FUNCTIONAL AND STRUCTURAL DESCRIPTORS FOR SOFTWARE COMPONENT RETRIEVAL

Yuhanis Yusof

A thesis submitted to the

School of Computer Science, Cardiff University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

December 2007

CARDIFF UNIVERSITY

DECLARATION
This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

[illegible] 10/01/08
Yuhanis Yusof Date

STATEMENT 1
This thesis is being submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (PhD).

[illegible] 10/01/08
Yuhanis Yusof Date

STATEMENT 2
This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references.

[illegible] 10/01/08
Yuhanis Yusof Date

STATEMENT 3
I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter library loan, and for the tittle and summary to be made available to outside organisations.

[illegible] 10/01/08
Yuhanis Yusof Date

ABSTRACT

FUNCTIONAL AND STRUCTURAL DESCRIPTORS FOR SOFTWARE COMPONENT RETRIEVAL

Identifying appropriate software components in a repository is an important task in software reuse; after all, components must be found before they can be reused. Program source code —— documents written in a computer programming language —— has the possibility to be a software component. Program source code is a form of data, containing both structure and function; it is therefore important to make use of this information in representing programs in a software repository. Existing approaches in software component retrieval systems focus on retrieving a component based on either its function or structure. Such an approach may not be suitable to users that require examples of programs that illustrate a particular function and structure, there is therefore a need for combining this information together. The objective of this research is to build a software repository of Java programs, to facilitate the search and selection of programs using the information about a program's function and structure. The hypothesis is that retrieval of program source code is better undertaken using a combination of functional and structural descriptors rather than using functional descriptors on their own.

This thesis presents a program retrieval and indexing model which can be used in developing a source code retrieval system. The model reveals on how functional and structural descriptors are identified and combined into a single representation. The functional descriptors are identified by extracting selected terms from program source code and a weighting scheme is adopted to differentiate the importance of terms. As programs in the repository are from open-source applications, extracting information that does not rely on semantic terms would be beneficial, as these programs are written by various developers with different programming background and experience. Structural descriptors that comprise of information generated based on structural relationships, such as design patterns and software metrics, are extracted from a program to be added as the program descriptor. The functional and structural descriptors are combined into a single index, known as a compound index, which is used as a program descriptor. The degree of similarity between a given query and programs in a repository is identified using measurements undertaken based on vector model and data distribution based approaches. Lessons learned from the experiments undertaken reveals that programs retrieved using the proposed method are less complex and easy to maintain. Furthermore, it is suggested that programs from different application domains contain different trends in their software metrics.

ACKNOWLEDGMENTS

First, I am ever grateful to God, the Creator and the Guardian, and to whom I owe my very existence. Second, I would like to express my deep and sincere gratitude to my supervisor, Prof. Omer F. Rana, for his support throughout the course of this thesis. I especially want to thank him for all the time he took to read and re-read this thesis so as to make it as good as possible. On the other hand, I also want to thank the people who served as evaluators in the evaluation phases of my thesis; research students at the Cardiff School of Computer Science.

On a different note, I am deeply indebted to my husband, Massudi Mahmuddin, whose patient love enabled me to complete this work. Besides pursuing his own PhD, he finds the time to help with the house chores; cook, clean, and looking after our daughter. In addition, he also provides stimulating suggestions and encouragement in all the time of research for and writing of this thesis. I also want to thank my daughter, Fatin Nabihah Massudi, who is the joy of my life, for letting *Ibu* to work on her thesis when she needed to do so.

Finally, a special thank you to my mum and dad, Siti Eshah Shariff and Yusof Abd. Ghani, to whom I dedicate my work. I am thankful for their love and best wishes, which despite of their physical absence have helped me in the successful completion of my study in Cardiff University. Last but not least, my special gratitude is due to my sister, my brothers and their families for their loving support.

Yuhanis Yusof

December 2007

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background

Software developers today need to create applications that are capable of providing various functionality and this has to be done quickly in order to overcome the issues of decreasing resources, time and budgets. Developing software that supports portability, flexibility, extensibility, and reliability is hard; developing high quality reusable software components is even harder [1]. To facilitate software developers in achieving such goals, software reuse, which is the process of developing software systems using existing software artefacts, has been a popular topic of debate and discussion for over 30 years in the software community. Software artefacts include software products, requirements and proposals, specifications, designs, program source code, program output, user manuals and test suites. Anything that is produced from a software development process can potentially be reused.

McClure [2] suggests that software artefacts have the possibility to be software components. Software component retrieval is an important task in software reuse; after all, components must be found before they can be reused. Based on existing work in software component retrieval [3, 4, 5, 6, 7, 8], there are two types of retrieval:

function-based and structure-based.

Given a query that describes what a required component should do, function-based retrieval presents developers with software components that *act* similarly. This means that the retrieved components illustrate the same function as that defined in the query. For example, a developer may require a program source code that illustrates the implementation of a solution to the `Tower of Hanoi` puzzle [9]. By defining the developer's query using the phrase *Tower of Hanoi*, a function-based retrieval system presents them with relevant programs that contain all or part of the search phrase. Similarity between the query and programs in the repository are performed using the textual analysis approach [10, 11, 12, 3], which uses term occurrences [13] to represent the function of a program. Nevertheless, various other methods have been used in representing functionality of a software component and these can be found in Chapter 2.

In contrast to function-based retrieval which identifies components that *act* similarly, structure-based retrieval presents developers with components that *look* alike. An example would be two distinct programs that illustrate factorial function using different approaches, e.g. recursion and looping; even though they have the same function, they have a different structure. Further elaboration on existing approaches of software component retrieval that identifies structural similarity can be seen in Chapter 2.

In most of the work undertaken in the area of software component retrieval, users are presented with components that are objects written to a specification such as Component Object Model (COM) [14], Java Beans [15], etc. It is only by adhering to the specification that the object becomes a component and gains features like reusability. Even though software component reuse has emerged strongly in software engineering, software developers who intend to use these components are inevitably restricted to the specification of the interfaces provided and required. When this information conflicts with developers' requirements, reusing the component is either impossible or requires the original system to be modified. Additionally, the developer could introduce a component adaptor [16] or some other wrapper [16] between the system and the component. As Holzle [17] shows, however, there are complications when multiple components must communicate with each other while they are contained within some form of wrapper object. Hence, as an alternative, developers tend to be opportunistic about reusing programs obtained from open-source applications.

