Science of Computer Programming 77 (2012) 131–151

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Remodularizing Java programs for improved locality of feature implementations in source code

Andrzej Olszak *, Bo Nørregaard Jørgensen

The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark

ARTICLE INFO

Article history:
Received 28 March 2010
Received in revised form 22 October 2010
Accepted 24 October 2010
Available online 6 November 2010

Keywords:
Features
Remodularization
Feature location
Fragile decomposition problem

ABSTRACT

Explicit traceability between features and source code is known to help programmers to understand and modify programs during maintenance tasks. However, the complex relations between features and their implementations are not evident from the source code of object-oriented Java programs. Consequently, the implementations of individual features are difficult to locate, comprehend, and modify in isolation. In this paper, we present a novel remodularization approach that improves the representation of features in the source code of Java programs. Both forward and reverse restructurings are supported through on-demand bidirectional restructuring between feature-oriented and object-oriented decompositions. The approach includes a feature location phase based on tracing of program execution, a feature representation phase that reallocates classes into a new package structure based on single-feature and multi-feature packages, and an annotation-based reverse transformation of code. Case studies performed on two open-source projects indicate that our approach requires relatively little manual effort and reduces tangling and scattering of feature implementations in the source code.

1. Introduction

Program comprehension was reported to consume more than half of all resources during software maintenance [1]. An important part of the overall comprehension effort is spent on relating program features to source code [2]. A *feature* is a coherent and identifiable bundle of system functionality that helps to characterize the system from the user's perspective [2]. Features serve software users as a means of formulating their requirements, change requests, and error reports. In turn, developers need to relate the task descriptions obtained from users to appropriate fragments of a program's source code in order to carry out the requested perfective, adaptive or corrective modifications [1]. Hence, the easier the correspondence between features and their implementations can be understood, the easier it is to modify a program's functionality [3].

In object-oriented programs, the relations between features and their implementations are generally not observable in the structure of the source code, because the object-oriented style of decomposing programs does not consider features as the units of decomposition. Instead, object-oriented programs are decomposed according to entities from a program's problem domain, and to particular architectural styles, i.e. layered arrangement of code [4,5]. In such decompositions, features are implemented by collaborations between objects. Thus, features have no explicit representation and they may crosscut multiple classes and multiple architectural units. This delocalization makes it a difficult and tedious task to identify the classes and methods that implement a given feature in unfamiliar code. As a result, programmers need to use additional effort to understand how features are implemented, and how their implementations relate to each other [6]. This effort

* Corresponding author.
E-mail addresses: ao@mmmi.sdu.dk (A. Olszak), bnj@mmmi.sdu.dk (B. Nørregaard Jørgensen).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.10.007

accumulates throughout a program's evolution, as understanding of feature implementations and their interrelationships is a prerequisite to modifying and assessing the impact of any functional change.

In this paper, we present an approach aimed at improving locality of feature implementations of source code of legacy Java programs, and thereby facilitating feature-centric program comprehension. We do this by proposing an approach to feature-oriented remodularization, which establishes explicit representations of features in the package structures of object-oriented programs. To address the identified problem of fragile decompositions, we equip our remodularization approach with the capability of bidirectional restructurings between the original and the feature-oriented decompositions. Forward restructuring consists of two phases: feature location and feature representation. First, program features are located by tracing the execution of methods based on the so-called feature entry points. Then, explicit representations of features are created, while preserving reified domain concepts, by reallocating classes from the original package structure into a new package structure, centered around single-feature and multi-feature packages. The whole process requires some manual effort during the feature location phase and is fully automatic for the feature representation phase. To address the fragile decomposition problem, i.e. situations where the decomposition of source code cannot be kept in further development, the approach includes automated reverse restructuring of code, so that transitions between the object-oriented and feature-oriented program decompositions can be performed both directions in an on-demand fashion.

To evaluate our approach, we have conducted case studies on two open-source software applications: JHotDraw SVG and BlueJ. The obtained results show that the application of our forward restructuring approach reduced the total scattering [7] of feature implementations by 29% and 23%, and their total tangling [7] by 49% and 38%. Furthermore, during these case studies we have observed that the manual effort required by our approach was relatively low.

The rest of this paper is organized as follows: Section 2 motivates our work and introduces the fragile decomposition problem. Section 3 provides an overview of our remodularization approach. Section 4 describes the implementation details of the approach. Section 5 presents the two case studies and discusses the results and threats to validity. Section 6 presents related work. Finally, Section 7 concludes the paper.

2. Motivation

Object-oriented decomposition of programs is based on systems of collaborating classes [4]. The *classes* and their interrelations reflect either *entities* found in the problem domain, or solution-specific concepts used by programmers to structure their programs (persistence classes, design patterns, etc.). The entities are typically identified during requirements analysis and usually captured in the form of a *domain model* [8]. Thus, domain models establish a mapping between entities in the problem domain and their respective classes in the program. Furthermore, it is common in object-oriented decompositions to represent certain concerns, like for instance the *domain model* concern, in terms of *layered architectures* [4] in order to separate them according to their high-level implementation responsibilities [5].

The focus of object-oriented programming on direct representation of real-world entities and their relations in source code is advantageous in several ways. Firstly, comprehension of code units and designs is improved, since the structure of source code mimics the structure of the real-world problem domain. Secondly, evolutionary modifications of programs caused by change of perception of entities in the problem domain can be confined within individual entities in the domain model. Despite these significant advantages, object-oriented decompositions are, however, no exception to the rule of the *tyranny of dominant decomposition* [9]. As is the case for any program decomposition, object-oriented decompositions can only modularize one dimension of concern at a time. Hence, other alternative dimensions, like the feature dimension, will be misrepresented.

Since features characterize a program from the users' perspective [2], they are inherently rooted in a program's problem domain and are implemented in the program either to mimic existing real-world *processes,* or to provide new processes that the users desire. This makes features natural units of change for implementing evolutionary changes concerned with user-identifiable functionality. However, because feature implementations crosscut multiple classes and multiple architectural layers due to the tyranny of the object-oriented decomposition, it is difficult to modify the implementations of existing features as many, seemingly unrelated, places in code have to be changed together. The two known sources of comprehension overhead in such situations are the phenomena of *delocalized plans*, which happen when a programming plan (such as an implementation of a feature) is being realized by lines of code scattered in different parts of the program [10], and *interleaving*, which occurs when lines of code responsible for accomplishing more than one purpose may be woven together in a single section [11]. In our setting, we associate the delocalized plans problem with the concern-oriented notion of *scattering* of feature implementations over multiple code units (methods, classes, packages) [7], and the interleaving problem with the notion of *tangling* of feature implementations in terms of individual code units [7].

The mentioned situation can be improved by recovering, and explicitly visualizing the implicit mappings between features and object-oriented source code [12,13]. However, providing an explicit bidirectional traceability is not enough to remedy the problems of delocalized plans and interleaving. To address these issues, it is necessary to change the program decomposition—i.e. to remodularize the decomposition according to the feature-oriented criteria, rather than purely the object-oriented criteria. The ultimate aim of such a feature-oriented decomposition is to create a one-to-one correspondence between features and the static structure of object-oriented source code, analogously to the way object-oriented decompositions correlate source code's structure with entities found in the problem domain. Only by means of feature-oriented remodularization is it possible to obtain the three outcomes of modularity [14]: feature-wise program comprehen-

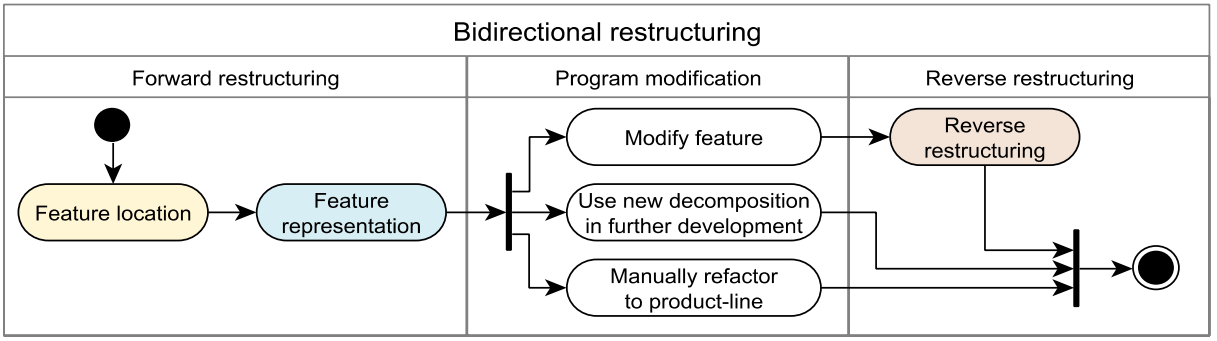

Fig. 1. Usage scenarios for feature-oriented remodularization.

sion, feature-wise division of work, and confinement of functional change propagation within individual program modules. However, the condition necessary for achieving reproducibility and scalability of feature-oriented remodularization with respect to program size is a high level of automation as suggested in [15].

2.1. Usages of feature-oriented remodularization

There are two scenarios of applying our approach to a legacy Java program. For discussing them, we follow the taxonomy of Chikofsky and Cross [16] by terming them as *restructurings*. Additionally, we indicate the direction of the restructurings to differentiate between *forward restructuring, reverse restructuring* and *bidirectional restructuring*. Forward restructuring is the transition from the original decomposition to feature-oriented decomposition. Consequently, *reverse restructuring* recovers the original decomposition from the feature-oriented one. *Bidirectional restructuring* is provided by granting both forward restructuring and reverse restructuring.

Firstly, our feature-oriented remodularization can be used in a *forward restructuring* scenario. As depicted in Fig. 1, this restructuring consists of *locating* feature implementations and *representing* feature implementations in source code (the coloring scheme in Fig. 1 marks the distinct stages; this scheme will be used further in the paper to allow for interrelating of figures). The thereby established feature-oriented decomposition of a program can serve as a basis for further development or it can be used as a starting point for further refactoring of an existing software product into a software product-line. Settling with a feature-oriented decomposition may be required due to organizational policy (e.g. introduction of a software product-line), or due to a need for feature-wise division of work in a program, which was found to be an important concern in the later stages of software projects [17].

The second scenario is the *bidirectional restructuring* usage. In this scenario, a legacy program is being forward-restructured to a feature-oriented decomposition in order to perform feature-wise modifications of source code. When the modifications are finalized, the program undergoes a reverse restructuring back to its original decomposition. This mechanism allows for performing remodularization in an on-demand fashion [9], since it makes it possible for the developers to select the decomposition best suited for accomplishing a task at hand. For instance, extending an existing use case of an application is best accomplished by using a feature-oriented decomposition, whereas correcting the persistence layer is best accomplished by using the original object-oriented decomposition.

Furthermore, there exist situations where the developers cannot base the further development on the introduced decompositions of source code and have to revert to the original decompositions as soon as the feature-centric modification tasks are accomplished. Such situations create the need for availability of a reverse restructuring. These situations are caused by what we refer to as *the fragile decomposition problem*.

2.2. The fragile decomposition problem

We define the fragile decomposition problem as the situation in which there exist dependencies on a program's decomposition. In most cases, such dependencies will be specified in the source code of programs in the form of static dependencies and thus will be compile-verifiable.

