An Approach for Solving the Feature Location Problem by Measuring the Component Modification Impact

Abhishek Rohatgi[1], Abdelwahab Hamou-Lhadj[2], Juergen Rilling[1]

[1]Department of Computer Science and Software Engineering

[2]Department of Electrical and Computer Engineering

Concordia University

1455 de Maisonneuve West Montreal, Quebec

{a_rohatg, abdelw, rilling@encs.concordia.ca}

Abstract

Maintaining a large software system is an inherently difficult task that often involves locating and comprehending system features prior to performing the actual maintenance task at hand. Feature location techniques were introduced to locate the source code components implementing specific software features. Common to these approaches is that they rely either on exercising several features of a system, and/or domain experts to guide the feature location process. In this paper, we present a novel hybrid feature location approach that combines static and dynamic analysis techniques. Our approach uses a component dependency graph of the system to provide a ranking of the components according to their feature relevance. The ranking itself is based on the impact of a component modification on the remaining parts of a system. Our approach can almost be completely automated without requiring an extensive knowledge of the system. A case study performed on two open source projects is presented to evaluate the applicability and effectiveness of our approach.

1. Introduction

System evolution, an important aspect of the software life cycle, depends greatly on the ability of a maintainer to identify these parts of the source code that implement specific features. A software feature can be defined as a function of the system that can be triggered by an external user [8]. Software maintainers typically do not need to analyze and comprehend an entire system prior to making modifications or adding new system functionality. Required software modifications often relate directly to features and their implementations [31]. As a result, programmers often adopt an *as-need* comprehension strategy [16], in which case programmers tend only to comprehend these portions of the program that are relevant the affected feature. As part of such an as-needed approach, programmers will first make an attempt to locate these code segments that implement the feature being modified, followed by understanding the code and making the required changes. However, identifying these relevant source code segments is an inherently difficult task due to several factors. (1) A lack of traceability between documentation and source code caused by the unavailability of roundtrip engineering tools and/or inadequate processes within organizations to enforce consistent and up-to-date documentation, and (2) the lack of a clear mapping between a system's features and their corresponding code segments. This disconnection is often a direct result of bad design decisions and excessive ad-hoc maintenance activities performed on the system. As a result, a feature tends to be often distributed over several different modules resulting in complex and often difficult to comprehend dependencies among source code segments.

A commonly used approach to support software evolution is based on the use of feature location techniques to help identifying and locating features in the source code [4]. Existing feature location techniques can be grouped into two main categories depending on the use of static and dynamic analysis techniques. Static analysis consists of analysing the source code in order to extract the system's components (e.g., classes, methods, or any other module of the system) and how they interact with each other, whereas, dynamic analysis techniques are based on the analysis of the system behavioural aspects, commonly through the study of their execution traces.

The first category, pure dynamic approaches, require the generation of execution traces that are then clustered or compared in order to identify the components of a single feature (e.g., Software Reconnaissance [32]). A general limitation with these pure dynamic approaches, however, is the need to generate as input for the analysis execution traces that are created by exercising several features of the system. The need to exercise several features can not only result in a significant number of traces to be generated, but it also relies on availability of domain knowledge about existing features and the inputs required to generate these traces.

The second category relies on a combination of static and dynamic analysis. The approaches in this category utilize static information to further process the execution trace generated for the feature under study. For this purpose, several techniques were presented, such as the ones based on concept analysis [8], latent semantic indexing [7], etc. The major limitation with these techniques is that they require a significant amount of user interaction and domain knowledge to allow the user to indicate which parts of the source code should be analyzed.

The work presented in this paper is a continuation of our previous work on feature location [23], where we discussed in general how impact analysis can be applied to address the feature location problem and [22] where we introduced two feature location techniques based on impact analysis. In this research, we further extend our work on feature location that combines dynamic analysis, requiring only a single feature execution trace as input, with impact analysis based on static dependency analysis.

The main contributions of this paper are as follows:

- We present a novel approach for feature location using impact analysis by measuring the impact of the components invoked during the execution of a given feature on the rest of the system. Our hypothesis is that the smaller the impact set of a component modification, the more likely it is that the component is specific to a feature. Conversely, we expect a component affecting many parts of a system to be invoked in multiple traces and therefore rendering it as less specific to a particular feature.

- As part of this research we introduce four different feature location algorithms that vary in the way the impact of a component modification is measured. The first impact metric considers only the impact due to modification of a component on the rest of the software. The second metric improves over the first metric by considering additionally information about the system architecture. The third metric considers both the impact due to the modification of a component on rest of the system as well as the number of components that affect this component. The last metric adds additional weight to the previous metric by adding information about the system architecture.

- We applied the algorithms to traces generated from two object-oriented software systems to evaluate the applicability of our approach.

The remaining part of this paper is organized as follows: Section 2 presents a survey of existing feature location techniques along with their advantages and limitations. The feature location algorithms and approach are presented in Section 3. The evaluation of our approach is presented in Section 4. We conclude the paper in Section 5 with a summary of the main contributions and some future work.

2. Related Work

In this section, we present a survey of what we consider the most influencing work in feature analysis. Wilde and Scully [32] introduced in 1995 the concept of Software Reconnaissance, which relies on dynamic analysis to locate source code components that implement a specific feature. The authors' approach necessitates two main steps. The first step consists of generating multiple execution traces by exercising several features of the system, which are then compared during the second step. The components specific to the feature under study are the ones that are only invoked in its corresponding trace. One of the drawbacks of this approach is that it requires exercising several features of the system although the final objective is to identify only the components of a single feature. In addition, it is not clear how many features need to be considered for the approach to be effective. Furthermore it is essential in this approach to select a balanced set of features (i.e., features that cover different parts of the system) for the software reconnaissance approach to be effective. This requires from the software engineers using this technique to be knowledgeable of the system under study.

Wong et al. enhanced the Software Reconnaissance approach proposed in [32], by including three measurements used to identify the extent to which a particular component belongs to a feature [33, 34]. Their first metric is called *Disparity* and measures the closeness of a feature to a given program component. The authors claim that the disparity is equal to the set of blocks in either a component or a feature under consideration (but not in both) divided by the union of a set of blocks in a feature and components under consideration. They define blocks as an execution slice or code statements. Their second metric is called *Concentration,* which they define as the intersection of the set of blocks in a component (under consideration) and the set of blocks in a feature (under consideration) divided by a set of blocks in the feature under consideration. Their third metric, called *Dedication*, measures how much of a program component is concentrated in a feature. They calculate this as the intersection of the set of blocks in the component under consideration and the set of blocks in the feature under consideration divided by the set of blocks in this component.

The Reconnaissance approach was also extended by Antoniol et al. [1]. Their main contribution was to filter out unwanted events from the execution traces prior to comparing them. Examples of such events included unwanted mouse motion events, frequently occurring events, automatically generated code, etc. For this purpose, the authors used a combination of knowledge-based filtering and probabilistic ranking techniques. Another contribution of their work is the application of software reconnaissance to traces generated from multi-threaded applications.

Eisenberg and De Volder [9] proposed a feature location technique based on ranking the components invoked in traces. According to them, a component occurring several times in the execution of a feature under different situations (i.e., normal and exceptional scenarios) should be regarded as an important component, whereas a component that occurs in traces of several features should be considered as a utility component and should be ranked lower in comparison with other components.

Eisenbarth et al. [8] proposed a feature location approach based on concept analysis. Their approach uses dynamic analysis to gather traces that correspond to software features of the system, similar to Wilde and Scully's technique [32]. They combined the content of traces with a static dependency graph to build a concept lattice that maps features to components. One of the shortcomings of this approach is that the concept lattice shows also overlapping components, i.e., the ones that implement several features. To overcome this issue, users are required to navigate the concept lattice and identify manually the components specific to each feature. This process necessitates a considerable effort from the users and a good understanding of the source code as well as the domain of the application.

Concept analysis has also been applied for different types of dependency analysis, including impact analysis, reuse and comprehension. Gold et al. [10] introduced a framework of concept slicing approaches that combine concept assignment and program slicing to extract executable domain level slices. Our approach differs from their approach in several aspects, including its overall objective. Gold et al. focus in their approach on the computation of executable subprograms for the purpose of reuse and testing, we on the other hand only identify and rank components based on their relevance with respect to the implementation of a particular feature. Our approach is mostly automatic, with no or only very limited user involvement. This compares to Gold's et al. approach that requires the existence of a domain level model and manual linking of the concepts to various artefacts.

Furthermore, we argue that in the context of feature location, maintainers do not require all the syntactical and semantic details that have to be included in an executable subprogram. Maintainers are often only interested in identifying the components that are the most specific to a specific feature.

In [11], the authors address one of the shortcomings of their previous work [10] by making the Hypothesis Based Concept Assignment approach used as input to the slicing algorithm [10] less restrictive, by allowing for the overlapping of concept boundaries. They introduce a genetic and a search based algorithm to address the problem of identifying overlapping concept boundaries. In Binkley et al. [3] the authors showed that based on evidence from experimental studies they performed, executable concept slices can aid in comprehending static and dynamic aspects of the program. The results collected in this paper indicate that slicing criteria can be raised to the domain level as an executable concept slice; thus increasing its applicability for testing and reuse.

