Pure

Syddansk Universitet

Detection of Seed Methods for Quantification of Feature Confinement

Olszak, Andrzej; Bouwers, Eric; Jørgensen, Bo Nørregaard; Visser, Joost

Published in:
TOOL Europe 2012

Publication date:
2012

Document version
Early version, also known as pre-print

Citation for pulished version (APA):
Olszak, A., Bouwers, E., Jørgensen, B. N., & Visser, J. (2012). Detection of Seed Methods for Quantification of Feature Confinement. In C. A. Furia, & S. Nanz (Eds.), TOOL Europe 2012 (Vol. LNCS 7304, pp. 252–268). Springer.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 20. Mar. 2019

Detection of Seed Methods for Quantification of Feature Confinement

Andrzej Olszak[1], Eric Bouwers[2,3],
Bo Nørregaard Jørgensen[1], and Joost Visser[2]

[1] University of Southern Denmark, Odense, Denmark
{ao,bnj}@mmmi.sdu.dk
[2] Software Improvement Group, Amsterdam, The Netherlands
{e.bouwers,j.visser}@sig.eu
[3] Delft University of Technology, Delft, The Netherlands
E.M.Bouwers@tudelft.nl

Abstract. The way features are implemented in source code has a significant influence on multiple quality aspects of a software system. Hence, it is important to regularly evaluate the quality of feature confinement. Unfortunately, existing approaches to such measurement rely on expert judgement for tracing links between features and source code which hinders the ability to perform cost-efficient and consistent evaluations over time or on a large portfolio of systems.

In this paper, we propose an approach to automating measurement of feature confinement by detecting the methods which play a central role in implementations of features, the so-called *seed methods*, and using them as starting points for a static slicing algorithm. We show that this approach achieves the same level of performance compared to the use of manually identified seed methods. Furthermore we illustrate the scalability of the approach by tracking the evolution of feature scattering and tangling in an open-source project over a period of ten years.

1 Introduction

Structural organization of software has a major influence on locality of changes during software evolution [9]. One of the important types of such changes are those concerned with extending and modifying the implemented functionality, i.e. *features*, of a system. To minimize the effort of performing such changes, it is important to control the confinement of features in the structural units of source code, so that they remain properly localized and separated from one another [6].

Therefore, it is important to incorporate the quantification of feature confinement into the quality assessment of software systems. A number of metrics for this purpose have already been defined based on the concepts of *scattering* and *tangling* [11]. Scattering describes the delocalization of concerns over units of source code, whereas tangling describes the simultaneous occurrence in the same units of source code. We refer to these two properties jointly as *feature confinement*.

C.A. Furia and S. Nanz (Eds.): TOOLS Europe 2012, LNCS 7304, pp. 252–268, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In order to measure these properties a link between features and source code of a software system needs to be defined. While a number of approaches for doing this exist, they are not fully automated because they rely on an association between units of source code and human-originated specifications of features defined by experts. This lack of automation prevents the cost-efficient evaluation of feature confinement on a large-scale.

To define a consistent, scalable and objective association between source code units and feature specifications this paper proposes an approach for the automatic detection of so-called *seed methods* of features. The approach detects such seed methods using the popularity of method names and the sizes of the static call-graph slices they yield.

The static slices produced from the identified seed methods do not provide an association between specific features and source code units (i.e., units 'x' and 'y' are involved in the implementation of feature "a"), but rather identify functional related code units in a system. These groups are used as a basis for quantification of feature confinement on the system level, e.g., forty percent of the units are involved in the implementation of twenty percent of the function groups of the system. We believe such quantifications to be useful in several quality assurance scenario's such as the tracking of feature confinement over time as well as determining those systems in a portfolio which implement the best/worst level of feature confinement.

We evaluate our approach on a group of open-source systems by comparing the coverage of source code achieved by slices produced from the automatically detected seed methods with that of the slices produced from manually-chosen seed methods. After applying our approach to a population of systems, we demonstrate its applicability to automatic measurement of multiple revisions of a single system by measuring system-level scattering and tangling in 27 revisions of an open-source project released over a period of 10 years.