This thesis employs the open-source program source code as the component that may be retrieved from a software repository. Further in the thesis, the program source code is referred to as a program and is defined as follows:

Definition 1 *A program is a single file containing segments of code statements that have been written to follow a particular language structure. For example, a program which has been written using the Java programming language, contains package and import statements, a class header and its body, and a method header and its body.*

As a greater number of software developers make their programs available, there is a need to store such open-source applications into a repository, and facilitate search through the repository. The work described in this thesis concerns the mechanism that supports the search and selection of programs from a software repository containing Java programs. Only programs written using Java programming language have been included in the repository due to the popularity of the language [18], and to ensure evaluation of the work can be performed adequately.

1.2 Research Problem

Existing approaches of function-based component retrieval [3, 4, 5, 6, 7, 8] use the function of a component as the focal point of comparison between a given query and components in a repository. Functionality of a component can be identified using various methods and this includes textual analysis of software components [10, 11, 12] and textual description of software components [4, 19, 20, 5]. Nevertheless, such methods concentrate only on the natural language text that exists in software documentation and/or programs. Therefore, only well-documented software is best suited for such retrieval methods. In the context of this thesis, what is meant by a well-documented software is the following:

Definition 2 *A well-documented software contains programs that use meaningful identifiers - identifiers are named based on their functionality. For example, a method named* add *illustrates the operation of adding two numbers. A well-documented software also includes software documentation (information describing the functionality of the software, and the required input and the expected output), program documentation (information explaining the functionality of classes and methods, and patterns used in implementing the code) and user manual (information on how to use the software).*

This means that software that is not accompanied with a user manual, and/or documentation would not benefit from textual analysis and/or textual description that are employed in function-based approaches. In addition, if a program is written using identifier names that do not explicitly reflect its function, then retrieval undertaken based on textual analysis and/or textual description may not present developers with relevant results. Furthermore, an inherent problem with many of these approaches [21, 19, 12, 22] is that they are based on constructing a specific domain model [12] or vocabulary [21, 19], which restricts the scope and flexibility of their solutions. Therefore, we need to support function-based retrieval by incorporating information that does not rely on semantics of a component (i.e meaning of statements in program source code or in software documentation).

Currently, there have been efforts to develop search engines specifically for retrieving source code. These include Google code search [23] and the Koders search engine [24], which make use of the term occurrences approach to represent the function of a program. A particular program is presented to the user if terms defined in a query occurred in code statements of the program. However, Google code search [23] does not perform a search in comment statements written in the program. Comments are a very useful index term within a program as they quite often adequately explain the functionality of the classes and methods contained in an object oriented program. This causes them to contain words that often would not be in the code itself. Furthermore, in the Google code search [23], there is a lack of domain-knowledge associated with its queries. For example, if a user requires programs that implement a connection to the SQL database, and the program should employ a particular design pattern such as Observer, how can such requirements be represented as a query in the Google code search? Another example would be the use of the term *add* as the query for programs in a Google code search. This search would present users with a very broad result; including how to add a record in a database, adding a panel into a GUI component, and the assignment of a value to the variable *add*. If the required program is from the application domain of Mathematics, i.e the term *add* is referred to a mathematical operation, programs relating to other application domains would not be useful.

Even though there has been work undertaken in retrieving components based on their structure [25, 26, 27], this work does not embrace knowledge buried in a program, such as design patterns that have used in developing the code itself. Existing work focuses on using pattern matching symbols [25, 26] in determining similarity between a source code query and programs in a repository. Such an approach may be beneficial to software maintainers who need to identify particular segments of code (e.g. a nested for) in an application, but would not help a developer whose intention is to find examples of programs that have been written to follow a particular pattern (e.g Observer design pattern). Furthermore, a pattern matching approach requires additional knowledge on pattern languages [25, 26] prior to defining a query for the source code.

Classification of programs into application domain would facilitate program retrieval as developers could identify useful programs quickly and easily [28]. As many developers are now posting their applications in open-source development repositories [29, 30], there is a need to automate organization of programs in such repositories. It is common to classify programs into application domains such as database, graphics, networking and security. Existing open-source repositories which include `Sourceforge.net` [29] and Freshmeat [30] classify a software into an application domain by using natural language descriptions provided by the developer and information extracted from the software documentation. However, such an approach may misclassify a software if it is not well-documented and/or is posted to the repository without relevant description by its developer. Nevertheless, neither existing function-based nor structure-based retrieval approaches have employed appropriate mechanisms to automate classification of components into application domains prior to retrieval.

While existing component retrieval approaches [23, 24, 25, 26, 27] are based on either the function or structure of a component, we are proposing to combine the two types of retrieval. In order to do so, the work described in this thesis concerns the use of two types of descriptors: functional and structural. These descriptors are defined as follows:

Definition 3 *Functional descriptors consist of information extracted from a program that represents the functionality of the program. This includes terms extracted from the code and comment statements written in a program.*

Definition 4 *Structural descriptors consist of structural information contained in a program that illustrate relationships between properties of the program. This includes class inheritance, interface hierarchies, method invocations and dependencies, parameters and return types, object creations, and variable access within a method. In addition, information inferred using structural information such as design patterns and software metrics are also considered as structural descriptors.*

By identifying functional and structural descriptors contained in a search query and programs in a software repository, software developers are not only presented with programs that function appropriately but also illustrate the required structure. Furthermore, no work has been undertaken in program retrieval that uses a combination of functional and structural descriptors. With an efficient organization of programs and the use of structural and functional descriptors, open-source applications stored in a software repository can be made better use of. By understanding a program's function and structure, software developers are able to better adapt a program for their own applications.

Our approach of combining functional and structural descriptors in representing programs in a software repository can also be extended to retrieve programs of other languages, such as C++. Relevant parsers can be developed to extract functional and structural descriptors from programs of different languages.