The situations, in which a decomposition of a program becomes fragile, include the cases where a program is used, e.g. as a library, by other independently developed programs. In case of such dependencies, refactoring of the library's decomposition in order to improve its quality (e.g. introduction of the information hiding principle [14]) can result in a chain of unforeseen crosscutting changes in a number of dependent applications. Moreover, a program being remodularized can physically include source code of other programs (e.g. this is usually the case for framework-based applications [18]), and thereby become unable to arbitrarily remodularize its structure without breaking compatibility with future versions of the incorporated source code. This is also the case when contributing to open-source projects—any modifications made in a feature-oriented decomposition have to be ported back to the original decomposition before the changes can be committed to a project's repository.

If dependencies on a program's decomposition exist, modification of the decomposition (e.g. through feature-oriented remodularization) will affect the dependent parties. In response to the decomposition's change, a dependent party needs to undergo syntactical modifications (e.g. a dependency on particular names of methods, classes, etc.) as well as semantical modifications (e.g. a dependency on semantics of classes in the context of problem domain, protocols of interaction with them, etc.). Hence, the fragile decomposition problem makes it difficult to modify the initial decompositions of evolving programs, since doing so can have unforeseen consequences for the program's clients.

2.3. Comprehending feature implementations

We claim that the fragile decomposition problem has its repercussions on program comprehension. This is because it is not only programs that depend on a particular shape of a program's decomposition, it is also programmers who depend on it.

The first case of such a dependency occurs when a programmer is already familiar with a particular decomposition of a program, with the semantics of the established classes and the patterns of their interactions. Thereby his understanding of the program's source code depends on the presence of these learned concepts. If the program decomposition is then changed, especially in a way that substantially invalidates the original semantics of existing classes (i.e. changing semantics and re-assigning responsibilities), then the programmer can no longer apply his previous knowledge and she has to use additional effort to re-learn the program in its new decomposition.

The second case of dependency of a programmer on a program's decomposition is not due to invalidation of the knowledge of previous decomposition of a program, but due to misalignment between the decompositions of the problem domain and the solution domain. The key observation here is that not all decompositions of programs are equally easy to understand. As it is difficult to pinpoint a set of precise, yet general, properties that would make a source-code abstraction easier to understand, it appears feasible to assume that in most cases the abstractions that *can* be related to a program's problem domain are easier to understand than the ones that cannot. For instance, in a banking application it would be relatively easy for a programmer, who knows what a bank is, to get an impression of the meaning and responsibilities of a class called *Account*, based solely on its name. Given this premise and the assumption that classes in legacy object-oriented programs are designed to reflect entities in their problem domains, we postulate that particular attention during remodularization has to be paid to not invalidating the human-understandable classes that originate from the problem domains of programs. This observation corresponds to observations of Rajlich and Wilde [19] on the role of familiar concepts, of Ratiu et al. [20] on the delocalization of domain knowledge in programs and of Janzen and De Volder [21] who aim at giving programmers "the advantages of open classes, without having to give up the ability to edit the program directly in terms of classes".

Since the classes and methods in legacy object-oriented source code take part in implementing multiple features (e.g. *Account* class could be used by *make transfer* and *close account* features) they would have to be divided into parts – which we will call "fragments" to use a technology-neutral term – to allow for complete separation of feature implementations in code during remodularization. Doing so would mean a division of a single localized concept into a number of smaller fragments, each of which is assigned a subset of the original responsibilities and roles. The thereby introduced *delocalization of knowledge*, means that whenever there is a need for understanding or modifying the concept of *Account* as a whole, each of these fragments has to be visited by the programmer. Furthermore, the fragments are not guaranteed to have any reasonable semantics in terms of the program's problem domain, or even any intuitive meaning at all, making it difficult to understand them in isolation from each other. Finally, any modification made to one of the fragments can have unforeseen consequences on the other fragments. This is because division of a single coherent class into a number of class-fragments inevitably leads to a strong coupling between the fragments (at least semantically, but also syntactically—depending on the separation of concerns mechanism used). As a result, it is difficult to modify or remove a fragment without considering the impact of doing so on the rest of the system.

Overlaps of feature implementations at the granularity of classes, methods or even individual code statements, occur in legacy programs that were not designed for separation of features in source code. As we aim to apply our remodularization approach to legacy programs, it is relevant to ask how to find a reasonable balance between supporting program understanding by preserving existing abstractions and separating feature implementations in source code. A reasonable answer would be to separate classes and methods as much as possible without impacting program comprehension and in such a way that the resulting fragments are intuitively understandable and loosely coupled to each other. Since automated measurements for evaluating such context-sensitive comprehension properties are not yet available, one would have to rely on human judgment to achieve that. This would prevent such an approach from automation and thus makes it difficult to apply it to large-scale legacy software. In order to achieve automation and avoid any of the discussed negative effects of remodularization on comprehension, even for the cost of a limited separation of feature implementations, we decide not to divide the classes present in the original decompositions of legacy programs. The details of this decision and its effects on the feature representation phase of our remodularization approach will be discussed in Section 3.3.

3. Our approach

To aid programmers in taking advantage of feature-oriented decomposition of source code, we propose a remodularization approach that reduces the implicitness, delocalization and interleaving of feature implementations in

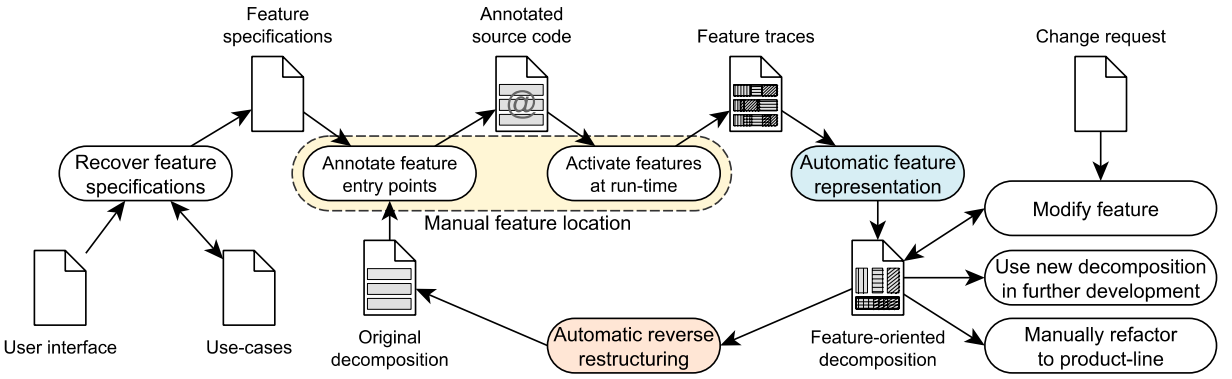

Fig. 2. Overview of the remodularization process.

source code, while preserving the familiar concepts from the original program. To tackle the fragile decomposition problem, we support bidirectional restructuring by providing reverse restructuring of code. To minimize the manual effort required by our approach we implement a lightweight feature location approach and we automate the phases of feature representation and reverse restructuring.

Fig. 2 shows a process overview of our approach. The main cycle is formed by the forward restructuring and the reverse restructuring that allow for bidirectional transition between the original decomposition and the feature-oriented decomposition.

As shown in Fig. 2, forward restructuring consists of two phases: user-driven *feature location* (consisting of *annotating feature entry points* and *activating features at run-time*) and automated *feature representation*. Having a legacy Java program structured according to object-oriented decomposition criteria as an input, we firstly discover subsets of its implementation responsible for implementing features. Based on this information, we establish a new package structure that re-groups the program's classes according to their participation in implementations of features. Such an explicit representation of features is aimed at reducing delocalization, and interleaving. The granularity of classes is used for separating feature implementations, in order not to invalidate a programmer's existing knowledge of the program, and not to break the correspondences between the program classes and the entities in its problem domain. The reverse restructuring allows for recovery of the original decomposition of source code by relocating classes based on a collected meta-information. This capability enables our approach to be applied in an on-demand fashion—i.e. to switch between feature-oriented and object-oriented decomposition of programs according to the particular character of a maintenance task being handled. The presence of reverse restructuring also addresses the different cases of the fragile decomposition problem, which we have discussed earlier.

The preconditions to the usage of our remodularization approach are the availability of source code of a target program and the availability of the specifications of its features. Out of the two, feature specifications are oftentimes not present in legacy programs. Therefore, we complement our approach with a procedure and guidelines for *recovering feature specifications* from use cases, or if use cases are not available, from the domain knowledge and the program's user interface.

The remainder of this section discusses the mentioned steps of our approach and the design decisions behind them. The respective implementation-specific details will be described in Section 4.

3.1. Recovering feature specifications

Feature location is concerned with identifying fragments of source code that contribute to implementations of program features [22,23]. Defined in terms of Rajlich's *name-intension–extension* scheme [24], feature location is the process of establishing traceability links between the feature implementations in source code (extensions) and the feature names (names) that represent feature specifications (intensions). Hence, a prerequisite to establishing the traceability between feature names and feature implementations is the availability of both the source code and the specifications of features. 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* 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 [2]. We propose that use-case groups for forming features are established around the domain concepts (nouns) rather than activities (verbs). In the course of the proposed grouping strategy, a common generalization of the grouped activities needs to be found to form the semantics for the feature. For example, having three use cases {"open document", "close document", "open help"}, our suggested grouping would be between the first two use cases to form "document management" feature, rather than between the first and the third use case to form "opening of artifacts" feature.

If no documentation is available, one needs to understand the program's problem domain and its run-time behavior in order to identify the use cases, which can thereafter be grouped into features. For identifying use cases, elements of a graphical *user interface* allowing for user-triggerable actions (e.g. main menus, contextual menus, buttons, etc.), keyboard

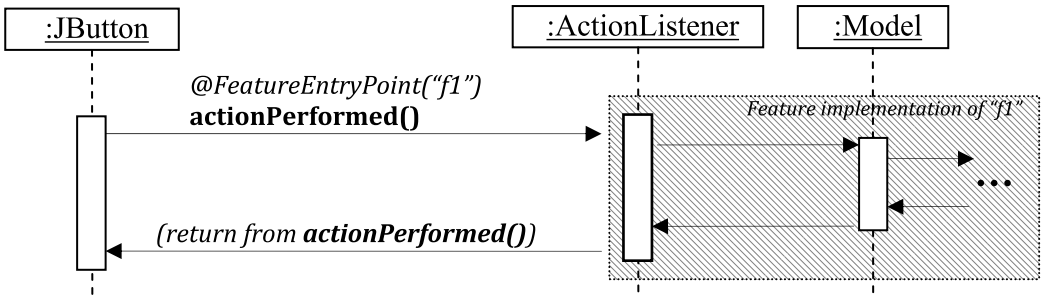

Fig. 3. Usage of feature entry points to locate features.

shortcuts and command-line commands are considered good candidates. Depending on the complexity of the problem domain and experience of the programmer, this process can be also performed using a more comprehensive methodology for requirements recovery, such as AMBOLS [25].

3.2. Feature location

The vast body of existing research on feature location, which we discuss in Section 6.1, indicates feasibility of various techniques, their combinations, and diverse sources of data for locating feature implementations. The choice of a particular technique determines the types of artifacts required, the amount of manual effort imposed on a programmer and the properties of precision and recall of the location process, as investigated in [51].

We propose a relatively simple, yet novel, dynamic feature location method, which does not require anything more than the source code of legacy programs. We chose to do so, as it suffices for generating the necessary input, equipped with information about feature implementation overlaps, for our feature presentation phase and, as we claim, it imposes no extensive manual effort on the user.