2 Related Work

Quantifying feature confinement Brcina and Riebisch [5] propose two metrics for assessing the confinement of features in architectural designs. The first one, *scattering indicator*, is designed to quantify the delocalization of features over architectural components of a system. The second metric, *tangling indicator*, captures the degree of reuse of architectural components among multiple features. For both of these metrics, the authors provide a list of problem resolution actions that can be applied to address the problems detected by the metrics.

Eaddy et al. [6] introduced and validated a suite of metrics for quantifying the degree to which a concern is scattered across components and separated within a component. The defined metrics include *concentration* of a concern in a component, *degree of scattering* of a concern over components, *dedication* of a component to a concern and *degree of focus* of a component. Furthermore, the authors provide a set of guidelines for manually identifying concerns in source code, a prerequisite to a practical application of any concern-oriented metrics.

Wong et al. [14] defined three metrics for quantifying *closeness between program components and features*. These metrics capture the *disparity* between a program component and a feature, the *concentration* of a feature in a program component, and the *dedication* of a program component to a feature. To support practical application of their metrics, the authors propose a dynamic-analysis approach for establishing traceability links between features and source code using an execution slice-based technique that identifies regions of source code invoked when a particular feature-triggering program parameter is supplied.

Locating Features in Source Code. The problem of feature location can be seen as an instance of the more general problem of *concern location*. In this context, Marin et al. [7] have proposed a semi-automatic approach to identify crosscutting concerns in existing source code, based on analysis of call relations between methods. This is done by identifying the methods with the highest fan-in values, filtering them, and using the results as *candidate seed* methods of concerns. These candidate seeds are then manually inspected to confirm their usefulness and associate them with the semantics of a particular concern they implement.

Similarly to Marin et al. [7], the majority of approaches to *feature location* employ the notions of seed methods and control flow. One of the first works associating features with control flow was the *software recoinnaissance* approach of Wilde and Scully [13]. Their approach is a dynamic feature location technique that uses run-time tracing of test execution. Wilde et al. propose that feature specifications are investigated in order to define a set of feature-exhibiting and non-exhibiting execution scenarios. Individual execution scenarios are implemented as a suite of dedicated test cases that, when executed on an instrumented program, produce a set of traceability links between features and source code.

Salah and Mancoridis [10] proposed a different approach to encoding the feature-triggering scenarios. Their approach, called *marked traces*, requires one to manually exercise features through a program's user interface. Prior to executing a feature-triggering scenario, a dedicated execution tracing agent is to be manually enabled and supplied with a name of the feature being exercised. Effectively, this approach removes the need for identifying starting-point methods in source code and the need for the up-front effort of implementing appropriate feature-triggering test cases. Though this is achieved at the price of manual scenario execution.

Similarly to Salah and Mancoridis, Olszak and Jørgensen [8] proposed an approach based on user-driven execution of features. However, they reduce the burden of manual activation and deactivation of a tracing agent by introducing the notion of so-called feature-entry points. Feature-entry points are methods analogous to the ones that need to be invoked by test cases in software reconnaissance - the methods through which control flow enters feature implementations. The approach presented in [8] requires a programmer to annotate such methods in the source code. Using this information, the tracing agent is able to activate itself and track the execution of individual features triggered by a user.

Feasibility of using designated methods as starting points for static, as opposed to dynamic, analysis was demonstrated by Walkinshaw et al. [12]. They

developed a feature location technique based on slicing a static call-graph according to user-supplied landmarks and barriers. There, landmarks are manually identified as the "methods that contribute to the feature that is under consideration and will be invoked in each execution", whereas barriers are the methods irrelevant to a feature. These two types of methods serve as starting points and constraints for a static slicing algorithm. This static mode of operation improves the overall level of automation by removing the need for designing and executing feature-exhibiting scenarios.

