems. Wiley.

[80] Harman, M. and Tratt, L. (2007). Pareto Optimal Search Based Refactoring at the Design Level. In GECCO'07: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 1106-1113.

[81] Bowman, M., Briand, L. C. and Labiche, Y. (2010). Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms. *IEEE Transactions on Software Engineering*, 36(6), 817-837.

[82] Praditwong, K., Harman, M. and Yao, X. (2011). Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, 37(2), 264-282.

[83] Cohen, J., Douence, R. and Ajouli, A. (2012). Invertible Program Restructurings for Continuing Modular Maintenance. In CSMR'12: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*.

[84] Janzen, D. and De Volder, K. (2004). Programming with crosscutting effective views. In ECOOP'04: *Proceedings of the 18th European Conference on Object-Oriented Programming*, 195-218.

[85] Desmond, M., Storey, M.-A. and Exton, C. (2006). Fluid Source Code Views. In ICPC'06: *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 260-263.

[86] NetBeans, http://netbeans.org

[87] Featureous, http://featureous.org/

[88] BeanShell2, http://code.google.com/p/beanshell2/

[89] Jacobsen, I. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach. ACM Press, Addison-Wesley.

[90] Liu, K., Alderson, A., Qureshi, Z. (1999). Requirements recovery from legacy systems by analyzing and modelling behavior. In ICSM'99: *Proceedings of the IEEE International Conference on Software Maintenance.*

[91] AspectJ, http://www.eclipse.org/aspectj/

[92] BTrace, http://kenai.com/projects/btrace/

[93] BlueJ, http://www.bluej.org/

[94] JHotDraw. http://www.jhotdraw.org/

[95] ConcernTagger data, http://www.cs.columbia.edu/~eaddy/concerntagger/

[96] Landi, W. (1992). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), 323-337.

[97] Kiczales, G., Des Rivieres, J. and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol.* (M. I. T. Press, Ed.)*POPL* (Vol. 61, p. 335). MIT Press.

[98] Li, W., Matejka, J., Grossman, T., Konstan, J., A. and Fitzmaurice, G. (2011). Design and evaluation of a command recommendation system for software applications. *ACM Transactions on Computer-Human Interaction.* 18(2).

[99] Robillard, M.P. and Murphy, G.C. (2002). Concern graphs: finding and describing concerns using structural program dependencies. In ICSE '02: *Proceedings of the 24th International Conference on Software Engineering*, 406–416.

[100] Egyed, A., Binder, G. and Grunbacher, P. (2007). STRADA: A Tool for Scenario-Based Feature-to-Code Trace Detection and Analysis. In ICSE'07: *Proceedings of 29th International Conference on Software Engineering*, 41-42.

[101] Storey, M., Wong, K. and Muller, H. (2000). How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3), 183-207.

[102] Lanza, M. (2003). CodeCrawler-lessons learned in building a software visualization tool. In CSMR'03: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering,* 409-418.

[103] Yau, S. S., Collofello, J. S. and MacGregor, T. (1978). Ripple effect analysis of software maintenance. In COMPSAC'78: *Proceedings of the Second International Computer Software and Applications Conference,* 60 – 65.

[104] Storey, M. A., Fracchia, F. D. and Muller, H. A. (1997). Cognitive design elements to support the construction of a mental model during software visualization. In IWPC'97: *Proceedings Fifth International Workshop on Program Comprehension,* 17-28.

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

[106] Sillito, J., Murphy, G. C. and De Volder, K. (2006). Questions programmers ask during software evolution tasks. In FSE'06: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering,* 23.

[107] Robillard, M. P., Coelho, W. and Murphy, G. C. (2004). How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 889-903.

[108] Prefuse, http://prefuse.org/

[109] Kothari, J., Denton, T., Mancoridis, S. and Shokoufandeh, A. (2006). On Computing the Canonical Features of Software Systems. In WCRE'06: *Proceedings of the 13th Working Conference on Reverse Engineering*, 93-102.

[110] Storey, M. A. (2005). Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In IWPC'05: *Proceedings of the 13th International Workshop on Program Comprehension*, 181-191.