Our feature location technique is based on the observation that an activation of a feature by a user triggers a chain of inter- and intra-object method invocations that carry out the requested feature collaboratively. While tracing of program execution is easily achievable using currently available tools, there exist several possibilities of making the tracing tool aware of activation and deactivation of features. The two approaches to date are based on encoding the feature activation in the form of either test cases (software reconnaissance [26]) or by requesting the user to manually start and stop the tracing tool for individual features (marked traces [27]). In our approach, we store the knowledge about activation of features in the source code of the program itself using meta-data, thus avoiding the extensive manual work and complexity behind creating feature-triggering test-case suites for legacy programs. Although our approach is user-driven, as it is for marked traces, we do not require users to start and stop the tracing process explicitly. This reduces the required user effort and makes the feature location process transparent.

One of the important repercussions of using a dynamic analysis technique, as opposed to other types of approaches, is its sensitivity to run-time-resolvable conditional branching and polymorphic method invocations. If the condition of multiplexing the flow of control is based on the concrete feature being executed (this is common practice in inheritance-based frameworks and in routines for handling keyboard shortcuts), a correct interpretation of polymorphic and conditional behaviors is necessary for accurate location of feature implementations. However, sensitivity to user input and run-time behaviors also means that additional care is required to explore all the execution paths of feature implementations.

Our procedure of feature location is as follows. When the knowledge of the features of a program is acquired in form of a feature specification, we can use it to prepare the source code for run-time tracing. To do so, we have to find and mark what we term *feature entry points* in the source code of a program. As visualized in Fig. 3, a feature entry point is a method, through which a thread of execution enters the implementation of a particular feature. This can for instance be the *actionPerformed()* method of an appropriate button in the graphical user interface. 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 implementation of a corresponding feature (marked in Fig. 3).

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 Fig. 3, 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 run-time by our execution-tracing tool, hereby allowing the tool to recognize a feature-membership of methods being executed. In our approach, a feature is allowed to have more than one feature entry point annotated in the code to cover cases where a feature is activated by more than one method in the code (e.g. when a feature consists of several independently activated use cases).

It is important to note that annotating only the feature entry points enables us to remove the proportionality between the actual size of feature implementations and the number of methods that need to be manually marked in the code. By abstracting away the details of the control flows inside of feature entry points, the number of manually marked entry points

depends solely on the number of a program's features, thus diminishing the amount of manual work involved in locating feature implementations in unfamiliar programs.

Activating the features in an instrumented version of a feature-entry-point-annotated program can be done either by creating and executing appropriate test cases, or by interacting with a program'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 the legacy system at hand and possibly modifications thereof to create appropriate feature-triggering test suites. 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 user-driven triggering of features is negligible, but this comes at a cost of lowered repeatability and a potential lack of exploration of all execution paths in the investigated program. Repeatability can be improved by using a GUI testing robot. GUI testing robots are, however, sensible to changes in the contents and layouts of the GUI. In our case studies, we opted for the user-driven triggering method due to its low manual workload and good applicability to unfamiliar source codes.

By applying our feature location technique, we obtain a set of *feature traces* that correlate a program's features with its source code. This information forms a basis for restructuring the source code towards improved representation of features. It needs to be mentioned that the only connection between the feature location and the code transformation is through the feature traces. Thus, in principle it is possible to use any of the potentially more sophisticated feature location techniques mentioned earlier together with our code transformation phase, given that the output of this mechanism can be converted to our feature-trace model, which we discuss in detail in Section 4.1.

3.3. Feature representation

The aim of feature representation is to represent implementations of features explicitly in the static structure of source code. This is done by creating a new package structure and populating it with unaltered classes of the original program according to their participation in implementing features.

As motivated in the discussion of the fragile decomposition problem, we decide to preserve the original definitions of classes without splitting them into increments in order to preserve their representation of human-understandable concepts from the original decomposition. However, we use this coarse granularity of separation at a cost of not being able to separate feature implementations interwoven at the level of classes, methods, or individual code statements. We leave any further finer-grained separation of classes and algorithms inside of methods up to the programmer. We perceive this decision adequate, as human expertise is necessary to decide how to split existing abstractions and algorithms in a meaningful fashion to meet the needs of a particular development scenario (bidirectional restructuring vs. refactoring to a software product-line vs. removal of a feature, etc.). This also leaves it to the programmers to choose the mechanism for advanced separation of concerns in the forward restructuring scenario. Note that our aim at preservation of original classes contrasts with some of the existing works in feature-oriented programming [53] and feature-oriented refactoring [28], which aim at completely splitting classes into feature-specific increments. In our approach, we automatically create a class-preserving decomposition, while leaving the decision about any further manual separation up to the programmers.

Our feature representation approach creates decompositions that consist of one single-feature package per feature and a number of multi-feature packages. Each *single-feature package* only contains classes that are unique to the implementation of that particular feature, whereas the *multi-feature packages* contain classes that are shared by multiple features and their dependent classes whose contribution to program features is not captured by feature traces. The distribution of classes among multi-feature packages is done in a way that minimizes delocalization and interleaving of feature implementations in terms of packages. By distinguishing between single-feature and multi-feature packages, we aim at creating a high-level structure of source code consisting of independent single-feature packages that build upon a common *core* [2] comprised of a number of multi-feature packages.

We use the standard package mechanism of Java to separate feature implementations. Java packages are sufficient for our granularity of feature separation and they possess the core properties desirable for a modularity mechanism—the ability to group classes, to separate the groups from each other in terms of namespaces, and to control access to members of the groups. Moreover, the package construct is a part of the standard Java language specification, which we perceive as an important factor for practicality of our approach.

There exist a number of useful properties of the described package structure. Firstly, single-feature packages provide a starting point for comprehending individual feature implementations in isolation from the rest of a program's functionality. Secondly, feature-specific modifications are contained to classes within the same package and have no effect on classes belonging to the implementations of other features. This corresponds to the *common closure principle,* which says that classes that change for the same reason should be grouped together [29]. In situations where the scope of comprehension, or modification, cannot be confined to single-feature packages, the number of multi-feature packages that have to be visited are kept to a minimum by defining multi-feature packages in a way that minimizes delocalization and interleaving of feature implementations. The relevant classes in multi-feature packages can be found by following the static dependencies of single-feature classes. The multi-feature packages are expected to follow the *common reuse principle* [29], i.e., to group classes that are reused together by multiple features, thus forming a common core of the system.

To aid programmers in identifying the implementations of individual features inside multi-feature packages and determine the levels of code sharing between them, we provide an IDE plug-in that marks source code of classes and

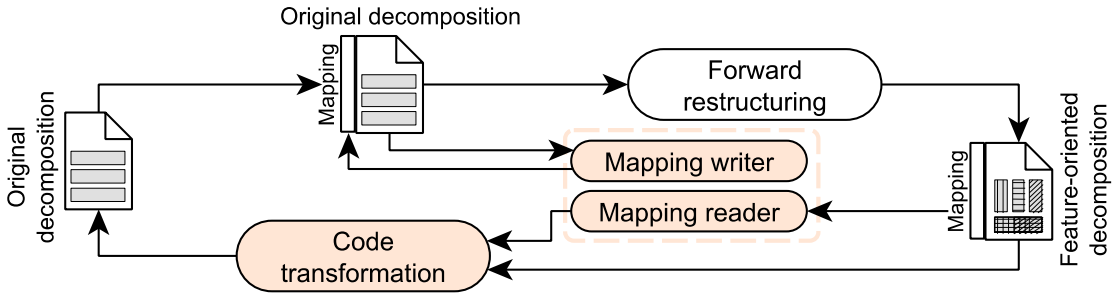

Fig. 4. Reverse restructuring.

methods with the identifiers of features. The explicit marking of single-feature methods helps to identify feature-specific parts of multi-feature classes, whereas marking of feature overlaps allows programmers to discover inter-feature relations more easily and assess the reuse level of core classes among implementations of features. The details of the IDE support for inspecting multi-feature packages are discussed in Section 4.2. By investigating the level and the nature of code sharing between features a programmer should be able to identify the reusable parts of feature implementations and to get an impression of a potential effort needed to modify a feature implementation in a class.

3.4. Reverse restructuring

The reverse restructuring aims at re-establishing the original decomposition of a program from its remodularized version. This is done by automatically re-creating the original package structure of a program and automatically re-assigning classes to their original packages. This is performed based on two inputs, as visualized in Fig. 4.

The first input required by the reverse restructuring is the source code of the feature-oriented remodularized program decomposition. This is obtained in the course of the forward restructuring, which consists of feature location and feature representation.

The second piece of information needed by the reverse restructuring is the mapping between classes of the feature-oriented remodularized program decomposition and the packages in the original program decomposition. Creation of this mapping is done by pre-processing the source code of the original program decomposition prior to forward restructuring. The information about the names of the original packages is extracted by the *mapping writer* and then stored as meta-data annotations on the declarations of individual classes in the source code. After the forward restructuring and any required modifications of source code are performed, the package-class mapping is read by the *mapping reader* and passed as one of the parameters to the *code transformation* routine. Based on the given mapping of classes to original packages and on the source code of the feature-oriented remodularized program decomposition, the original package structure is restored.

Since the presence of these annotations is idempotent to the forward restructuring, the process of creating class-package mappings as well as the reverse restructuring can be taken in and out from the remodularization process to realize either the forward- or the bidirectional restructuring usage scenarios—depending on the needs of the programmer.

In the bidirectional restructuring scenario, where the feature-oriented decomposition is introduced to support feature-wise modifications, we have to consider not only modifications to algorithms within methods but also modifications of the static structure of software. Accordingly, modifications to the remodularized program such as {changing the name, adding, removing} of {classes, methods, attributes} have to be handled properly by our reverse restructuring. This concern is addressed in our approach by making the package-class mappings evolve together with the source code. This is achieved by using per class-basis annotations parameterized with string-based names of their original packages to specify the package-class mappings. 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 changing the name of the class, or modifying its implementation.

The behavior of our reverse restructuring is, however, unspecified for the cases when a new class is created or is modified semantically (e.g. a business logic class refactored into a utility class) in the remodularized program. To correctly resolve such situations, we rely on programmers to decide to which package in the original decomposition such a class belongs and declare the decision in the source code by annotating the class with the appropriate package name. This policy is sufficient for the scope of our current investigations, yet it remains possible to replace it in the future by a more sophisticated one (e.g. clustering based on static cohesion and coupling, co-change, etc.)—the only required change would be implementing a new pair of *mapping writer* and *mapping reader* routines.

Last but not least, in our remodularization approach, we use the presented code transformation mechanism based on class-attached meta-data mappings only for the purpose of reverse transformation of code. The mechanism is, however, general enough to be used also for reproducing previously established feature decompositions. This would be done by collecting package-class mappings of the remodularized program and storing them in the memory space of the *mapping reader* routine, while writing a new mapping to the source code. Then, based on the original mapping read by *mapping reader* the code containing the new mapping would be transformed back to the original decomposition. By doing so, it is

```
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){ ... }

    });
  }
}
```

Fig. 5. Annotating feature entry points in code.

possible to repetitively switch between two decompositions, as the information about the previous structure of code would always be preserved as meta-data in the new decomposition. This procedure can be used to quickly reproduce the exact shapes of any previously established feature-oriented decomposition without the need for re-executing the multi-objective optimization process used to establish our feature-oriented decomposition.

4. Implementation

In this section, we describe the implementation-specific details of our approach. We start by explaining the localization of feature implementations and move on to the representation of features.