3 Problem Statement

Following the methodology of Basili et al. [4], we define the goal of our study to be to *automatically quantify the confinement of functional concerns* to *provide a high-level indication of this confinement* for the purpose of *automated evaluation of the confinement of functional concerns* from point of view of *software quality evaluators* in the context of *evolutionary and large-scale portfolio analysis.*

Surveying the related work, three important steps in the quantification of the confinement of functional concerns arise:

- Identification of entry points to functional concerns of an application
- Identification of those parts of the application that are being executed when the application's functionality is invoked by a user
- Usage of this information to calculate metrics of feature confinement to compare multiple systems or to analyze the evolution of a single system

In order to quantify confinement of functional concerns on a large scale, these steps need to be automated. Fortunately, Walkinshaw et al. [12] showed the feasibility of using static analysis to identify the parts of the application that are involved in the implementation of a functional concern. There, so-called "landmark"-methods representing starting point of feature implementations are used as seed nodes for static slicing of inter-method call graphs. Unfortunately, the lists of suitable landmarks still have to be established manually.

A close examination of the literature does not provide a solution for automatic identification of methods serving as starting points of functional concern implementations. These starting points, which we call "seed methods", play a key role in identifying functionally related code units, since they are the methods through which the control flow enters functionality-specific parts of a program's source code. By defining an approach to automatically detect those methods, the quantification of the confinement of functional concerns can be fully automated.

4 Detection of Seed Methods

To detect methods that play central roles within the implementation of a software system's functionality (i.e., seed methods), the total set of the system's methods needs to be filtered. The filtering approach proposed in this paper is explained

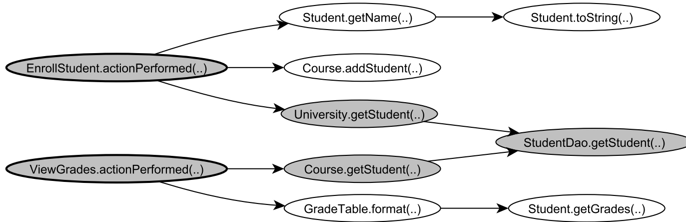

Fig. 1. Call graph of an example program

using the example in Fig. 1 which represents a call-graph of the methods in a small Java system. Note that the use of Java in the examples is only for explanatory purposes, the heuristic is not limited to only this language.

A simple heuristic for filtering the methods is to keep only those methods which are not called from within the system itself, assuming that these methods are used either as call-backs from the interface or are exposed as part of an API of a library. Within Fig. 1 this would lead to identifying the "actionPerformed"-methods as the seed methods.

Unfortunately, this heuristic does not perform well. First of all, for programs defined with a command-line interface the only method that is kept is the static "Main"-method that starts the program. Even though this method is an important part of the system the functional concern of starting an application too general to be considered an important part of a specific application. Secondly, when a system implements an internal event dispatching mechanism, the interesting methods are likely called directly from within the system by the dispatching infrastructure and thus not found by the heuristic.

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

The assumption behind this approach is that, due to polymorphic mechanisms and programming conventions, the methods with the same name but a different implementation implement variations of a functional concern specific to this system. Given the example in Fig. 1 the method "actionPerformed" is implemented multiple times since this method is enforced by a generic interface provided by the Swing GUI framework to handle actions taken by the user. Similarly, the "getStudent"-method is implemented, either because of polymorphism or a convention, by both "Course" and "University" classes.

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

To filter out these uninteresting methods we take into account the number of methods needed to implement a specific method. This is done by counting the number of distinct methods called by the specific method, and then recursively counting the distinct methods used by those methods. Our assumption is that a higher number of methods used in the implementation of a method corresponds to a method which implements more sophisticated functionality. By only keeping those methods which are a) implemented proportionally more often and b) which use many other methods in their implementation we expect to discover the interesting seed methods within a system.

