An Approach towards Feature Location Based on Impact Analysis

Abhishek Rohatgi

A Thesis

In

The Department

Of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2008

Your file Votre référence
ISBN: 978-0-494-45500-5
Our file Notre référence
ISBN: 978-0-494-45500-5

ABSTRACT

An Approach towards Feature Location Based on Impact Analysis

Abhishek Rohatgi

System evolution depends greatly on the ability of a maintainer to locate these parts of the source code that implement specific features. Until recently, quite a number of feature location techniques have been proposed. These techniques suffer from a number of limitations. They either require exercising several features of the system, or rely heavily on domain experts to guide the feature location process.

In this thesis, we present a novel approach for feature location that combines static and dynamic analysis techniques. An execution trace is generated by exercising the feature under study (dynamic analysis). A component dependency graph (static analysis) is used to rank the components invoked in the trace according to their relevance to the feature. Our ranking technique is based on the impact of a component modification on the rest of the system. We hypothesize that the smaller the impact of a component modification, the more likely it is that this component is specific to the feature. The proposed approach is automatic to a large extent relieving the user from any decision that would otherwise require extensive knowledge of the system.

We present a case study involving features from two software systems to evaluate the applicability and effectiveness of our approach.

Acknowledgements

During my Masters of Computer Science, I had the privilege to work with some expert and experienced professionals, which was truly an enriching experience.

My heartiest thanks to Dr. Abdelwahab Hamou-Lhadj, my supervisor, for showing confidence in me, providing me with initial spark for the topic, listening to my ideas no matter how vague they were and for constantly providing me support from his vast source of knowledge.

Many thanks to Dr. Juergen Rilling, my co-supervisor for giving me a positive response whenever I approached him, for extensively reviewing my research documents and also for providing me with incredible ideas on the topic.

In addition, I would like to thank my lab mates for their help. They took time out of their research to discuss with me about my topic and sometimes providing me useful tips and tricks to get the work done.

Lastly I would like to thank my family in India. Although, they do not have much of an understanding about what I do but they always backed me up emotionally and had been supportive throughout the course of my studies.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Feature location has long been recognized as an important reverse engineering activity to identify the implementation of a given system functionality in the source code. In this thesis, we present a powerful approach for solving the feature location problem using impact analysis. The presented approach combines two different sources of information: an execution trace that corresponds to the software feature under study and a static component dependency graph (CDG). Using the CDG, we rank the components invoked in the trace by measuring the impact of a component modification on the rest of the system. Our hypothesis is that the smaller the impact of a component modification, the more likely it is that the component is specific to the feature under study.

In the remainder of this chapter, we describe the main motivations behind the thesis, our contributions, and the thesis outline.

1.1 Problem and Motivation

System evolution, an important aspect of the software life cycle, depends on the ability of a maintainer to identify the parts of the source code that implement specific features. Software maintainers typically do not need to analyze an entire system before making modifications or adding new functionality, since required software changes often relate directly to features implementations [Wilde 03]. Instead, they apply an as needed

approach, by locating the most relevant code with respect to the feature or source code to be modified, understand it, and make the necessary changes. Due to a lack of traceability between documentation and source code locating these features in the source code becomes a major challenges for maintainers. This lack of feature traceability is caused by the unavailability of roundtrip engineering tools, lack of adequate processes in organizations to enforce consistent and up-to-date documentation, etc.

In an ideal situation, there should be a clear mapping between a system's features and the corresponding code segments. However, this is not the case for many existing systems where bad design decisions and/or excessive ad-hoc maintenance activities complicate this mapping. As a result, a feature is often distributed over several different modules that interact in complex ways, making, in particular for large systems, the identification of the source code implementing a particular feature inherently difficult.

One approach to support maintainers during activities like feature evolution, maintenance, reverse engineering and program comprehension is based on the use of feature location techniques that aim to provide maintainers with guidance in identifying and locating features in the source code [Brooks 83].

This idea of location of features in source code is not new. Existing feature location techniques can be grouped into two main categories depending on the use of static and dynamic analysis techniques. 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. An example of these techniques is the one proposed by Wiled and Scully [Wilde 95] and known as Software Reconnaissance. The major

limitation with these methods is that they require as input execution traces for all (most) system features to be generated and processed.

The second category relies on a combination of static and dynamic analysis. These approaches utilize static information to further process the execution trace that corresponds to the feature under study. For this purpose, several approaches were presented such as the ones based on concept analysis [Eisenbarth 03], latent semantic indexing [Deerwester 90], etc. The major limitation of these techniques is that they require from the user to indicate what parts of the source code to analyze, a task that can be tedious for software engineers who have little knowledge of the system.

In this thesis, we present a novel technique for feature location in source code that combines both static and dynamic analysis. Our technique operates on only one trace, which is generated by exercising the feature under consideration. In addition, the proposed approach is automatic to a large extent and therefore does not require users to have extensive system knowledge.

1.2 Research Contributions

The main contributions of this thesis are as follows:

- A novel idea of using impact analysis to solve the feature location problem is presented. The feature location techniques used in the presented approach are based on impact analysis. More precisely, components invoked in the execution trace for a given feature are ranked based on measuring the impact of component modifications 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 feature location algorithms that vary depending on 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 fourth metric refines the previous metric by adding information about the system architecture.

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

1.3 Thesis Outline

The rest of the thesis is structured as follows:

- In Chapter 2, we present background information, including a brief overview of related topics, namely, software maintenance, program comprehension, and reverse engineering. A detailed survey of existing feature location techniques is presented along with their advantages and limitations.

- The feature location algorithms are presented in Chapter 3. The chapter starts by presenting our definition of what constitutes in the context of our research a software feature. It continues with an overview of the feature location process. Next, we present impact analysis and how it is applied to locate features in source code. The four impact metrics are then presented and explained through an example and concludes with a discussion on the applicability of the presented metrics

- The evaluation of our approach is presented in Chapter 4. The chapter introduces the target systems used in the case study and describes the features on which we apply the algorithms. The chapter also covers the trace generation process and results of applying the feature location algorithms are described and discussed in details.

- We conclude the thesis in Chapter 5 with a summary of the main contributions, some future directions, and a concluding remark.

Chapter 2 Background

Feature location research pertains to three inter-related software engineering topics, namely, software maintenance, program comprehension and reverse engineering. In this chapter, we briefly describe these topics and discuss how they relate to feature location. We also present a survey of existing feature location techniques, followed with a discussion on the advantages and disadvantages of these techniques.

2.1 Related Topics

2.1.1 Software Maintenance

"Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. ... (We must) lose our preoccupation with the first release and focus on the long term health of our products."

D. L. Parnas [Parnas 94]

Software aging is one of the major reasons that trigger the need for software maintenance. Although software aging is inevitable, effective software maintenance can help slow down the aging process. According to IEEE Standard 1219, software maintenance is defined as a modification process that takes place after the delivery of

software in order to correct faults, improve performance or other attributes, or to make the program adaptable to new surroundings. The importance of software maintenance is evidenced from the fact that it constitutes of a major part of the overall software life cycle as shown by Lientz and Swanson in the late 1970's [Lientz 80].

Software evolution is often used as a substitute term for software maintenance. According to Bennett and Rajlich, the software maintenance phase starts after the software development phase. That is, maintenance activities generally take place after the software system is released. They further introduced the concept of a staged software lifecycle model in which development and maintenance were considered different phases in the software life cycle [Bennett 00].

Lientz and Swanson classified maintenance activities into four categories [Lientz 80]:

- Adaptive Maintenance: This type of maintenance includes user enhancements and modification to the existing software system to meet new user requirements.

- Perfective Maintenance: This involves making changes to the structure of the system in order to make it easier to extend, modify, and maintain.

- Corrective Maintenance: This type of maintenance deals with fixing software bugs in existing system functionality.

- Preventive Maintenance: This type of maintenance focuses on restructuring the existing system to prevent the system from bugs that may occur in the future.

Most of the above types of software maintenance activities require feature location techniques since changes to an existing system often relate to a particular feature. For example, Feature location techniques can be used to identify the source code components implementing a particular feature that led to a software defect. Knowing such components can help software engineers narrow down the space of components that need to be explored in order to repair the defective feature.

2.1.2 Program Comprehension

Rugaber defines program comprehension as a process of gaining knowledge about the system under study for the purpose of fixing a system's defect, enhancing the system, reusing and improving system's documentation [Rugaber 95]. According to a survey conducted by Fjeldstad and Hamlen, program comprehension accounts for 50% of the time spent on software maintenance activities [Fjeldstad 83].

According to Mayrhauser et al., program comprehension requires existing knowledge of a software system in order to acquire new knowledge [Mayrhauser 95]. Any newly acquired knowledge becomes then an integrated part of the system knowledge that is essential to support the understanding the program code. Based on their study, the authors conclude that software engineers possess two types of knowledge:

- General Knowledge: This type of knowledge is gained from past experience in the software engineering domain and is independent of the software under consideration.

- Software-Specific Knowledge: This knowledge represents their level of understanding of the software application under consideration.

A software engineer comprehending a system uses general software engineering knowledge together with the knowledge obtained from exploring the system under consideration in order to understand the system completely [Mayrhauser 95].

Program comprehension is not an easy task. This is partly attributable to the fact that existing documentation represents high-level views of the system whereas the implementation of the system contains many low-level programming details that are not necessary captured at a higher level [Rugaber 95]. The software engineer comprehending the system must map the high-level design elements to these low-level implementation details. This task is very challenging especially in situations where the initial high-level design documents have not been updated for a long time, which is commonly the case in practice [Rugaber 95]. Due to a lack of traceability between documentation and source code, it becomes a major challenge for a maintainer to identify a system's components that need to be analyzed in order to enhance an existing feature. This lack of feature traceability is caused by the unavailability of roundtrip engineering tools, lack of adequate processes in organizations to enforce consistent and up-to-date documentation, etc. The situation is further complicated by the fact that it is very common to have features spread across many system components that are not even tightly coupled. One of the main objectives of this thesis is to assist software engineers in the program comprehension process by allowing them to map automatically high-level software features to the specific components that implement them.

2.1.3 Reverse Engineering

Chicofsky and Cross define reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [Chicofsky 90].

Reverse engineering tools can be used by software engineers to facilitate the program comprehension process, and hence improve their productivity when performing software maintenance tasks.

Figure 2.1 (taken from [Chicofsky 90]), shows the relationship between forward and reverse engineering.