Poshyvanyk et al. present an approach that is based on processing a single trace derived from the feature under investigation [24, 25]. They applied information retrieval techniques to extract knowledge from the source code that describes the components invoked in the trace. Through an iterative process, the resulting knowledge is explored by queries containing key terms that describe the feature are executed in order to locate the source code components implementing the feature. The advantage of this approach is that it requires only one trace instead of many traces as it is the case in other approaches. The disadvantage is that it relies heavily on informal knowledge such as source code comments, identifiers, etc. to extract knowledge about the trace components.

Greevy et al. [12] exploited the relationship between features and classes to analyze the evolution of features and to detect changes in the code from a feature perspective. Rather than detecting feature specific components, the main focus of this work is to investigate how the role of classes may change during the evolution of a system (for example, by understanding the number of features they participate in as the system evolves).

Kothari et al. [17] presented an approach for identifying canonical features of a system and their comprehension at the source code level. A canonical feature in their context is a small set of features that implement different parts of the system. They compute their canonical feature set, by first building test cases to exercise all the features of the system and record the resulting traces. In the next step, for each of the traces a dynamic call graph is generated. Using a similarity measurement tool, they computed the pair wise similarity among the call graphs using a similarity matrix. Call graph similarity can be measured using simple metrics such as (a) the number of function nodes the call graphs have in common, (b) the number of call edges they have in common, or (c) a more sophisticated approximate graph matching algorithm. In their approach they used the similarity among subgraphs to compute the degree to which features share common significant amount of code, with two similar features sharing several vertices (functions) and edges (function call relations). The amount of code of a particular feature that is not shared with the other features (i.e., through their dynamic call graphs) is deemed to be the most specific to this particular feature. One of the main drawbacks of this technique is that it requires computing the similarity matrix by exercising each and every feature of the system under consideration.

Salah et al. [28] introduced a feature location approach that combines three system interaction views, an object interaction, a class interaction and a feature interaction view. The object interaction view is constructed from the execution traces that are generated by exercising a subset of its features. This view shows how objects interact with each other through method invocation. The class interaction view is simply an abstraction of the object view by grouping objects by their class types. The feature interaction view shows the relationships between the features based on the objects (or classes) invoked in their corresponding traces. The mapping between the feature interaction view and the object (or class) interaction view enables the analyst to uncover the components implementing a specific feature. In an attempt to reduce the number of components invoked in multiple traces, the authors proposed using marked traces, which are traces where an analyst needs to manually indicate the beginning and the end of the trace generation process. However, marked traces not only require user interactions, they also do not guarantee that the resulting traces will contain only the components that are most relevant to a traced feature.

Robillard et al. proposed a technique to locate concerns in source code [27]. A concern, also called a software aspect, can be considered as a particular feature where the implementing components crosscut many modules of the system. The authors introduced the concept of a concern graph, which abstracts the implementation details of a particular concern. The vertices of the graph consist of the components (e.g., routines) involved in the implementation of this particular software aspect. The edges represent the relationships among these components. The process of creating a concern graph encompasses two steps. In the first step, the software engineer builds a component dependency graph from the system. This step is usually performed automatically. The second step consists of iteratively querying the component dependency graph to identify the components specific to a particular concern. This step requires from the developer to have some knowledge of the system under study. The authors have also developed a tool called FEAT (Feature Analysis and Exploration Tool) that partially automate the tasks of creating a program model from the source code, formulating queries, extracting concern graphs, and displaying the concern graphs to the developer in a convenient and manageable form. Using this tool the developer can also view the implementation details of a concern graph into source code.

Common to most of these reviewed techniques is that they suffer from two major limitations. Even if a user might only be interested in locating a specific feature, most of these techniques require exercising several system features in order to be able to identify the components associated with a specific feature. Exercising several features however not only requires the generation of the appropriate input data for each feature execution, but also domain knowledge in selecting the appropriate features that need to be used in order to obtain a balanced input set that would result in unbiased results.

The second drawback with most existing techniques is that they require significant human intervention either during locating or interpreting the feature related data, resulting in a significant cost and effort overhead. As a result, there is a clear need for more automated techniques to reduce the cost associated with these interactive feature location approaches. In addition, many existing techniques require users to have a good prior understanding of the system before applying the techniques. This contradicts the goal of feature location, which is to assist software engineers in *comprehending* how a particular feature is implemented.

Similar to other approaches, our feature location approach combines static and dynamic analysis. However, in comparison to the existing work, our approach operates on only *one* trace that corresponds to the feature under study, and it facilitates the automatic identification of feature-specific components. Our approach applies a ranking mechanism to guide software engineers in locating feature-specific components without the need for prior knowledge of the system.

3. Feature Location Methodology

In this section, we present our feature location methodology, which combines static and dynamic analysis techniques. We use dynamic analysis to generate a trace that corresponds to the feature under study. Static analysis is applied to rank the components invoked in the generated trace according to their relevance with respect to the executed feature. The ranking technique presented in this paper is based on impact analysis, i.e., by measuring the impact of a component modification on the rest of the system.

The organization of this section is as follows: In Section 3.1, we present our definition of a software feature. In Section 3.2, we describe our overall approach. The detailed steps of the proposed approach are further elaborated in Sections 3.2.1 and 3.2.2, which cover the generation of traces from software features, and the application of impact analysis for locating feature-specific components.

3.1. What is a Software Feature?

Perhaps the most commonly used definition of a software feature in feature location research is the one proposed by Eisenbarth et al. in [8]. The authors define a software feature as a behavioural aspect of the system that represents a particular functionality, triggered by an external user [8].

In [8], Eisenbarth et al. discuss the relationship between a software feature, a scenario, and a computation unit (Figure 1). A scenario is an instance of a software feature where the user needs to specify a series of inputs to trigger that feature. A scenario can invoke a number of features at the same time. A computational unit refers to the source code components that are executed by exercising the feature on the system. A feature is implemented by many computational units, and at the same time a given computational unit can be used in the implementation of multiple features.

In our research, we also define a software feature as any specific scenario of a system that is triggered by an external user. However, we further extend this definition by adding that a software feature is similar to the concept of use cases found in UML [30]. As a result, we assume that a particular instance of a feature (based on a selected data input) corresponds to a scenario. We also do not distinguish between primary and exceptional scenarios although it is advisable to include at least the primary scenario, since these scenarios tend to correspond to the most common program execution associated with a particular feature.

3.1 Overall Approach

Figure 2 provides a general overview of our approach used to identify components implementing a specific feature. In this paper, we limit the components of interest to classes in the system. However, we believe that our approach is also applicable at other component abstraction levels, such as methods, packages, or any other modules.

Figure 2 illustrates our feature location approach which is based on a combination of static and dynamic analysis. An execution trace is generated by exercising the feature under study. There exist various techniques for generating traces such as inserting probes in the source code, instrumentation of the binaries, or modifying the run-time environment to support the generation of traces. These techniques are automatic and do not require human intervention. In our approach we used source code instrumentation due to its simplicity and the abundance of available tool support. Once the system is instrumented, we execute the instrumented version by exercising the feature to be analyzed. From this feature trace, we extract the distinct classes invoked, while executing the particular feature (i.e., on the fly). We call the distinct classes invoked in a feature trace the *execution profile* of the feature. It should be noted that the trace does not need to be saved. We use the term *feature trace* to refer to a trace that corresponds to a particular feature execution. Next a class dependency graph (static analysis) is created to rank the distinct classes invoked during the feature execution to identify their relevance to the feature. The ranking technique itself is based on the impact set of a component (i.e., a class) modification on the other parts of the system. We hypothesize that the smaller the impact of a component modification is, the more likely it is that this component is specific to the particular feature. The rationale behind this is as follows: classes that impact many other parts of the system will most likely be invoked in many other feature traces, making them non-feature specific. It has been shown previously [14, 13] that these classes often correspond to utility classes that help implementing the core functionality of the system. On the other hand, one would expect a feature-specific class to be self-contained (i.e., low coupling and high cohesion), and a modification to such a class should result in a very low impact on the remaining parts of the system. Furthermore, there will be situations where the impact set of a class is in between these two cases, indicating classes that implement functionality shared by similar features.

Based on the above discussion, we characterize the components invoked in a feature-trace according to the following three categories:

- Relevant Components: Components that are most relevant to the feature at hand. In other words, these components are not invoked in any other feature trace.
- Related Components: Components that are involved in the implementation of related features, and therefore, are expected to appear in these related feature traces.
- Utility Components: Components that are mere utilities and therefore are used by many other features of a system.

In the next section, we discuss the applicability of impact analysis to the feature location problem. In particular, we introduce four impact metrics to measure the degree to which a specific component can be deemed relevant to the studied feature.

3.2. Impact Analysis

Impact analysis is the process of identifying these parts of a program that are potentially affected by program modification. Impact analysis is an essential activity for planning system changes, making changes, accommodating certain types of software changes, and tracing through the effects of changes [19, 29].

3.2.1. Building a Class Dependency Graph

A class dependency graph is a directed graph where the nodes are the system's classes and the edges represent a dependency relationship among the classes. Several types of relationships may exist between two classes such as the ones based on method calls, generalization and realization relationships, etc. It should be noted that the accuracy of the impact analysis depends greatly on the types of dependency relations supported by the analysis. For our impact analysis we are in particular interested in the method invocations. These function calls are traced using a static call graph including require points-to analysis for polymorphic calls. There exist various techniques for the points-to analysis, among them are, Unique Name (UN) [5], Class Hierarchy Analysis (CHA) [2, 6], and Rapid Type Analysis (RTA). Each algorithm has its own advantages and imitations. In this paper, we use RTA for its simplicity, efficiency, and tool support [2].