1.2.1 Why the Research is Important

This research is important to help developers answer questions that arise prior to retrieving relevant programs from a software repository. Such questions include, *what is the functionality of the program – what can be achieved when we execute the program?* and *which application domain can we use the program for? For example, is the program suitable for use in database applications?* In addition, a developer might notice that someone else's code seems simpler and works better than theirs, and they wonder how that particular developer achieves this simplicity. Therefore, a developer whose intention is to use the retrieved programs as guidelines in developing their own application may also like to know if the program has been written to follow a particular pattern. This raises questions such as *Are there any design patterns employed in the program? If so, what design patterns are they?*. Furthermore, what if the developer is also concerned about software quality [31]. As a developer performs a search and is presented with a list of programs that illustrate the required function, they are most likely to adapt a program that illustrates less complexity (e.g. containing fewer method dependencies).

Based on the programming task illustrated in Figure 1.1, a retrieval system which presents programs that function as required and illustrate the desired structure is best suited for those who have the knowledge of design patterns and are keen to use the knowledge in developing the application. These developers may define their query consisting of relevant terms and a particular design pattern. Based on their knowledge, they are able to identify suitable design patterns to be employed in the application by inferring structural information (i.e dependencies between *WeatherReporter* and *TextReport*) in the given task. Therefore, these developers may define their query as: *weather report application* AND *observer design pattern.*

Nevertheless, developers with little knowledge of design patterns can still benefit

Develop a weather reporting application **WeatherReporter** *class that stores the latest weather data on-screen, the weather is displayed by two classes:* **GraphicReport** *(cloud, sun, rain icons) and* **TextReport** *(Temperature: 25C, Sunny). When the weather changes,* **WeatherReporter** *sends updates to* **TextReport** *object and* **GraphicReport** *object.*

Figure 1.1 Programming Task - Weather Reporting Application

from the retrieval system. By using an existing program which they are currently writing for the programming task as a search query, they may still be able to retrieve relevant programs from a repository. Structural descriptors contained in the query program will contribute to the identification of similar programs in the repository.

1.3 Research Hypothesis and Questions

This thesis argues that retrieval of programs is better undertaken using a combination of structural and functional descriptors rather than using functional descriptors on their own.

The research hypothesis will be verified by developing a program retrieval system that is built upon open-source applications and that presents users with examples of programs that illustrate similar structure and function as illustrated in the query program. The research is structured around four central questions discussed in sections 1.3.1, 1.3.2 and 1.3.3. Performance of the program retrieval system is later evaluated through (1) objective analysis and (2) subjective experiments. The objective analysis involves measuring the processing time of the retrieval system upon receiving a search query and identifying the importance of structural descriptors in supporting program retrieval. On the other hand, in the subjective experiments, retrieval effectiveness of the proposed program retrieval system is analyzed through field experiments with developers.

1.3.1 Information Extraction

Question 1: *How can we extract information from a program that can be used as functional or structural descriptors in a program retrieval system?*

One of the first challenges that a retrieval system has to cope with is indexing the programs contained in a repository. This is achieved by extracting relevant information from a program to be used as the program descriptors. Open-source programs typically contain irregularities as they are written by different developers with different programming background and experience. Examples of such irregularities are identifier names used in a program that illustrate the content of the program and the practice of writing a program following a particular pattern. Programs in the repository may use similar identifiers, nevertheless they are employed in different context in the program. The challenge is to extract these identifiers and represent them (as indices) based on their contexts. An additional challenge is to find (new) abstraction information that is not explicitly available in the program (e.g design patterns) and can be used to represent a program. Thus appropriate parsers are required to extract different types of information that are explicitly or implicitly contained in a program.

1.3.2 Creating New Representation

Question 2: *How can we combine functional and structural descriptors of a program to represent a query and programs in a repository?*

Upon identifying functional and structural descriptors, how can this information be integrated as a program descriptor? An issue that should be considered is the flexibility of the proposed mechanism. It should be flexible enough so that additional descriptors (functional and/or structural) can easily be incorporated into it.

1.3.3 Supporting Program Retrieval

Question 3: *How can we use the information obtained in the first two questions to support and improve program retrieval?*

Several issues have to be addressed before the identified information in Question 1 can be used to improve program retrieval: how to deal with similar identifiers that represent the different contexts of a program (e.g. variable name, class name)? Also, what information can be inferred from structural information extracted from a program? In order to identify the benefits of incorporating structural descriptors as program descriptors, we need to perform relevant analysis on the programs retrieved for a given query.

Question 4: *How is similarity measurement undertaken between a query and the program in a software repository?*

The challenge is to identify measurements that can be used to determine similarity between a query and programs in the repository, which have been represented using the mechanism identified in Question 2. Prior to that, we need to identify how a query for source code is defined.

1.4 Scope of the Research

Although this thesis is set in the context of software reuse, the work undertaken is discussed based only on retrieval perspective, e.g how different retrieval indexing (functional or combination of functional and structural) affects the performance of a source code retrieval system. Issues related to whether the mechanisms used in determining functional and structural descriptors are sufficient enough for a source code retrieval system are not the focal point of the thesis. We are focusing to learn if the combination of functional and structural descriptors would generate a better retrieval when compared to using functional descriptors on their own.

The work in this thesis focuses on program written using the Java language. Functional descriptors identified from a Java program are restricted to keywords extracted based on a program structure. On the other hand, structural descriptors that were used in this work are the design patterns, application domains and software metrics. Three design patterns, namely Singleton, Composite and Observer, are identified using the proposed design pattern identification mechanism. Currently, only programs from database and graphics domains are included in this work and the classification of programs into application domains is performed based on their software metrics.

1.5 Research Contributions

Several of the results set this research apart from other related approaches. The overall solution is general and applicable to a wide range of programming languages and application domains. The research proposes and validates a new model that supports source code retrieval and is applicable to a wide range of programming language. The experiments shows that this approach provides better support for a number of programming tasks (e.g connecting to a database system, retrieving data from a database and organizing a set of images contained in a folder). The contributions of this thesis are as follows:

- A model for extracting functional and structural descriptors contained in a program source code. These descriptors are identified separately and later combined into a single representation which is known as a compound index. Such an approach can be extended to include other descriptors as identified and/or required by the user.