[111] Myers, D., Storey, M.-A. and Salois, M. (2010). Utilizing Debug Information to Compact Loops in Large Program Traces. In CSMR'10: *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, 41-50.

[112] Simmons, S., Edwards, D., Wilde, N., Homan, J. and Groble, M. (2006). Industrial tools for the feature location problem: an exploratory study. *Journal of Software Maintenance and Evolution Research and Practice*, 18(6), 457-474.

[113] Apel, S. and Beyer, D. (2011). Feature cohesion in software product lines: an exploratory study. In ICSE'11: *Proceedings of the 33rd International Conference on Software Engineering*, 421-430.

[114] Coppit, D., Painter, R. R. and Revelle, M. (2007). Spotlight: A Prototype Tool for Software Plans. In ICSE'07: *Proceedings of the 29th International Conference on Software Engineering*, 754-757.

[115] NDVis, http://ndvis.sourceforge.net/

[116] Korson, T. and McGregor, J. D. (1990). Understanding object-oriented: a unifying paradigm. *Communications of ACM,* 33(9), 40-60.

[117] Rugaber, S. (2000). The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1), 143-192.

[118] Rajlich, V. and Wilde, N. (2002). The role of concepts in program comprehension. In IWPC'02: *Proceedings 10th International Workshop on Program Comprehension*, 271-278.

[119] Project Jigsaw, http://openjdk.java.net/projects/jigsaw/

[120] Lehman, M. M. and Ramil, J. F. (2000). Software evolution in the age of component-based software engineering, *IEEE Software*, 147(6), 249-255.

[121] Briand, L., Daly, J. and Wust, J. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 13(2), 115-121.

[122] Briand, L., Daly, J. and Wust, J. (1998). A unified framework for cohesion measurement in Object-Oriented systems. *Empirical Software Engineering*, 3(1), 65-117.

[123] Briand, L. C., Morasca, S. and Basili, V. R. (1995). Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering*, 22(1), 68-86.

[124] Marin, M., Van Deursen, A., Moonen, L., & Van Der Rijst, R. (2009). An integrated crosscutting concern migration strategy and its semi-automated application to JHotDraw. *Automated Software Engineering*, 16(2), 323-356.

[125] Lopez-Herrejon, R. E., Montalvillo-Mendizabal, L. and Egyed, A. (2011). From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. In SPLC'11: *Proceedings of the 15th International Software Product Line Conference*, 181-190.

[126] Batory, D., Sarvela, J. N. and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6), 355-371.

[127] Clarke, J., Dolado, J. J., Harman, M., Jones, B., Lumkin, M., Mitchell, B., Rees, K. and Roper, M.: (2003). Reformulating software engineering as a search problem. *IEEE Proceedings on Software*, 3, 161–175.

[128] Harman, M. and Clark, J. (2004). Metrics Are Fitness Functions Too. In METRICS'04: *Proceedings of the IEEE International Symposium on Software Metrics*, 58-69.

[129] Simon, H.A. (1969). The Sciences of the Artificial. MIT Press.

[130] Zhang, Y., Harman, M. and Mansouri, S. A. (2007). The multi-objective next release problem. In GECCO'07: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 1129-1137.

[131] Rosenman, M. A. and Gero, J. S. (1985). Reducing the pareto optimal set in multicriteria optimization. *Engineering Optimization*, 8(3), 189-206.

[132] Chaudhari, P. M., Dharaskar, R. V. and Thakare V. M. (2010). Computing the Most Significant Solution from Pareto Front obtained in Multiobjective Evolutionary Algorithms. *International Journal of Advanced Computer Science and Applications*, 1(4), 63-68.

[133] Recoder, http://recoder.sourceforge.net/

[134] Apel, S., Liebig, J., Kästner, C., Kuhlemann, M. and Leich, T. (2009). An Orthogonal Access Modifier Model For Feature-oriented Programming. In FOSD'09: *Proceedings of the First International Workshop on Feature Oriented Software Development,* 27-33.

[135] Melton, H. and Tempero, E. (2007). An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12(4), 389-415.