4.1. Feature location—FeatureTracer

We have implemented *FeatureTracer*—a method-granularity feature location library based on the proposed notion of feature entry points. A program that is subject to feature location is manually annotated with *@FeatureEntryPoint* annotations prior to using the library. Fig. 5 demonstrates how to annotate the action handler of a button-triggerable feature. In the provided example, the *actionPerformed* method of the anonymous subclass of *ActionListener* is the entry point of the feature *Feature1*. Consequently, each method called at run-time within the flow of control of the marked method is considered a part of the implementation of *Feature1*.

After all entry points of the program features are found and annotated, the program is automatically instrumented with FeatureTracer's tracing code. The inserted tracing code will transparently detect the execution of features and save the collected trace data in the form of feature-trace files upon the program's shutdown.

The tracing tool is implemented as an aspect using *AspectJ,* an aspect-oriented programming tool for Java [30]. We use AspectJ's load-time weaver to transparently instrument classes of a traced program at their load time. The essentials of our tracing aspect are shown in Fig. 6. The aspect is based on the *before* and *around* advices that match the *call* and *execution* join points for every method and constructor in a subject Java program. The aspect is configured to trace within the packages of a particular program by specializing the aspect and implementing the abstract pointcut *packages*—note that this can be done by using a standard *aop.xml* descriptor file of AspectJ's load-time weaver.

The presented aspect allows us to extract the signatures of executions (both methods and constructors), to identify the caller–callee pairs of execution (this is why we need both *call* and *execution* pointcuts), their enclosing classes, and the identities of concrete objects used at run-time. To determine the contribution of a currently executing method to the implementation of a program's features, FeatureTracer maintains internal stack-based representations of feature entry points entered by a program's threads. The collected data is updated upon entering and leaving every execution through the calls to the *enterExecution* and *leaveExecution* methods.

The per feature-trace data is stored in the form of *feature-trace models*. The definition of our feature-trace model is given in Fig. 7.

Feature-trace models capture the executions of methods and constructors that occurred at run-time in the context of their corresponding features. A method is placed in a feature trace only if it was executed at least once during program execution—an analogous rule applies to types. In the case of reentrancy or parallel executions of features, the new data are aggregated in an already existing feature model, so that at any point in time a single feature is represented by exactly one, consistent, model. Collected feature-trace models are automatically saved to files in a serialized form on termination of the traced program.

In designing FeatureTracer, some of the important design decisions were the policies for interpreting polymorphic invocations, inherited methods and abstract methods. We handle this issue by only capturing the classes that define the executed method bodies. Consequently, this implies that interfaces and some abstract classes are not registered in feature traces. The rationale for this decision was to better describe the concrete executions of features at run-time.

```
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);
    }
  }
}
```

Fig. 6. The tracing aspect.

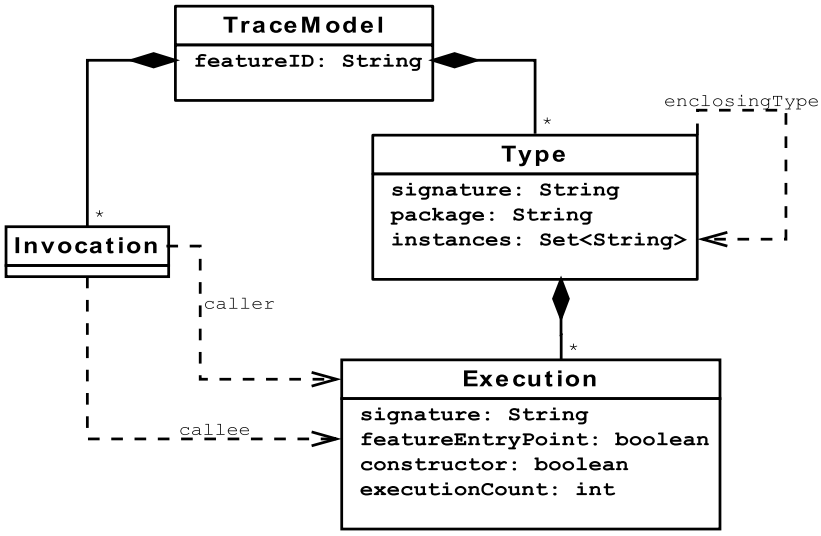

Fig. 7. Feature-trace model.

The last concern worth mentioning here is the memory usage of FeatureTracer. Memory usage is important, since the primary usage scenario of FeatureTracer is to trace a Java program while a user is interacting with it, imposing no constraints on how long this interaction should take. We avoid the proportionality relation between FeatureTracer's memory usage and the duration of subject program execution by collecting neither the information on the order of method invocations, nor of the timestamps of their invocation. The library increases its memory footprint only when the sets containing method-, class-, and object-identifiers are expanded, which happens only during execution of elements unique to these sets.

The FeatureTracer library is available as a part of our feature-centric analysis tool Featureous, which we introduced in [31].

4.2. Feature representation

This section discusses the process of creating explicit feature representations based on packages.

4.2.1. Remodularization of classes

The feature traces produced during feature location are used to automatically reallocate the classes of a program to achieve an explicit representation of its feature within its package structure.

Firstly, our remodularization tool analyzes feature traces to discover classes dedicated to single features. These classes are then assigned to packages named after their respective features. The physical modification of source code done here, as well as in the rest of the remodularization activities, is performed using the Spoon program processor library [32].

$$\boldsymbol{afd}(F,P) = \frac{\sum_{f \in F} |\{p \in P : f \rightsquigarrow p\}|}{|F|}$$

$$\boldsymbol{api}(F,P) = \frac{\sum_{p \in P} |\{f \in F : f \rightsquigarrow p\}|}{|P|}$$

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

$$\boldsymbol{tpcoup}(P) = \sum_{\substack{p \in P \\ T \Rightarrow p}} coup(p,T), \boldsymbol{where}: coup(p,T) = \sum_{\substack{t1 \in T \\ t1 \Rightarrow p}} \sum_{\substack{t2 \in T \\ t2 \not\Rightarrow p}} \left| DD_{t1,t2} \cup DM_{t1,t2} \right|$$

Fig. 8. Multi-objective formulation of the package assignment problem.

Secondly, the remaining classes are reallocated to a number of multi-feature packages, based on their participation in implementing individual features and their static dependencies to other classes. The goal of the reallocation algorithm is to obtain a set of multi-feature packages that minimize feature tangling and scattering, while maximizing package cohesion and minimizing package coupling. Using static dependencies helps to find appropriate packages for the remaining classes that are not covered in any feature trace, as they will be placed in packages with which they are most strongly related. To optimize both for feature-wise and dependency-wise assignment of classes to multi-feature packages we define this task as a multi-objective optimization problem.

The criteria that we use as optimization objectives for assignment of classes to multi-feature packages are: average number of packages that features are delocalized over, average number of features that are interwoven in packages, average static cohesion of packages, and total coupling of packages. The intended aim of these criteria is to localize feature implementations and separate them from each other, while keeping the resulting packages cohesive and loosely coupled. The concrete measures that we use for evaluating these properties are shown in Fig. 8.

The average feature delocalization (*afd*) measures the average number of packages p that contribute to implementing program features f (i.e. packages that fulfill the $\rightsquigarrow$ ("*implemented by*") relation with features). The value of this measure needs to be minimized in order to reduce delocalization of feature implementations.

The measure for average package interleaving (*api*) is analogous to *afd,* with the only difference that it focuses on the number of features f interwoven in terms of packages p. Our approach minimizes the value of *api* in order to reduce interleaving of features in terms of packages.

The definitions of package cohesion and coupling are based on cohesion and coupling measures proposed in [33,34]. For our formulation, we use the notions of *interactions between data declarations* (*DD-interactions*) and *interactions between data declarations and methods* (*DM-interactions*) that were also introduced in [33], and the $\Rightarrow$ operator for specifying containment relations between types and packages. Our formulation of package coupling corresponds to a sum of the *ACAIC*, *OCAIC*, *ACMIC*, and *OCMIC* measures defined in [34], whereas cohesion is the package-level version of the *RCI* metric proposed in [33]. In order to use these two package-scope metrics in assessment of class-package assignments we compute mean cohesion (due to the *normalization* property of this measure—see [35] for an extensive discussion) and summary coupling of created packages. It can be shown that our definitions fulfill all the properties for, respectively, cohesion and coupling that were proposed in the software measurement framework [35].

For solving the described optimization problem our approach uses a Pareto-front-based genetic algorithm. The fundamental principles of our implementation of this technique are analogous to the ones proposed in [36]. In our formulation, each chromosome in a population being evolved represents a concrete distribution of classes among packages. Classes are represented as genes in the chromosomes, whereas their assignment to packages is represented by the discrete values of genes. Evaluation of the quality of the solutions is done by computing the four metrics discussed earlier. Based on the four computed values, chromosomes can be compared against each other in terms of a Pareto-dominance relation, which states that one out of two chromosomes is only better off than the other chromosome, if all values of its four metrics are not worse, and at least one of the values is better. Computation of concrete values is done by measuring them on manipulated feature traces, and a simple static dependency model that we extract from the source code of a subject program. The genetic algorithm is initialized with a randomized population, into which we insert a number of chromosomes that correspond to the original package structure of target classes to make the original decomposition a starting point for the algorithm. A Pareto front obtained in the course of iterative optimization is then being searched for best values of *afd* and *api* in order to choose a new structure of core packages. This structure is then physically established in the source code.

4.2.2. Inspecting feature representations

The grouping of single-feature classes into single-feature packages provides programmers with an explicit correspondence between a feature and its specific classes, whereas grouping of multi-feature classes into multi-feature packages does not provide programmers with a similar correspondence. Even though programmers know that classes in multi-feature packages are not dedicated to implementations of single features, it is difficult for them to discover how many feature implementations a given class participates in. Similarly, it is not clear how many feature implementations the

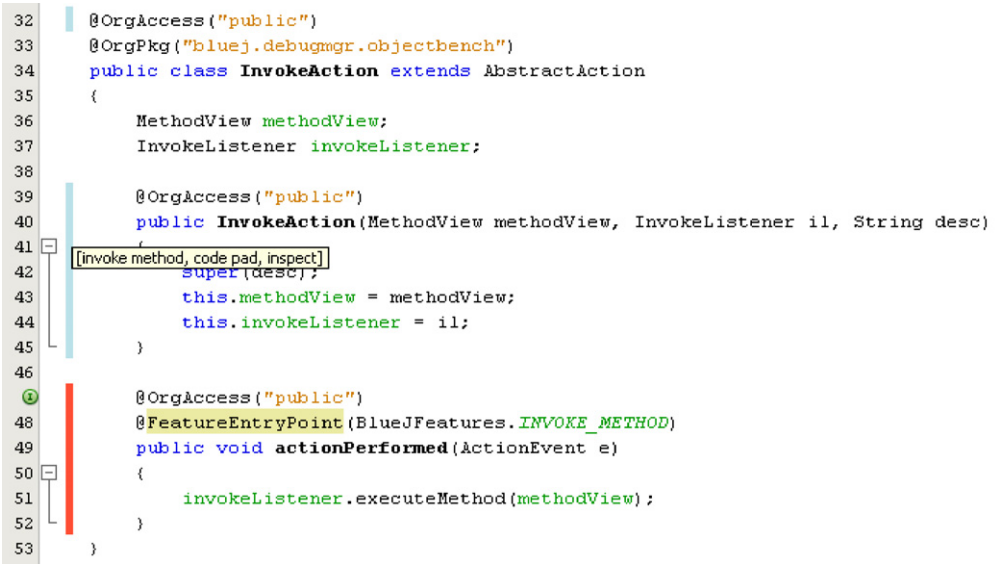

Fig. 9. Code-editor coloring based on feature traces.

methods of multi-feature classes participate in. Some methods in a class may just participate in the implementation of a single feature, while others participated in many, thereby making their respective class a multi-feature class. We address this issue by providing a visualization plug-in to the source-code editor in the NetBeans IDE. An example usage of our visualization is illustrated in Fig. 9.