- A model for retrieving programs based on a user providing the requirements of a program, in a form of a query program (i.e program source code). This model includes how similarity between a query program and programs from a repository is identified and how programs that are relevant to the search are sorted in the retrieval hit list.

- A new way of identifying design patterns employed in programs contained in a repository. The identification mechanism is solely based on structural relationships, hence it can easily be modified to be used on programs written using programming languages other than Java. Moreover, the proposed mechanism can be extended to identify other design patterns as elaborated by Gamma et. al [1].

- Classification of programs into applications domains; database and graphics. The classification is performed using software metrics contained in the program and such an approach can be used to support program classification undertaken based on semantic meanings.

1.6 Organization of the Thesis

Chapter 2 of this thesis presents a review of current work in the area of software component retrieval. In particular, it focuses on component retrieval undertaken using functional descriptors and structural descriptors on their own.

In Chapter 3, we describe the use of term occurrences, which are accompanied by a weighting scheme to be used as query and program descriptors. Chapter 4 focuses on utilizing structural information contained in a program where we propose the use of design patterns as structural descriptors of a program. In this chapter, we demonstrate the identification of three design patterns contained in Java programs.

Chapter 5 illustrates how software metrics extracted from a program are used to support program retrieval. The metrics are used to classify a program into appropriate application domain and to represent program reusability. It is demonstrated later in the chapter that program retrieval that includes program classification, undertaken based on software metrics, is better than the retrieval performed based only on semantic terms.

Chapter 6 is central to this work, and gives details of how functional and structural descriptors identified in Chapters 3, 4 and 5 are incorporated into the program retrieval system. We describe here how the similarity measurement is undertaken between a query and programs in the repository.

Evaluation of the program retrieval system is described in Chapter 7, which also includes empirical subjective evaluations. In addition, lessons learned from the evaluation are also presented and discussed in this chapter.

Chapter 8 concludes the thesis by summarizing the contributions made and discussing future research directions.

1.7 Origins of the Chapters

Parts of this thesis were published previously. Portions of Chapter 3 are based on the work presented in Yusof and Rana [32]. Portions of Chapters 4 and 6 are extended from Yusof and Rana [33] and Yusof and Rana [34]. Most of Chapter 5 contains the content of an article submitted to the IEEE Software Engineering, which is currently under review, and most of Chapter 6 are based on the work presented in Yusof and Rana [35].

CHAPTER 2

LITERATURE REVIEW

The development of a system for retrieving programs from a software repository involves an understanding of software component retrieval. Issues related to representation of programs in the repository are described in this chapter since they form the basis for the research described in the subsequent chapters. The general area of research under investigation here is related to applications of functional and structural descriptors to address software component retrieval tasks.

2.1 Software Component Retrieval

A component retrieval mechanism works in the following way, as described by Mili et al. [36](Figure 2.1): when faced with a programming task, the user understands it in his or her own way, and then formulates a query, which may be as simple as a set of keywords or as complex as specifications in a formal language. An example of this is when a user wants to write a Java program to solve the `Tower of Hanoi` puzzle [9]. One possible way to represent a query for source code is by using a set of keywords, such as *Java program Tower of Hanoi*. In practice, this first process results in the loss of information since the user is not always capable of exactly understanding the problem, or being definite about the required problem solution or of encoding the required solution in the query language. If the user is not aware of the different ways of implementing the problem using the Java language, which includes recursive and non-recursive solutions, then s/he will not include the required solution in the query. Once a query has been generated, it is passed to the Matcher as shown in Figure 2.1, that is responsible for identifying similarity between the query and indices in the code library. This process of classification (also known as indexing), may be manual or automatic, and also results in the loss of information. This occurs because since a component embodies various features, it is difficult to identify all of these features and use them as code indices. For example, several pieces of information can be used to represent the functionality of a component (functional descriptors), such as the terms extracted from code statements, formal specifications of the component and the sample of input/output data related to the component. However, if the indexing is based only on a particular descriptor, for example sample of input/output data, then a query that is represented using formal specifications may generates irrelevant results. The search itself consists of comparing the query with the index and returning the components that match the query. This information loss is the focus of all the work in this area —— representing a program based on information that is anticipated to be included in a query.

An application may contain more than a single artefact (e.g program source code, user manuals, design documentation). Perhaps the most well known reusable artefact is the program source code. This is because it is the most up to date artefact. Developers may have made many changes in programs in order to achieve the desired functionality but these changes may not be reflected in the documentation included in the application. Hence, programs demonstrate best what function the application offers and how it is implemented. The common practice in existing retrieval systems

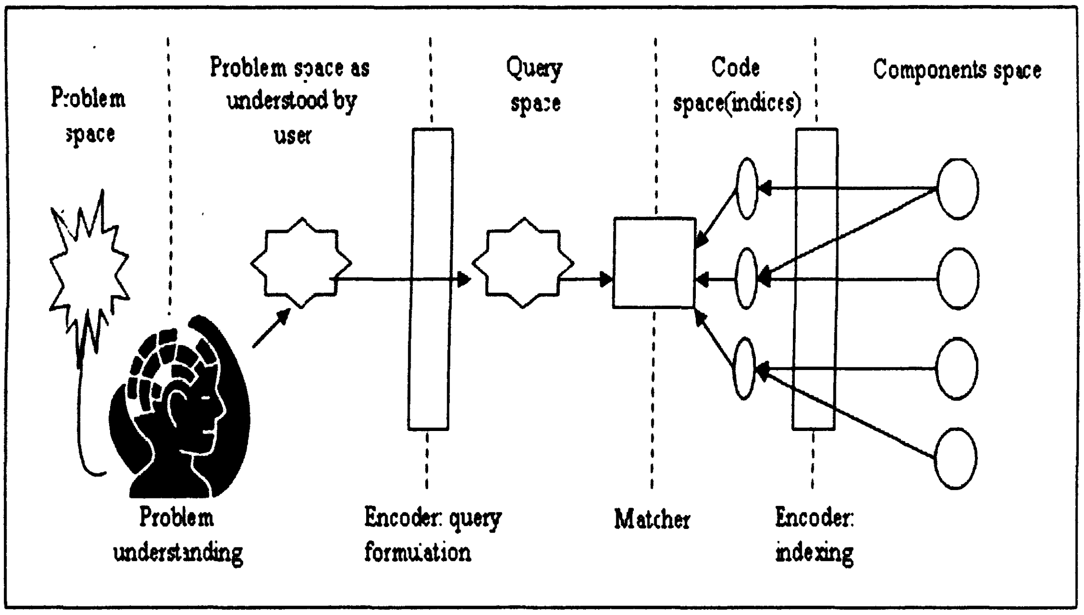


Figure 2.1 Component Retrieval Model (in Mili et al., 1995 [36])