[136] FreeMind, http://freemind.sourceforge.net/

[137] RText, http://fifesoft.com/rtext/

[138] Räihä, O., Kundi, H., Koskimies, K. and Mäkinen, E. (2011), Synthesizing Architecture from Requirements: A Genetic Approach., *Relating Software Requirements and Architectures*, Springer, 307-331.

[139] Bodhuin, T., Di Penta, M. and Troiano, L. (2007). A search-based approach for dynamically re-packaging downloadable applications. In CASCON'07: *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, 27-41.

[140] Walkinshaw, N. D., Roper, R. M. and Wood, M. I. (2007). Feature location and extraction using landmarks and barriers. In ICSM'07: *Proceedings of the International Conference on Software Maintenance*, 54-63.

[141] Checkstyle, http://checkstyle.sourceforge.net/

[142] Alves, T. L., Ypma, C. and Visser, J. (2010). Deriving metric thresholds from bench-mark data. In ICSM'10: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 1-10.

[143] Alves, T. L., Correia, J. P. and Visser, J. (2011). Benchmark-based aggregation of metrics to ratings. In IWSM/MENSURA'11: *Proceedings of the Joint Conference of the 21th International Workshop on Software Measurement and the 6thInternational Conference on Software Process and Product Measurement*, 20-29.

[144] Baggen, R., Correia, J. P., Schill, K. and Visser, J. (2011). Standardized code quality benchmarking for improving software maintainability. In SQM'10: *Proceedings of the 4th International Workshop on Software Quality and Maintainability*, 1-21.

[145] Marin, M., Deursen, A. V. and Moonen, L. (2007). Identifying Crosscutting Concerns Using Fan-In Analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1), 1-37.

[146] Feature Driven Development, http://www.featuredrivendevelopment.com/

[147] Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10), 70-77.

[148] Cockburn, A. (2004). Crystal Clear a Human-Powered Methodology for Small Teams. Addison-Wesley Professional.

[149] Albrecht, A. J. (1979). Measuring Application Development Productivity. In: *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, 83–92.

[150] Jones, C. (1996). Programming Languages Table. *Software productivity research, Inc.*

[151] Cockburn, A. (2006). *Agile Software Development: The Cooperative Game*. Addison-Wesley Professional.

[152] Cusumano, M. A. and Selby, R. W. (1997). How Microsoft builds software. *Communications of the ACM*, 40(6), 53-61.

[153] Mylyn, http://www.eclipse.org/mylyn/

APPENDIX

A1. APIS OF FEATUREOUS WORKBENCH

This section discusses the APIs of Featureous Workbench. The concrete interfaces and implementations of the discussed APIs can be found in the open-source repository of the project [187].

A1.1 Extension API

The most important API of Featureous Workbench is the Extension API that allows for adding new feature-oriented views to the tool.

In order to extend Featureous Workbench with a new view, a new NetBeans module needs to be created. Apart from the dependency on the Featureous Core module, the new module has to be made dependent on the Lookup API, Utilities API and Window System API modules provided by the NetBeans RCP.