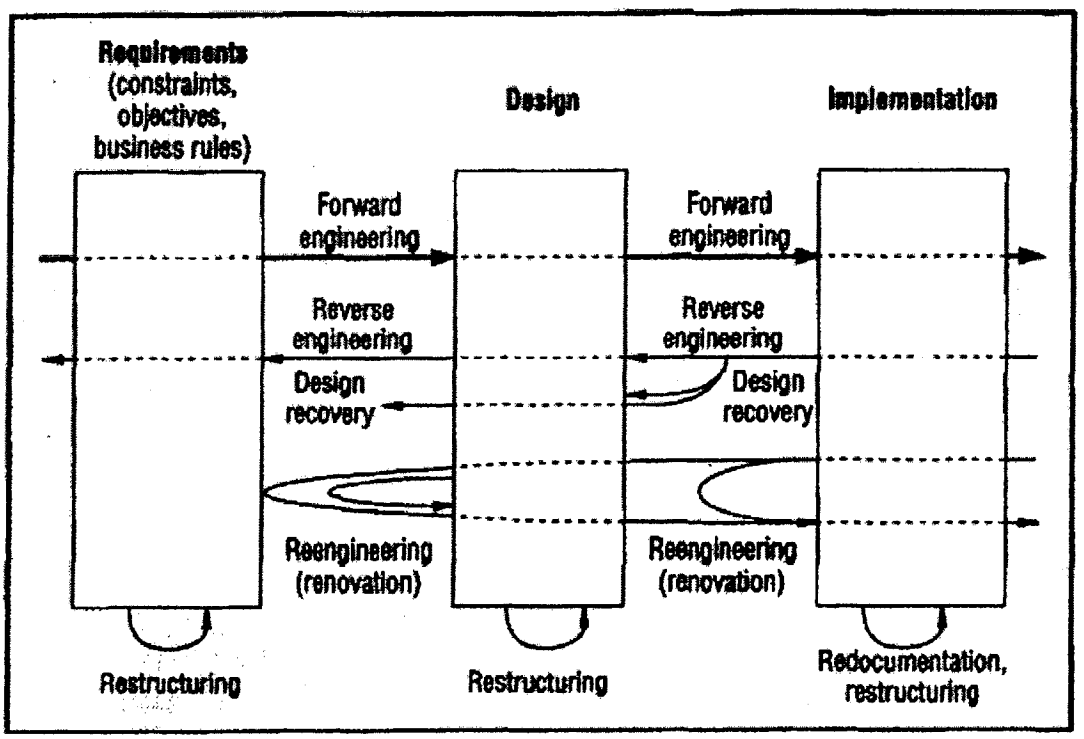

Figure 2.1. Relationship between forward and reverse engineering

- Forward Engineering: The process of taking higher-level design elements as input and transforming them into low-level implementation details. As shown in Figure 2.1, forward engineering involves a sequence of steps that map the requirements to design, and finally to implementation [Chicofsky 90].

- Reverse Engineering: It is the reverse of forward engineering, where high-level design elements are extracted from lower-level implementation details. As depicted in Figure 2.1, reverse engineering involves a sequence of recovery stages starting from implementation to design. Reverse engineering can be applied at any abstraction level of the system under consideration [Chicofsky 90].

- Re-documentation: Re-documentation is the creation of alternative views within the same abstraction level to assist the process of comprehending the lower level software details [Chicofsky 90].

- Design Recovery: This is one of the main activities of reverse engineering, and involves a combination of knowledge gained by analyzing the system and domain knowledge. The objective is to recover design views from low-level implementation details [Chicofsky 90].

- Restructuring: This process consists of making a change to the system at the same abstraction level (e.g., migrating an existing system from C to Java). The changes should not alter the external behaviour of the system [Chicofsky 90].

- Re-engineering: The basic goal of re-engineering is to renovate an existing software system to improve code comprehension, performance, etc. It involves a combination of reverse engineering to help the software engineers understand an existing system, and forward engineering for the purpose of reexamining the functionalities that need to be deleted, added or modified [Chicofsky 90].

The feature location techniques presented in this thesis can be easily embedded into a reverse engineering tool to support many reverse engineering activities. For example by re-documenting a system's functionality by identifying and locating the source implementing the particular feature. Similarly, design recovery can take advantage of feature location techniques to recover behavioural design models that represent the high-level components that are specific to a given feature and its interaction with other parts of the system.

2.2 A Survey of Existing Feature Location Techniques

In this section, we present a survey of the most cited feature location techniques. We did not attempt to include all studies that exist in the literature. However, we believe that the ones presented in this section reflect the current state of the art in feature location research.

Wilde and Scully [Wilde 95] introduced 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 technique is that it requires exercising several features of the system although the objective is to identify the components of only one feature. In addition, it is not clear how many features need to be considered for the approach to be effective. For example, there might be situations where a particular component is invoked by chance in only one trace and would therefore be considered as specific to the corresponding feature using software reconnaissance. Finally, it is very important 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.

In [Wong 99, Wong 05], Wong et al. improved the results of applying software reconnaissance by measuring the extent to which a particular component belongs to a feature. To achieve this, they computed several metrics that aim at determining the component dedication to a feature. They proposed three metrics. Their first metrics is called Disparity. This metric measures how close is a feature to a given program component. According to them disparity is equal to set of blocks in either a component or a feature under consideration but not in both divided by the union of set of blocks in a feature and component under consideration. They define blocks as an execution slice or code statements. Their second metrics is called Concentration. This metrics measures how much a feature is concentrated into a program component. They calculate concentration as the intersection of the set of blocks in a component (under consideration) and set of blocks in a feature (under consideration) divided by set of blocks in the feature under consideration. Their third metrics is called Dedication which

is a measure about how much 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 and Gael-Gueheneuc [Antoniol 06]. 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 [Eisenberg 05] proposed a feature location technique based on ranking the components invoked in the trace. 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. In addition, the authors used the call depth of a method in execution of different test cases.

Eisenbarth et al. [Eisenbarth 03] proposed a feature location approach that combines static and dynamic analysis techniques. They used dynamic analysis to gather traces that correspond to software features of the system, similar to Wilde and Scully's technique [Wilde 95]. 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 of this approach 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.

Poshyvanyk et al. [Poshyvanyk 07a, Poshyvanyk 07b], their approach is based on processing one trace only, which is the one that corresponds to the feature under investigation. They used information retrieval techniques to extract knowledge from the source code that describes the components invoked in the trace. Using this approach, a user needs to formulate queries that contain key terms describing the feature. The query terms are then compared to the knowledge gathered from the source code in order to identify the corresponding components of the trace. The user may need to write several queries before the system can detect any component. The advantage of this approach is that it uses only one trace instead of many traces as it is case in the previous approaches. The disadvantage is that it relies on informal knowledge such as source code comments, identifiers, etc. to extract knowledge about the trace components.

Greevy et al. [Greevy 05] exploited the relationship between features and classes to analyze the way features of a system evolve and to detect changes in the code from a feature perspective. Rather than detecting feature specific components, the main focus of the authors' approach is on studying how the classes may change their roles during software evolution, for example, by understanding the number of features they participate in as the system evolves.

Kothari et al. [Kothari 06] worked on computing canonical features of a system and understanding their implementation. A canonical feature set is a set of a small number of features that implement different parts of the system. To compute the canonical feature set, they first built test cases to exercise all the features of the system. The features were executed under the supervision of a dynamic analysis tool that captures the objects, functions, and variables involved. A call graph tool was also used with each test case to produce a dynamic call graph for each feature of the system. Using a similarity measurement tool, they computed the pair wise similarity between the call graphs that were generated in the previous step and created 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, to measure the similarity among subgraphs they computed the degree to which features share common significant amount of code. The dynamic call graphs of two similar features should have several vertices (functions) and edges (function call relations) in common. 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. [Salah 04] proposed a feature location approach that combines three views of a system, namely, an object interaction view, a class interaction view, and a feature interaction view. The object interaction view is constructed from the execution traces of the program by exercising a subset of its features. This view shows how objects interact

with each through method invocation. The class interaction view is simply an abstraction of the object view by grouping objects by their class type. 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. 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 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 [Robillard 03]. 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 in to source code.

2.3 Discussion

Feature location is a process of identifying the specific components that implement a given feature. This requires mapping high-level features to low-level implementation details. Feature location is considered as an important reverse engineering activity that can enable software engineers to perform software maintenance and evolution tasks in a more efficient manner. This is due to the fact that changes to a system usually relate to a particular software feature.

Recently, there has been an increase in the number of feature location techniques. These techniques are based on either dynamic analysis, static analysis, or a combination of both. Static analysis techniques rely on analyzing the relationships among the source code components, whereas dynamic analysis focuses on the study of the execution traces generated from a running system.

An analysis of these techniques reveals that they suffer from two major limitations. First, most techniques require exercising several features of the system to identify the components of only one feature. Exercising several features requires determining the appropriate input data for each feature, setting the execution environment, etc. In addition, there is usually a need to pre-process the generated traces to filter out unwanted

events such as unnecessary repetitions due to loops, recursion, or the way the system is used during the trace generation process. In addition, it is difficult to determine in advance the features that need to be used in order to obtain a balanced set of features that would lead to unbiased results.

The second drawback of most existing techniques is that they require as part of their analysis significant human intervention which results in a significant cost and effort overhead. Therefore, there is a clear need for more automated techniques to allow for a reduction of the cost and effort associated with these interactive approaches. In addition, many existing techniques require that users have already a good understanding of the system to be analyzed, which contradicts the high-level goal of feature location techniques: To help software engineers *understand* how a particular feature is implemented.

Our feature location approach also combines static and dynamic analysis. It operates on only *one* trace, i.e., the one corresponding to the feature under study. However, the main difference between our approach and existing work is that it facilitates the identification of feature-specific components to a great extent, by utilizing a ranking technique that will allow software engineers to quickly spot feature-specific components without having to be very knowledgeable of the system under study.

Chapter 3 Feature Location Methodology

In this chapter, we present our methodology for solving the feature location, 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 used to rank the components invoked in the generated trace according to their relevance to the feature. The ranking technique presented in this thesis is based on impact analysis, i.e., by measuring the impact of a component modification on the rest of the system.

3.1 What is a Software Feature?

Perhaps the most popular definition of the concept of software feature in the context of feature location research is the one proposed by Eisenbarth et al. in [Eisenbarth 03]. The authors define a software feature as a behavioural aspect of the system that represents a particular functionality, triggered by an external user [Eisenbarth 03].

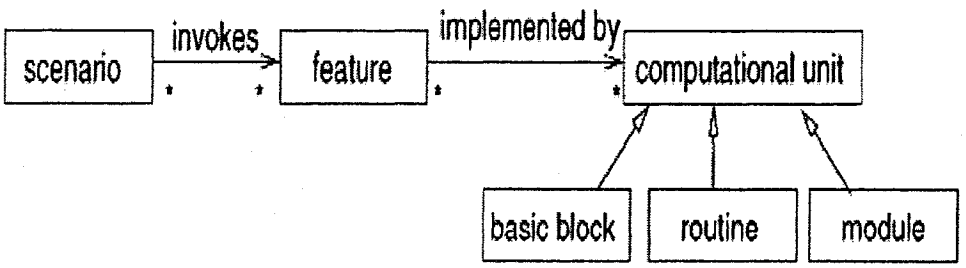

Figure 3.1. Relationship between software feature, scenario, and computational units

In [Eisenbarth 03], Eisenbarth et al discuss the relationship between a software feature, a scenario, and a computation unit (Figure 3.1). 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 that is 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 the context of 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 the previous definition by adding, that a software feature is similar to the concept of use cases found in UML [UML 2.0]. As a result, in our context, a particular instance of a feature (based on a selected data input) corresponds to a scenario. Furthermore, we 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.