3.2.2. Impact Metrics

We have developed four metrics for evaluating the impact of a class modification on the remaining parts of the system. These metrics differ in the way the impact of a component modification is computed.

3.4.2.1 Definitions

Definition 1: Impact Set

We define the impact set for modifying a component *C,* as the set of components that depend directly or indirectly on *C*. More formally, a class dependency graph can be represented using a directed graph $G = (V, E)$ where V is a set of classes and E a set of directed edges between classes. The impact set of *C* consists of the set of predecessors of *C*. A predecessor of a node is defined as follows: Consider an edge *e = (x, y)* from node *x* to *y*, If there exist a path that leads from *x* to *y*, then *x* is said to be a predecessor of *y*. For example, the impact set of class *C5* in Figure 3 consists of the classes *C6, C7 C4, C3*, and *C1* (i.e., the predecessors of node *C5*) since there exist a path between each of these classes and the class *C5*. Note that the same class may occur in multiple paths. In this case, such a class is considered only once in the impact set. This also handles situations where two classes are mutually dependent on each other.

It should be noted that the fact that two components depend on each other does not necessarily lead to a change in one component if the other one has been modified. This is because the change may not affect the dependency itself. However, when using static analysis to measure the impact of a component modification, we can only refer to the potential impact since a component can be modified in all possible ways.

Definition 2: Class Afferent Impact

The Class Afferent Impact (CAI) of a class *C* consists of the number of classes that are affected (directly or indirectly) when *C* is modified (i.e., the cardinality of the impact set of C).

Definition 3: Class Efferent Impact

The Class Efferent Impact (CEI) of a class *C* is the number of classes that will affect (directly or indirectly) *C* if they change. These are the classes in the directed graph that can be reached through *C*, also called the descendants of C in the class dependency graph. It should be noted that the intersection between CAI and CEI is not necessarily empty, since some components can be affected by a change to *C* while at the same time they can affect *C*.

Definition 4: Package Afferent Impact

We define the Package Afferent Impact (PAI) of a class *C* as the number of packages that are affected by a modification of *C*. The package afferent impact will be used to weigh some of the metrics presented in this section. It should be note that we consider all packages of the system as separate packages no matter if they belong to another package or not.

The class (and package) afferent and efferent impact should not be confused with the afferent and efferent couplings proposed by Robert Martin [21], used to assess the quality of a design by analyzing the stability of its subsystems. The afferent and afferent couplings focus on measuring fan-in and fan-out of a subsystem using a subsystem dependency graph, whereas our focus on measuring the impact of a component change on the rest of the system.

3.4.2.2 The One Way Impact Metric (OWI)

There exist several metrics that measure the relationships among a system's components (e.g., MOOSE metrics [18]). However, these metrics are used to assess the overall quality of a design and do not necessarily measure the impact of a component on the overall system. In this paper, we propose four simple and yet powerful metrics that focus specifically on measuring this impact set.

The first metric is referred to as the *One Way Impact* (OWI) metric and considers exclusively the impact modification of a class on the system, i.e., CAI.

We define the OWI metric of a class *C* as:

- S = A set that contains all the classes of the system under study. We assume in this paper that the system under study has more than one class. That is the cardinality of set S is always greater than 1.

$$OWI(c) = \frac{|CAI(c)|}{|S|}$$

The OWI has a range from 0 to 1. It converges to 0 if the class has a small impact on the rest of the system, which is a good indicator that the class is specific to the feature in question. On the other hand, a class with an OWI value close to 1, indicates that a class might be used in various features and therefore a change in this class might cause many parts of the systems to change. This indicates that this class is used to support the implementation of various features.

The running example in Figure 4 will be revisited throughout this section to illustrate how our impact analysis metrics can identify feature related components. In this example, we assume that the classes that are relevant to the specific feature all located in package P1. However, the feature profile created from this specific feature trace also contains additionally the classes C6, C7, and C8.

Table 1 shows the result of applying the one way impact metric to the example in Figure 4. The table is sorted in an ascending order based on the OWI values and shows that the metric was able to group successfully all P1 classes (the most relevant classes) in the top part of the table. The class C6 is used by many other classes of the system, which suggests that it is a utility class. Similarly classes C7 and C8 where close ranked at the bottom together with C6, since they are used by the utility class C6.

Table 1. Applying OWI to Example of Figure 4

Package	Class	CAI	OWI
P1	C1	0	0.000
P1	C2	1	0.083
P1	C3	1	0.083
P1	C5	1	0.083
P1	C12	2	0.167
P1	C4	5	0.417
P2	C6	7	0.583
P2	C8	8	0.667
P2	C7	9	0.750

3.2.2.3 The Two Way Impact Metric (TWI)

The Two Way Impact metric considers both, the impact of a class modification on the rest of the system (i.e., its afferent impact), as well as the number of classes that impact this class if these classes change (i.e., the efferent impact, CEI, of the class).

The rationale for using the efferent impact is based on a study conducted by Hamou-Lhadj et al. to automatically detect utility components that exist in a software system [15, 13]. The authors used in their work, fan-in analysis to measure the extent to which a routine can be considered a utility. According to their findings, a routine with high fan-in (incoming edges in the call graph) should be considered a utility as long as its fan-out (outgoing edges) is not high. They argued that the more calls a routine has from different places then the more purposes it likely has, and hence the more likely it is to be a utility. On the other hand, if a routine has many calls (outgoing edges in the call graph), this is evidence that it is performing a complex computation and therefore it is needed to understand the system.

The TWI metric uses a similar approach, except that it considers the impact of a component modification rather than its mere fan-in. In other words, we do not only considers the direct impact associated with a component change by including all components that are directly associated with it, but also the ones that are indirectly affected by this component change. This allows us to measure the fact that the afferent impact of a component can be very high without necessarily having a high fan-in. For example in Figure 5, the class C2 has a very low fan-in (one incoming edge) but a high afferent impact value (five classes are affected by changing C2).

We define the two way impact metric (TWI) of a class C as follows:

$$TWI(c) = \begin{cases} \dfrac{CAI(c)}{|S|} \times \dfrac{Log(\frac{|S|}{CEI(c)})}{Log(|S|)} & if \quad CEI \mathrel{!}= 0 \\[2ex] \dfrac{CAI(c)}{|S|} \quad if \; CEI = 0 \end{cases}$$

If a class does not depend on any other class (i.e., CEI = 0) then TWI is the same as the one way impact metric. The interesting case is when CEI is different from zero. In this case $\frac{CAI(c)}{|S|}$ reflects the fact that the classes with large CAI (class afferent impact) are the ones that are most likely to be non-feature specific classes, as previously discussed. Through $Log(\frac{|S|}{CEI(c)})$ the metric also takes into account the efferent impact although with a lower weight than the afferent impact using the Log function. The reason behind this is that we believe that the afferent impact should be weighted more than the efferent impact since a class modification that causes a considerably large number of changes in the system should be classified as utility no matter what the value of its efferent impact is.

We divide the result of both parts by $Log(|S|)$ to ensure that the entire formula varies from 0 to 1, with 0 being a component that is feature specific and not shared by any component in the system and 1 being a component that is shared among all components in the system.

When applying the metrics to the running example, the two way impact metric favoured the class C6 over the class C4 as being the most relevant to the feature under study. This is because C4 does not depend on any other class (CEI = 0), whereas C6 depends on two classes (CEI = 2), which might suggest that it is more important than C4. This classification is not necessarily incorrect since utility classes might also have a local scope. For example, C4 could be a utility class for the P1 package, whereas C6 is a utility class for the entire system. Therefore, having these two classes at the bottom of the table should be seen as a good outcome of the algorithm.

Table 2. Applying TWI to example of Figure 4

Package	Class	CAI	CEI	TWI
P1	C1	0	11	0.000
P1	C2	1	6	0.018
P1	C5	1	5	0.023
P1	C3	1	2	0.046
P1	C12	2	1	0.120
P2	C6	7	2	0.325
P1	C4	5	0	0.417
P2	C8	8	1	0.481
P2	C7	9	0	0.750

Next, we introduce our *Weighted* OWI and *Weighted* TWI metrics to improve on the OWI and TWI metric by also considering the package afferent impact as part of the measurement.

3.4.2.4 The Weighted One Way Impact Metric (WOWI)

The Weighted One Way Impact metric uses available information about the system architecture to further enhance the already introduced one way impact metric. More specifically, for the WOWI metric, also the number of packages is considered that are affected by a class modification (i.e., the package afferent impact, PAI). The rationale behind this is that a class affecting more packages (that is affecting classes belonging to majority of packages in the system) is more likely to be a feature irrelevant class in comparison to a class affecting less number of packages. For example, a class that affects five classes from three different packages will more likely be included in the execution profile of several features than a class that affects five classes of the same package. In other words, two classes that have the same OWI value may be ranked differently if they affect a different number of packages, and in such a case, the component that crosses the least number of packages will be given more importance than the one that affects a larger number of packages.