In order for a Java class in the created module to be recognized as a view by Featureous Workbench, a class needs to be annotated as a provider of the `dk.sdu.mmmi.featureous.explorer.spi.FeatureTraceView` service, as shown in Figure A.1. In order to actually provide the declared service, the view class needs also to extend the `dk.sdu.mmmi.featureous.explorer.api.AbstractTraceView` class. As a result, it will need to implement four abstract methods. These four methods (i.e. `createInstance`, `createView`, `updateView`, `closeView`) will be called by the Featureous Workbench infrastructure upon the creation, opening, update of traces and closing of a view.

```
@ServiceProvider(service=FeatureTraceView.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 loaded traces: " + ftms.size();
    JLabel status = new JLabel(msg);
    this.add(status, BorderLayout.CENTER);
  }

  public void updateView() { ... }

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

Figure A.1: Extending Featureous Workbench with a new feature-oriented view

In the simple example in Figure A.1, the name and the icon of the view are declared in the class constructor. The `createView` method is used to create a simple view that displays the number of currently loaded feature traces. The access to the currently loaded feature trace models is obtained through the `Controller` singleton class exposed by the *Core* module of Featureous Workbench.

Such a view class providing the `FeatureTraceView` service will be dynamically looked-up by the Featureous Workbench infrastructure when its module is loaded. The view will be represented by the declared name and icon on the toolbar of the main Featureous Workbench window. When such a button is clicked by a user to bring up the view, a new instance of the class will be created using the class's `createInstance` factory method. Subsequently, the `createView` method will be invoked to allow the view to populate its UI. In the case the set of traces is updated, the `updateView` method will be called, so that the view can reflect the changes. Lastly, the `closeView` method will be called to allow for custom deinitialization when the view is being closed down by the user.

A1.2 Source Utils API

The Source Utils API provides Featureous Workbench with static analyses of source code and with object-oriented metrics built on top of the analyses.

The static analysis infrastructure is centered on the concept of *static dependency model* (SDM) of Java source code. The SDM is a simple model that represents packages, classes and static dependencies among them. While this model currently provides only a partial representation of Java source code, it constitutes a sufficient basis for

computing a number of design-level metrics that are used in Featureous. The overall design of SDM is presented in Figure A.2.

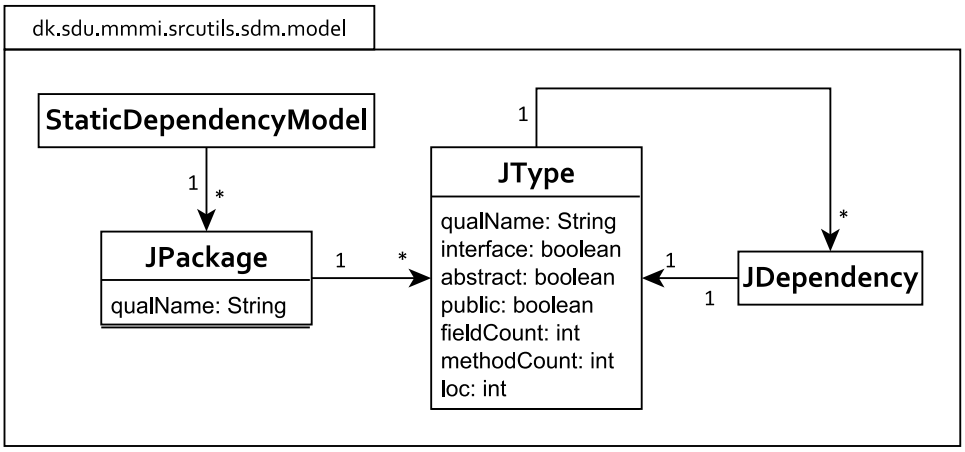


Figure A.2: The static dependency model used by Featureous Workbench

SDM can be instantiated for any Java codebase by using an importer implemented using the Recoder library [133]. In important property of the importer is the ability to extract a SDM based solely on source code parsing, i.e. full compilation of source code is not required. In consequence, it becomes possible to extract SDM from source code without supplying its dependent JAR files, extract SDM from subsets of the whole source code and even from source code that contains syntax errors (however, the erroneous units will be discarded). The independence of JAR files makes SDM a feasible basis for scalable automated measurement of static dependencies in Java systems.

Apart from the extractor, Source Utils API provides persistence support for SDM, reusable implementations of the most common queries and integration with the NetBeans source code infrastructure.

```
RecoderModelExtractor.extractSdmAndRunAsync(new RecoderModelExtractor.RunnableWithSdm() {

    public void run(StaticDependencyModel sdm) {
      JPackage pkg = sdm.getPackages().iterator().next();
      List<JType> classes = pkg.getTopLevelTypes();

      PCoupExport coupling = new PCoupExport();
      Double res = coupling.calculate(sdm, pkg);
    }