3.2 Overall Approach

Figure 3.2 provides a general overview of our approach for identifying the components that implement a specific feature. In our research, we limit the components of interest to classes in the system. However, we believe that the approach in this thesis can also be easily adaptable to other modules of the system such as methods or packages.

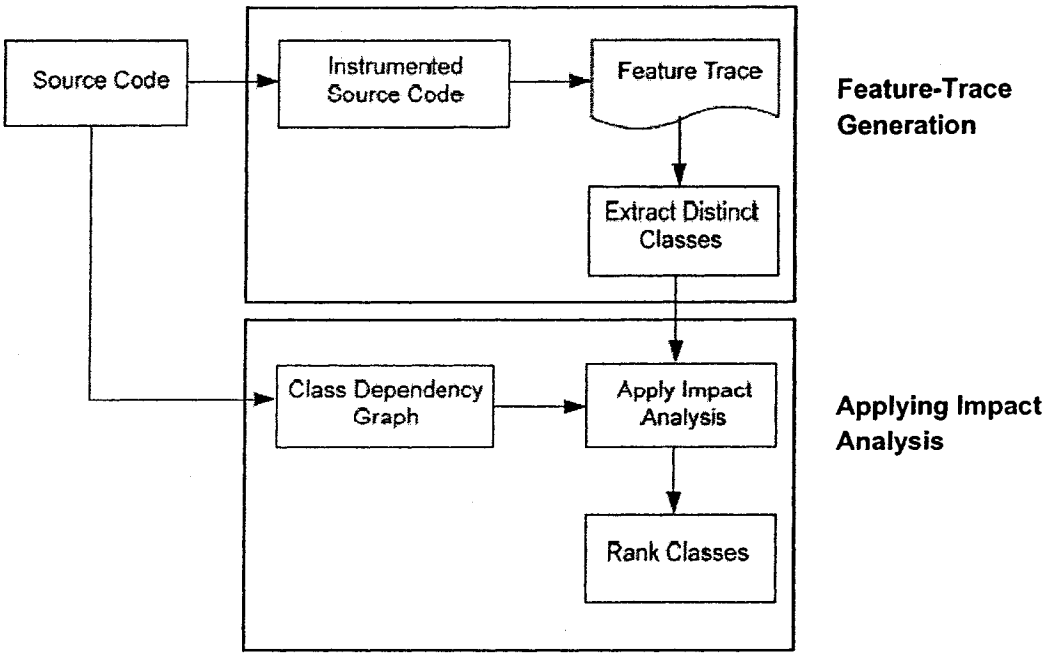

Figure 3.2. Overall Approach

Our approach for locating features in code, we apply a combination of static and dynamic analysis. An execution trace is generated by exercising the feature under study (dynamic analysis). For this purpose, we use code instrumentation to insert probes at various locations in the source code. More discussion on the trace generation process is presented in Section 3.3.1. Next a class dependency graph (static analysis) is used 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 of a component (class) modification on the rest 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. They often correspond to utility classes that help implementing the core functionality of the system [Hamou-Lhadj 04, Hamou-Lhadj 06]. On the other hand, we 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. In addition, we anticipate that 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 propose to characterize the components invoked in a feature-trace according to the following three categories:

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

The remaining part of the chapter is organized as follows: The trace generation process is discussed in detail in Section 3.3. In Section 3.4, 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.

3.3 Feature Trace Generation

The first step of our approach consists of generating a *feature trace* that corresponds to the software feature under study. We use the term *feature trace* to refer to a trace that corresponds to a particular feature. There exist various techniques for generating traces. The first technique is based on instrumenting the source code, which consists of inserting print statements at selected locations in the source code. To generate a trace of method calls, for example, one needs to insert a print-out statement at least at each entry and exit of a method. The second technique consists of instrumenting the execution environment in which the system runs (e.g., the Java Virtual Machine). Unlike source code instrumentation, this technique does not require the modification of the source code. Finally, it is also possible to run the system under the control of a debugger. In this case, breakpoints are set at locations of interest. This technique has the advantage of modifying neither the source code nor the environment but has been shown to slow down considerably the execution of the system, which makes it impractical for large systems.

We have used source code instrumentation for its simplicity and the abundance of 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.

3.4 Impact Analysis

Impact analysis is the process of determining the parts of a program that are potentially affected by a change made to the program. Impact analysis has been shown to be useful for planning system changes, making changes, accommodating certain types of software changes, and tracing through the effects of changes [Law 03, Turver 94].

As previously mentioned, we apply in our approach impact analysis to identify feature specific classes, by measuring at a class dependency graph level, the potential impact that a modification of each distinct class in the feature trace has on the rest of the system. The four metrics presented in this thesis measure the impact of a class modification on the other parts of the system. Dependencies among system components are computed based on a static class dependency graph. Building class dependencies graphs is discussed in the next subsection. The impact metrics are presented in Section 3.3.2.2.

3.4.1 Building a Class Dependency Graph

Impact analysis is based on the exploration of the class dependency graph (CDG) of a system. A CDG is a directed graph where the nodes are the system's classes and the edges represent a dependency relationship among the classes as shown in Figure 3.3.

The construction of a class dependency graph typically requires parsing of the source code. 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. One of the important dependencies that exist between classes

is method invocation. These function calls are traced using a static call graph. The construction of call graphs from object-oriented systems requires the resolution of polymorphic calls. There exist various techniques in the literature that achieve this including Unique Name (UN) [Calder 94], Class Hierarchy Analysis (CHA) [Bacon 96, Dean 95], and Rapid Type Analysis (RTA). Each algorithm has its own advantages and imitations. In this thesis, we use RTA for its simplicity, efficiency, and tool support [Bacon 96].

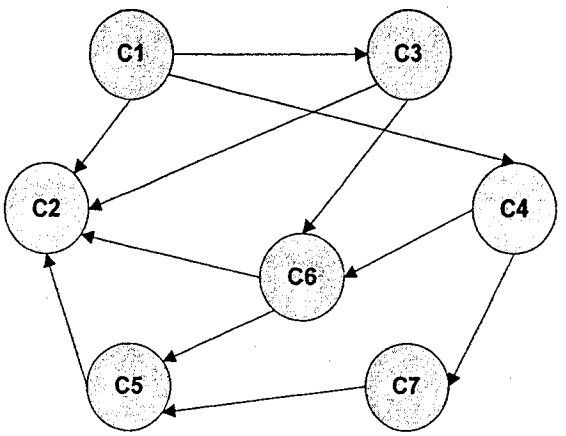

Figure 3.3. Example of a class dependency graph

3.4.2 Impact Metrics

We have developed four metrics for measuring the impact of a class modification on the other parts of the system. These metrics vary depending on the way impact of a component modification on the rest of the system is computed.

3.4.2.1 Definitions

Definition 1: Impact Set

We define the impact set of modifying a component *C* as a 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 is a path in the graph 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 3.3 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.

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 [Martin 95], 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 in this thesis, we 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 exists several metrics in the literature that measure the relationship among the system components (e.g., the MOOSE metrics presented in [Lanza 2006]). These metrics are used to assess the overall quality of a design and do not necessarily measure the impact of a component on the rest of the system. In this thesis, we propose four simple and yet powerful metrics that achieve this goal.

The first metric is called the one way impact metric and considers exclusively the impact modification of a class on the system, i.e., CAI.

We define the one way impact metric of a class *C* as:

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

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

OWI ranges 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 it is specific to the feature in question according to our hypothesis. On the other hand, a class with an OWI value reaching 1 indicates that a change in this class causes many parts of the systems to change. This indicates that this class is used to support the implementation of various features.

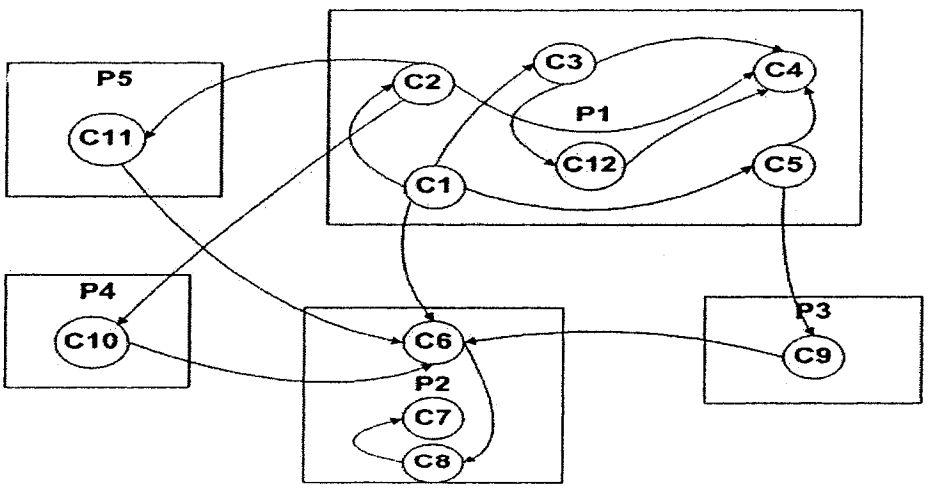

Figure 3.4. A class dependency graph

The example in Figure 3.4 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 3.1 shows the result of applying the one way impact metric to classes of Figure 3.4. The table is sorted in an ascending order based on the OWI column. From the table one can see that the metric was able to group successfully all P1 classes (the most relevant classes) in the top of the table. The class C6 is used by many other classes of the system, which suggests that it is a utility class. In this example, it was ranked among the last classes along with C7and C8 on which it depends.

Table 3.1. Applying OWI to Example of Figure 3.4

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

3.4.2.3 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 behind 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 [Hamou-Lhadj 05, Hamou-Lhadj 06]. The authors used 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 explained 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 two way impact 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 3.5, 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).

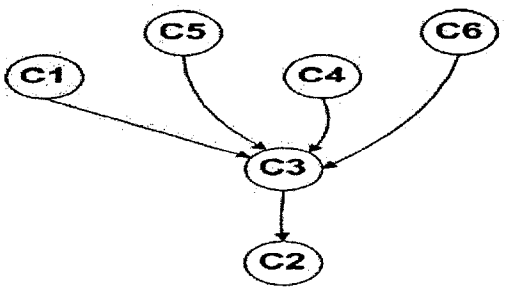

Figure 3.5. The difference between fan-in and CAI

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

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

$$TWI\ (c) = \begin{cases} \frac{CAI\ (c)}{|\ S\ |} \times \frac{Log\ (\frac{|\ S\ |}{CEI\ (c)})}{Log\ (|\ S\ |)} & if\ \ CEI\ != 0 \\ \frac{CAI\ (c)}{|\ S\ |} & if\ CEI\ = 0 \end{cases}$$

If the 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, the formula is divided into two parts. The first part $\frac{CAI(c)}{|S|}$ 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. The second part 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 to the example in Figure 3.4, 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.