Given the above, we can introduce now the following metric:

$$WOWI(c) = OWI(c) x \frac{PAI(c)}{|P|}$$

- P is a set that contains the packages of the system.

The range for the $WOWI(c)$ is from 0 to 1, with 0 being a component that is feature specific and not shared by any other component or package in the system, and 1 being a component that is shared among all components and packages of the system.

Table 3. Applying WOWI on the example of Figure 4

Package	Class	CAI	PAI	WOWI
P1	C1	0	1	0.000
P1	C2	1	1	0.017
P1	C5	1	1	0.017
P1	C3	1	1	0.017
P1	C12	2	1	0.033
P1	C4	5	1	0.083
P2	C6	7	5	0.583
P2	C8	8	5	0.667
P2	C7	9	5	0.750

The weighted one way impact metric results in a similar outcome as the non-weighted one way metric when applied to the running example (see Table 3). However, it should be noticed that the gap between the relevant classes (P1 classes) and the non-relevant classes (P2 classes) is considerably larger than the gap between these two categories of classes when we applied the non weighted OWI.

3.2.2.5 The Weighted Two Way Impact (WTWI)

Similar to the Weighted One Way Impact metric, the weighted two way impact (WTWI) can be seen as a further improvement over the two way impact metric by also considering the number of packages affected due to a component modification.

The WTWI metrics therefore corresponds to:

$$WTWI(c) = TWI(c) x \frac{PAI(c)}{|P|}$$

Table 4 shows the result after applying the WTWI to the example in Figure 4. As a result of using the WTWI metric, class C4 was placed back in the pool of relevant classes. This is because it only affects one package as opposed to the classes of P2 package which affect five packages. In addition, the WTWI improves over the non-weighted TWI by enlarging the gap between the relevant and non-relevant classes.

Table 4. Applying WTWI on the example of Figure 4

Package	Class	CAI	CEI	PAI	WTWI
P1	C1	0	11	1	0.000
P1	C2	1	6	1	0.004
P1	C5	1	5	1	0.005
P1	C3	1	2	1	0.009
P1	C12	2	1	1	0.024
P1	C4	5	0	1	0.083
P2	C6	7	2	5	0.325
P2	C8	8	1	5	0.481
P2	C7	9	0	5	0.750

3.5 Summary

In this section, we presented our approach for addressing the feature location by focusing on the identification of the classes that are the most relevant to the feature to be analyzed.

Our approach combines both static and dynamic analysis. A trace is generated by exercising a feature under study. The invoked classes in the trace are ranked based on identifying the impact of a class modification on the rest of the system. Our hypothesis is that the higher the impact, the less relevant the component. To measure the impact of a component modification on the rest of the system, we proposed four impact metrics that operate on the class dependency graph. The first metric, the One Way Impact Metric (OWI), considers only the impact of a class modification on the rest of the system. The second metric, the Two Way Impact metric (TWI), considers both, the impact of a class modification on the rest of the system (i.e., its afferent impact), as well as the number of classes that impact this class if these classes change (i.e., the efferent impact, CEI, of the class). The two other metrics, the Weighted One Way Impact (WOWI) metric and the Weighted Two Way Impact (WTWI) metric, both use architectural information to further refine the OWI and TWI metrics.

4. Evaluation

In this section, we present an initial evaluation of our feature location techniques. The objective of the study is to investigate the following questions:

Q1: How effective the above feature location techniques are in their ability to detect the components that are most relevant to a specific feature?

Q2: What is the difference between applying the various impact metrics presented in the paper?

For this purpose, we applied our approach to traces generated from two open source object-oriented software systems described in the following section.

4.1 Target Systems

We apply the proposed feature location techniques on feature traces generated for two Java-based systems called Weka[1] (Vers.3.0), and Checkstyle[2] (Vers. 3.3). Weka has been developed at the University of Waikato, New Zealand. It is a machine learning tool that supports several algorithms such as classification algorithms, regression techniques, clustering and association rules. It consists in the total of 10 packages, 142 classes, and 95 KLOC.

[1] http://www.cs.waikato.ac.nz/ml/weka/ last visit: July 2008
[2] http://checkstyle.sourceforge.net/ last visit: July 2008

The second system, Checkstyle, which is a development tool to guide programmers in writing Java code that adheres to a specified coding standard. The tool allows programmers to create XML-based files to represent almost any coding standard. Checkstyle uses ANTLR[3] (ANother Tool for Language Recognition) and the Apache regular expression pattern matching package[4]. These two packages have been excluded from this analysis. Checkstyle (without ANTLR and the Apache module) has 17 packages, 210 classes, and 130 KLOC.

We selected Weka and Checkstyle because both systems are well documented. For Weka, packages and most important classes are documented in a book dedicated to the tool and machine learning in general [33]. A detailed description of the Checkstyle architecture can be found on the tool website. Having this documentation available enabled us to validate the results obtained from our approach against the documented feature implementations.

4.2 Applying Feature Locations Algorithms

4.2.1 Feature Selection

We applied our feature location techniques to two software features, one for each system. For the Weka system, we applied our approach to identify the classes that are specific to the implementation of the M5 algorithm. The M5 algorithm is a classification algorithm based on the so-called model trees [26]. For the Checkstyle system, we selected the CheckCode feature that is used to check Java code for coding problems such as uninitialized variables, etc.

4.2.2 Generation of Feature-Traces

To generate the corresponding traces, we instrumented Weka and CheckStyle using our own instrumentation tool based on the Bytecode Instrumentation Toolkit framework [20]. Probes were inserted at each entry and exit method (including constructors) of both systems. For each feature discussed in the previous section, we generated two execution traces, which correspond to the selected features, by executing the instrumented version of Weka and CheckStyle. We used as input for exercising the features sample data provided in the corresponding system documentations. In our approach there is no need to store the entire feature trace, instead we only

[3] http://www.antlr.org/ last visit: July 2008
[4] http://jakarta.apache.org/regexp/ last visit: July 2008

store the distinct classes invoked. Table 5 shows the number of distinct classes invoked in M5 and CheckCode traces.

Table 5. Distinct Classes in the traces for M5 and Checkcode.

Feature (System)	**Number of Classes**
M5 (Weka)	26
CheckCode (Checkstyle)	68

4.2.3 Applying the Impact Metrics

For the evaluation we used the Structural Analysis for Java (SA4J)[5] tool to parse the source code and generate a global class dependency table that contains various metrics including the class afferent and efferent impacts. SA4J supports also a large spectrum of relations among classes such as: accesses, calls, contains, extends, implements, instantiates, references, etc.

In addition, the tool provides architectural information of a system such as the number of packages, the content of each package, and the relationships between packages. We used the package dependency graphs to compute the package afferent impact, which is needed for the computation of the weighted one way impact and weighted two way impact metrics. In the following subsections, we present the result of applying our feature location approach based on the OWI, WOWI, TWI, and WTWI metrics.

4.2.3.1 The One Way Impact Metric (OWI)

Table 6 shows the result of applying the OWI metric (in ascending order of OWI) for the Weka feature trace M5. We use bold font to show classes that are ranked as most relevant component, italics font for the related components, and underline font to represent utility components.

The execution profile of the M5 feature (i.e., the distinct classes invoked in the M5 feature trace) consists of classes that belong to the following packages: `m5` (12 classes), `classifiers` (2 classes), `filters` (3 classes), `estimators` (1 class), and `core` (8 classes).

[5] http://www.alphaworks.ibm.com/tech/sa4j last visit: August 2008.

Table 6. Applying OWI to M5 feature

Package	Class	|CAI|	OWI*1000
weka.classifiers.m5	**M5Prime**	**1**	7.04
weka.classifiers.m5	**Node**	**2**	14.08
weka.classifiers.m5	**Options**	**3**	21.13
weka.classifiers.m5	**SplitInfo**	**3**	21.13
weka.classifiers.m5	**Function**	**3**	21.13
weka.classifiers.m5	**Errors**	**4**	28.17
weka.classifiers.m5	**Ivector**	**4**	28.17
weka.classifiers.m5	**Dvector**	**4**	28.17
weka.classifiers.m5	**Impurity**	**4**	28.17
weka.classifiers.m5	**Values**	**5**	35.21
weka.classifiers.m5	**Matrix**	**7**	49.30
weka.filters	*ReplaceMissingValuesFilter*	*7*	49.30
weka.filters	*NominalToBinaryFilter*	*7*	49.30
weka.classifiers.m5	**M5Utils**	**10**	70.42
weka.filters	*Filter*	*33*	232.39
weka.classifiers	*Evaluation*	*33*	232.39
weka.core	Queue	34	239.44
weka.classifiers	*Classifier*	*35*	246.48
weka.estimators	*KernelEstimator*	*37*	260.56
weka.core	Statistics	49	345.07
weka.core	Instances	108	760.56
weka.core	Instance	108	760.56
weka.core	Attribute	109	767.61
weka.core	Utils	126	887.32
weka.core	FastVector	127	894.37

According to the Weka documentation [33], the package `m5` contains the key classes that implement the M5 model tree algorithm. The `classifiers` package (excluding its sub-packages) contains common classes that most Weka classifications algorithms (including M5) use. The classes defined in the `filters` package are used to extract the data used by the Weka classification algorithms. The classes used by the M5 algorithm are: `NominalToBF` and `ReplaceMissingVF`. The class `Filter` is a superclass from which all filters are inherited. The `estimators` package contains classes that implement various techniques for estimating the machine learning models used by Weka classification algorithms. The class `KernelEstimator` invoked in the M5 trace is used by M5 and many other classification algorithms as well. Finally, the `core` package contains general-purpose utilities used by all Weka algorithms whether they are classification algorithms or not.