);
```

Figure A.3: Using Recoder to obtain static dependency model for main project in the IDE

Figure A.3 demonstrates how to use Source Utils API to extract an SDM from a project opened and marked as main in the NetBeans IDE. The provided example shows how an obtained SDM can be queried and how it can be used as an input to computing one

of the metrics provided by Source Utils module of Featureous Workbench. This module provides implementations of afferent/efferent cohesion and afferent/efferent coupling metrics. These implementations can be found in the `dk.sdu.mmmi.srcUtils.sdm.metrics` package.

A1.3 Feature Trace Model Access API

The `Core` module of Featureous Workbench exposes an API for accessing and manipulating trace models that encompass the traceability links captured by Featureous Location. In order to decouple the evolution of Featureous Location model format from the model format exposed to feature-oriented views, Featureous Workbench wraps the Featureous Location models. While the wrappers closely correspond to the original models discussed in Chapter 4, they are decorated with a number of additional utility methods and fields. The set of traces currently loaded in Featureous Workbench can be obtained from the `dk.sdu.mmmi.featureous.core.controller.Controller` singleton object by invoking: `TraceSet traceSet = Controller.getInstance().getTraceSet()`. The exposed classes of the feature trace model reside in the `dk.sdu.mmmi.featureous.core.model` package.

The primary scenario for using the trace models exposed by Featureous Workbench is adding a new feature-oriented view to the tool, as discussed earlier. A second supported scenario is custom exporting of the traceability without implementing a complete view. This can be done by using the BeanShell2 Java console [88] provided with Featureous Workbench. The console allows a user to insert and execute arbitrary Java routines that access or manipulate the current `TraceSet` object. One of the example routines provided with the console is shown in Figure A.4. This code demonstrates how to iterate over the models of traces and classes that they contain and how to query these models for various data.