Table 3.2. Applying TWI to example of Figure 3.4

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

Therefore, having these two classes at the bottom of the table should be seen as a good outcome of the algorithm.

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, it is more likely that this class will 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.

Taking the above rationale into account, we introduce the following metric:

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

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.

In the earlier example we saw that our metric worked fine according to our hypothesis but if the system was rather complex and packaged the previous metrics would not had been that accurate. We enhance the previous example by introducing packages to show the effectiveness of our new metric in comparison to the pervious example.

The weighted one way impact metric results in a similar outcome as the non-weighted one way metric when applied to the example in Figure 3.4 (see Table 3.3). 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.

Table 3.3. Applying WOWI on the example of Figure 3.4

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

3.4.2.5 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 3.4 shows the result after applying the WTWI to the example in Figure 3.4. 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 5 packages. In addition, the WTWI improves over the non-weighted TWI by enlarging the gap between the relevant and non-relevant classes.

Table 3.4. Applying WTWI on the example of figure 3.4

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 chapter, we presented our approach for solving the feature location problem with a focus on identifying 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 metric and the weighted two way impact metric, use architectural information to further enhance the previous metrics.

Chapter 4 Evaluation

In this chapter, we evaluate the applicability of our feature location techniques by applying them to traces generated from two object-oriented software systems.

4.1 Target Systems

We have applied the proposed feature location techniques to traces generated from two Java-based system called Weka[1] (version 3.0), and Checkstyle[2] (version 3.3). Weka has been developed in 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 has 10 packages, 142 classes, and 95 KLOC.

The second software system used for evaluating our approach is Checkstyle. Checkstyle is a development tool that aims to help programmers write Java code that adheres to a coding standard. This is a useful tool in projects where enforcing a coding standard is important. 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

[1] http://www.cs.waikato.ac.nz/ml/weka/
[2] http://checkstyle.sourceforge.net/
[3] http://www.antlr.org/
[4] http://jakarta.apache.org/regexp/

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. Weka packages and most important classes are documented in a book dedicated to the tool and machine learning in general [Witten 99]. A detailed description of Checkstyle architecture can be found on a website dedicated to the tool. Having this documentation available will allow 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 have applied our feature location techniques to several software features of Weka and Checkstyle. In this thesis, we report on the result of applying these techniques to two features (one for each system), which reflect the overall result obtained. For the Weka system, we wanted to identify the classes that are specific to the implementation of the M5 algorithm, which is a classification algorithm based on the so-called model trees [Quinlan 92]. 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 [Lee 97]. 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 a sample input data provided in the documentation of both systems to exercise the M5 and CheckCode features. We saved the distinct classes invoked in each trace while the system was executing. It should be noted that we did not have to save the entire trace. Table 4.1 shows the number of distinct classes invoked in M5 and CheckCode traces.

Table 4.1. 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

We applied the impact analysis metrics to both, the Weka and Checkstyle systems using a tool called Structural Analysis for Java (SA4J)[5]. The tool parses the source code and generates a global class dependency table that contains various metrics including the class afferent and efferent impacts. SA4J supports a large spectrum of relations among classes such as: accesses, calls, contains, extends, implements, instantiates, references, etc.

[5] http://www.alphaworks.ibm.com/tech/sa4j

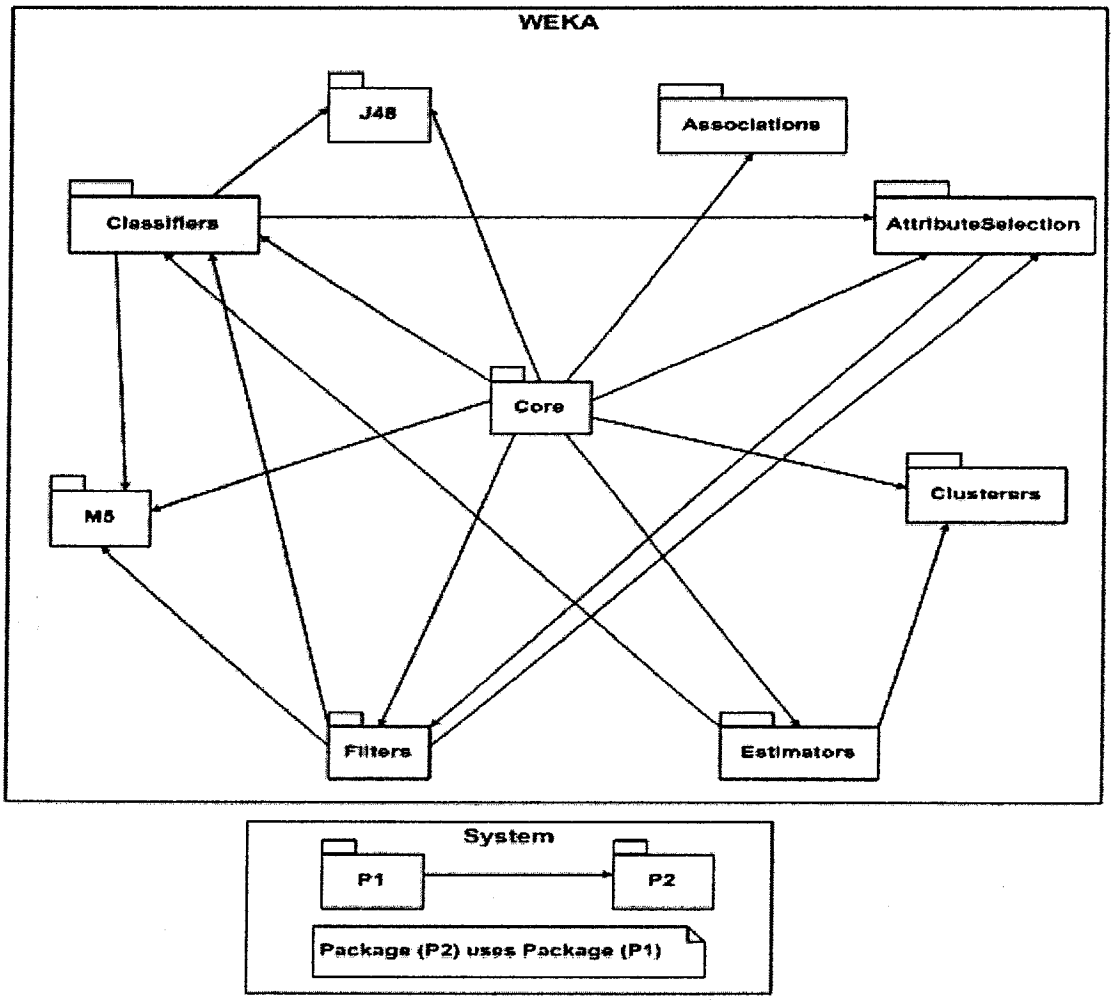

Figure 4.1. Example of a package dependency graph generated using SA4J

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. Figure 4.1, for example, shows a package dependency graph extracted from the Weka system. 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 OWI, WOWI, TWI, and WTWI based feature location approaches to the two selected features.

4.2.3.1 The One Way Impact Metric (OWI)

Table 4.2 shows the result of applying the one way impact metric to the Weka feature trace M5 (we multiplied OWI by a 1000 to improve the clarity of the presentation of the results). The table is sorted in the ascending order of OWI and does not show all classes to avoid cluttering. The detailed results are presented in Appendix A.

Table 4.2. Applying OWI to M5 feature

Package	Class	|CAI|	OWI*1000
weka.classifiers.m5	**M5Prime**	**1**	**7.04**
10 other classes here of M5 Package OWI*1000 value ranging between 7.04 and 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
5 other classes of core Package OWI*1000 value ranging between 345.07 and 894.37			

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).

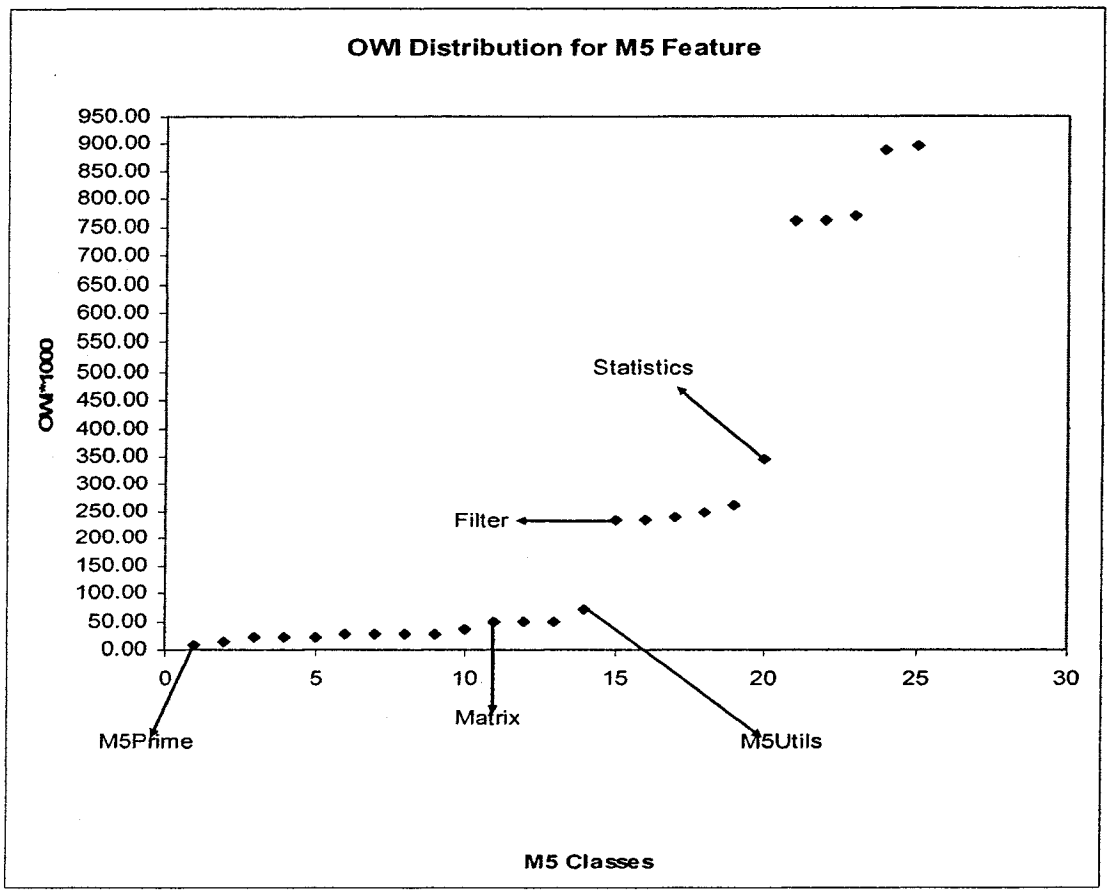

Figure 4.2. OWI distribution for M5 Feature

According to the Weka documentation [Witten 99], 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 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, there is no clear cut between the classes that belong to the relevant components category and the ones contained in the related components category. Figure 4.2 shows this graphically. Although the algorithm succeeded to distinguish between most of the utility classes (on the top of the figure) and the other categories of classes, it 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 4.3, sorted in the ascending order of OWI, shows the result of applying the one way impact metric to the Checkstyle feature trace CheckCode. We 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 4.3. Applying OWI to CheckCode Feature