By further analyzing the Weka documentation we were able to verify that the OWI-based feature location technique ranked successfully most of the M5 specific classes (shown in bold) as relevant components, except for the class `M5Utils`. It also grouped at the bottom of the table most classes of the utility package `core` (underlined). The classes shown in Italics represent the related components, used by M5 and some other Weka's classification algorithms.

Although the OWI-based feature location technique produced good results, a closer look at the values of OWI revealed that the value for classes `Matrix` from the m5 package, and `ReplaceMissingValuesFilter` and `NominalToBinaryFilter` from the filterers package are identical (OWI = 49.30/1000). In other words, the algorithm did not provide a clear cut between the relevant and the related components.

For the CheckCode feature we followed a similar assessment process. The execution profile of the CheckCode feature revealed that it consists of classes belonging to the following packages: `coding` (32 classes), `checkstyle` (12 classes), `checks` (5 classes), `grammars` (2 classes), and `apis` (17 classes). As for Weka, we consulted the documentation of Checkstyle to understand the most components of the CheckCode feature. The package `coding` is the one that contains the key classes that implement the various checking procedures most relevant to this feature.

Table 7, sorted in ascending OWI order , shows the result after applying the OWI metric to the Checkstyle feature trace CheckCode (note some classes are omitted to avoid cluttering the table). From Table 7, one can observe that the OWI-based feature location ranked successfully most of the CheckCode feature specific classes (shown in bold). The only major exceptions are the classes: `AbstractSuperCheck` and `AbstractNestedDepthCheck.` These classes have a large class afferent impact (CAI = 3) compared to all other classes of the `coding` package (CAI = 1). This is due to the fact that they are abstract classes, and as such, they implement general purpose functions used by many other classes.

Table 7. Applying OWI to CheckCode Feature

Package	Class	|CAI|	OWI	OWI*1000
coding	**ExplicitInitializationCheck**	**1**	**0**	4.76
28 other classes of the coding package are omitted here, all of them with a OWI value of 4.76				
coding	**StringLiteralEqualityCheck**	**1**	**0**	4.76
checkstyle	*DefaultConfiguration*	*1*	*0*	4.76
checks	*DescendantTokenCheck*	*2*	*0.01*	9.52
checks	*GenericIllegalRegexpCheck*	*2*	*0.01*	9.52
checkstyle	*Checker*	*3*	*0.01*	14.29
coding	**AbstractSuperCheck**	**3**	**0.01**	14.29
coding	**AbstractNestedDepthCheck**	**3**	**0.01**	14.29
checkstyle	*DefaultLogger*	*3*	*0.01*	14.29
checkstyle	*TreeWalker*	*3*	*0.01*	14.29
checkstyle	*ConfigurationLoader*	*3*	*0.01*	14.29
checkstyle	*PropertiesExpander*	*3*	*0.01*	14.29
checks	*AbstractTypeAwareCheck*	*3*	*0.01*	14.29
checkstyle	*PackageNamesLoader*	*4*	*0.02*	19.05
checkstyle	*PropertyCacheFile*	*4*	*0.02*	19.05
checkstyle	*StringArrayReader*	*4*	*0.02*	19.05
grammars	*GeneratedJava14Lexer*	*4*	*0.02*	19.05
grammars	*GeneratedJava14Recognizer*	*4*	*0.02*	19.05
checkstyle	*PackageObjectFactory*	*5*	*0.02*	23.81
apis	FilterSet	6	0.03	28.57
checkstyle	*DefaultContext*	*7*	*0.03*	33.33
checkstyle	*AbstractLoader*	*7*	*0.03*	33.33
checks	*CheckUtils*	*8*	*0.04*	38.10
apis	AbstractFileSetCheck	8	0.04	38.10
apis	TokenTypes	9	0.04	42.86
apis	AuditEvent	13	0.06	61.90
checks	*AbstractFormatCheck*	*17*	*0.08*	80.95
apis	ScopeUtils	19	0.09	90.48
11 other classes of the apis package are omitted here, all of them with an OWI value ranging from 95.24 to 704.76				
apis	SeverityLevel	150	0.71	714.29

The packages `checkstyle`, `checks`, and `grammars` contain classes that implement common functionality used by most checks performed within Checkstyle. For example the classes of the `grammars` package contain operations that build a grammar from the code inputted for analysis. Most of these classes, represented in Italics, are ranked after classes of the coding package with a few exceptions. For example, the class `checkstyle.DefaultConfiguration`, `checks.DescendantTokenCheck`, `checks.GenericIllegalRegexpCheck` and `checkstyle.checker` were ranked among the most important classes. These classes are not as important as those of the `coding` package as they are used by many features of the Checkstyle tool. In Table 7, these classes are shown in Italics, indicating that they are neither specific to the CheckCode feature nor utility classes. The One Way Impact metrics also detected successfully the utility classes, which are packaged in the `apis` package, with a few exception, such as the class `FilterSet`, which was misplaced by our approach.

Table 8. Applying WOWI to M5 feature

Packages	Class	|CAI|	|PAI|	WOWI*1000
weka.classifiers.m5	**M5Prime**	**1**	**1**	0.70
weka.classifiers.m5	**Node**	**2**	**1**	1.41
weka.classifiers.m5	**Options**	**3**	**1**	2.11
weka.classifiers.m5	**SplitInfo**	**3**	**1**	2.11
weka.classifiers.m5	**Function**	**3**	**1**	2.11
weka.classifiers.m5	**Errors**	**4**	**1**	2.82
weka.classifiers.m5	**Ivector**	**4**	**1**	2.82
weka.classifiers.m5	**Dvector**	**4**	**1**	2.82
weka.classifiers.m5	**Impurity**	**4**	**1**	2.82
weka.classifiers.m5	**Values**	**5**	**1**	3.52
weka.classifiers.m5	**Matrix**	**7**	**1**	4.93
weka.classifiers.m5	**M5Utils**	**10**	**1**	7.04
weka.filters	*ReplaceMissingValuesFilter*	*7*	*3*	14.79
weka.filters	*NominalToBinaryFilter*	*7*	*3*	14.79
weka.filters	*Filter*	*33*	*4*	92.96
weka.classifiers	*Evaluation*	*33*	*4*	92.96
weka.classifiers	*Classifier*	*35*	*4*	98.59
weka.core	Queue	34	5	119.72
weka.estimators	*KernelEstimator*	*37*	*5*	130.28
weka.core	Statistics	49	8	276.06
weka.core	Instances	108	8	608.45
weka.core	Instance	108	8	608.45
weka.core	Attribute	109	8	614.08
weka.core	Utils	126	9	798.59
weka.core	FastVector	127	9	804.93

Similar to Weka, the OWI-based feature location technique did not succeed to provide a clear cut between the different categories of classes (i.e., relevant components, related components, and utilities). For example, the OWI metric value for the classes from the `coding` package and `defaultconfiguration` from the `checkstyle` package are similar (OWI = 4.76/1000), although these two classes should be in different categories.

4.2.3.2 The Weighted One Way Impact Metric (WOWI)

Table 8 shows the result of applying WOWI metric to the M5 feature in Weka. As shown in Table 8, this approach can produce better results than the non-weighted OWI metric, by improving the grouping of the classes. More specifically, the grouping created with WOWI shows compared to the OWI metrics a wider gap and therefore a clearer distinction between the feature specific classes and the ones which are less relevant to the M5 feature.

Table 9. Applying WOWI to CheckCode feature

Package	Class	|CAI|	|PAI|	WOWI*1000
coding	**ExplicitInitializationCheck**	**1**	**1**	0.28
28 other classes of the coding package are omitted here, all of them with a WOWI value of 0.28				
coding	**StringLiteralEqualityCheck**	**1**	**1**	0.28
checkstyle	*DefaultConfiguration*	*1*	*1*	0.28
checkstyle	*Checker*	*3*	*1*	0.84
coding	**AbstractSuperCheck**	**3**	**1**	0.84
coding	**AbstractNestedDepthCheck**	**3**	**1**	0.84
checkstyle	*DefaultLogger*	*3*	*1*	0.84
checkstyle	*ConfigurationLoader*	*3*	*1*	0.84
checkstyle	*PropertiesExpander*	*3*	*1*	0.84
checks	*DescendantTokenCheck*	*2*	*2*	1.12
checks	*GenericIllegalRegexpCheck*	*2*	*2*	1.12
checkstyle	*PackageNamesLoader*	*4*	*1*	1.12
checkstyle	*PackageObjectFactory*	*5*	*1*	1.40
checkstyle	*TreeWalker*	*3*	*2*	1.68
checkstyle	*PropertyCacheFile*	*4*	*2*	2.24
checkstyle	*StringArrayReader*	*4*	*2*	2.24
checks	*AbstractTypeAwareCheck*	*3*	*3*	2.52
grammars	*GeneratedJava14Lexer*	*4*	*3*	3.36
grammars	*GeneratedJava14Recognizer*	*4*	*3*	3.36
checkstyle	*DefaultContext*	*7*	*2*	3.92
checkstyle	*AbstractLoader*	*7*	*2*	3.92
apis	FilterSet	6	3	5.04
checks	*CheckUtils*	*8*	*3*	6.72
apis	AuditEvent	13	3	10.92
apis	TokenTypes	9	5	12.61
apis	AbstractFileSetCheck	8	6	13.45
checks	*AbstractFormatCheck*	*17*	*4*	19.05
apis	ScopeUtils	19	7	37.25
11 other classes of the apis package are omitted here, all of these classes have a WOWI value ranging from 39.22 to 663.31.				
apis	SeverityLevel	150	16	672.27