4.1 Heuristic Formalization

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

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

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

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

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

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

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

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

Lastly, to calculate the score for each short name the values of the two functions need to be combined. Ideally, the aggregation function prevents compensation, i.e., a high value on one approach should not overly compensate a low value

Table 1. Normalized scores for the methods as shown in Fig. 1

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

on the other approach. Given this property, two simple aggregation functions can be chosen: the minimum and the product. For our heuristic the product is used to ensure a higher level of discriminative power, the total score for a given short name thus becomes:

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

Applying the heuristic to the example in Fig. 1 provides us with the scores in Table 1, note that the scores are normalized against the total number of methods defined within the system. The "actionPerformed"-methods receive the highest score because these methods occur twice in the system and the average number of methods needed to implement them is 4.5. The methods called "getStudent" are second in rank, occurring three times in the system and needing on average 0.66 methods to be implemented.

4.2 Automated Quantification of Feature Confinement

As explained in Section 3, the calculation of feature-confinement metrics requires two steps; identification of seed methods and identification of those parts of the system that are executed when a seed method is executed.

For the first step the score function, as defined above, can be used to identify the δ most interesting methods. For practical reasons we use the $\delta = 10$ best methods as seed methods throughout the rest of this paper. Nevertheless, the optimality of this value and the potential context-dependency of the δ parameter needs to be investigated in the future.

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

5 Evaluation of the Approach

The evaluation of the proposed approach is two-fold. First, in Section 6, we *validate* the proposed heuristic for detecting seed methods by comparing it to a structured manual approach. This is done by comparing the regions of source

Table 2. Subject systems used in the study

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

code covered by the static traces produced by both approaches. Our hypothesis is that the traces stemming from seed methods found by our heuristic cover the same amount and the same regions of code as the traces stemming from manually-identified seed methods.

Secondly, in Section 7, we apply the proposed approach to measuring the evolution of feature confinement in an open-source project. The goal of this study is to evaluate the applicability of the measurements produced by our approach for enriching the analysis of long-term evolution of scattering and tangling of features. This is done by interpreting the fluctuations of the quality of feature confinement over time, in order to generate informed hypotheses about the nature of the performed evolutionary changes.

6 Validation

To validate the heuristic for identifying seed methods the following steps are taken. First, a set of subject programs is chosen (Section 6.1). For each of the programs, we manually identify a ground-truth set of seed methods enforced by the respective interfacing technologies and libraries being used (Section 6.2). This data is then used to compute static traces, whose aggregated source code coverage allows us to reason about the completeness of the constructed ground-truth. Then, the aggregated coverage of ground-truth slices is compared against aggregated coverage of traces generated by the heuristic (Section 6.3). Based on this, we evaluate whether our approach covers similar amounts and similar regions of source code as the manually-established ground truth.

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

Table 3. Correlation of subjects with technologies and their ground-truth seed methods

Technology	seed methods	ArgoUML	CheckStyle	GanttProject	Gephi	JHotDraw	k9mail	Mylyn	NetBeans RCP	OpenMeetings	Roller	Log4J	Spring	Hibernate	Glassfish
JDK	run, call, main	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Swing	actionPerformed, stateChanged, keyTyped, keyPressed, mouseClicked, mousePressed	✓	✓	✓	✓	✓			✓						
Eclipse/SWT	handleEvent, keyPressed, mouseDown, mouseDoubleClick, widgetSelected, widget-DefaultSelected, runWithEvent, run, start, execute							✓							
Servlet	doGet, doPost										✓				
Android	onCreate, onOptionItemSelected, onClick, onLongClick, onKey, onKeyDown, onTouch, onStartCommand, startService						✓								
Spring	handle, handleRequest, onSubmit, start, initApplicationContext									✓			✓		
Struts	execute, invoke, intercept										✓				
Log4J	getLogger, log											✓			
Hibernate	buildSessionFactory, openSession, update, save, delete, createQuery, load, beginTransaction, commit, rollback													✓	
Glassfish	start, execute, load														✓

6.1 Subject Systems

The evaluation experiment is performed on a set of 14 open-source Java programs summarized in Table 2. The chosen population is intentionally diversified in order to observe how our approach deals with discovering seed methods in not only stand-alone applications but also libraries, frameworks, web applications and application containers. Thereby, we aim at validating the ability of our approach to detect seed methods that are triggered not only by GUI events, but also by command-line parameters, calls to API methods, HTTP requests, etc.