Based on the collected feature traces, our plug-in visualizes the contribution of methods to implementation of program features. We use color bars next to the code editor to indicate participation of a method in implementing a feature. The single-feature methods are marked in red, whereas multi-feature ones are marked in blue. Furthermore, the shades of the multi-feature blue bars qualitatively indicate the number of features sharing a method—the more features using a method, the darker the shade. The information about the concrete features can be displayed in the form of a tooltip by pointing at the color bars with the mouse cursor. In the overall idea of annotating source code with feature-specific information, our visualization bears similarities with Spotlight [37] and CIDE [38]. However, we put the primary emphasis on the level of sharing of feature implementations (indicated by colors of the bars) rather than on identifying the concrete features that use the code (displayed on request as a tooltip). The visualization plug-in is available as a part of the Featureous tool [31].

4.2.3. Access control at package boundaries

As our feature presentation technique alters the existing package structure of a program, it also invalidates the original criteria for assigning access control at the boundaries of packages. Hence, we have to redefine access control for the newly created packages to reflect the criteria of the new decomposition. The two particular concerns addressed with respect to access modifiers are compilability of the resulting programs, and usage of access restrictions for improving encapsulation of feature implementations.

First of all, we ensure compilability of the program by handling the *default* and *protected* scope modifiers. If they are present in the original program, they constrain the remodularization process, since too restrictive usage of them can cause compilation errors in remodularized programs. This occurs when, for instance, a class in the original program 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 together with it, or unless its access restrictions will be relaxed. We deal with this issue by replacing *default* and *protected* access declarations with the *public* modifier, which removes the access restrictions.

After any required relaxation of access restrictions, it is important to consider the opposite process—using access modifiers to improve encapsulation of feature implementations in code. We do this 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 feature implementations (e.g. using access modifiers proposed in [39]) would have been possible if compatibility with Java language specification were not a concern to our approach.

4.3. Reverse restructuring

The reverse restructuring recovers the original package structure of a remodularized program using three routines implemented using the Spoon pre-processor library [32]. As introduced in Section 3.4 these routines are: code transformation that moves classes between packages according to a given mapping data, and a writer–reader pair of

$$\boldsymbol{fsca}(\boldsymbol{F}) = \frac{\sum_{f \in F} sca(f)}{|F| \cdot |P|}, fsca \in [0,1), \boldsymbol{where}: sca(f) = |\{p \in P : f \rightsquigarrow p\}| - 1$$

$$\boldsymbol{ftang}(\boldsymbol{P}) = \frac{\sum_{p \in P} tang(p)}{|F| \cdot |P|}, ftang \in [0,1), \boldsymbol{where}: tang(p) = |\{f \in F : f \rightsquigarrow p\}| - 1$$

Fig. 10. Measures for quality of feature representation in source code.

mapping–handling routines that store and retrieve the package-class mappings from the source code of a remodularized program. The reverse restructuring as a whole will work correctly and allow for bidirectional restructuring of programs, as long the input source code can compile and as long it contains appropriate meta-data.

The information about the original location of classes in packages is collected by iterating over classes in the Spoon model of source code. Each class is annotated with the *@OrgPkg* Java annotation parameterized by the string-based qualified name of the class' package in the original decomposition of the program. This meta-data present directly in the source code of the program throughout the possible modification made to the remodularized program is used to relocate the classes back to their original packages, when and if the original decomposition is to be recovered. As we discussed previously, in the case of addition of new classes during modifications of the remodularized programs or significant re-assignments of responsibilities among classes, it is up to programmers to annotate these classes with correctly parameterized *@OrgPkg* annotations, so that they become assigned to appropriate packages when the original decomposition is recovered.

Due to the manipulation of the access modifiers in the code done during the feature representation step, we need to include a mechanism for recovering the original access restrictions in the code. We do this by capturing the original access-modifier information, apart from the package-class mapping, 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 program by the same routine that places the *@OrgPkg* annotations. It is parameterized by a string value that corresponds to the declarations 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 programmer to decide on their visibility in the original decomposition by placing appropriately parameterized annotations.

5. Case study

We have conducted two case studies in order to evaluate our remodularization approach in terms of required manual effort and the quality of the obtained feature-oriented decompositions.

For evaluating the manual effort required by our approach, we register the time spent on performing manual activities with relation to sizes of the investigated programs and number of their features.

The quality of the feature-oriented decompositions that our approach produces we evaluate in terms of two feature-oriented metrics defined in [7]. The exact definitions of the feature scattering among packages *fsca* and the feature tangling in terms of packages *ftang* that we use are shown in Fig. 10. The measures are defined over the set of a program's features *F*, and the set of its packages *P*. The used definitions ensure that the measured values will not be influenced by the total number of packages in a program, but only by packages that actually contribute to implementing features [7].

Since the presented measures directly correspond to the extent of delocalization (represented by scattering) and interleaving (represented by tangling) of features in terms of packages, they provide us with quantitative criteria for assessing the impact of the remodularization process on feature-wise comprehensibility of source code. Therefore, our hypothesis in these case studies is that our remodularization approach successfully reduces scattering and tangling of feature implementations in terms of packages.

As subjects for the case studies, we chose the two programs, BlueJ and JHotDraw SVG.

BlueJ [40] is an open-source interactive programming environment created to help students learn the basics of object-oriented programming in Java. Since BlueJ is a non-trivial application, whose operation involves compilation, dynamic loading, execution, and debugging of Java code, we were particularly interested in testing our approach on it. The case study used BlueJ version 2.5.2.

JHotDraw [41] is an open-source graphical framework created as an example of a well-designed object-oriented application. Its wide usage in software case studies and its claimed property of being a well-designed framework was what motivated us to include it in our case study. We focus our investigations on the example application *SVG*, which is based on the framework and distributed together with it. The version used here is 7.2.

5.1. Results

5.1.1. Manual effort

During the manual part of the remodularization process, we needed to identify features, annotate feature entry points, and activate features from the GUI for both programs.

Table 1
Summary of manual effort.

	BlueJ	JHotDraw SVG
Program size	78 KLOC	62 KLOC
Number of identified use cases	127	90
Number of identified features	41	31
Total time of identifying features and use cases	2 h	1 h
Number of activated features	39	29
Number of feature entry points	228	91
Total time of annotating and activating features	6 h	4 h

Table 2
Feature-characterization of the programs' top-level types.

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 top-level types	535	493

In the case of BlueJ we were able to infer almost all program features from the available user documentation, since it is arranged into a list of usage scenarios. This way we have identified 127 use cases, which we have then grouped into 41 coherent groups representing features. For JHotDraw SVG, we found no documentation that allowed us to identify its features. Therefore, we have relied solely on the contents of the program's menus, contextual menus, and toolbars. In the course of inspecting run-time behavior of JHotDraw SVG we have identified 90 use cases, which we have grouped into 31 coherent features. The recovery of use cases and grouping was performed according to the guidelines given in Section 3.1.

After identifying the features of both subject programs, we investigated their respective source codes in order to annotate feature entry points. Our effort was focused on annotating the GUI layers of both programs. Most often, we placed annotations on "*actionPerformed*" methods of a corresponding *action listener* that handles events produced by GUI elements that allow for activating features (menus, buttons, etc.). In the case of BlueJ, we, furthermore, annotated most of the use cases at the level of the program's domain model. These annotations ensure that even if multiple ways for activating a feature exist in the program's GUI (e.g. using mouse vs. using a keyboard shortcut), execution of features will be detected when the flow of execution reaches the domain model. Annotation of the domain model was, however, difficult to perform consistently in the case of JHotDraw, and therefore we have focused on only annotating the event handlers in its GUI classes.

By instrumenting the annotated programs with the FeatureTracer's tracing tool and by using their GUIs to execute their use cases, we have collected two sets of feature traces. During this process, we did not observe any significant execution performance overhead due to tracing.

A summary of the performed manual work is shown in Table 1. The table depicts the effort required to enumerate the programs' functionality, and to place the feature-entry-point annotations on appropriate methods. It is worth mentioning that we did not know the source codes, or the architectures of the two programs beforehand.

5.1.2. Feature-oriented decomposition

Before applying our remodularization approach to BlueJ and JHotDraw SVG, we calculated a summary distribution of top-level types (as opposed to nested types, and inner types that are supported by the Java language) among several categories of relations that a class can be in with respect to features. This feature-wise characterization is shown in Table 2.

We distinguish four characteristic ways of how a class can participate in implementing features: single-feature classes are classes that participate only in a single feature; multi-feature classes participate in implementing multiple features; non-covered classes are classes that exist in a program's static structure, but were not used at run-time by the execution features. Furthermore, we have listed interfaces, as they are not included in feature traces, as discussed earlier, and thereby contribute to non-covered types. Finally, we detected a number of types that belong to the "dead code" category. By "dead code", we mean classes that are not statically referenced from the programs' main codebases, even though these types are present physically. For BlueJ the majority of these types implements parsing of code tokens of programming languages other than Java, and are located in package *org.syntax.jedit.tokenmarker*. In the case of JHotDraw SVG "dead code" classes are the framework's classes not used by the investigated JHotDraw SVG application, but for some reason not excluded from SVG's package. For the rest of our analysis, we have decided to remove the "dead code" types from the codebases of the investigated programs.

After obtaining the feature-wise characterizations of the two programs, we measured their original decompositions with respect to quality of their feature representation. For this, we have used the *fsca* and *ftang* metrics introduced earlier.

Table 3
Summary impact on feature representation's quality.

	BlueJ			JHotDraw SVG		
	Pre	Post	Δ	Pre	Post	Δ
fsca	0.247	0.189	−23%	0.280	0.199	−29%
ftang	0.309	0.192	−38%	0.374	0.190	−49%

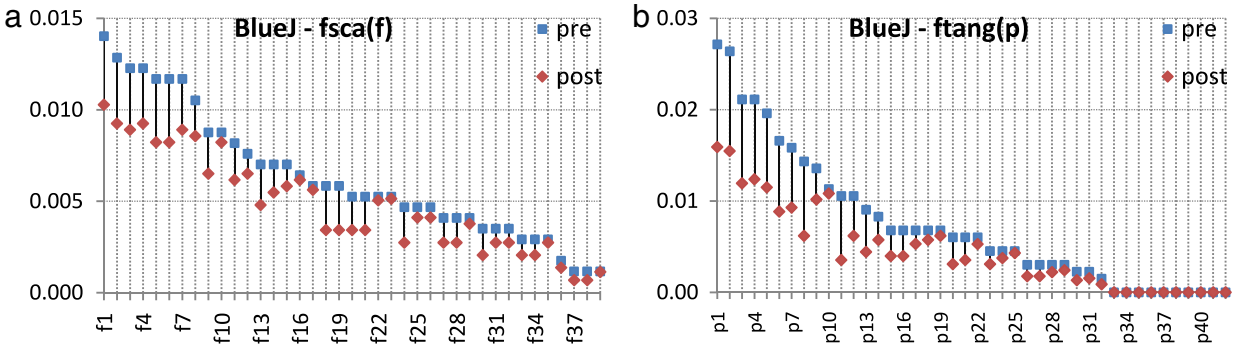

Fig. 11. Scattering values per feature (a) and tangling values per package (b) in BlueJ.