Similar to the observation made for the Weka case study, the weighted OWI provides also better results than the non-weighted OWI when applied to the CheckCode feature example (Table 9). It not only improves the grouping of the classes of the coding package but also enlarges the gap between among the relevant and the related classes. More precisely, the gap between the misplaced classes (`AbstractSuperCheck` and `AbstractNestedDepthCheck`) of `coding` package and the rest of the `coding` package classes has been reduced. However, the problem of having the same WOWI metric value for the classes from the `coding` package and `defaultconfiguration` from the `checkstyle` package still remains, as in the previous technique. The WOWI metrics also shows an improvement in the ranking of classes in the `apis` package. Using the WOWI metrics classes are now group closer to each other at the bottom of the table. However, the approach also misplaced two classes `CheckUtils` and `AbstractFormatCheck` at the end with `apis` package utility classes.

Table 10. Applying TWI to M5 Feature

Packages	Class	|CEI|	|CAI|	TWI*1000
weka.classifiers.m5	**M5Prime**	**35**	**1**	**1.95**
weka.classifiers.m5	**Node**	**19**	**2**	**5.57**
weka.classifiers.m5	**Function**	**12**	**3**	**10.19**
weka.classifiers.m5	**SplitInfo**	**11**	**3**	**10.53**
weka.classifiers.m5	**Options**	**9**	**3**	**11.31**
weka.classifiers.m5	**Impurity**	**10**	**4**	**14.54**
weka.classifiers.m5	**Dvector**	**9**	**4**	**15.08**
weka.classifiers.m5	**Values**	**8**	**5**	**19.6**
weka.classifiers.m5	**Errors**	**1**	**4**	**24.23**
weka.classifiers.m5	**Ivector**	**1**	**4**	**24.23**
weka.filters	*ReplaceMissingValuesFilter*	*11*	*7*	*24.58*
weka.filters	*NominalToBinaryFilter*	*11*	*7*	*24.58*
weka.classifiers.m5	**Matrix**	**5**	**7**	**31.47**
weka.classifiers.m5	**M5Utils**	**8**	**10**	**39.2**
weka.classifiers	*Evaluation*	*18*	*33*	*94.32*
weka.filters	*Filter*	*10*	*33*	*119.95*
weka.classifiers	*Classifier*	*8*	*35*	*137.2*
weka.estimators	*KernelEstimator*	*7*	*37*	*151.23*
weka.core	Queue	1	34	205.95
weka.core	Statistics	1	49	296.81
weka.core	Instances	7	108	441.43
weka.core	Instance	7	108	441.43
weka.core	Attribute	5	109	490.08
weka.core	Utils	4	126	599.16
weka.core	FastVector	2	127	696.1

4.2.3.3 The Two Way Impact Metric (TWI)

Table 10 shows the result of applying the TWI metric to locate the M5 feature in the Weka system.

Table 11. Applying TWI to CheckCode feature.

Package	Class	|CAI|	|CEI|	TWI*1000
coding	**ExplicitInitializationCheck**	**1**	**22**	**1.97**
28 classes of the coding package are omitted here, all of these classes have a TWI value ranging from 1.92 to 2.14				
coding	**StringLiteralEqualityCheck**	**1**	**18**	**2.14**
checkstyle	*DefaultConfiguration*	*1*	*2*	*3.78*
checks	*DescendantTokenCheck*	*2*	*19*	*4.19*
checks	*GenericIllegalRegexpCheck*	*2*	*19*	*4.19*
checkstyle	*TreeWalker*	*3*	*32*	*4.94*
checkstyle	*Checker*	*3*	*21*	*6.03*
checks	*AbstractTypeAwareCheck*	*3*	*20*	*6.15*
coding	**AbstractSuperCheck**	**3**	**20**	**6.15**
coding	**AbstractNestedDepthCheck**	**3**	**18**	**6.42**
checkstyle	*DefaultLogger*	*3*	*11*	*7.65*
checkstyle	*ConfigurationLoader*	*3*	*3*	*10.58*
checkstyle	*PropertiesExpander*	*3*	*2*	*11.35*
checkstyle	*PackageNamesLoader*	*4*	*4*	*13.31*
grammars	*GeneratedJava14Lexer*	*4*	*3*	*14.11*
checkstyle	*PropertyCacheFile*	*4*	*2*	*15.13*
grammars	*GeneratedJava14Recognizer*	*4*	*2*	*15.13*
checkstyle	*StringArrayReader*	*4*	*1*	*16.58*
checkstyle	*PackageObjectFactory*	*5*	*2*	*18.92*
apis	FilterSet	6	5	19
apis	AbstractFileSetCheck	8	13	19.29
checkstyle	*DefaultContext*	*7*	*2*	*26.48*
checks	*CheckUtils*	*8*	*3*	*28.22*
checkstyle	*AbstractLoader*	*7*	*1*	*29.01*
checks	*AbstractFormatCheck*	*17*	*18*	*36.38*
apis	TokenTypes	9	1	37.3
13 classes of the apis package are omitted here, all of these classes have a TWI value ranging from 45.86 to 584.39				
apis	SeverityLevel	150	1	621.69

In general, the results achieved using the TWI metric did not provide any further improvement over the previous metrics, except for its ability to allow for a grouping of all utility classes within one block. As shown in Table 10, the `core` package classes form now one block located at the bottom of the table. The TWI metric however did misplace an additional relevant component (the class `m5.Matrix`). The overall TWI distribution presents an improvement compared to the previous metrics, with the TWI distribution showing a better ranking of the components from the most relevant to the least relevant, with the exception of the few misplaced classes.

When applied to the CheckCode feature (see Table 11), the results obtained are similar to the one obtained by the OWI and WOWI metrics, except a better classification of the utility classes (i.e., the ones belonging to the `apis` package) was achieved. For example, the classes `apis.TokenType` and `apis.AuditEvent`, which were both misplaced using the OWI and WOWI metric, were correctly grouped by the TWI metrics with the other `apis` classes.

Table 12. Applying WTWI to M5 feature

Packages	Class	|CEI|	|CAI|	|PAI|	WTWI*1000
weka.classifiers.m5	**M5Prime**	**35**	**1**	**1**	**0.2**
weka.classifiers.m5	**Node**	**19**	**2**	**1**	**0.56**
weka.classifiers.m5	**Function**	**12**	**3**	**1**	**1.02**
weka.classifiers.m5	**SplitInfo**	**11**	**3**	**1**	**1.05**
weka.classifiers.m5	**Options**	**9**	**3**	**1**	**1.13**
weka.classifiers.m5	**Impurity**	**10**	**4**	**1**	**1.45**
weka.classifiers.m5	**Dvector**	**9**	**4**	**1**	**1.51**
weka.classifiers.m5	**Values**	**8**	**5**	**1**	**1.96**
weka.classifiers.m5	**Errors**	**1**	**4**	**1**	**2.42**
weka.classifiers.m5	**Ivector**	**1**	**4**	**1**	**2.42**
weka.classifiers.m5	**Matrix**	**5**	**7**	**1**	**3.15**
weka.classifiers.m5	**M5Utils**	**8**	**10**	**1**	**3.92**
weka.filters	*ReplaceMissingValuesFilter*	*11*	*7*	*3*	*7.37*
weka.filters	*NominalToBinaryFilter*	*11*	*7*	*3*	*7.37*
weka.classifiers	*Evaluation*	*18*	*33*	*4*	*37.73*
weka.filters	*Filter*	*10*	*33*	*4*	*47.98*
weka.classifiers	*Classifier*	*8*	*35*	*4*	*54.88*
weka.estimators	*KernelEstimator*	*7*	*37*	*5*	*75.62*
weka.core	Queue	1	34	5	102.97
weka.core	Statistics	1	49	8	237.45
weka.core	Instances	7	108	8	353.15
weka.core	Instance	7	108	8	353.15
weka.core	Attribute	5	109	8	392.06
weka.core	Utils	4	126	9	539.24
weka.core	FastVector	2	127	9	626.49

4.2.3.4 The Weighted Two Way Impact Metric (WTWI)

Table 12 shows the result of applying the feature location techniques based on the WTWI metric. The results achieved by this metric are considerably better than the ones obtained by the previous approaches. One can observe (Table 12) that the classes of the `m5` package were all identified and grouped together as the most relevant classes to the M5 feature. Also, the `core` package utility classes were grouped together at the end showing that these classes are least important for this feature. As a result, using the WTWI, none of the classes were misplaced.