Package	Class	|CAI|	OWI	OWI*1000
coding	**ExplicitInitializationCheck**	**1**	**0**	**4.76**
29 other classes of coding package with OWI*1000 value of 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
12 more classes of apis package here with OWI*1000 value ranging from 90.48 to 714.29				

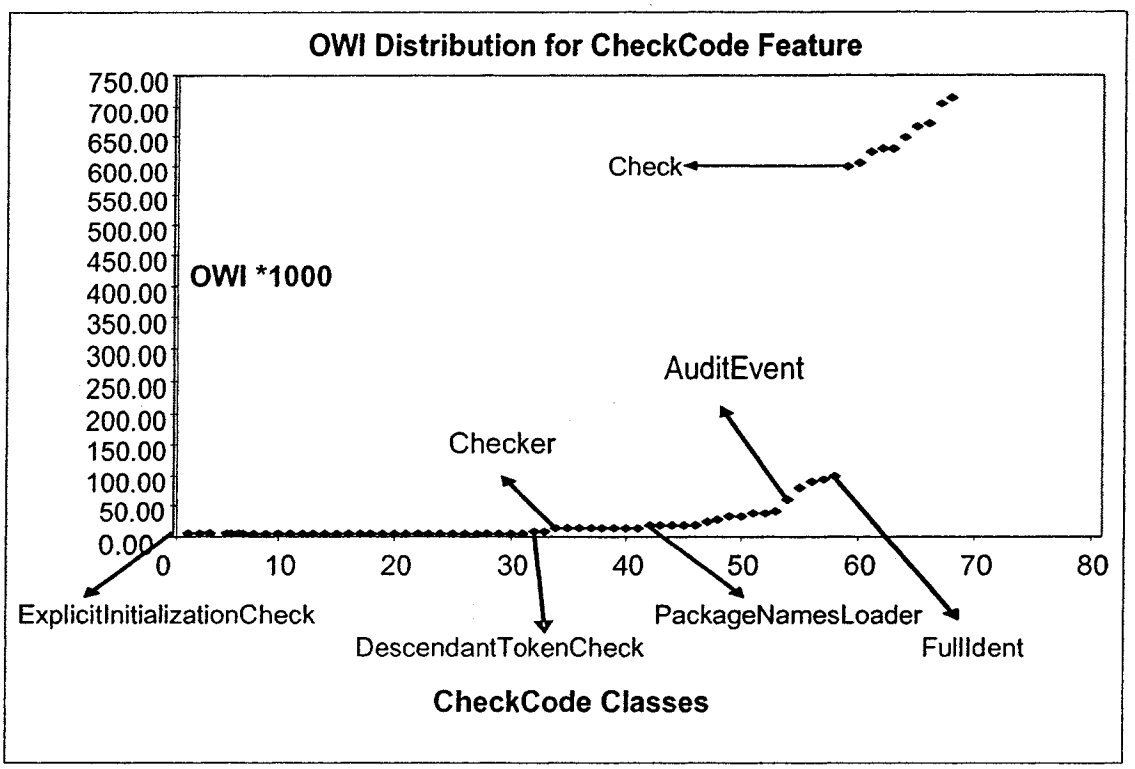

Figure 4.3. OWI distribution for CheckCode feature

The packages `checkstyle`, `checks`, and `grammars` contain classes that implement common functionality used by most checks performed by 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 4.3, 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.

Similar to Weka, the OWI-based feature location technique did not succeed to have 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 4.4 shows the result of applying the weighted one way impact metric to the M5 feature in Weka. As shown in the table, this technique can provide better results than the non-weighted one way impact metric by improving the grouping of the classes within the `M5` package by enhancing the gap between the most important classes and the ones which are less relevant to the M5 feature as shown graphically in Figure 4.4.

Table 4.4. Applying WOWI to M5 feature

Packages	**Class**	**|CAI|**	**|PAI|**	**WOWI*1000**
weka.classifiers.m5	**M5Prime**	**1**	**1**	0.70
11 other classes of M5 package with WOWI*1000 value ranging between 0.70 and 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
5 other classes of core package with WOWI*1000 value ranging between 276.06 and 804.93				

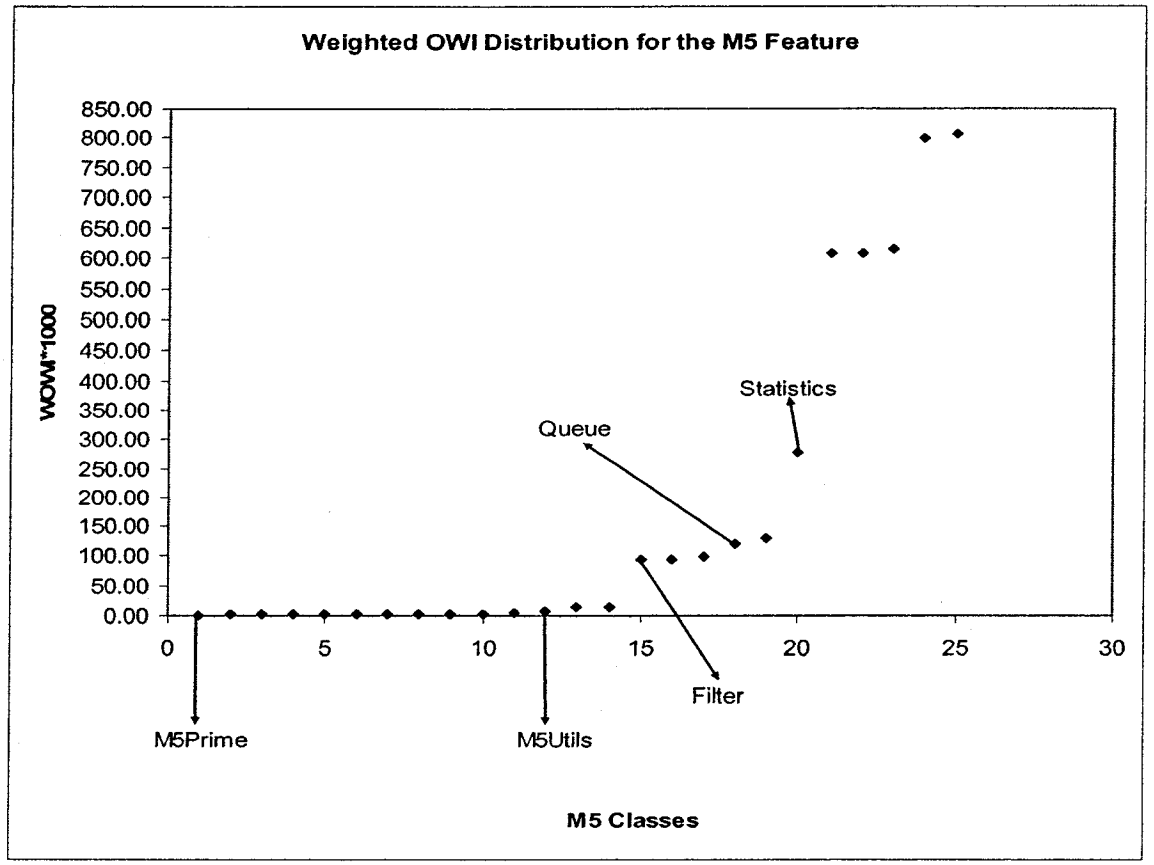

Figure 4.4. WOWI distribution for M5 feature

Similar to Weka, the weighted OWI provides better results than the non-weighted one way impact when applied to the CheckCode feature (Table 4.5). 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.

Table 4.5. Applying WOWI to CheckCode feature

Package	Class	|CAI|	|PAI|	WOWI*1000
coding	**ExplicitInitializationCheck**	**1**	**1**	**0.28**
29 other classes of coding package with WOWI*1000 value of 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
12 other classes of apis package with WOWI*1000 value ranging from 10.92 to 672.27				

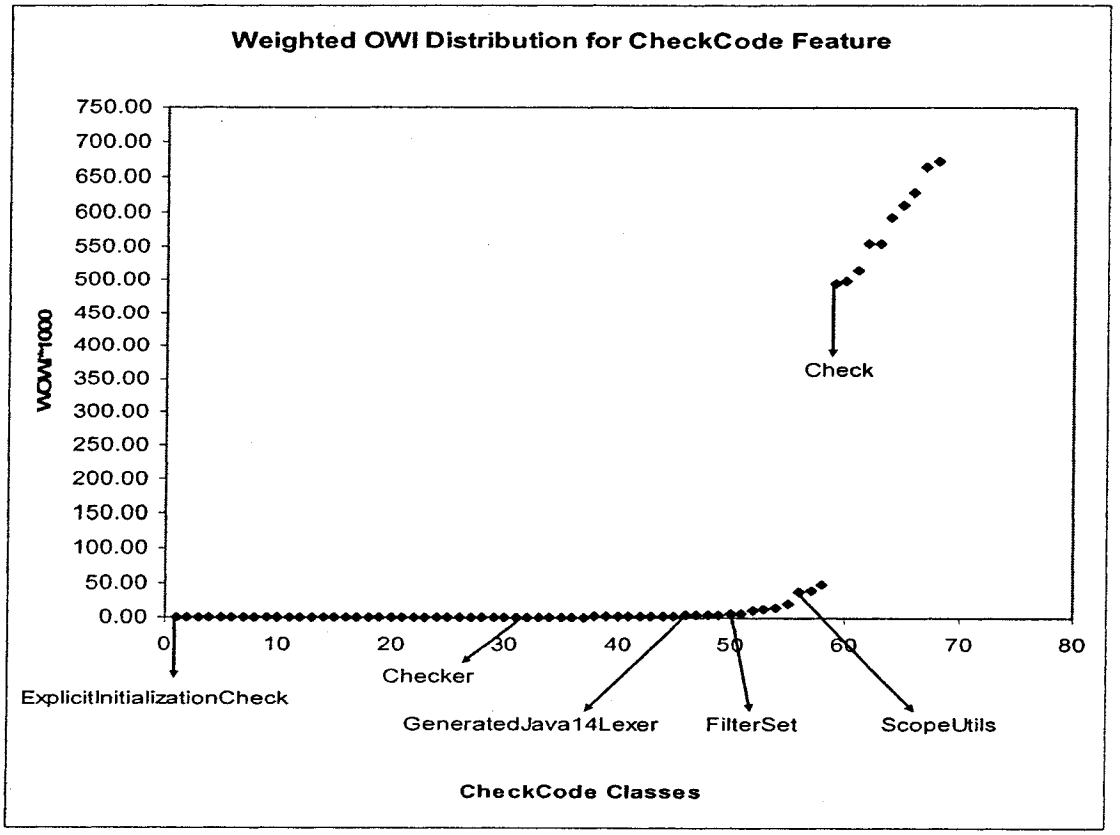

Figure 4.5. WOWI distribution for CheckCode feature.