is to identify relevant programs based on their functionality. Therefore, a program that is written to achieve a particular function (e.g adding two values) should be represented by descriptors that abstract its most relevant functional (semantics) features. Nevertheless, it should also be represented in a way that focuses on its relevant structural (syntactic) features. This is because prior to software implementation, developers tend to model the problem (programming task) using various modelling tools (e.g UML [37]). Such a process generates structural features of the components to be developed, for example, the relationships between two objects. A combination of functional and structural descriptors (refer to Definitions 3 and 4 on page 7) to represent programs in a repository would help developers to retrieve programs that illustrate the required function and structure as modelled in the design documentation (e.g entity relationship diagram (ERD)).

As existing source code retrieval systems such as Google code search [23] and Koders search engine [24] only use functional descriptors in identifying similarity between a query and programs in a repository, structural descriptors of a program have not been utilized. Nevertheless, developers may require programs that illustrate a particular function in a certain way. Developers should not only benefit from *cutting and pasting* code statements from a program, other information embedded in the code can also be reused. This can be achieved by including structural descriptors in representing a program. Basili et al. [38] defined a reusable program as the realization of some software development experience. Such an experience refers to the way how a problem solution is designed prior to implementation.

2.1.1 Function-based

In the literature, several efficient ways to retrieve various types of software components have been found [39, 4, 40, 41, 42, 43, 44, 7]. We present related work on software component retrieval based on functional and structural descriptors as defined in Definitions 3 and 4 on page 7. Approaches undertaken in software component retrieval based on functional descriptors are identified as using the following methods:

Information retrieval: these are methods that depend on a textual analysis of software components. Components are represented using text and relevant components are identified by understanding the meanings of the text that represents the component [10, 11, 12, 3, 45].

Descriptive: descriptive methods depend on abstract representation of the components. Such representation includes the use of a set of keywords or a set of facet definitions [4, 20, 46, 47, 48, 5]. In deriving a faceted classification scheme, the objective is to create and structure a controlled vocabulary [49] that is standard not only for classifying but also for describing a component in a domain specific collection. Retrieval of relevant components is undertaken by identifying components that minimize some measure of distance to the user query [39, 41, 50, 51, 7]. Given a query that describes some required features of a component, the retrieval system retrieves components that most closely match a description of the features.

Operational semantics: these methods depend on the operational semantics of the software components. This means that components are represented by how they function. They exploit the executable nature of components by comparing the input/output data specified by a search query to the one produced by stored components [52, 53, 6].

Denotational semantics: these are methods that depend on the denotational semantic definition of the software component. Denotational semantics is an approach to formalizing the semantics of a component by constructing mathematical objects (called denotations or meanings) which express the semantics of these components. These methods proceed by identifying a semantic relation between the user query and software components [54, 40, 43, 55, 56, 57, 42, 8].

Nevertheless, only work undertaken using information retrieval and descriptive methods are elaborated upon in this thesis as the proposed program retrieval system employs a similar approach to these methods.

Information retrieval methods

Related work in information retrieval methods and their applications to the domain of software is of importance. The research that has been conducted on the specific use of applying information retrieval methods to source code includes Fischer [11], Frakes and Nejmeh [10] and Maarek et al. [12]. Notable is work by Maarek et al. on the use of an information retrieval approach for automatically constructing software libraries. Their method relies on a natural language description of software components and search queries. The indexing process automatically extracts a set of indices that define its profile based on uncontrolled vocabulary. The uncontrolled vocabulary, also referred to as free-text analysis, consists in analyzing term frequencies in natural text [58]. On the other hand, controlled vocabulary consists of terms that are established in order to group similar components [49]. The idea of a controlled vocabulary is to reduce the variability of expressions used to characterize the component being indexed, e.g. by avoiding synonyms and remove ambiguity (homonyms). Such an approach can be seen in the work undertaken by Prieto-Diaz [4] and Yang et al. [5] described under the descriptive methods.

Because of the unlimited number of terms (uncontrolled vocabulary) used to represent a component, the search space in identifying relevant components is large.

hence generating a greater possibility of having false positive results. To overcome such a problem, Lindig [3] proposes that a user incrementally specifies a set of keywords that the searched components are required to have. Such an approach is based on precalculated *concepts* of the library, which are natural pairs of component and keyword sets. The concepts form a lattice of super and subconcepts and are obtained by *formal concept analysis* [59].

Marcus et. al [45] employ the term occurrences approach to indicate domain knowledge and concepts embedded in a program source code. Identifier names and comments are extracted from the program before latent semantic indexing (LSI) [60] is performed. In addition to recording which keywords a program contains, the LSI examines the program collection as a whole, to see which other programs contain some of those same words. LSI considers programs that have many words in common to be semantically close, and ones with few words in common to be semantically distant.

Similar to the work undertaken by Marcus et. al [45], we extract identifier names from a program to represent the function of the program. In order to overcome the drawback of using uncontrolled vocabulary (i.e large search space), we include information on the program context for each of the extracted identifiers. Details of the approach can be seen in Chapter 3.