6.2 Ground-Truth

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

For instance, the Swing Java GUI framework defines a set of methods, such as *actionPerformed, onClick*, etc., that are meant to be implemented by a client application and are called by Swing upon the reception of a given event from a user. Such methods, exhibiting individual functional concerns in response to external events, are used as ground-truth seed methods in our experiment. We reckon that such chosen methods could also be appropriate candidates for execution by *software recoinnaissance's* test-cases [13], annotating as feature-entry-points [8], marking as landmark methods [12], or starting points for static analysis [15].

Table 4. Percentage of LOC covered for both approaches

Program	Ground truth	Intersection	Approach
ArgoUML	81,5 %	79,3 %	82,2 %
Checkstyle	51,8 %	48,6 %	73,6 %
GanttProject	93,9 %	93,6 %	96,1 %
Gephi	90,7 %	89,2 %	92,0 %
JHotDraw	87,3 %	85,9 %	88,9 %
k9mail	97,1 %	97,0 %	97,0 %
Mylyn	80,7 %	78,1 %	81,9 %
NetBeans RCP	81,5 %	79,9 %	89,0 %
OpenMeetings	75,9 %	73,6 %	79,5 %
Roller	80,7 %	79,8 %	83,6 %
Log4J	90,1 %	86,3 %	88,6 %
Spring	69,3 %	66,5 %	76,9 %
Hibernate	84,1 %	82,1 %	84,9 %
Glassfish	71,4 %	70,5 %	78,8 %

Based on the mentioned observation, we manually identified interfacing technologies used by the subject programs. This was done based on static dependencies found in source code. For each of the discovered technologies, we identified methods that are intended to be implemented/overridden by client programs in order to provide a client's functionality. We identified such methods by surveying the available official documentation of the investigated libraries. The summary results of this process are listed in Table 3.

6.3 Results

For each of the subject programs, the seed methods of its interfacing technologies served as a starting point for static call-graph slices. Their *aggregated coverage*, being the union of these slices, was used as the ground-truth. The aggregated coverage percentages of both the ground truth and the proposed heuristic are shown in Table 4. In the "Ground truth" column the percentage of code covered by the static-slices originating from the ground truth is shown. The "Approach" column shows the percentage of code covered by the static-slices originating from the methods found by our heuristic. In the "Intersection" column the percentage of code covered by intersection of both result-sets is shown.

We can observe that for most systems the ground-truth coverages remain over 75% of the LOC, which suggests a high degree of completeness of the established ground truth. The only exceptions here are Checkstyle, Spring and Glassfish that are covered in 51,8%, 69,3% and 71,4% respectively. The result of Checkstyle seems to suggest incompleteness of the used ground truth. However, a closer look reveals that there exist four other systems that managed to achieve over 75% coverage based on exactly the same set of seed methods as Checkstyle. As we discuss later, this particular result of Checkstyle had a different cause.

Comparison of columns one and three indicates that aggregated coverage generated by our approach surpasses that of the ground truth for all the systems but k9mail. While the differences for most of the systems appear negligible (below 5% LOC), there are four notable exceptions, namely Checkstyle with difference of 21,8%, Spring with 7,6%, NetBeans RCP with 7,5% and Glassfish with 7,4%.

Interestingly, three of these systems are also the ones that exhibit the lowest ground-truth coverage.