There is also an improvement in the ranking of the classes of the `apis` package, which are now closer to each other at the bottom of the table. However, the approach misplaced two classes `CheckUtils` and `AbstractFormatCheck` at the end with `apis` package utility classes.

4.2.3.3 The Two Way Impact Metric (TWI)

Table 4.6 shows the result of applying the two way impact metric to locate the M5 feature in the Weka system.

The overall results achieved using this technique provided no 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 4.6, the `core` package classes form now one block located at the bottom of the table. This approach however did misplace an additional relevant component (the class `m5.Matrix`). The overall TWI distribution presents an improvement compared to the previous metrics (see Figure 4.6), where the distribution of TWI forms a curve showing the components from the most relevant to the least relevant, with a few classes misplaced.

Table 4.6. Applying TWI to M5 Feature

Packages	Class	|CEI|	|CAI|	TWI*1000
weka.classifiers.m5	**M5Prime**	**35**	**1**	**1.95**
9 other classes of M5 package with TWI*1000 value ranging from 1.95 to 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
6 other classes from package core with TWI*1000 value ranging from 205.95 and 696.1				

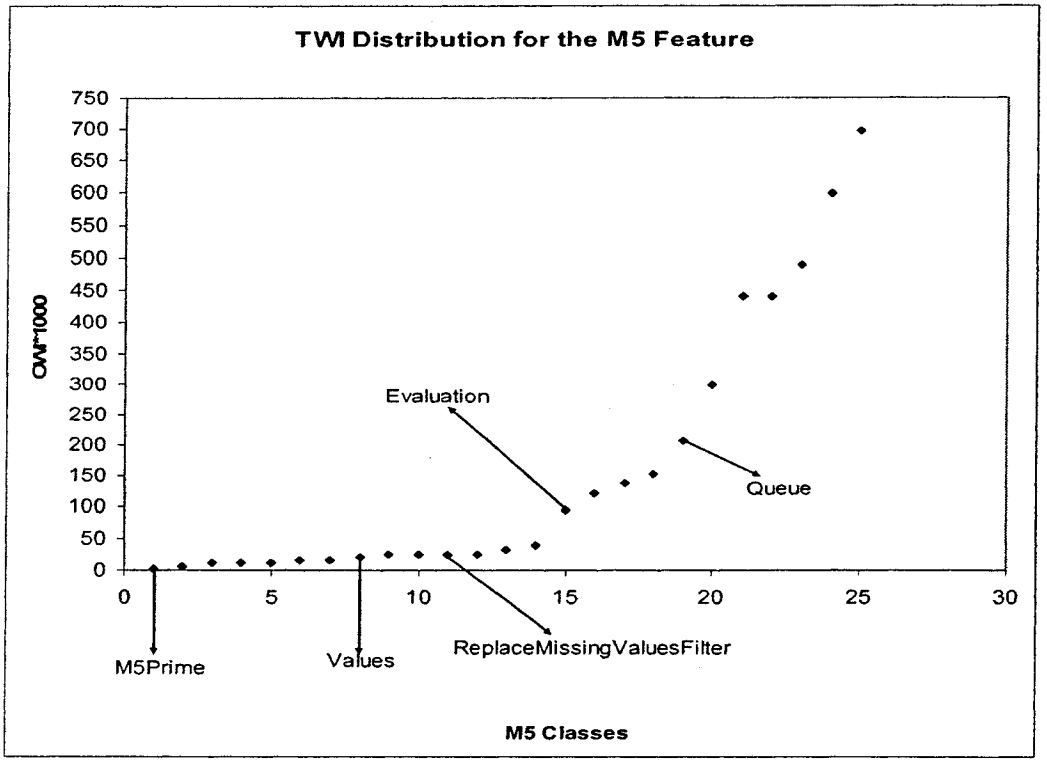

Figure 4.6. TWI distribution for M5 feature

When applied to the CheckCode feature (see Table 4.7), the results obtained were similar to the OWI and WOWI metrics except a better classification of the utility classes (i.e., the ones belonging to the `apis` package). For example, the classes `apis.TokenType` and `apis.AuditEvent` which were misplaced by OWI and WOWI were rightly placed with the other `apis` classes using TWI.

Table 4.7. Applying TWI to CheckCode feature.

Package	Class	|CAI|	|CEI|	TWI*1000
coding	**ExplicitInitializationCheck**	**1**	**22**	**1.97**
29 other coding package classes here with TWI* 1000 value ranging from 1.97 to 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
14 other apis package classes here with TWI*1000 value ranging from 37.3 to 621.69				

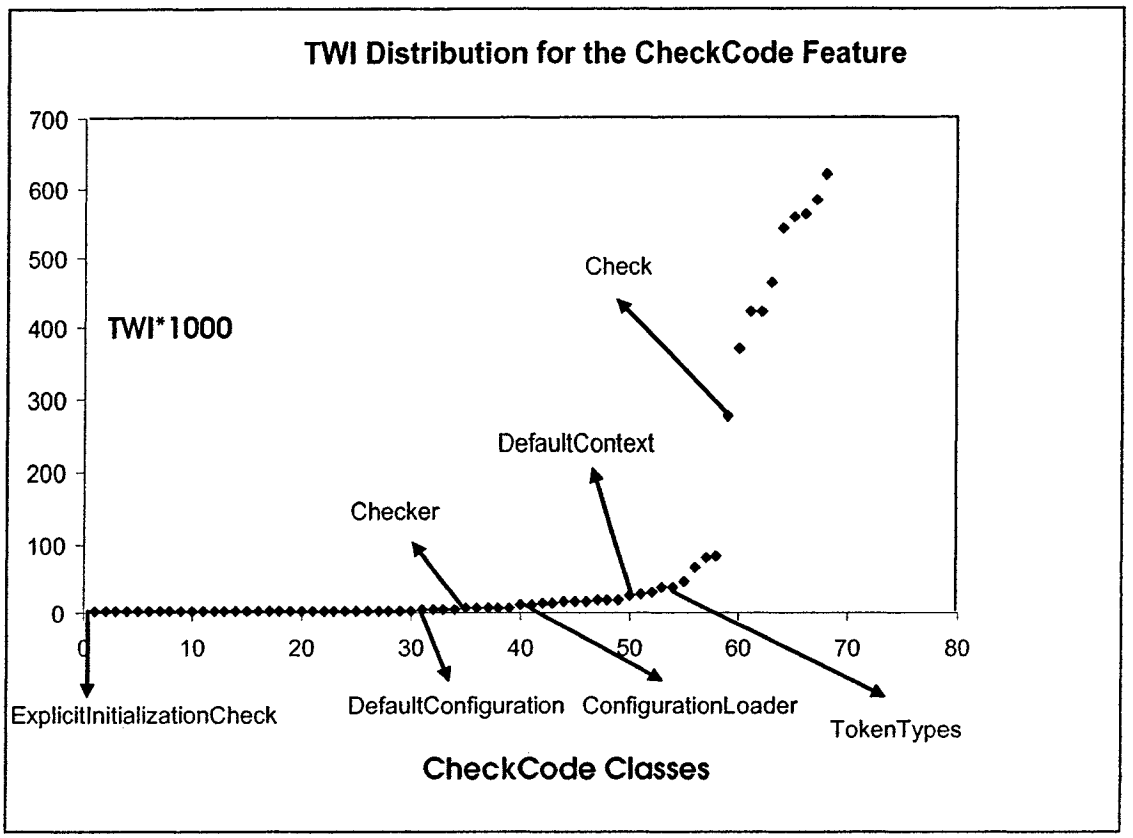

Figure 4.7. TWI distribution for CheckCode feature

4.2.3.4 The Weighted Two Way Impact Metric (WTWI)

Table 4.8 shows the result of applying the feature location techniques based on the weighted two way impact metric. The results achieved by this metric are considerably better than the ones obtained by the previous approaches. One can observe that the classes of the `m5` package were all identified and grouped together as the most relevant classes to the M5 feature. In addition, the `core` package utility classes were grouped together at the end showing that these classes are least important for this feature. Using the WTWI, none of the classes has been misplaced. As shown in Figure 4.8, WTWI also results in a better distribution compared to the other metrics.

Table 4.8. Applying WTWI to M5 feature

Packages	Class	|CE|	|CA|	|PA|	WTWI*1000
weka.classifiers.m5	**M5Prime**	**35**	**1**	**1**	**0.2**
11 other classes of M5 package here with WTWI*1000 value ranging from 0.2 to 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
6 other classes from core package with WTWI*1000 value ranging from 102.97 and 626.49					

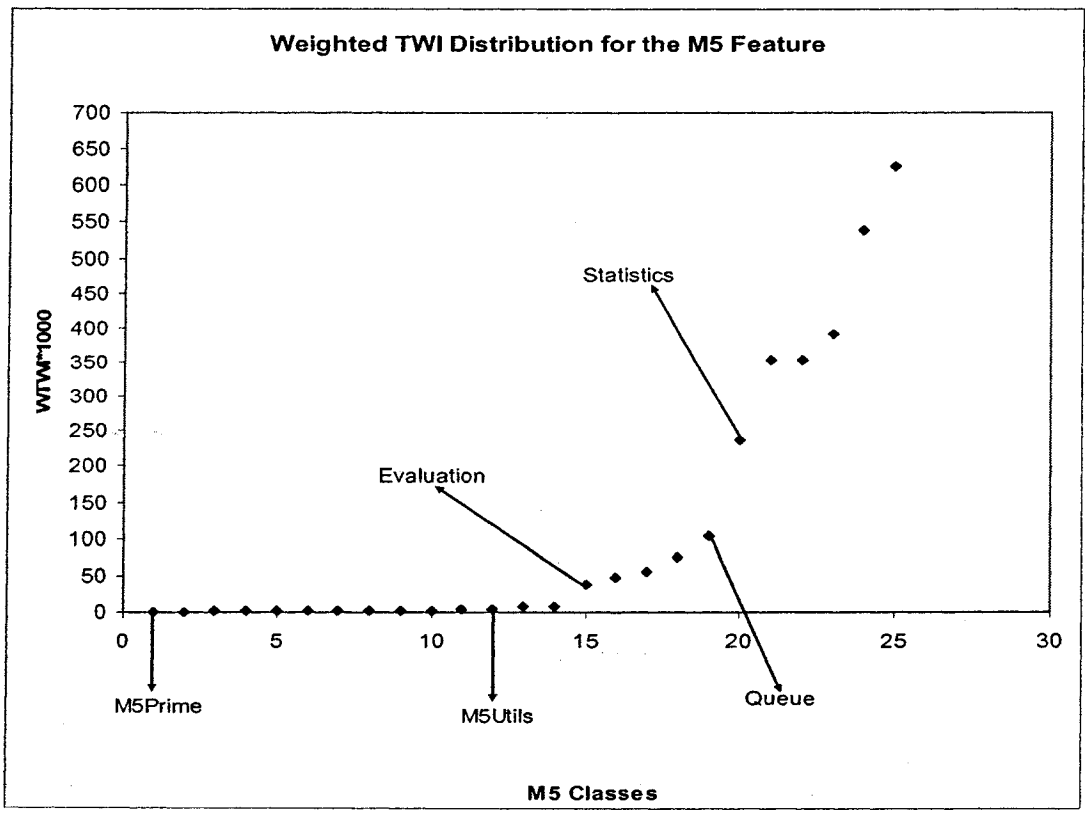

Figure 4.8. WTWI Distribution for M5 feature

The results of applying the WTWI-based feature location technique to the CheckCode feature are shown in Table 4.9. WTWI shows similar results as achieved by WOWI-based technique. However, the WTWI approach misplaced some classes of the `checks` 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 of the same packages. 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. The distribution of WTWI is also better than the TWI distribution as shown in Figure 4.9.