Descriptive methods

Prieto-Diaz [4] extended the use of keywords into a multi-dimensional search space through the use of a facet, consisting of a set of predefined keywords. There are three steps involved in retrieving relevant software components. First, users need to formulate the query and this is undertaken by selecting appropriate terms from a list of provided terms (known as term space) for each facet in the classification. To solve ambiguities, a thesaurus is designed by the researcher for each facet to make sure the keyword matched can only be within the facet context. Examples of the facet might be a function. object/item-type. and system-type. Thus, organizing a collection of software components into n facets implies that a query into the search space of the collection would be made up of an n-tuple of keywords with the ith keyword drawn from the term space of the ith facet. To determine similarity between a query and software components, a weighted conceptual graph [4] is used to measure closeness according to the conceptual distance among terms in a facet. The third step is to rank the retrieved components. The ranking subsystem is based on reuse related metrics. This estimates, for each of the retrieved components, the relative effort it would take to reuse the component, that is, the effort required to adapt and integrate the component into the new system. Components requiring the least amount of effort are ranked at the top of the retrieval list.

Building on the work undertaken by Prieto-Diaz [4], Yang et al. [5] focus on the problem of how to determine the ranks of the components retrieved by users. Factors which can influence the ranking are extracted and identified through the analysis of an ER-Diagram of the facet-based component retrieval system. Faceted classification and retrieval has proven to be very effective in retrieving suitable components from repositories [4, 5], but the approach is labour intensive. The reason for this is the need for deriving and defining terms by experts so that the terms can later be used in representing concepts relevant to the facet. From this, it has also been learned that faceted classification is more effective for domain-specific collections than for broad, heterogeneous collections such as an open-source repository. Even though this method is gaining increasing attention because it takes domain knowledge into account when designing facets [7], there exists a major concern in designing the facets. If facets are designed too simple or few, there will be too many components in the retrieval list, which will require users to examine the components manually in order to determine the relevant ones. On the other hand, if facets are designed to be too complex, it is hard for users to understand them and hard for the repository administrator to classify all components into different categories. Moreover, the process to classify the components is susceptible to being subjective, so that two different people may choose different keywords or facets to describe the same component. In this sense, we employ automatic indexing to extract, from code and comment statements, terms that describe a component.

In the work undertaken by Girardi and Ibrahim [47], an acquisition mechanism automatically extracts from software documentations the knowledge needed to catalogue them in a software base. The system extracts lexical, syntactic and semantic information and this knowledge is used to create a frame-based internal representation for the software component. The interpretation mechanism used for the analysis of a software documentation does not pretend to understand the meaning of a description. It attempts to automatically acquire information to construct indexing terms for a software documentation. The WordNet [61] lexicon is used to obtain morphological information, grammatical categories of terms and lexical relationships between terms. The software base contains a collection of frames, and each software component (i.e software documentation) has a set of associated frames containing the internal representation of its description along with other information associated with the component (e.g program source code). The retrieval mechanism looks for and selects components from the repository, based on the closeness between the frames associated with a query and the software components. Closeness measures [62] are derived from the semantic formalism and a conceptual distance between the terms in the frames under comparison. Software components are scored according to their closeness value with the user query. The ones with a score higher than a controlled threshold become the retrieved software components.

Similar to the work undertaken by Girardi and Ibrahim [47], Gu et al. [48] also represent components to be stored in the repository using frames. They adopt a frame-based representation and reasoning system, *CREEK* [63], which unifies component-specific cases and general domain knowledge within a single representation system. In *CREEK*, information describing the functionality of a component is represented as concepts, and a concept takes the form of a frame-based structure, which consists of a list of slots. A slot acts as a relation from the concept to a value related with another concept. Viewed as a semantic network, a concept (frame) corresponds to a node, and a relation (slot) corresponds to a link between nodes. Slot values have types or roles, referred to as facets. Similar to the work undertaken by [4], such a approach is also labour intensive as participation of an expert is required to design the frame.

We are taking a similar approach to [4, 47] to represent functionality of components by extracting relevant information from software components. Nevertheless, our approach does not require the participation of an expert to design the facet and determine suitable terms to be included in the term space. We employ program structure as the facets and use relevant terms extracted from a program as the term space for the appropriate facet. Furthermore, we include a weighting scheme to illustrate the importance of the facets. Elaboration on program structure and the weighting scheme can be found in Chapter 3.

2.1.2 Structure-based

It is fair to say that most of existing software component retrievals identify relevant components solely on the basis of their function: the system decides whether to select a software component by matching the functional descriptors (refer to Definition 3 on page 7) of the candidate component against desired functional features.

An alternative rationale is to select software components not on the basis of their function but rather on the basis of their structure: the system selects a software component whenever there is a reason to believe that a possible solution to the query has the same structure as the software component under consideration. Function-based retrieval is very important, as it can provide effective and precise retrieval results. Unfortunately the semantics of a software component identified using information retrieval and descriptive methods may be hard to determine if the software is not well-documented (refer to Definition 2 on page 4). Therefore, an alternative to using functional descriptors in retrieving relevant components from the software repository would be beneficial. In addition, the software repository used in our work contains applications obtained from open-source repositories. This means that:

- it is contributed to by various developers, each with a different style of writing programs. This includes not naming objects and methods based on the functionality that they offer. In addition, the repository might also include applications that are not well-documented. If the accompanied documentation is poor, how can the existing information retrieval and descriptive methods that rely mainly on text description be used as program descriptors?

- it may contain an application that requires a different environment or platform. With this in mind, the possibility of identifying the desired components using operational methods is lessened if developers do not have the appropriate environment.

- there are possibilities that there is only program source code included in an application. Since most of the developers do not include specification documents in the application to be stored in the repository, retrieval methods based on denotational semantics are not suitable. In addition, most of the denotational semantics approaches require a representation of the program in formal language. Even though one could translate the appropriate repository contents into a formal specification, it is unlikely to happen as such a process requires additional effort especially if it is done manually. Furthermore, currently there are no tools created to automate the translation process.