Table 13. Applying WTWI to CheckCode feature

Package	Class	|CAI|	|CEI|	|PAI|	WTWI*1000
coding	**ExplicitInitializationCheck**	**1**	**22**	**1**	**0.12**
28 other classes of the coding package are omitted here, all of these classes have a WTWI value ranging from 0.12 to 0.13.					
coding	**StringLiteralEqualityCheck**	**1**	**18**	**1**	**0.13**
checkstyle	*DefaultConfiguration*	*1*	*2*	*1*	*0.22*
checkstyle	*Checker*	*3*	*21*	*1*	*0.35*
coding	**AbstractSuperCheck**	**3**	**20**	**1**	**0.36**
coding	**AbstractNestedDepthCheck**	**3**	**18**	**1**	**0.38**
checkstyle	*DefaultLogger*	*3*	*11*	*1*	*0.45*
checks	*DescendantTokenCheck*	*2*	*19*	*2*	*0.49*
checks	*GenericIllegalRegexpCheck*	*2*	*19*	*2*	*0.49*
checkstyle	*TreeWalker*	*3*	*32*	*2*	*0.58*
checkstyle	*ConfigurationLoader*	*3*	*3*	*1*	*0.62*
checkstyle	*PropertiesExpander*	*3*	*2*	*1*	*0.67*
checkstyle	*PackageNamesLoader*	*4*	*4*	*1*	*0.78*
checks	*AbstractTypeAwareCheck*	*3*	*20*	*3*	*1.09*
checkstyle	*PackageObjectFactory*	*5*	*2*	*1*	*1.11*
checkstyle	*PropertyCacheFile*	*4*	*2*	*2*	*1.78*
checkstyle	*StringArrayReader*	*4*	*1*	*2*	*1.95*
grammars	*GeneratedJava14Lexer*	*4*	*3*	*3*	*2.49*
grammars	*GeneratedJava14Recognizer*	*4*	*2*	*3*	*2.67*
checkstyle	*DefaultContext*	*7*	*2*	*2*	*3.12*
apis	FilterSet	6	5	3	3.35
checkstyle	*AbstractLoader*	*7*	*1*	*2*	*3.41*
checks	*CheckUtils*	*8*	*3*	*3*	*4.98*
apis	AbstractFileSetCheck	8	13	6	6.81
apis	AuditEvent	13	3	3	8.09
checks	*AbstractFormatCheck*	*17*	*18*	*4*	*8.56*
apis	TokenTypes	9	1	5	10.97
12 other classes of the apis package are omitted here,. Their WTWI value ranges from 27.6 to 530.51					
apis	SeverityLevel	150	1	16	585.12

The results of applying the WTWI-based feature location technique to the CheckCode feature are shown in Table 13. WTWI shows similar results as achieved by WOWI-based technique. However, the WTWI approach misplaced some classes of the `checkstyle` package such at the classes `AbstractLoader`, `AbstractFormatCheck` (two abstract classes), and `CheckUtils` with the `apis` classes. As mentioned previously, abstract classes seem to behave differently from the other classes. As for `CheckUtils`, it is a utility class whose scope is within the `checks` package, unlike the classes of the `apis` package, which are system-level utilities.

4.3 Discussion

In this section, we discuss how the evaluation of our approach addresses the questions that the study aimed to investigate.

Q1: How effective the above feature location algorithms are in their ability to detect the components that are most relevant to a specific feature?

The overall results achieved by our feature location techniques show that they can be effective in locating the components that implement a particular feature, which answers the first research question investigated by this evaluation. In terms of features, applying our approach for the Weka M5 feature showed better results compared to the Checkstyle CheckCode feature. This is perhaps due to the fact that the Weka system is considerably smaller that Checkstyle, and hence less complex.

Q2: What is the difference between applying the various impact metrics presented in the paper?

We were able to observe settle differences among the results generated by our four impact metrics.

One Way Impact Metric: This metric worked quite well in detecting most important components for the feature M5 and CheckCode. With the exception of just one specific class of M5 feature and two specific classes of CheckCode feature it provided us with a correct grouping of the classes. However, one of the major disadvantage of this metric is its inability to create a clear-cut difference among the different categories of classes (relevant, related and utilities). Secondly the metric also misplaced a few classes with high OWI value by grouping them with the utility classes.

Weighted One way Impact Metric: Using this technique an improvement in the grouping of the classes was achieved. Specifically it ranked all the most important classes of the M5 feature rightly at the top. It also reduced the gap between the misplaced CheckCode feature classes and the rest of the important classes.

Two Way Impact Metric: This technique showed no additional improvement in grouping of the relevant classes over the OWI and WOWI metrics. The classes that were initially misplaced by OWI Metric remained misplaced using the TWI metrics. However, the metric showed some improvement in grouping the utility classes closer together as one block. All the utility classes of the `Core` package were placed at the end and also the gap between misplaced `apis` package classes of CheckCode feature was reduced.

Weighted Two Way Impact Metric: For the feature M5 this technique showed the best results in comparison with the previous techniques. A clear cut difference was achieved between the different categories (relevant, related and utilities) of classes involved in the feature- trace of M5 feature. In case of CheckCode feature some classes still remained misplaced and this technique showed results similar to those achieved by WOWI Metric.

5. Conclusion and Future Work

In this paper, we presented a novel feature location technique that supports the mapping of features to code. We in particular focused on identifying automatically these classes that are implementing the code most relevant to the feature being analyzed.

Our approach is based on a combination of static and dynamic analysis. A trace is generated by exercising the feature under study (dynamic analysis). A static class dependency graph (static analysis) is then used to rank the classes invoked in the trace according to their relevance to the feature. We ranked the classes by measuring their impact on the rest of the system. The rationale behind measuring the impact is that classes with a small impact set are likely to be specific to the feature at hand, whereas classes with a large(r) impact set have typical multiple purposes, and hence are less feature specific.

Based on this hypothesis we introduced four new techniques to measure the impact set of a class modification on the rest of a system. Each metric has its own characteristics with respect to its ability to locate and group the identified classes based on their degree of feature relevance.

We applied our techniques to several features of the software systems Weka and Checkstyle. The overall results of our experimental evaluation were very promising. Not only were we able to identify the components involved in the feature, but also rank them according to their relevance to the traced features. Further, analysis of our metrics showed that our weighted techniques

(WOWI and WTWI) delivers overall better results compared to the non- weighted ones (OWI and TWI). In addition, our approach is very simple and almost completely automatic, requiring almost no human intervention.

The immediate future work consists of conducting further experimentation on other feature traces. In particular, we intend to target larger systems with less well designed architectures to further evaluate our approach.

There is also a need to determine a threshold above which to consider classes as relevant to the feature. We anticipate that each system might have its own threshold, and that software engineers will dynamically change the threshold depending on their expertise of the system. Therefore, an analysis tool that would support the techniques presented in this paper should allow enough flexibility for the users to change the threshold.

Finally, during the analysis of the results of the experimental studies, we noticed that abstract classes showed a different behaviour compared to other classes. They tend to have a higher afferent impact. There is a need to further investigate this behaviour and propose other means to rank abstract classes.

5. References

[1]. G. Antoniol, and Y. G. Gueheneuc, "Feature Identification: An Epidemiological Metaphor", *IEEE Transactions on Software Engineering,* 32(9), pp.627-641, 2006.

[2]. D. F. Bacon, and P. F. Sweeney, "Fast static analysis of C++ Virtual function calls", *In Proc. of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, pp. 324-341, 1996.

[3]. D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, "An empirical study of the relationship between the concepts expressed in source code and dependence", *Journal of Systems and Software, 81(12)*, pp. 2287-2298, 2008.

[4]. R. Brooks, "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies, 18(6)*, pp. 542-554, 1983.

[5]. B. Calder, and D. Grijnwald, "Reducing indirect function call overhead in C++ programs". *In Proc. of the 21st ACM Symposium on Principles of Programming Languages (POPL),* ACM Press, pp. 397-408, 1994.

[6]. J. Dean, D. Grove, and Chambers, "Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis", *In Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, Springer-Verlag, pp. 77-101, 1995.

[7]. S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science,* 41(6), pp. 391-407, 1990.

[8]. T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering, 29(3),* pp. 210 – 224, 2003.

[9]. D. Eisenberg, and K. De Volder, "Dynamic feature traces: finding features in unfamiliar code", *In Proc. of the 21st IEEE International Conference on Software Maintenance,* pp. 337-346, 2005.

[10]. N. E. Gold, M. Harman, Z. Li, K. Mahdavi, "Allowing Overlapping Boundaries in Source Code using a Search Based Approach to Concept Binding", *In Proc. of the 22nd Conference on Software Maintenance and Evolution*, pp. 310-319, 2006.

[11]. N. E. Gold, M. Harman, D. Binkley, R. M. Hierons, "Unifying program slicing and concept assignment for higher-level executable source code extraction*", Journal of Software - Practice & Experience, 35(10),* pp. 977-1006, 2005

[12]. O. Greevy, S. Ducasse, and T. Girba, "Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis", *In Proc. of 21st IEEE International Conference on Software Maintenance*, pp. 347-356, 2005.

[13]. A. Hamou-Lhadj, and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the 12th International Conference on Program Comprehension,* pp. 181-190, 2006.

[14]. A. Hamou-Lhadj, and T. Lethbridge, "Reasoning About the Concept of Utilities", *ECOOP International Workshop on Practical Problems of Programming in the Large, Lecture Notes in Computer Science (LNCS), Vol 3344*, pp. 10-22, 2004.