4.3 Discussion

The overall results achieved by our feature location techniques depict that our proposed hypothesis has proven to be effective in the detection of components most relevant to the feature in question. More specifically to test our hypothesis stating we used on two features: M5 from Weka and CheckCode from Checkstyle software.

In terms of features, M5 feature showed more accurate results over CheckCode feature. However the advantages and disadvantages of each proposed feature location techniques in terms of Impact metric utilized are as follows:

Table 4.9. Applying WTWI to CheckCode feature

Package	Class	|CAI|	|CEI|	|PAI|	WTWI*1000
coding	**ExplicitInitializationCheck**	**1**	**22**	**1**	**0.12**
29 classes from coding package with WTWI*1000 value ranging from 0.12 and 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
13 other classes of apis package with WTWI*1000 value ranging from 10.97 to 585.12					

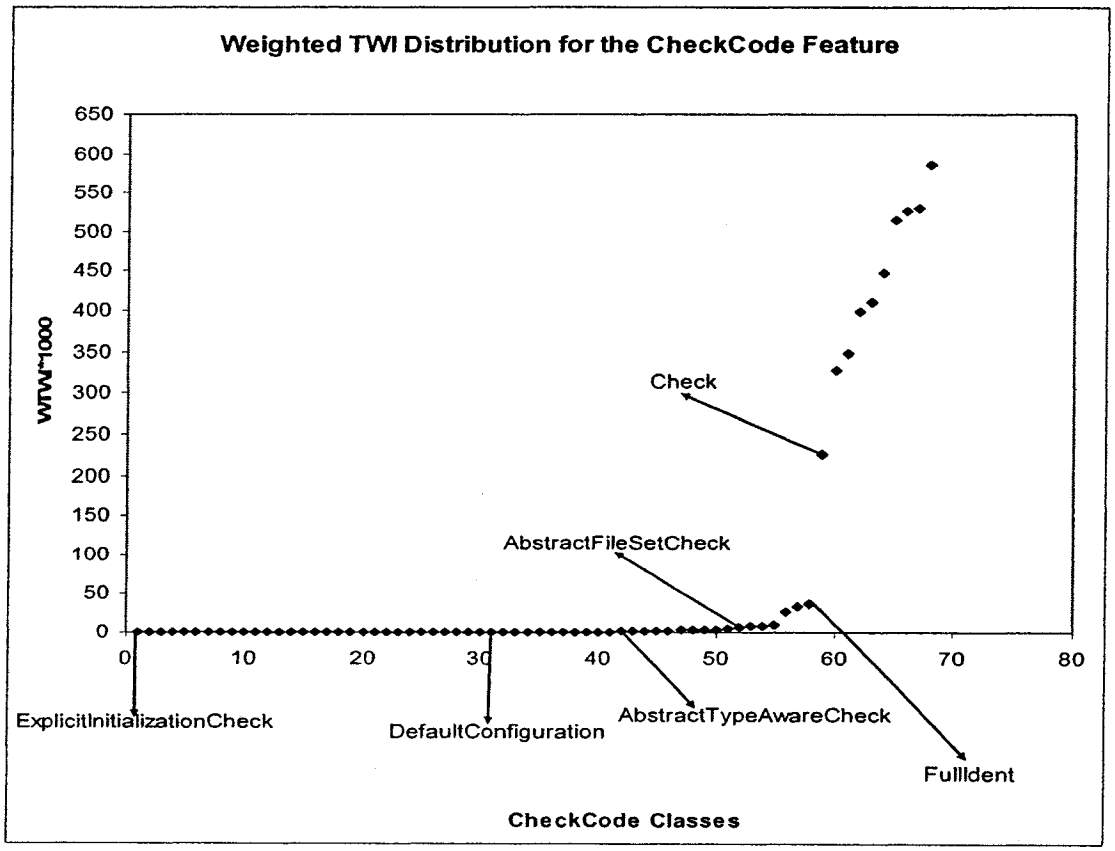

Figure 4.9. WTWI distribution for CheckCode feature.

Feature location technique based on One Way Impact Metric: This technique worked quite well in detecting most important components for the feature M5 and CheckCode. More specifically it misplaced just one specific class of M5 feature and two specific classes of CheckCode feature. However one of the major disadvantage of this approach is its inability to detect a clear-cut difference between the categories of classes (relevant, related and utilities). Secondly this technique also misplaced a few classes of relatively high degree of importance with the utility classes.

Feature location technique based on 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 M5 feature rightly at the top, it further reduced

the gap between the misplaced CheckCode feature classes and the rest of the important classes. This validates the effectiveness of weighting the previous metric by a factor of |PAI|/ |P|.

Feature location technique based on Two Way Impact Analysis: This technique showed no further improvement over the last two techniques. The classes that were initially misplaced by OWI Metric remained misplaced using this technique. However this technique showed an improvement in grouping the utility classes at the end together as one block. All the utility classes of `Core` package were placed at the end and this technique also reduced the gap between misplaced `Apis` package classes of CheckCode feature.

Feature location Technique based on Weighted Two Way Impact Analysis: 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.

Chapter 5 Conclusion

In this chapter we conclude our thesis by summarizing our research contributions in Section 5.1, which also includes a discussion about the results achieved by our approach and its effectiveness. In Section 5.2, we elaborate on opportunities for future research to further improve the effectiveness and accuracy of the present approach. Finally in section 5.3 we provide our closing remarks for this thesis.

5.1 Research Contributions

In this dissertation, we presented a new approach towards solving the feature location problem – mapping features to code. In particular, we focused on identifying these classes that are 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 used to rank the classes invoked in the trace according to their relevance to the feature. We ranked the classes invoked by measuring the impact of each component modification on the rest of the system. The rationale is that classes with small impact are likely specific to the feature at hand, whereas classes that have a large impact have many purposes, and hence they are less specific.

Based on our hypothesis we introduced four new techniques that measure the impact of a class modification on the rest of a system. Common to these metrics is their fundamental

assumption that the lower the impact of a modification on the remaining parts of the system, the more relevant the class is towards a specific feature. Each technique has its own criterion based on the software architecture for measuring the impact of a component on rest of the system in order to acquire the degree of its relevance to the feature of interest.

We applied our techniques to several features of the software systems Weka and Checkstyle. The overall results were very satisfactory. We were not only able to identify the most important components but also rank these components according to their relevance to the traced features. In addition, our approach is very simple and does not require a lot of human intervention. Further, on comparing our techniques we came to the conclusion that our weighted techniques (WOWI and WTWI) showed better results as compared to the non- weighted ones (OWI and TWI).

5.2 Opportunities for Further Research

The immediate future work consists of conducting further experimentation on other feature traces. In particular, we intend to target larger systems with poor architecture.

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 thesis should allow enough flexibility for the users to change the threshold.

Finally, during the analysis of the results of the case study, we noticed that abstract classes do not behave the same way as other classes. They tend to have a higher afferent impact. There is a need to investigate this and propose other means to rank abstract classes.

5.3 Closing Remarks

The architecture of the system degrades with time and the documentations becomes outdated. This leaves the maintainer with no other option but to reverse engineer the complete system exhaustively. This process utilizes plenty of time and resources. Here feature location technique comes handy. The main objective of feature location activities is to assist maintainer in order to save time and resources. We have designed our proposed technique "Feature location based on Impact analysis" to fulfil this objective. The technique is simple, easy to use and yet powerful. Our technique operates on one trace only, which is generated by exercising the feature under consideration. In addition, the proposed approach is automatic to a large extent relieving the user from any decision that would otherwise require extensive knowledge of the system.

Appendix A: The Detailed Result of Case Study.

A.1. Applying OWI

Table A.1.1. 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

Table A.1.2. Applying OWI to CheckCode Feature

Package	Class	|CAI|	OWI	OWI*1000
coding	ExplicitInitializationCheck	1	0	4.76
coding	MagicNumberCheck	1	0	4.76
coding	IllegalTokenTextCheck	1	0	4.76
coding	IllegalTypeCheck	1	0	4.76
coding	NestedIfDepthCheck	1	0	4.76
coding	RedundantThrowsCheck	1	0	4.76
coding	SuperCloneCheck	1	0	4.76
coding	SuperFinalizeCheck	1	0	4.76
coding	DeclarationOrderCheck	1	0	4.76
coding	HiddenFieldCheck	1	0	4.76
coding	IllegalCatchCheck	1	0	4.76
coding	JUnitTestCaseCheck	1	0	4.76
coding	MissingSwitchDefaultCheck	1	0	4.76
coding	CovariantEqualsCheck	1	0	4.76
coding	IllegalInstantiationCheck	1	0	4.76
coding	IllegalTokenCheck	1	0	4.76
coding	NestedTryDepthCheck	1	0	4.76
coding	ArrayTrailingCommaCheck	1	0	4.76
coding	AvoidInlineCondCheck	1	0	4.76
coding	DoubleCheckedLockCheck	1	0	4.76
coding	EmptyStatementCheck	1	0	4.76
coding	EqualsHashCodeCheck	1	0	4.76
coding	FinalLocalVariableCheck	1	0	4.76
coding	InnerAssignmentCheck	1	0	4.76
coding	PackageDeclarationCheck	1	0	4.76
coding	ParameterAssignmentCheck	1	0	4.76
coding	ReturnCountCheck	1	0	4.76
coding	SimplifyBooleanExpCheck	1	0	4.76
coding	SimplifyBooleanReturnCheck	1	0	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
apis	Scope	20	0.1	95.24
apis	FullIdent	21	0.1	100.00
apis	Check	126	0.6	600.00
apis	FileContents	127	0.6	604.76
apis	DetailAST	131	0.62	623.81
apis	AbstractViolationReporter	132	0.63	628.57
apis	LocalizedMessages	132	0.63	628.57
apis	Utils	136	0.65	647.62
apis	AutomaticBean	140	0.67	666.67
apis	StrArrayConverter	141	0.67	671.43
apis	LocalizedMessage	148	0.7	704.76
apis	SeverityLevel	150	0.71	714.29