A closer investigation of the reasons for the difference of 21,8% for Checkstyle revealed that the results generated by our approach contained a number of methods that we can categorize as non-technology-originated seed methods. For instance, the methods *process, processFiltered, verify, traverse* and *createChecker*, were found to yield slices containing the highest numbers of classes. These methods constitute important domain-specific abstractions that were established by Checkstyle's developers for implementing functionality, instead of relying on the general-purpose abstractions provided by the JDK or by Swing. Similarly, we found a similar pattern in other subjects, i.e. *afterPropertiesSet, invoke, postProcessBeforeInitialization* and *find* in Spring, or *execute, addNotify* and *propertyChange* in the NetBeans RCP.

Comparison of the columns one and two shows that the proposed heuristic manages to cover most of the regions of source code covered by the manually extracted ground truth, with the average loss of only 2,5% LOC. While this result is something that is expected for the highest sets of coverages (e.g. for the intersection of two result-sets achieving 95% coverage, the maximum possible loss is 5%), it is especially significant in the context of the lowest-scoring ground-truth values, i.e., Checkstyle (for which the maximum possible loss is 26,4%) and Spring (for which the maximum possible loss is 23,1%). This indicates that our approach not only covers as much source code as the manually-established ground truth, but that it also identifies largely the same regions of source code, thus providing analogous input to measuring feature confinement.

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

7 Evolutionary Application

In this section, we apply our approach to evaluating the quality of features confinement in an evolving program. We do this by automatically measuring long-term evolutionary trends of confinement metrics in the release history of Checkstyle[1], a library for detecting violations of coding style in Java source code. The units of functionality in Checkstyle library, and whose historical quality we intend to assess using feature-oriented metrics, are the individual *detectors* for various types of style violations, as well as the core infrastructure of the library responsible of parsing source code, reporting results, etc. In this investigation, we measure 27 major releases of the library since version 1.0 until version 5.4.

7.1 Measuring Feature Confinement

The existing literature proposes and demonstrates the usefulness of a number of diverse metrics for measuring this *confinement*, e.g., [14,6,5]. A common theme

[1] `http://checkstyle.sourceforge.net/`

that tends to re-appear in many works is formulating measures for quantifying *locality of features* in structural units of source code (i.e. packages, classes or methods) and for quantifying *overlap of features* in terms of structural units. Having this in mind, for the purpose of this work we use the most elementary and intuitive formulations of metrics for capturing these properties. The two metrics used here are called *scattering* and *tangling* and they are based on simply counting the number of related classes or features. They are defined as follows:

- Scattering: denotes the delocalization of a functional concern over computational units of a program. In this work, we measure scattering for each seed method as the total number of classes that appear in its static trace.
- Tangling: denotes the interweaving of functional concerns in a structural unit of a program. In this work, we measure tangling for each class as the number of seed methods in whose static traces a given class appears.

The metrics chosen to quantify feature confinement are not directly calculated on the system level, but rather on the level of a single trace (scattering) or on the level of the class (tangling). In order to come to a system level measurement the values of the measurements on the lower level need to be aggregated. To compare a variety of systems in a consistent manner the aggregation needs to be done in such a way that the influence of other factors, for example the size of the system or the number of concerns evaluated, do not influence the aggregated measurement.

7.2 Aggregation of Confinement Metrics

Alves et al. [2] proposed an aggregation strategy based on benchmarking with these characteristics which has been applied successfully [3]. In this aggregation strategy a repository of systems is used to derive thresholds for categorizing unit of measurement in system (i.e., the class or the trace) into one of four categories. By summing up the size of all entities in the four categories a system-level profile is calculated, which in turn is used to derive a system-level rating [1].

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

The repository used to calibrate the rating for both scattering and tangling consists of industry software systems previously analyzed by the Software Improvement Group (SIG), an independent advisory firm that employs a standardized process for evaluating software systems of their clients [3]. These industry systems were supplemented by open source systems previously analyzed by SIG's research department.