[15]. A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, "Recovering Behavioral Design Models from Execution Traces", *In Proc. of the IEEE European Conference on Software Maintenance and Reengineering*, pp. 112-121, 2005.

[16]. J. Koenemann, S. Robertson. Expert Problem Solving Strategies for Program Comprehension, *Human Factors in Computing Systems*, ACM Press, 1991

[17]. J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh, "On Computing the Canonical Features of Software Systems", *In Proc. of the 13th IEEE Working Conference on Reverse Engineering*, pp. 93-102, 2006.

[18]. M. Lanza, R. Marinescu. Object-Oriented Metrics in Practice, Springer, 2006.

[19]. J. Law, G. Rothermel, "Whole program Path-Based dynamic impact analysis", *In Proc. of the 25th International Conference on Software Engineering,* pp. 308-318, 2003.

[20]. H. Lee, B. G. Zorn, BIT, "A tool for Instrumenting Java Bytecodes", *In Proc. of the USENIX Symposium on Internet technologies and Systems,* pp. 73-82, 1997.

[21]. R. Martin, "Object Oriented Design Quality Metrics: An Analysis of dependencies", ROAD, Vol. 2, No. 3, Sep-Oct, 1995.

[22]. A. Rohatgi, A. Hamou-Lhadj, J. Rilling, "An Approach for Mapping Features to Code Based on Static and Dynamic Analysis", *In Proc. of the 16th IEEE International Conference on Program Comprehension*, pp. 236-241, 2008.

[23]. A. Rohatgi, A. Hamou-Lhadj, and J. Rilling, "Feature Location based on Impact Analysis" *In Proc. of the Conference on Software Engineering and Applications*, ACTA Press, 2007.

[24]. D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering,* 33(6), pp.420-432, 2007.

[25]. D. Poshyvanyk, and D. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", *In Proc. of 15th IEEE International Conference on Program Comprehension*, pp. 37-48, 2007.

[26]. J. R. Quinlan, "Learning with continuous classes", *In Proc. of the 5th Australian Joint Conference on Artificial Intelligence*, pp 343-348, 1992.

[27]. M. P. Robillard, and G. C. Murphy, "FEAT: A tool for locating, describing, and analyzing concerns in source code," *In Proc. of the 25th International Conference on software Engineering,* pp. 822-823, 2003.

[28]. M. Salah, and S. Mancoridis, "A hierarchy of dynamic software views: from object-interactions to feature-interactions", *In Proc. of the 20th IEEE International Conference on Software Maintenance,* pp. 72-81, 2004.

[29]. R. J. Turver and M. Munro, "An Early Impact Analysis Technique for Software Maintenance", *Journal of Software Maintenance: Research and Practice, 6(1),* pp. 35-52, 1994.

[30]. UML 2.0 Specification: www.omg.org/uml. Last visit: December 2008

[31]. N. Wilde , M. Buckellew, H. Page , V. Rajlich , L. Pounds, "A comparison of methods for locating features in legacy software", *Journal of Systems and Software, 65(2),* pp.105-114, 2003.

[32]. N. Wilde, and M. Scully, "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice, 7(1)*, pp. 49-62, 1995.

[33]. H. Witten, and E. Frank. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, 1999.

[34]. W. E. Wong, and S. Gokhale, "Static and dynamic distance metrics for feature-based code analysis", *Journal of Systems and Software, 74(3),* pp. 283-295, 2005.

[35]. W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi, "Locating program features using execution slices", *In Proc. of Application-Specific Systems and Software Engineering and Technology,* pp. 194 – 203, 1999.

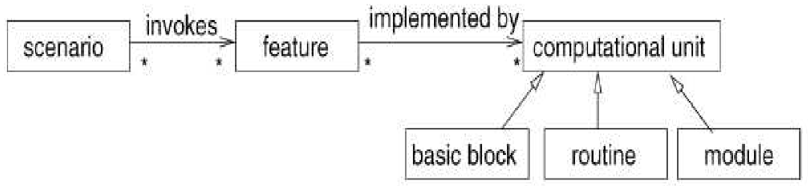

Figure 1. Relationship between software feature, scenario, and computational units [8]

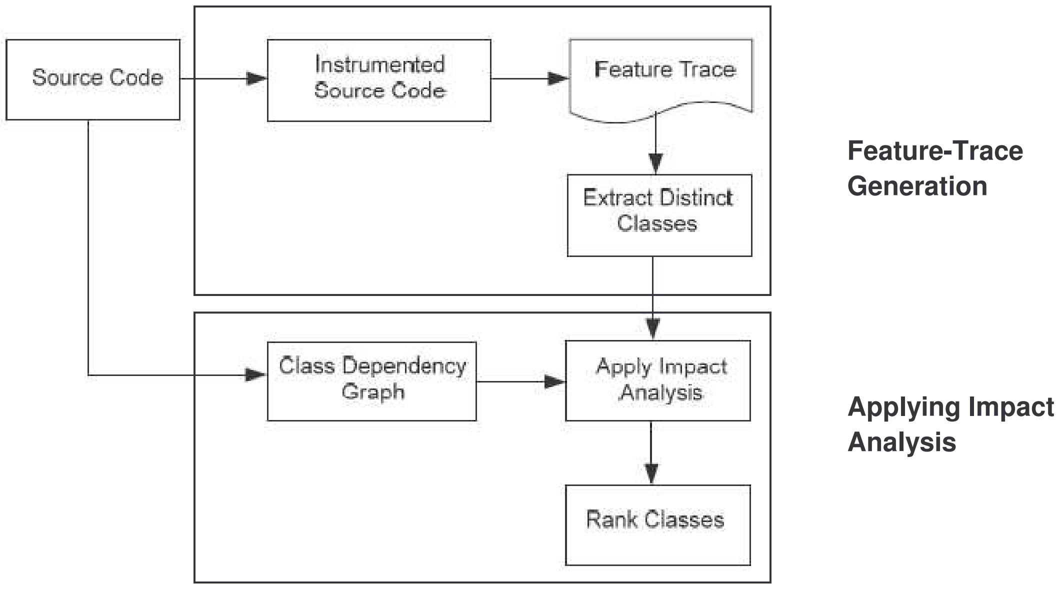

Figure 2. Overall Approach

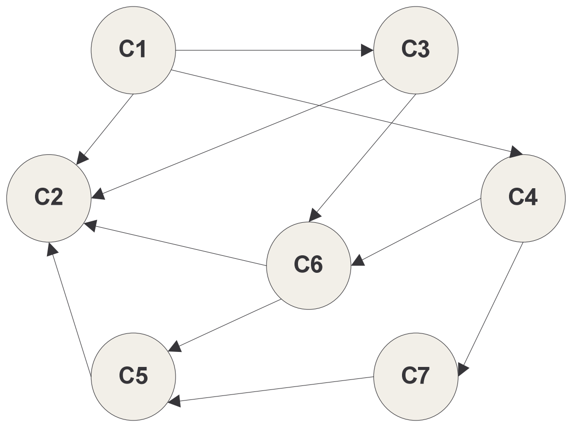

Figure 3. Example of a class dependency graph

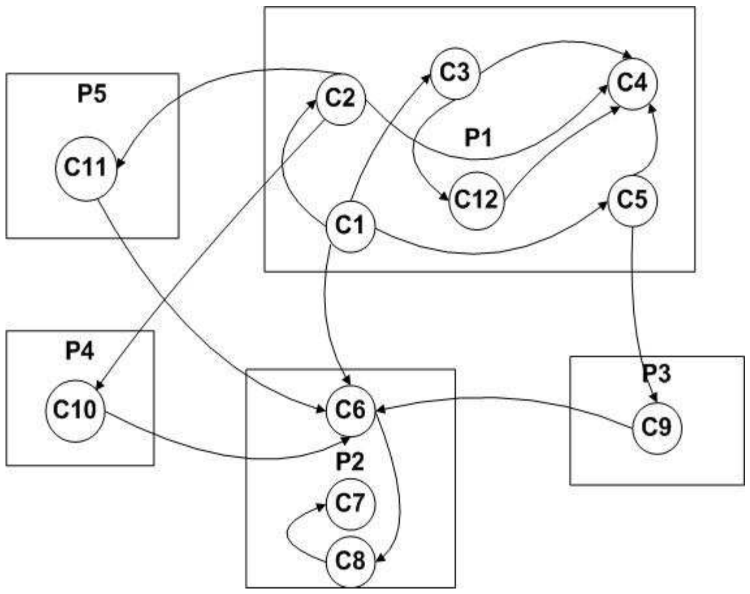

Figure 4. A class dependency graph

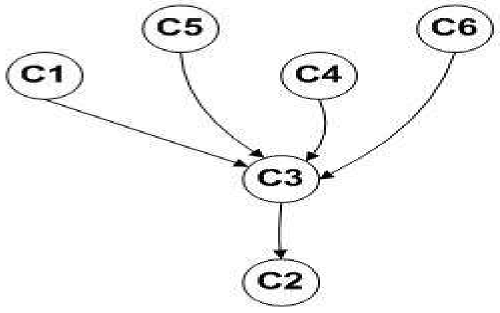

Figure 5. The difference between fan-in and CAI

Fan-in (C2) = 1 and CAI (C2) = 5