```
for(TraceModel tm : traceSet){
  String name = tm.getName();
  print(name + ": \n");
  //Set<OrderedBinaryRelation<String, Integer>> invs = tm.getMethodInvocations();
  for(ClassModel cm : tm.getClassSet()){
    String pkgName = cm.getPackageName();
    String className = cm.getName();
    print(" - " + className + "\n");
    Set<String> methods = cm.getAllMethods();
    //Set<String> objsCreatedByFeat = cm.getInstancesCreated();
    //Set<String> objsUsedByFeat = cm.getInstancesUsed();
  }
};
```

Figure A.4: Example BeanShell2 script

Using the trace model access API combined with the static dependence model presented earlier, Featureous Workbench implements a number of state of the art feature-oriented metrics that are applied throughout this thesis. These implementations can be found in the `dk.sdu.mmmi.featureous.metrics.concernmetrics` package. Furthermore, Featureous Workbench provides two interesting static analyses

in the `dk.sdu.mmmi.featureous.core.staticlocation` package: a static control-flow-based slicing algorithm and algorithm for retrospective static approximation of read and write accesses to fields of the collected Featureous Location traces. While these two analyses are not used within this thesis, it is envision them a good starting point for development of third-party plugins concerned with static feature location and detection of resource-sharing-based feature interactions [51].

A1.4 Affinity API

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 Workbench with customized domain-specific affinity schemes.

A1.5 Selection API

Featureous Workbench is equipped with a global selection mechanism driven by the three traceability views. The remaining views are made to automatically react to on events of selecting features in the *feature explorer* view, selecting code units in *feature inspector* or call-tree views or by moving the caret of the *feature-aware source code editor* to classes and methods of interest. Most of the views react to such events by dynamically highlighting the selected entities or by dynamically re-layouting their visualizations. The global selection mechanism is also exposed for both writing and observing through the `SelectionManager` class.

A2. FEATURE LISTS

This thesis uses several open-source software systems as case studies. The lists of their features, as recognized by applying Featureous Location, are presented in the remainder of this section.

A2.1 JHotDraw SVG

JHotDraw SVG 7.2
Size: 62 KLOC
Annotated Feature-entry points: 91

Name	Summary description
Basic editing	cut, copy, paste figures
Selection tool	select click, select drag, deselect, move figure, resize figure
Drawing persistence	clear recent files, load drawing, load recent, save drawing
Manage drawings	new drawing, close drawing
Path editing	combine, split path
Text area tool	create text area, display text area, edit text
Grouping	group figures
Font palette	display palette
Attribute editing	pick attributes, apply attributes to figure
View palette	create, zoom in, zoom out
Align palette	display palette, align left, center, right, top, middle, bottom
Figure palette	display palette
Fill palette	display palette
Stroke palette	display palette
Automatic selection	clear selection, select all, select same
Canvas	display, resize
View source	view svg source
Link palette	display palette
Undo redo	undo, redo
Line tool	display line, create line, close path
Export	export drawing
Scribble tool	create bezier path
Arrange	bring to front, send to back
Text tool	create text, display text
Image tool	create image, display image, load image
Ellipse tool	create ellipse, display ellipse
Rectangle tool	create rectangle, display rectangle
Tool palette	display palette
Application startup	initialize application

A2.2 BlueJ

BlueJ 2.5.2
Size: 78 KLOC
Annotated Feature-entry points: 228

Name	Summary description
Project persistence	open project, save, save as
Create test method	create test method
Manage projects	new project, close project