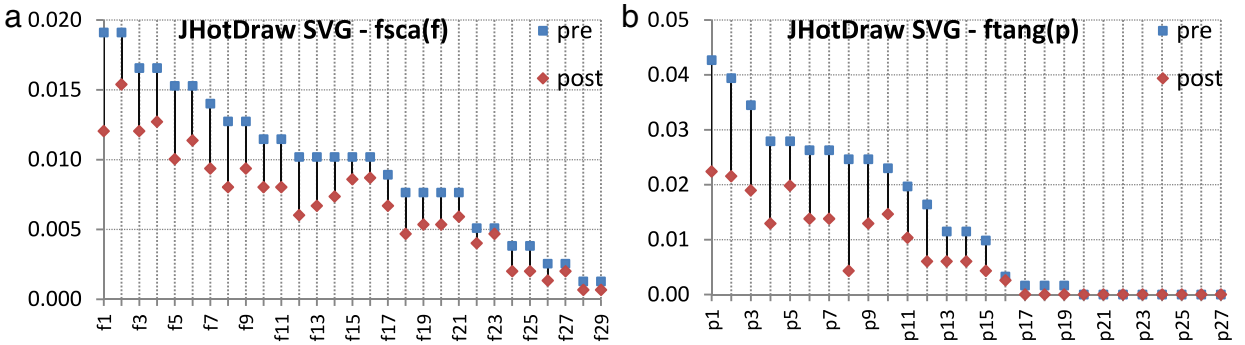

Fig. 12. Scattering values per feature (a) and tangling values per package (b) in JHotDraw SVG.

Analogously, we did similar measurements after applying our remodularization approach. A comparative summary of the obtained results is shown in Table 3.

The details of the impact of the remodularization process on the distribution of scattering values among individual features, as well as the distribution of tangling values among packages are shown in Figs. 11 and 12. Scattering of feature implementations is presented on a feature-basis, allowing for direct comparison of values between concrete features in the original and the remodularized version of the programs. For the distributions of tangling, we have excluded single-feature packages from the tangling charts, as their tangling values are equal to zero, and they cannot be compared across program versions as these packages are a product of the remodularization process and therefore absent in the original decomposition. For presentation purposes, all the *pre*- and *post*-value pairs in displayed data series are sorted with respect to their *pre*-values and the differences are marked with a high–low line and the names of features and packages are represented by short identifiers. A complete list of features and packages represented by the used identifiers is included in Table 4.

5.2. Discussion

5.2.1. Manual effort

Our experience is that the amount of work required by our approach was relatively low. The effort we measure in terms of time spent on: identifying features and use cases, annotating feature entry points in source code and activating the features at run-time. In total, we needed 13 h in total to finalize these tasks for both programs, which we consider low taken into account that we did not know the source code or the designs of the subject programs beforehand. Unfortunately, we are not able to directly compare this result to the other existing approaches due to a lack of quantitative data on their required manual effort.

Qualitatively, we have found the manual work involved not to be complex, and not to require extensive knowledge of the source code. We experienced that in order for a feature location's results to be comparable between the two programs,

Table 4
Identifiers of features and packages in BlueJ and JHotDraw SVG.

BlueJ				JHotDraw SVG			
Features		Packages		Features		Packages	
Id	Name	Id	Name	Id	Name	Id	Name
f1	Project persistence	p1	bluej	f1	Basic editing	p1	org.jhotdraw.draw
f2	Create test method	p2	bluej.pkgmgr	f2	Selection tool	p2	org.jhotdraw.samples.svg.figures
f3	Manage projects	p3	bluej.utility	f3	Drawing persistence	p3	org.jhotdraw.util
f4	Code pad	p4	bluej.pkgmgr.target	f4	Manage drawings	p4	org.jhotdraw.samples.svg.gui
f5	Invoke method	p5	bluej.prefmgr	f5	Path editing	p5	org.jhotdraw.gui
f6	Compilation	p6	bluej.pkgmgr.actions	f6	Text area tool	p6	org.jhotdraw.beans
f7	Team	p7	bluej.editor.moe	f7	Grouping	p7	org.jhotdraw.gui.plaf.palette
f8	Testfixtures	p8	bluej.graph	f8	Font palette	p8	org.jhotdraw.samples.svg
f9	Application close	p9	bluej.debugger.jdi	f9	Attribute editing	p9	org.jhotdraw.draw.action
f10	Export	p10	bluej.extmgr	f10	View palette	p10	org.jhotdraw.geom
f11	Debugger control	p11	bluej.debugmgr.objectbench	f11	Align palette	p11	org.jhotdraw.undo
f12	Edit class implementation.persistence	p12	bluej.pkgmgr.target.role	f12	Figure palette	p12	org.jhotdraw.text
f13	Startup	p13	bluej.debugmgr	f13	Fill palette	p13	org.jhotdraw.app
f14	Inspect	p14	bluej.parser	f14	Stroke palette	p14	org.jhotdraw.app.action
f15	Inheritance arrow management	p15	bluej.pkgmgr.dependency	f15	Automatic selection	p15	org.jhotdraw.samples.svg.action
f16	Package management	p16	bluej.terminal	f16	Canvas	p16	org.jhotdraw.io
f17	Add class from file	p17	bluej.classmgr	f17	View source	p17	org.jhotdraw.util.prefs
f18	Edit class implementation.code editing	p18	bluej.pkgmgr.graphPainter	f18	Link palette	p18	org.jhotdraw.gui.fontchooser
f19	Insert method into code	p19	bluej.debugmgr.texteval	f19	Undo redo	p19	org.jhotdraw.xml.css
f20	Preferences	p20	bluej.extensions.event	f20	Line tool	p20	org.apache.batik.ext.awt
f21	Edit class implementation.basic editing	p21	bluej.parser.symtab	f21	Export	p21	org.jhotdraw.xml
f22	Class management	p22	bluej.views	f22	Scribble tool	p22	org.apache.batik.ext.awt.image
f23	Import	p23	bluej.groupwork.ui	f23	Arrange	p23	org.jhotdraw.gui.event
f24	Generate project documentation	p24	bluej.debugger	f24	Text tool	p24	org.jhotdraw.samples.svg.io
f25	Print project	p25	bluej.editor	f25	Image tool	p25	net.n3.nanoxml
f26	Class instance management	p26	bluej.testmgr	f26	Ellipse tool	p26	org.jhotdraw.gui.datatransfer
f27	Debugger.display state	p27	bluej.testmgr.record	f27	Rectangle tool	p27	org.jhotdraw.gui.plaf
f28	Package description	p28	bluej.utility.filefilter	f28	Tool palette		
f29	Create objects from lib classes	p29	bluej.compiler	f29	Application startup		
f30	Open class documentation	p30	bluej.debugger.gentype				
f31	Run tests	p31	bluej.groupwork.actions				
f32	Open class implementation	p32	bluej.debugmgr.inspector				
f33	Terminal	p33	bluej.extensions.editor				
f34	Breakpoint management	p34	bluej.groupwork.cvsnb				
f35	Uses arrow management	p35	bluej.groupwork.svn				
f36	Key bindings management	p36	bluej.parser.ast.gen				
f37	Machine icon	p37	org.syntax.jedit				
f38	Program statics display	p38	bluej.extensions				
f39	Program dynamics display	p39	bluej.parser.ast				
		p40	bluej.groupwork				
		p41	bluej.runtime				
		p42	org.syntax.jedit.tokenmarker				

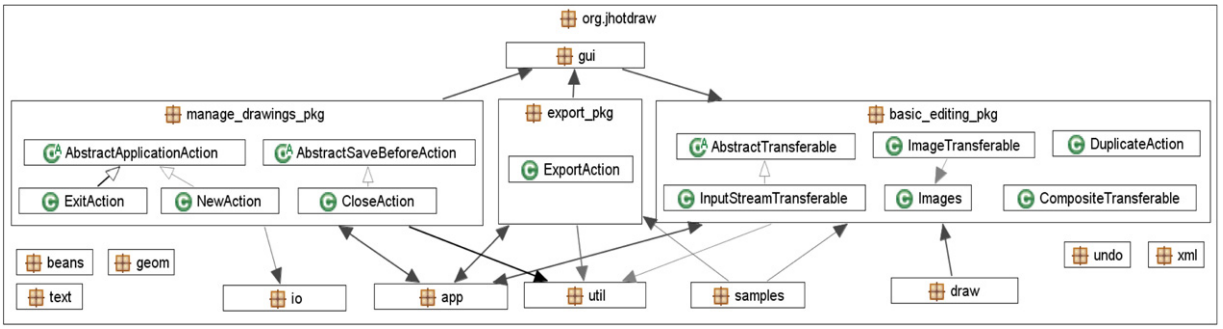

Fig. 13. Relations between three single-feature packages and multi-feature packages in JHotDraw SVG.

a consistent set of guidelines is necessary on how to recover features from requirements, annotate their entry points in code, and activate them at run-time. With respect to the first two activities, we have found the guidelines proposed in this paper adequate. The third aspect, the activation process, we have structured according to the lists of identified use cases—we have used them as checklists for activating features at run-time. In our case studies, we have performed feature activation manually, but, as mentioned earlier, this process could be automated by using a GUI testing robot for interacting with the graphical user interfaces of programs.

5.2.2. Feature-oriented decomposition

From the general characterizations of both programs in terms of features presented in Table 2, it can be seen that in both cases single-feature classes are only about 10% of all top-level types. This indicates a relatively low potential of both programs for achieving a complete separation of features without a major refactoring of source code. The registered number of classes not covered by the run-time tracing suggests the existence of alternative feature-execution paths that were not explored in the course of dynamic analysis. Our approach, however, compensates for incomplete coverage of execution paths by including the notions of cohesion and coupling among the remodularization objectives. As discussed earlier, usage of these criteria promotes a placement of the non-covered classes together with feature classes that statically reference them the most.

Comparison between the original programs and their remodularized versions reveals that our approach has reduced the total tangling and scattering of feature implementations. For BlueJ, scattering of feature implementations was reduced by 23%, whereas in the case of JHotDraw SVG it was reduced by 29%. Total tangling of feature implementations in terms of program packages was reduced by 38% for BlueJ and by 49% for JHotDraw SVG. These results indicate that our remodularization approach has significantly improved the overall representation of features in the two programs. The concrete value distributions of feature scattering and feature tangling reveal improvement, or at least preservation, of these properties for each feature and package in both the programs.

A closer inspection of the remodularized programs reveals a number of interesting details about the nature of the created feature-oriented decompositions. Fig. 13 shows three representative cases of single-feature packages in the feature-oriented decomposition of JHotDraw SVG: *export_pkg, manage_drawings_pkg, basic_editing_pkg*. The contents of the single-feature packages and their static dependency relations with the remaining multi-feature packages are indicated by arrows. Shades of the arrows depict the strength of the dependencies.

The feature *export*, whose task is to export created drawings to various formats, has only one single-feature class in its package *export_pkg*. 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 tells us that it would be a non-trivial task to modify one of these features in isolation from another (bidirectional restructuring scenario) or completely separating the features into independent increments of functionality (forward restructuring scenario; e.g. when refactoring towards a software product-line).

The second feature shown in Fig. 13, *manage drawings,* is interesting because of the relations between the single-feature classes included in the *manage_drawings_pkg* package. Not only for this feature, but also for many others in the remodularized version of JHotDraw SVG, we have observed that many single-feature packages enclose both the base classes defined by the JHotDraw framework, and their concrete subclasses created as part of the SVG application of the framework. This arrangement of base classes and subclasses eases feature-wise comprehension of framework-based applications, as it allows developers to easily discover the framework's classes that take part in implementations of individual features. Since frameworks often make use of the dependency-inversion principle [18] and polymorphism, we argue that the grouping of classes established by our approach makes it easier to comprehend how the source code of a framework realizes the user-identifiable functionality of an application built on top of the framework.