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

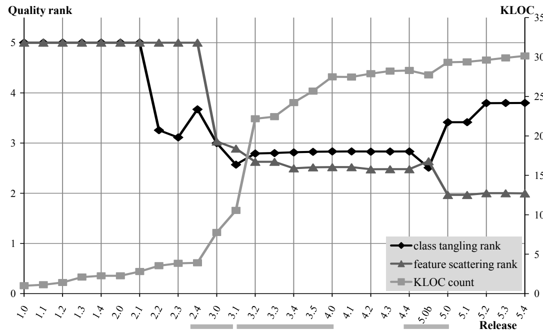

Fig. 2. Evolution of Checkstyle

7.3 Results

Fig. 2 shows a plot of the measured evolutionary trends of Checkstyle. The figure shows the values of KLOC metrics and the *ranking values* of scattering and tangling for each release. Please note that as a result of benchmarking, the quality rankings have to be interpreted inversely to the metrics they originate from - e.g. a high quality rank of scattering means low scattering of features.

The evolutionary trends plotted in Fig. 2 indicate that feature-oriented quality, represented by ranks of scattering and tangling tends to degrade over time. In the following, we investigate three periods marked in Fig. 2 that exhibit particularly interesting changes of the measured ranks.

Versions 2.4 – 3.1: a significant degradation of both scattering and tangling quality ranks is observed. The observed degradation was initiated by changes done in release *3.0*, where one of the major changes was a restructuring to a "completely new architecture based around pluggable module"[2]. This restructuring involved factoring-out a common infrastructure from existing detectors. Doing so was bound to increase the number of classes that features are scattered over, and create a number of infrastructural classes meant to be reused by multiple features, thus contributing to tangling.

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

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

[2] `http://checkstyle.sourceforge.net/releasenotes.html`

reverse changes in tangling and scattering ranks in version *5.0b*, where these detectors are removed.

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

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

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

8 Discussion

The results presented in Section 6 show that seed methods automatically identified by our approach yield static slices that capture largely the same regions of source code as a manually-established ground truth. Moreover, the heuristic improves on the ground-truth coverage results by identifying non-technology-originated seed methods that reflect important domain-specific functional abstractions. Given these observations, we conclude that the seed methods computed by our approach are adequate substitutes to manually identified seed methods for the purpose of system-level quantification of feature confinement.

The application of our approach presented in Section 7 shows that the automated measurement of the evolution of scattering and tangling properties provides a valuable perspective on the evolution of an existing software system. We demonstrated how to interpret these metrics in the context of Checkstyle's change log by generating informed hypotheses about the impact of the individual changes on the feature-oriented quality of the program. While the generated

hypotheses need additional validation, they provide a sound and firm starting point for evaluating the evolutionary quality of feature confinement.

Algorithm Parameters. As explained in Section 4.2, the parameter δ is used to limit the number of best-ranked methods to be chosen as seed methods. Theoretically, such a value should preserve all the methods that contribute significantly to aggregated program coverage, whereas all the remaining methods should be filtered out. Even though the chosen δ seems to be correct for our current case-study (i.e., adding more methods to the list of seed methods did not increase the program coverage substantially), more work is needed to determine the optimal value of δ. Additionally, it is important to investigate whether a single optimal value of δ can be found for a portfolio of programs, or whether each program needs an individually-chosen δ value.

Limitations. One of the limitations of the performed experiments is the lack of a direct comparison against outputs of existing feature location approaches. Ideally, a correlation study of system-level scattering and tangling metrics contrasting our approach with the existing ones could be conducted. However, such a study requires a significant number of data points, being software systems, to achieve a satisfactory level of statistical confidence. While in the case of our approach this data can be generated automatically, to the best of our knowledge no sufficiently large data sets exist for existing feature location approaches.