Programming Cliches

Over the past couple of decades, researchers have been investigating tools that can help in the process of program understanding. One such tool attempts to recognize common *programming cliches.* In this context, a cliche is a pattern that appears frequently in many different programs (and possibly many different languages). Developers learn these patterns and use them to speed up the process of code construction: when they need to produce some behaviour that matches a pattern, they do not need to think about each line of code they write, but instead let their subconscious memory of the pattern generate the required statements. For example, developers have probably already learnt the pattern for iterating through an array and can write such behaviour quickly and reliably.

There are two types of programming cliches: general purpose cliches and specific-domain cliches. The former refers to cliches that occur in programs throughout all problem domains, such as iteration, while the latter are cliches that can be found only in a particular domain. Typically, the specific domain cliches can be built on top of the general purpose cliches. For example, such cliches can be found in programs that sequentially simulate parallel systems. An elaboration of the example can be found in the work undertaken by Wills [27].

Before any cliche can be retrieved by users, it needs to be identified. The recognition methods of programming cliches can be categorized into two categories: textual analysis [64] and graph parsing [27]. In the latter work, *GRASPR* is used to translate a program into a language-independent graphical representation (i.e flow graph). The cliches and the relationship between them are encoded in graph grammar rules. Before a program is translated into a flow graph, it is first translated into a Plan Calculus representation [65]. The structure of this graph explicitly captures data and control flow, as well as aggregate data structure accessors and constructors and recursion. Later, the translation process encodes the plan into an attributed flow graph representation [27]. Recognition of programming cliches is undertaken by parsing the program graphical representation in accordance with the graph grammar encoding of the cliches.

The specification of a generic problem results in the creation of a problem schema that is analogous to the notion of cliches in the Programmers' Apprentice [64]. Waters [64] used a variation of Ada's procedure notation in representing cliches. Such a form specifies the name of the cliche, and some declarations that define the important features of the cliche, as well as the computation that corresponds to the cliche. The cliches are stored in a library that is structured by the hierarchical generality relation. Examples of cliches are *FileEnumeration*, which sequentially enumerates all the records of a file and *SimpleReport*, which produces a report from a file, according to a predefined format. Retrieval is later undertaken by matching cliches' names against queries that are submitted using natural language. Building on this work, other types of component retrieval based on such structural descriptors has been undertaken. For example, Waters and Rich [66], later expanded the work done by Waters [64] by implementing the idea in the Design Apprentice [67]. The knowledge of the Design Apprentice is personified in its cliches for typical specifications, design and hardware characteristics. Examples of these cliches include initializing a device, a generic device driver and an interactive display device. While a cliche may represent the implementation of a piece of an object, it does not illustrate the interaction of the object with other objects, which may be depicted in a design documentation (e.g Entity Relationship Diagram (ERD)). Therefore the existing methods [64, 66, 67] are less useful for developers who are seeking for components that illustrate a similar design as defined in their design documentation.

Pattern Language

Santanul and Atul [25, 26] presented a framework in which pattern languages are used to specify the required code features. The pattern language is derived by extending the source programming language with pattern-matching symbols. *SCRUPLE*, a finite state machine-based program search tool implements the proposed framework. In *SCRUPLE* [25], the extensions include a set of symbols that can be used as substitutes for syntactic entities in the programming language. For example, a code statement of `x = x + 1` is represented as `$v3 = $v3 + 1` in the proposed pattern language. When a search specification is written using one or more of these symbols, it plays the role of an abstract template that can potentially match different code fragments. If no symbol is used, the specification consists only of constructs that are valid in the programming language, which effectively makes it a valid code fragment in itself, and hence leads to only precise matches. While this is a powerful method for maintainers of large software projects, it lacks the common retrieval *fuzziness* where components are relevant for a query, but do not necessarily match it. Additionally, the method requires some training prior to usage, because its query language is not standard. If a user fails to understand and use the pattern language effectively, s/he may be presented with a limited set of code fragments or even worse, s/he may not get any results at all.

Design Patterns

One of the characteristics of developing a reusable software component is to follow existing standards so that it can later be used by not only the developer himself but by other people who require components with the same capability. This includes patterns which are devices that allow software developers to share knowledge about their software design. In daily programming, developers encounter many problems that have occurred, and will occur again. The question that may arise is how the developer is going to solve it this time. Documenting patterns is one way that developers can reuse and possibly share the information that they have learned about how it is best to solve a particular problem (i.e programming task). Gamma et al. [1] employs Alexander's idea of explicitly describing implicit design knowledge and best practices [68] in software design and such an approach has rapidly spread to various scenes in software development. Like Alexander's pattern language [68], a design pattern is considered a well-formed language to represent software design. A design pattern names, abstracts and identifies the key aspects of a common design structure that can be used to develop a reusable program. Design patterns also identify the participating classes and instances, their roles and collaborations, and the distribution of responsibilities [1].

A design pattern is a way to pursue an intent - that uses classes and their methods in an object-oriented language [69]. A description of design patterns can be found in a documentation format such as described by Gamma et al. [1]. The authors [1] presented 23 design patterns using a template containing of 13 characteristics - *Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses* and *Related Patterns.* Gamma et al. [1] claimed that the template lends a uniform structure to the information, making design patterns easier to learn, compare and use. Even though software developers can learn about the use of design patterns through this documentation which includes examples of code fragments, they may later have the problem of modifying the code to illustrate the required function that suits a particular domain. As a retrieval system may be used as a learning tool, we see the need of presenting users with programs that not only functioned appropriately but also illustrate the required design pattern. With this, the users can use the retrieved programs as guidelines in creating their own applications.

A work conducted by Prechelt et al. [70] suggests the idea of using patterns in developing an application can often result in components that are more easily maintained and modified. Given the frequency with which the need for modifications arises in software development, the added flexibility that comes from using a pattern seems to be a more optimized structure. Therefore, design patterns are clearly a useful addition to the developer's vocabulary and programming skill. Indeed, it can be argued that even if design patterns are not widely employed in programs, because of the complexity, simply studying them will itself encourage the development of clearer thinking about design problem solution, and will convey some of the benefits of experience. Even though design patterns are mostly likely to be used in forward engineering process, such as when developers move from the design to the implementation phase, they are equally important in the reverse engineering process. In this thesis, reverse engineering is focused on the task of identifying design patterns embedded in programs contained in an application.