The last feature depicted in Fig. 13 is called *basic editing* and is concerned with the general editing that can be performed on every figure on the canvas; the use cases encompassed by this feature are {*copy*, *cut, paste, delete, duplicate*} *figure.* Taken into account only the semantics of the feature, it is not obvious whether the classes such as *AbstractTransferable, Images*

or *InputStreamTransferable* should be associated with that feature. This shows the importance of establishing localized representations of features in the source code in an automated manner. Relevance of conceptually unrelated classes is difficult to discover in an unfamiliar source code during manual inspection.

With respect to the fragile decomposition problem, in the case of remodularizing a framework application such as JHotDraw SVG, as opposed to a standalone application like BlueJ, it is advisable to use reverse restructuring to recover the program's original decomposition after successfully performing any modification tasks. This is to be done to preserve the compliance with the original framework's API, and thus makes it possible to upgrade SVG's code to future revisions of the JHotDraw framework, or to commit the adjustments made to the framework itself to its SVN repository. In contrast, the established feature-oriented decomposition of the standalone application BlueJ can be used as a basis for further development, as well as a starting point for further manual refactoring and separation of feature implementations at the level of the program's methods to create a software product-line.

5.2.3. Threats to validity

We see the following threats to validity of our studies.

Construct validity. In our study, we aimed at reducing the program-comprehension phenomena of delocalized plans and interleaving of feature implementations in packages. This was done by associating these phenomena with the notions of scattering and tangling, which it was possible to measure using code metrics. Thus, the validity of the study construct depends on the validity of the metrics used. The validity of the conclusions on the impact on program comprehension depends on the proposed associations between delocalization and scattering, and interleaving and tangling.

Internal validity. We perceive two threats to validity of our conclusions that we draw from the collected results. Firstly, there exists a certain degree of arbitrariness in partitioning of software requirements into feature specifications. In this paper, we state our criteria for performing this and we follow them in the case studies. Yet, choosing a different set of criteria is likely to produce a different set of feature specifications and thus to affect the pre- and post-remodularization results. A difference in the nature of recognized features could also be present across the case study programs due to different criteria for formulating use cases. This is because in one of the programs (BlueJ) the use cases were given by the program authors in the program's documentation, whereas in the second subject program (JHotDraw SVG), use cases were recovered manually from the program's user interface. Secondly, the correctness of the feature location phase, and therefore of the feature representation phase, depends on the correctness of manual annotation of feature entry points. The difference in annotating BlueJ (both in GUI action handlers and at the level of domain model classes) and JHotDraw SVG (only in GUI action handlers) could bias the feature location results achieved for one of the programs.

External validity. The small sample of subject programs in our study was enough to demonstrate feasibility of the approach, but is insufficient to generalize the results to other types of applications or other object-oriented languages. The particular type of applications used in the studies was a user-driven GUI-intensive Java application, therefore no basis is provided for reasoning about applicability of our approach to e.g. embedded systems, source code compilers, database engines, etc. A substantially larger and more diverse array of subjects would be needed to confirm or refute such generalizations.

6. Related work

This paper extends our earlier work presented in [42]. In the current work, we refine the notion of feature packages to single-feature and multi-feature packages, formulate the task of assigning classes to packages as a multi-objective optimization problem, define the notion of fragile decomposition, introduce a reverse restructuring of source code, and place feature-oriented remodularization in terms of a software development process through forward restructuring and bidirectional restructuring scenarios.

The approach that relates the most to ours is the approach of Mehta and Heineman described in [43]. The authors present a method for locating and refactoring features into fine-grained reusable components. Feature location is done by test-driven gathering of execution traces. Feature implementations are then manually analyzed, and manually refactored into components that follow a proposed component model. The aim of the approach is to improve reusability of the feature implementations. The main differences between our remodularization approach and the approach presented in [43] is the amount of required manual work (we do not require creation of dedicated test suites and automate the feature representation phase) and the presence of a reverse restructuring. Despite these differences, we reckon that in principle the two approaches could be combined by using the feature-oriented decompositions created by our approach as a starting point for the manual refactoring towards the component model of Mehta and Heineman.

Liu et al. [28] proposed the *feature-oriented refactoring* (*FOR*) approach to restructuring legacy programs to feature-oriented decompositions. The aim of FOR is to completely separate feature implementations in the source code, in order to use them in a software product-line. This is done by means of *base modules*, which contain classes and introductions, and *derivative modules*, which contain manually implemented method refinements. The approach encompasses a firm algebraic foundation, a tool, and a refactoring methodology. A case study involving a 2KLOC program with seven features is presented. The first difference between FOR and our approach is that FOR aims at complete separation of feature implementations in terms of bases and derivatives, whereas our approach does not divide legacy classes into increments and only separates single-feature classes from multi-feature ones in terms of Java packages. Secondly, feature location in [28] is performed by

means of tool-assisted manual labeling of source code, which requires major effort for larger programs. Finally, unlike in our approach, it is not the aim of FOR to provide a reverse restructuring of source code for recovering the original decompositions of legacy programs.

Our approach follows the idea of *on-demand remodularization* proposed by Ossher and Tarr [9]. Our discussions about the tradeoffs between the object-oriented and feature-oriented decompositions are based on their observations that software 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 the future concerns that will have to be modularized. To address this, the authors propose that it should be possible to remodularize software on-demand, accordingly to the needs that arise throughout software's evolution process. Our approach, where we focus on transitions between object-oriented and feature-oriented decompositions, can be seen as a specific instantiation of the general concept of on-demand remodularization.

A different approach to address this problem known as the tyranny of the dominant decomposition was proposed by Janzen and De Volder [21]. Using their notion of *effective views* they aim at reducing the problem of crosscutting concerns by providing editable textual views on two alternative decompositions of a single program. Fluid alternation and synchronization of the views allow programmers 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 enhanced to support it. In Decal it is possible to view both the alternative decompositions simultaneously, whereas in our approach code transformation has to be performed for each transition. We reckon 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 [21] is not applicable to legacy software and it appears that equipping it with such a mode of operation would be difficult to achieve.

6.1. Feature location

Feature location based on feature entry points can be perceived as a middle ground between two existing feature location approaches: manual inspection of source code [55,38] and *marked traces* [27].

A static approach to feature location based on manual inspection of program's dependence graph was proposed by Chen and Rajlich [55]. Their approach relies on an expert's judgment about the relevance of individual units of source code to program features. Discovery of complete feature-code mappings is thus performed gradually by following a proposed computer-aided search process. In order to declare the discovered feature-code traceability links, manual marking of source code with feature-information can be used. To facilitate this process, Kästner et al. [38] proposed their tool CIDE that allows for fine-grained manual marking of source code using colors, directly in IDE code editor.

Salah and Mancoridis [27] proposed a dynamic feature location technique that uses run-time execution tracing to reduce the need for manual inspection of source code. Their approach, called *marked traces,* requires program users to recognize and selectively enable run-time tracing to capture appropriate feature-exhibiting program behaviors in a running program. This also reduces the manual work required by the earlier dynamic approaches of Wilde and Scully called *software reconnaissance* [26] and of Wong et al. called *execution slices* [46], where dedicated test suites were used for triggering features. Moreover, the approach of Salah and Mancoridis classifies all the run-time events captured for individual features as relevant to their implementations—whereas the two earlier approaches only looked for feature-specific parts of feature implementations and thus required additional filtering of results' relevance, as postulated by Eisenberg and De Volder [45] and Koschke and Quante [50].

In our approach, we require manual marking of the feature entry points directly in the source code of the program, whereas the remaining boundaries of features are discovered by means of dynamic analysis. With respect to annotating source code with feature entry points, our approach follows the principles of manual inspection and marking [55,38]; the difference is, however, that we require marking of only entry point methods and not whole feature implementations.

To discover the remaining code contributing to program features we use user-driven dynamic analysis. However, in contrast to Salah and Mancoridis [27], our approach is transparent to the user—i.e. we do not require users to explicitly start and stop the tracing process to capture individual features. By using user-driven triggering, we avoid creating extensive test suites for unfamiliar legacy programs—designing and implementing of which is non-trivial, especially for GUI classes and for improperly exposed feature implementations. Despite not using test suites, our approach still has a potential for automation—as shown by Greevy and Ducasse through their TraceScraper tool [44], user-driven triggering of features can be automated by capturing it in the form of GUI-automation scripts.

Apart from the mentioned purely static and purely dynamic feature location approaches, there exist a number of hybrid approaches that aim at combining different techniques and information sources. Examples of such approaches include enhancing dynamic analysis with concept analysis and static analysis [48], information retrieval and prune dependency analysis [49], web mining techniques [52], change sequences from code repositories and issue tracking tickets [51]. The comparative analyses of multiple techniques and information sources (including execution traces and source code) performed by Ratanotayanon et al. [51] and Revelle et al. [52] revealed that there exist tradeoffs between existing approaches in terms of the properties of precision and recall. Hence, the choice of a particular feature location technique should depend on the tolerance for false-negative and false-positive location results.

6.2. Feature representation

In our approach, we do not divide legacy classes into increments due to comprehension concerns and we settle with Java packages as the modularity mechanism for establishing feature-oriented decompositions. Nevertheless, a number of approaches exist that untangle feature implementations by dividing classes and methods into fine-grained increments.

Murphy et al. [15] explored tradeoffs between three policies of splitting tangled feature implementations: a lightweight class-based mechanism, *AspectJ*, and *Hyper/J*. By manually separating a set of independent features at different levels of granularity, the authors confirm the limited potential for tangling reduction of the lightweight approach. In the case of AspectJ and Hyper/J, they have discovered that the usage of these makes certain code increments difficult to understand in isolation from one another. Furthermore, aspect-oriented techniques were found to be sensitive to the order of composition, which resulted in coupling the implementations of features to each other. The problem of order-sensitivity of feature composition was dealt with by McDirmid et al. [47]. They have used their *open class pattern* to remove the composition order constraints of the *mixin*-based feature-increments proposed earlier by Prehofer [53]. The open class pattern is based on cycling component linking, which reveals the final static shape of the program being composed to individual mixin-based feature implementations.

In contrast, our approach is not affected by any constraints in the order of feature composition, since we do not split legacy feature implementations to an extent that would create a need for an explicit composition step. However, this also means that our approach does not allow for immediate derivation of programs consisting of subsets of the original features—instead, manual removal of optional features would be necessary. In addition, we find that the observations of Murphy et al., about non-trivial semantics and a tight coupling between feature-increments developed using the current separation of concerns techniques, support our argumentation about the practicality of coarse-grained separation of feature implementations.

Kästner et al. [38] compared their annotative approach based on C/C++-like pre-processor with *AHEAD*—a *step-wise refinement* approach of Batory et al. [54], based on composition of template-encoded *class refinements*. The observed drawbacks of the template-based compositional mechanisms include limited granularity and the inability to express 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. The authors addressed the code obfuscation problem by employing coloring of feature implementations in IDE code editor using their CIDE plug-in.

In comparison, the granularity at which our approach manipulates source code is coarser than both pre-processor-based separation and AHEAD. However, it is worth noting that in the case of single-feature classes there is no need for any finer separation, whereas in the case of multi-feature classes, our approach facilitates visual separation of feature implementations by providing code-coloring of source code similar to CIDE.