A.2. Applying WOWI

Table A.2.1. 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

Table A.2.2. Applying WOWI to CheckCode Feature

Package	Class	|CAI|	|PAI|	WOWI*1000
coding	ExplicitInitializationCheck	1	1	0.28
coding	MagicNumberCheck	1	1	0.28
coding	IllegalTokenTextCheck	1	1	0.28
coding	IllegalTypeCheck	1	1	0.28
coding	NestedIfDepthCheck	1	1	0.28
coding	RedundantThrowsCheck	1	1	0.28
coding	SuperCloneCheck	1	1	0.28
coding	SuperFinalizeCheck	1	1	0.28
coding	DeclarationOrderCheck	1	1	0.28
coding	HiddenFieldCheck	1	1	0.28
coding	IllegalCatchCheck	1	1	0.28
coding	JUnitTestCaseCheck	1	1	0.28
coding	MissingSwitchDefaultCheck	1	1	0.28
coding	CovariantEqualsCheck	1	1	0.28
coding	IllegalInstantiationCheck	1	1	0.28
fcoding	IllegalTokenCheck	1	1	0.28
coding	NestedTryDepthCheck	1	1	0.28
coding	ArrayTrailingCommaCheck	1	1	0.28
coding	AvoidInlineCondCheck	1	1	0.28
coding	DoubleCheckedLockCheck	1	1	0.28
coding	EmptyStatementCheck	1	1	0.28
coding	EqualsHashCodeCheck	1	1	0.28
coding	FinalLocalVariableCheck	1	1	0.28
coding	InnerAssignmentCheck	1	1	0.28
coding	PackageDeclarationCheck	1	1	0.28
coding	ParameterAssignmentCheck	1	1	0.28
coding	ReturnCountCheck	1	1	0.28
coding	SimplifyBooleanExpCheck	1	1	0.28
coding	SimplifyBooleanReturnCheck	1	1	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
apis	Scope	20	7	39.22
apis	FullIdent	21	8	47.06
apis	Check	126	14	494.12
apis	FileContents	127	14	498.04
apis	DetailAST	131	14	513.73
apis	AbstractViolationReporter	132	15	554.62
apis	LocalizedMessages	132	15	554.62
apis	StrArrayConverter	141	15	592.44
apis	Utils	136	16	609.52
apis	AutomaticBean	140	16	627.45
apis	LocalizedMessage	148	16	663.31
apis	SeverityLevel	150	16	672.27

A.3. Applying TWI

Table A.3.1. Applying TWI to M5 Feature

Packages	Class	|CE|	|CA|	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

Table A.3.2. Applying TWI to CheckCode Feature.

Package	Class	|CAI|	|CEI|	TWI*1000
coding	**ExplicitInitializationCheck**	**1**	**22**	**1.97**
coding	**MagicNumberCheck**	**1**	**22**	**1.97**
coding	**IllegalTokenTextCheck**	**1**	**21**	**2.01**
coding	**IllegalTypeCheck**	**1**	**21**	**2.01**
coding	**NestedIfDepthCheck**	**1**	**21**	**2.01**
coding	**RedundantThrowsCheck**	**1**	**21**	**2.01**
coding	**SuperCloneCheck**	**1**	**21**	**2.01**
coding	**SuperFinalizeCheck**	**1**	**21**	**2.01**
coding	**DeclarationOrderCheck**	**1**	**20**	**2.05**
coding	**HiddenFieldCheck**	**1**	**20**	**2.05**
coding	**IllegalCatchCheck**	**1**	**20**	**2.05**
coding	**JUnitTestCaseCheck**	**1**	**20**	**2.05**
coding	**MissingSwitchDefaultCheck**	**1**	**20**	**2.05**
coding	**CovariantEqualsCheck**	**1**	**19**	**2.09**
coding	**IllegalInstantiationCheck**	**1**	**19**	**2.09**
coding	**IllegalTokenCheck**	**1**	**19**	**2.09**
coding	**NestedTryDepthCheck**	**1**	**19**	**2.09**
coding	**ArrayTrailingCommaCheck**	**1**	**18**	**2.14**
coding	**AvoidInlineCondCheck**	**1**	**18**	**2.14**
coding	**DoubleCheckedLockCheck**	**1**	**18**	**2.14**
coding	**EmptyStatementCheck**	**1**	**18**	**2.14**
coding	**EqualsHashCodeCheck**	**1**	**18**	**2.14**
coding	**FinalLocalVariableCheck**	**1**	**18**	**2.14**
coding	**InnerAssignmentCheck**	**1**	**18**	**2.14**
coding	**PackageDeclarationCheck**	**1**	**18**	**2.14**
coding	**ParameterAssignmentCheck**	**1**	**18**	**2.14**
coding	**ReturnCountCheck**	**1**	**18**	**2.14**
coding	**SimplifyBooleanExpCheck**	**1**	**18**	**2.14**
coding	**SimplifyBooleanReturnCheck**	**1**	**18**	**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
apis	AuditEvent	13	3	45.86
apis	ScopeUtils	19	3	67.02
apis	FullIdent	21	2	79.45
apis	Scope	20	1	82.89
apis	Check	126	17	275.67
apis	AbstractViolationReporter	132	8	370.28
apis	FileContents	127	4	422.73
apis	AutomaticBean	140	6	424.05
apis	LocalizedMessages	132	3	465.61
apis	DetailAST	131	1	542.94
apis	LocalizedMessage	148	2	559.96
apis	Utils	136	1	563.67
apis	StrArrayConverter	141	1	584.39
apis	SeverityLevel	150	1	621.69

A.4. Applying WTWI

Table A.4.1. 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

Table A.4.2. Applying WTWI to CheckCode Feature

Package	Class	|CAI|	|CEI|	|PAI|	WTWI*1000
coding	ExplicitInitializationCheck	1	22	1	0.12
coding	MagicNumberCheck	1	22	1	0.12
coding	IllegalTokenTextCheck	1	21	1	0.12
coding	IllegalTypeCheck	1	21	1	0.12
coding	NestedIfDepthCheck	1	21	1	0.12
coding	RedundantThrowsCheck	1	21	1	0.12
coding	SuperCloneCheck	1	21	1	0.12
coding	SuperFinalizeCheck	1	21	1	0.12
coding	DeclarationOrderCheck	1	20	1	0.12
coding	HiddenFieldCheck	1	20	1	0.12
coding	IllegalCatchCheck	1	20	1	0.12
coding	JUnitTestCaseCheck	1	20	1	0.12
coding	MissingSwitchDefaultCheck	1	20	1	0.12
coding	CovariantEqualsCheck	1	19	1	0.12
coding	IllegalInstantiationCheck	1	19	1	0.12
coding	IllegalTokenCheck	1	19	1	0.12
coding	NestedTryDepthCheck	1	19	1	0.12
coding	ArrayTrailingCommaCheck	1	18	1	0.13
coding	AvoidInlineCondCheck	1	18	1	0.13
coding	DoubleCheckedLockCheck	1	18	1	0.13
coding	EmptyStatementCheck	1	18	1	0.13
coding	EqualsHashCodeCheck	1	18	1	0.13
coding	FinalLocalVariableCheck	1	18	1	0.13
coding	InnerAssignmentCheck	1	18	1	0.13
coding	PackageDeclarationCheck	1	18	1	0.13
coding	ParameterAssignmentCheck	1	18	1	0.13
coding	ReturnCountCheck	1	18	1	0.13
coding	SimplifyBooleanExpCheck	1	18	1	0.13
coding	SimplifyBooleanReturnCheck	1	18	1	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
apis	ScopeUtils	19	3	7	27.6
apis	Scope	20	1	7	34.13
apis	FullIdent	21	2	8	37.39
apis	Check	126	17	14	227.02
apis	AbstractViolationReporter	132	8	15	326.72
apis	FileContents	127	4	14	348.13
apis	AutomaticBean	140	6	16	399.11
apis	LocalizedMessages	132	3	15	410.83
apis	DetailAST	131	1	14	447.13
apis	StrArrayConverter	141	1	15	515.64
apis	LocalizedMessage	148	2	16	527.02
apis	Utils	136	1	16	530.51
apis	SeverityLevel	150	1	16	585.12

Bibliography

Antoniol 06 — G. Antoniol, and Y. G. Gueheneuc, "Feature Identification: An Epidemiological Metaphor", *IEEE Transactions on Software Engineering,* 32(9), pp.627-641, 2006.

Bacon 96 — 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.

Bennett 00 — K. H. Bennett , V. T. Rajlich, "Software maintenance and evolution: a roadmap", *In Proc. of the Conference on the Future of Software Engineering*, pp.73-87, 2000.

Brooks 83 — R. Brooks, "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies, 18(6)*, pp. 542-554, 1983.

Calder 94 — 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.

Chicofsky 90 E. J. Chicofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy", *IEEE Software, 7(1),* pp. 13–17, 1990.

Dean 95 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.

Deerwester 90 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.

Eisenbarth 03 T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering, 29(3),* pp. 210 – 224, 2003.

Eisenberg 05 A. 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.

Fjeldstad 83 K. Fjeldstad and W. T. Hamlen., ''Application Program Maintenance Study: Report to Our Respondents'', *In Proc. of GUIDE 48, The GUIDE Corporation, Philadelphia,* pp. 13-30, 1983.

Greevy 05 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.

Hamou-Lhadj 06 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.

Hamou-Lhadj 04 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.

Hamou-Lhadj 05 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.

Kothari 06 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.

Lanza 06 M. Lanza, R. Marinescu. Object-Oriented Metrics in Practice , Springer, 2006.

Law 03 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.

Lee 97 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.

Lientz 80 B. P. Lientz and E. B. Swanson. Software Maintenance Management. Addison Wesley, 1980.

Martin 95 R. Martin, "Object Oriented Design Quality Metrics: An Analysis of dependencies", ROAD, Vol. 2, No. 3, Sep-Oct, 1995.

Mayrhauser 95 A. V. Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer, 28(8),* pp. 44-55, 1995.

Parnas 94 D. L. Parnas, "Software Aging", *In Proc. of the 16th International Conference on Software Engineering,* pp. 279-287, 1994.

Poshyvanyk 07a 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.

Poshyvanyk 07b 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.

Quinlan 92 — J. R. Quinlan, "Learning with continuous classes", *In Proc. of the 5th Australian Joint Conference on Artificial Intelligence*, pp 343-348, 1992.

Robillard 03 — 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.

Rugaber 95 — S. Rugaber, "Program comprehension", In Encyclopaedia of Computer Science and Technology, 35(20), pp 341-368, 1995.

Salah 04 — 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.

Turver 94 — R. J. Turver and M. Malcolm, "An Early Impact Analysis Technique for Software Maintenance", *Journal of Software Maintenance: Research and Practice, 6(1),* pp. 35-52, 1994.

UML 2.0 — UML 2.0 Specification: www.omg.org/uml.

Wilde 03 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.

Wilde 95 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.

Witten 99 I. H. Witten, and E. Frank.Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, 1999.

Wong 05 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.

Wong 99 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.