In our evolutionary investigation, the differences among the sets of identified seed methods for subsequent versions of Checkstyle could have influenced our results. We observed this behavior when new types of detectors using new seed methods were added. While such a flux of the sets of seed methods reflects the evolution of how feature implementations change over time, it may turn out problematic with respect to comparability of measurements across versions. As a means of addressing this threat to validity we used the metric aggregation discussed earlier. Additionally, we confirmed that even tough the set of seed methods changed over time the coverage remained between 68% and 75%.

Lastly, because only open-source Java systems where used in the evaluations, the results cannot be generalized to systems with different characteristics (i.e., systems using a different programming paradigm). However, since the heuristic is largely technology agnostic, it remains possible to validate the approach using a more diverse set of systems.

9 Conclusion

Cost-efficiency of applying feature-oriented measurement is constrained by lack of automation of measurement collection procedures. This hinders applicability of feature-oriented metrics in large-scale and evolutionary scenarios. As a result, it remains difficult to assess quality of feature implementations, control it over time, and thoroughly validate new feature-oriented metrics.

In this paper we have proposed an approach for the automated measurement of system-level feature confinement, based on statically slicing the call-graph of

a software system starting from a set of seed methods. The contributions of this paper are:

- The definition of a heuristic to automatically detect seed methods in software systems, based on popularity of method names and size of the static call-graph slices they yield.
- The validation of the heuristic by comparing the performance of static slices produced by our approach against slices produced from a set of manually selected seed methods.
- A demonstration of the practical applicability of the proposed approach in a case-study of measuring feature confinement over time.

References

1. Alves, T.L., Correia, J., Visser, J.: Benchmark-based aggregation of metrics to ratings. In: Proceedings of the IWSM/MENSURA 2011, The Joint Conference of the 21th International Workshop on Software Measurement (IWSM) and the 6th International Conference on Software Process and Product Measurement, Mensura (2011)
2. Alves, T.L., Ypma, C., Visser, J.: Deriving metric thresholds from benchmark data. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM 2010, pp. 1–10. IEEE Computer Society, Washington, DC (2010)
3. Baggen, R., Schill, K., Visser, J.: Standardized code quality benchmarking for improving software maintainability. In: 4th International Workshop on Software Quality and Maintainability (SQM 2010), Madrid, Spain, March 15 (2010)
4. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley (1994)
5. Brcina, R., Riebisch, M.: Architecting for evolvability by means of traceability and features. In: 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, ASE Workshops 2008, pp. 72–81 (September 2008)
6. Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V.: Do crosscutting concerns cause defects? IEEE Transactions on Software Engineering 34, 497–515 (2008)
7. Marin, M., van Deursen, A., Moonen, L.: Identifying crosscutting concerns using fan-in analysis. ACM Transactions on Software Engineering and Methodology 17, 3:1–3:37 (2007)
8. Olszak, A., Jørgensen, B.N.: Remodularizing java programs for improved locality of feature implementations in source code. Science of Computer Programming (2010) (in press, corrected proof)
9. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15, 1053–1058 (1972)
10. Salah, M., Mancoridis, S.: A hierarchy of dynamic software views: From object-interactions to feature-interactions. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 72–81. IEEE Computer Society, Washington, DC (2004)
11. Turner, C.R., Fuggetta, A., Lavazza, L., Wolf, A.L.: A conceptual basis for feature engineering. Journal of Systems and Software 49, 3–15 (1999)
12. Walkinshaw, N., Roper, M., Wood, M.: Feature location and extraction using landmarks and barriers. In: IEEE International Conference on Software Maintenance (ICSM 2007), pp. 54–63 (October 2007)

13. Wilde, N., Scully, M.C.: Software reconnaissance: mapping program features to code. Journal of Software Maintenance 7, 49–62 (1995)
14. Wong, W.E., Gokhale, S.S., Horgan, J.R.: Quantifying the closeness between program components and features. Journal of Systems and Software 54, 87–98 (2000)
15. Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F.: Sniafl: Towards a static noninteractive approach to feature location. ACM Transactions on Software Engineering and Methodology 15, 195–226 (2006)