Code pad	evaluate expression, error, exception, inspect object, show result
Invoke method	invoke method, dialog, show result
Compilation	compile project, compile selected, show warnings, rebuild
Team	checkout, commit, import, show log, settings, show status, update
Testfixtures	objects to fixture, fixture to objects
Application close	close application
Export	export project as jar
Debugger control	stop, step, step into, continue, terminate
Edit class implementation.persistence	show file, open file, reload file
Startup	initialize application
Inspect	inspect object, class, method result
Inheritance arrow management	new, remove, show
Package management	new, remove
Add class from file	insert Java file as class
Edit class implementation.code editing	display line number, paint syntax highlighting, match brackets,
Insert method into code	insert existing method
Preferences	show, class manager, editor, extension manager, misc, preference manager,
Edit class implementation.basic editing	find, replace, code message, warning, undo, redo, comment, uncomment, indent, deindent, go to line
Class management	new, remove,
Import	import non-Bluej project
Generate project documentation	project documentation, class documentation
Print project	print, page setup
Class instance management	add, remove object
Debugger.display state	show, instance field, static field, variable, step marks
Package description	readme edit, save, load
Create objects from lib classes	run constructor, run method
Open class documentation	open class doc
Run tests	run single, run all, show results
Open class implementation	open code editor
Terminal	show, clear, save
Breakpoint management	toggle breakpoint, remove breakpoint, display breakpoint in editor
Uses arrow management	new, remove, show
Key bindings management	key binding dialog, add, remove
Machine icon	show status
Program statics display	package editor
Program dynamics display	object bench

A2.3 FreeMind

FreeMind 0.7.1
Size: 14 KLOC
Annotated Feature-entry points: 87

Name	Summary description
Cloud node	cloud, cloud color
Display map	draw on screen, layout, fold nodes, fold node children, node up, node down, anti-aliasing
Documentation	show documentation, faq
Edit basic	cut, paste, copy, copy single
Edit map	new child node, new sibling, new prev sibling, delete, join nodes, rename, edit long
Evaluate	patterns
Exit program	exit program
Export map	export to html, properties
Icons	icon, set image
Import export branch	export branch, import, import linked, import without root, import folder structure, import explorer favorites, export branch to html
Init program	initialize program
Link node	set link by file, set link by text, follow link
Modify edge	linear style, sharp linear, bezier style, sharp bezier, color, width parent, width thin, width number
Modify node	fork style, bubble style, font, font size, font style, smaller font, larger font, color, blend color
Multiple maps	list, switch, previous command, next command, close map
Multiple modes	mindmap, file, list modes, switch mode
Navigate map	next, previous, forward, backward, move to root, find, find next
New map	create a new mind map
Open map	open a persisted mind map
Print map	page setup, print
Save map	save, save as
File mode	switch to file mode
Browse mode	switch to browse mode
Zoom	zoom in, zoom out

A2.4 RText

RText 0.9.8
Size: 55 KLOC
Annotated Feature-entry points: 82

Name	Summary description
Display text	render text, decrease font, increase font, line numbers, java syntax coloring
Edit basic	copy, paste, cut, select all
Edit text	write, delete, insert date/time
Exit program	exit program
Export document	export as html
Init program	initialization, splash screen
Modify document properties	line terminator, encoding
Modify options	interface options, printing options, text area options, language, file filters, source browser options, shortcuts
Modify text	delete end of line, join lines, to lower case, to upper case, invert case, intend, un-intend, toggle comment