7. Conclusion

In this paper, we proposed a novel approach to remodularization of Java programs that improves the locality of feature implementations in source code, by reallocating the classes of a program to single-feature and multi-feature packages. Inspection of overlapping feature implementations in multi-feature packages is improved by providing a plug-in to the NetBeans IDE that visualizes the contribution of classes and methods to feature implementations. Using two open-source projects it was shown that the proposed remodularization approach reduces the scattering and tangling of feature implementations considerably. As scattering and tangling are associated with the two program-comprehension problems of delocalized plans and interleaving, the proposed remodularization approach achieves an improvement in the comprehension of feature implementations for the programs of these two projects.

As comprehension of feature implementations is important to program maintenance, we believe that the proposed approach provides programmers with an efficient and useful tool for responding to change requests and error reports formulated in terms of user-observable program functionality. However, as not all projects allow their original decomposition to be changes, identified as the fragile decomposition problem, the proposed approach also supports reverse restructuring. Hence, bidirectional restructuring between object-oriented and feature-oriented decompositions of programs is possible.

References

[1] K.H. Bennett, V.T. Rajlich, Software maintenance and evolution: a roadmap, in: Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, June 04–11, 2000, ICSE'00, ACM, New York, NY, pp. 73–87.
[2] C.R. Turner, A. Fuggetta, L. Lavazza, A.L. Wolf, A conceptual basis for feature engineering, J. Syst. Softw. 49 (1) (1999) 3–15.
[3] T. Shaft, I. Vessey, The role of cognitive fit in the relationship between software comprehension and modification, MIS Q. 30 (1) (2006) 29–55.
[4] G. Booch, Object-oriented development, IEEE Trans. Softw. Eng. 12 (2) (1986) 211–221.
[5] G.E. Krasner, S.T. Pope, A cookbook for using the model-view controller user interface paradigm in Smalltalk-80, J. Object Oriented Program. 1 (3) (1988) 26–49.
[6] H. Benestad, B. Anda, E. Arisholm, Understanding cost drivers of software evolution: a quantitative and qualitative investigation of change effort in two evolving software systems, Empir. Softw. Eng. 15 (2) (2010) 166–203.
[7] R. Brcina, M. Riebisch, Architecting for evolvability by means of traceability and features, in: 23rd IEEE/ACM International Conference on Automated Software Engineering—ASE Workshops, 2008, pp. 72–81.
[8] T. Korson, J.D. McGregor, Understanding object-oriented: a unifying paradigm, Commun. ACM 33 (9) (1990) 40–60.
[9] H. Ossher, P. Tarr, On the need for on-demand remodularization, in: ECOOP'2000 workshop on Aspects and Separation of Concerns, 2000.
[10] S. Letovsky, E. Soloway, Delocalized plans and program comprehension, IEEE Softw. 3 (3) (1986) 41–49.

[11] S. Rugaber, K. Stirewalt, L.M. Wills, The interleaving problem in program understanding, in: Proceedings of 2nd Working Conference on Reverse Engineering, 14–16 Jul 1995, pp. 166–175.
[12] D. Röthlisberger, O. Greevy, O. Nierstrasz, Feature driven browsing, in: Proceedings of the 2007 international Conference on Dynamic Languages: In Conjunction with the 15th international Smalltalk Joint Conference 2007, Lugano, Switzerland, August 25–31, 2007, ICDL'07, vol. 286, ACM, New York, NY, pp. 79–100.
[13] B. Cornelissen, A. Zaidman, B. Van Rompaey, A. van Deursen, Trace visualization for program comprehension: a controlled experiment, in: Proceedings of the 17th International Conference on Program Comprehension, ICPC, IEEE Computer Society, 2009, pp. 100–109.
[14] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.
[15] G.C. Murphy, A. Lai, R.J. Walker, M.P. Robillard, Separating features in source code: an exploratory study, in: Proceedings of the 23rd international Conference on Software Engineering, ICSE, Toronto, Ontario, Canada, May 12–19, IEEE Computer Society, Washington, DC, 2001, pp. 275–284.
[16] E.J. Chikofsky, J.H. Cross, Reverse engineering and design recovery: a taxonomy, IEEE Softw. 7 (1) (1990) 13–17.
[17] O. Greevy, T. Girba, S. Ducasse, How developers develop features, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, CSMR, March 21–23, IEEE Computer Society, Washington, DC, 2007, pp. 265–274.
[18] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, Object-oriented frameworks—problems & experiences, Tech. rep., 1997.
[19] V. Rajlich, N. Wilde, The role of concepts in program comprehension, in: Proceedings of the 10th international Workshop on Program Comprehension, IWPC, June 27–29, IEEE Computer Society, Washington, DC, 2002, p. 271.
[20] D. Ratiu, R. Marinescu, J. Jurjens, The logical modularity of programs, in: Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE'09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 123–127.
[21] D. Janzen, K. De Volder, Programming with crosscutting effective views, in: 18th ECOOP, 2004, pp. 195–218.
[22] N. Wilde, J.A. Gomez, T. Gust, D. Strasburg, Locating user functionality in old code, in: Proceedings of International Conference on Software Maintenance, 1992, pp. 200–205.
[23] T.J. Biggerstaff, B.G. Mitbander, D. Webster, The concept assignment problem in program understanding, in: Proceedings of the 15th international Conference on Software Engineering, Baltimore, Maryland, United States, May 17–21, 1993, International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, pp. 482–498.
[24] V. Rajlich, Intensions are a key to program comprehension, in: International Conference on Program Comprehension, ICPC'09, Vancouver, BC, Canada, pp. 1–9.
[25] K. Liu, A. Alderson, Z. Qureshi, Requirements recovery from legacy systems by analyzing and modelling behavior, in: Proceedings of the IEEE International Conference on Software Maintenance, ICSM'99, IEEE Computer Society, Washington, DC, USA, 1999, p. pp. 3+.
[26] N. Wilde, M.C. Scully, Software reconnaissance: mapping program features to code, J. Softw. Mainten. 7 (1) (1995) 49–62.
[27] M. Salah, S. Mancoridis, A hierarchy of dynamic software views: from object-interactions to feature interactions, in: Proc. of 20th IEEE International Conference on Software Maintenance, ICSM'04, 2004, pp. 72–81.
[28] J. Liu, D. Batory, C. Lengauer, Feature oriented refactoring of legacy applications, in: Proceedings of the 28th international Conference on Software Engineering, ICSE '06, Shanghai, China, May 20–28, ACM, New York, NY, 2006, pp. 112–121.
[29] R.C. Martin, Design principles and design patterns, Available: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.
[30] G. Kiczales, J. Irwin, J. Lamping, J.M. Loingtier, C.V. Lopes, C. Maeda, A. Mendhekar, Aspect-oriented programming, ACM Comput. Surv. 28 (1996).
[31] A. Olszak, B.N. Jørgensen, Featureous: a tool for feature-centric analysis of java software, in: International Conference on Program Comprehension 2010, ICPC'10, Braga, Portugal.
[32] http://spoon.gforge.inria.fr/.
[33] L.C. Briand, J.W. Daly, J. Wüst, A unified framework for cohesion measurement in object-oriented systems, Empir. Softw. Eng. 3 (1) (1998) 65–117.
[34] L.C. Briand, J.W. Daly, J.K. Wüst, A unified framework for coupling measurement, in: Object-Oriented Systems, IEEE Trans. Softw. Eng. 25 (1) (1999) 91–121.
[35] L.C. Briand, S. Morasca, V.R. Basili, Property-based software engineering measurement, IEEE Trans. Softw. Eng. 22 (1) (1996) 68–86.
[36] K. Praditwong, M. Harman, X. Yao, Software module clustering as a multi-objective search problem, IEEE Trans. Softw. Eng. 99 (2010).
[37] D. Coppit, R.R. Painter, M. Revelle, Spotlight: a prototype tool for software plans, in: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 754–757.
[38] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: Proceedings of the 30th international Conference on Software Engineering, ICSE'08, Leipzig, Germany, May 10–18, ACM, New York, NY, 2008, pp. 311–320.
[39] S. Apel, J. Liebig, C. Kästner, M. Kuhlemann, T. Leich, An orthogonal access modifier model for feature-oriented programming, in: Proceedings of the First international Workshop on Feature-Oriented Software Development, FOSD'09, Denver, Colorado, October 06–06, ACM, New York, NY, 2009, pp. 27–33.
[40] http://www.bluej.org/.
[41] http://www.jhotdraw.org/.
[42] A. Olszak, B.N. Jørgensen, Remodularizing java programs for comprehension of features, in: Proceedings of the First international Workshop on Feature-Oriented Software Development, FOSD'09, Denver, Colorado, October 06–06, ACM, New York, NY, 2009, pp. 19–26.
[43] A. Mehta, G.T. Heineman, Evolving legacy system features into fine-grained components, in: Proceedings of the 24th international Conference on Software Engineering, ICSE'02, Orlando, Florida, May 19–25, ACM, New York, NY, 2002, pp. 417–427.
[44] O. Greevy, S. Ducasse, Correlating features and code using a compact two-sided trace analysis approach, in: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering, CSMR, March 21–23, IEEE Computer Society, Washington, DC, 2005, pp. 314–323.
[45] A.D. Eisenberg, K. De Volder, Dynamic feature traces: Finding features in unfamiliar code, in: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, vol. 0, IEEE Computer Society, Washington, DC, USA, 2005, pp. 337–346.
[46] W.E. Wong, J.R. Horgan, S.S. Gokhale, K.S. Trivedi, Locating program features using execution slices, in: ASSET '99: Proceedings of the 1999 IEEE Symposium on Application—Specific Systems and Software Engineering and Technology, IEEE Computer Society, Washington, DC, USA, 1999, p. 194+.
[47] S. McDirmid, M. Flatt, W.C. Hsieh, Jiazzi: new-age components for old-fasioned Java, in: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'01, Tampa Bay, FL, USA, October 14–18, ACM, New York, NY, 2001, pp. 211–222.
[48] T. Eisenbarth, R. Koschke, D. Simon, Locating features in source code, IEEE Trans. Softw. Eng. 29 (3) (2003) 210–224.
[49] M. Eaddy, A.V. Aho, G. Antoniol, Y.G. Guéhéneuc, Cerberus: tracing requirements to source code using information retrieval, dynamic analysis, and program analysis, in: ICPC'08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2008, pp. 53–62.
[50] R. Koschke, J. Quante, On dynamic feature location, in: ASE'05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, 2005, pp. 86–95.
[51] S. Ratanotayanon, H.J. Choi, S.E. Sim, My repository runneth over: an empirical study on diversifying data sources to improve feature search, in: International Conference on Program Comprehension 2010, ICPC'10, Braga, Portugal.
[52] M. Revelle, B. Dit, D. Poshyvanyk, Using data fusion and web mining to support feature location in software, in: International Conference on Program Comprehension 2010, ICPC'10, Braga, Portugal.
[53] C. Prehofer, Feature-oriented programming: A fresh look at objects, in: M. Akşit, S. Matsuoka (Eds.), ECOOP'97—Object-Oriented Programming, in: Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, Berlin, Heidelberg, 1997, pp. 419–443 (chapter 18).
[54] D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: ICSE'03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2003, pp. 187–197.
[55] K. Chen, V. Rajlich, Case study of feature location using dependence graph, in: Proceedings of the 8th international Workshop on Program Comprehension, IWPC, June 10–11, IEEE Computer Society, Washington, DC, 2000, p. 241.