Multiple documents	list, switch, close, close all
Navigate text	find, find next, replace, replace next, replace all, find in files, replace in files, go to, quick search bar, go to matching bracket
New document	create new document
Open document	text, java, other formats, open in new window, document history
Playback macro	load, playback last
Print document	print, print preview
Record macro	begin, stop, record temporary
Save document	save, save as, many formats, save all
Show documentation	show help topics
Undo redo	undo, redo
Use plugins	plugin init, plugin list, plugin options

A2.5 JHotDraw Pert

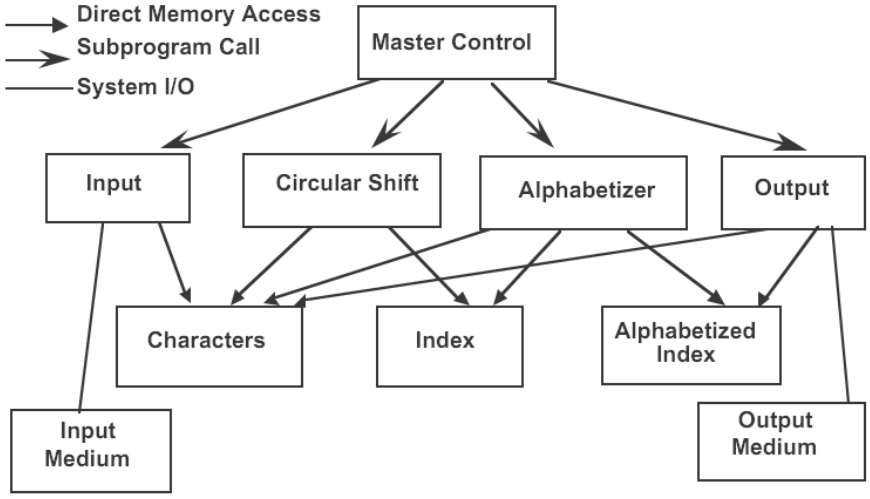
JHotDraw Pert 7.2
Size: 72 KLOC
Annotated Feature-entry points: 135

Name	Summary description
Align	left, center, right, top, middle, bottom
Dependency tool	create dependency, edit dependency, render dependency
Edit basic	cut copy paste
Edit figure	duplicate delete
Exit program	exit program
Export drawing	export drawings to several formats
Group figures	group ungroup figures
Init program	initialize program
Modify figure	fill color, pen color, pen width, pen dashes, pen stroke, pen stroke placement, pen caps, pen corners, no arrow, arrow at start, arrow at end, arrow kind, font, font size, font style, font color, pick attributes, apply attributes, move by unit
Multiple windows	new window, close window
New drawing	create a new drawing
Open drawing	xml format, open recent
Order figures	bring to back, bring to front
Save as drawing	xml format
Selection tool	select click, select drag, deselect, move figure, resize figure, select all
Snap to grid	snap, show grid
Task tool	create task, render task
Text tool	write text, edit text, edit text in figure, render text
Undo redo	undo, redo
Zoom	zoom in, zoom out

A3. FOUR MODULARIZATIONS OF KWIC

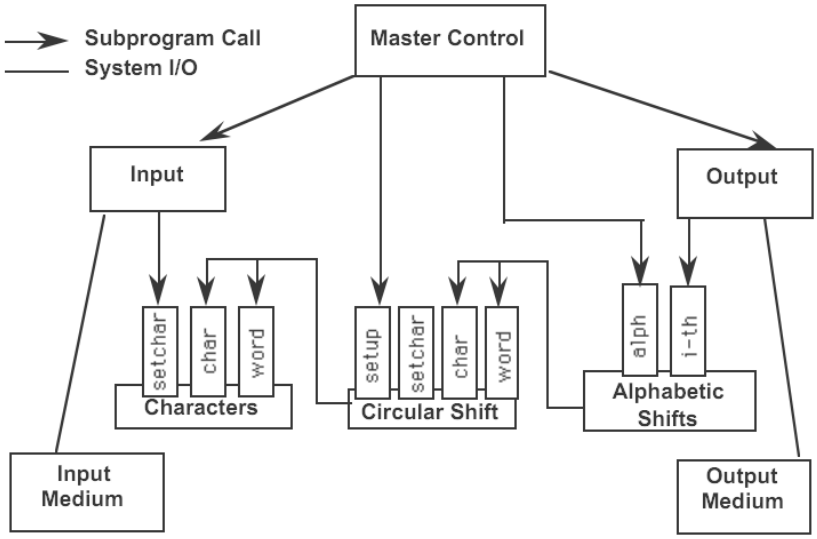
The following description of four modularizations of KWIC was reproduced from [23]:

A3.1 Shared data modularization



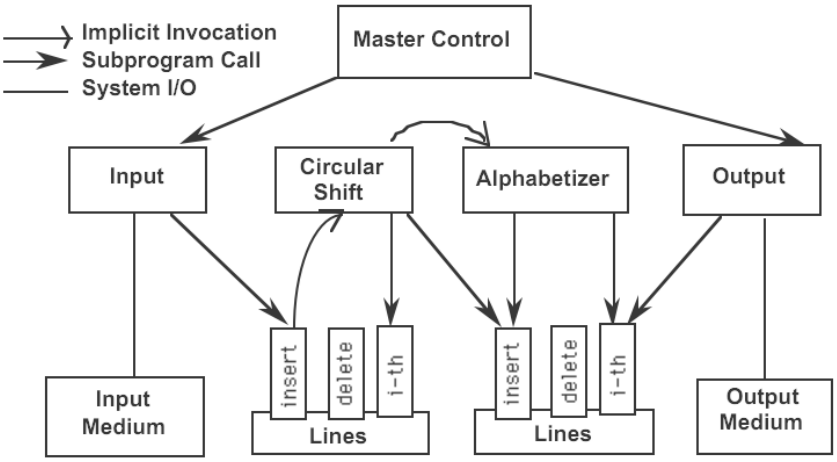
"The first solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output. These computational components are coordinated as subroutines by a main program that sequences through them in turn. Data is communicated between the components through shared storage ("core storage"). Communication between the computational components and the shared data is an unconstrained read-write protocol. This is made possible by the fact that the coordinating program guarantees sequential access to the data."

A3.2 Abstract data type modularization



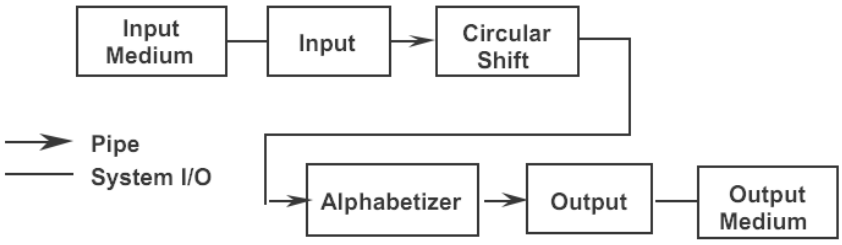
"*The second solution decomposes the system into a similar set of five modules. However, in this case data is no longer directly shared by the computational components. Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface.*"

A3.3 Implicit invocation modularization



"The third solution uses a form of component integration based on shared data similar to the first solution. However, there are two important differences. First, the interface to the data is more abstract. Rather than exposing the storage formats to the computing modules, data is accessed abstractly (for example, as a list or a set). Second, computations are invoked implicitly as data is modified. Thus interaction is based on an active data model. For example, the act of adding a new line to the line storage causes an event to be sent to the shift module. This allows it to produce circular shifts (in a separate abstract shared data store). This in turn causes the alphabetizer to be implicitly invoked so that it can alphabetize the lines."

A3.4 Pipes and filters modularization



"The fourth solution uses a pipeline solution. In this case there are four filters: input, shift, alphabetize, and output. Each filter processes the data and sends it to the next filter. Control is distributed: each filter can run whenever it has data on which to compute. Data sharing between filters is strictly limited to that transmitted on pipes."