Current approaches of design patterns detection can be categorized according to the kind of analysis they perform: static [71, 72], dynamic [73, 74] or a combination of static and dynamic [75]. Static analysis is performed by examining the code without executing the program and such a process provides an understanding of the code structure. On the other hand, dynamic analysis involves the execution of the analyzed program. As open-source applications may require different execution environment for it to be executed, the dynamic and a combination of static and dynamic analysis in design pattern detection may not be suitable. Therefore, we only focus on existing work that detect design patterns based on static analysis.

Most of the work undertaken using static analysis requires the analyzed program to be represented in an intermediate form such as an abstract syntax tree (AST) [76, 77, 78] or an American Standard Code for Information Interchange (ASCII)-based representation [71, 72]. Using AST as the intermediate format, every source file is entirely represented as a tree of AST nodes. The first step in *SPQR* [76] is to translate the AST obtained by GNU Compiler Collection (GCC) [79] to a format recognized by a theorem prover. GCC is an integrated distribution of compilers for several major programming languages which currently includes C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada. In *SPQR*, the *gcctree2oml* tool was included to read a tree file and later produces an XML description of the object structure features. A second tool, *oml2otter* then reads this XML description and produces a feature-rule input file to the automated theorem prover, *OTTER* [80]. *OTTER* finds instances of design patterns by inference based on pre-defined rules employed as denotational semantics. This approach relies heavily on the accuracy of the information extracted in the first stage. Although extracting structural relationships seems straightforward, it is complicated by variations in the implementations of some relationships, such as aggregation [78]. Thus, these approaches can result in higher false positive or false negative rates.

The design pattern detection mechanism introduced in *FUJABA* [78] works on the abstract syntax tree (AST) which is produced by the JavaCC source code parser [81]. The design pattern detection mechanism is based on graph grammars working on the AST and the patterns to be detected are defined by graph transformation rules.

Each rule transforms a particular graph structure, i.e. a pattern or a subpattern, and annotates it with an additional node to indicate the found design pattern instance and additional edges to indicate the participants of this instance. All patterns and structures are organized in a graph that shows the compositions of patterns and substructures and builds a dependency hierarchy between them. It analyses the rules applied to the AST, and also tries to apply the transformation rules of patterns which depend on these rules.

Previous work [71, 72] has used a structural analysis of code structure to identify design patterns defined by Gamma et al. [1]. Keller et al. [72] use the C++ program analysis system, *GEN++* [82] to generate an American Standard Code for Information Interchange (ASCII)-based representation of the relevant source code elements (UML/CDIF Intermediate Source Model). They [72] adopt the CDIF transfer format [83] as the syntax and the UML metamodel 1.1 [37] as the semantic model of the intermediate format. Keller et al. [72] extract structural relationships from the C++ source code and stores this information in an object oriented database. However, their approach requires developers to manually group design elements, such as classes, methods, attributes, or relationships to reflect a pattern.

The approaches [78, 76, 71, 72] discussed above are restricted to having an intermediate mechanism in detecting design patterns embedded in a program. They require either translation of patterns [78] or programs under analysis [72] into a particular representation.

Software Metrics

Software metrics can be classified as either product metrics or process metrics [84]. Process metrics are measures of the software development process, such as overall development, type of methodology used, or the average level of experience of the programming staff. On the other hand, product metrics are measures of the software product (e.g program source code, software design, software documentation) at any stage of software process, from requirements to installed systems. Product metrics may measure the complexity of the software design, the size of the final program (source code), or the number of pages of documentation produced. Examples of product metrics are as follows:

1. Number of Modules (Nom) - modules in terms of a grouping of member functions. For example, the C++ classes, Java classes and interfaces and Ada packages are defined as modules.

2. Lines of Code (Loc) - this count follows the standard of counting non-blank, non-comment lines of source code. Preprocessor lines are treated as blank. In the context of this thesis, class and function declarations are counted, but declarations of global data are ignored as such declarations(if the variables are of the same type) can be made on a single line.

3. McCabe's Cyclomatic Complexity (Mvg) - a measure of the decision complexity of the functions that make up the program. The definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph that maps the flow of control of a given code fragments. An analyzer counts this by recording the number of distinct decision outcomes contained within each function, which yields a good approximation to the formally defined version of the measure. Cyclomatic Complexity essentially represents the number of paths through a particular section of code, which in object-oriented languages applies to methods. Cyclomatic Complexity's equation from graph theory is as follows: $CC = E - N + P$ where E represents the number of edges on a graph, N the number of nodes, and P the number of connected components.

Cyclomatic complexity can be explained as follows: every decision point in a method (e.g `if`, `for`, `while`, or `case` statement) is counted; additionally, one is added for the method's entry point, resulting in an integer-based measurement denoting a method's complexity. For example, the code fragments illustrated in Figure 2.2 will yield a cyclomatic complexity value of 3. There are two decision points: an `if` and an `else`. Another value is obtained by adding the method's entry point which automatically adds one. The less the complexity, the better. More complexity means developers have more decision making and branching occurring inside the code fragments. This makes it harder to test the function.

```
public int getValue(int param1) {
int value = 0;
if (param1 == 0) {
value = 4;
}
else {
value = 0; }
return value; }
```

Figure 2.2 Method getValue

4. Depth of Inheritance Tree (Dit) - measures the depth of a class in the inheritance tree. If the whole inheritance graph is a tree, then Dit is the path length from the root to the class under investigation. This metric can be used to determine the complexity of a class based on its ancestors, since a class with many ancestors is likely to inherit much of the complexity of its ancestors.

5. Coupling Between Object (Cbo) - the use of another object's methods or instance variables. Since this creates dependencies between objects, the higher the number, the greater the possibility that reusability of class may decrease. When either one object uses another object, then both objects are said to be coupled. One major source of coupling is that between a superclass and a subclass. A coupling is also introduced when a method or field in another class is accessed, or when an object of another class is passed into or out of a method invocation. The more independent a class is, the more likely it is that it will be possible to reuse. When a class is coupled to another class, it becomes sensitive to changes in that class, thereby making maintenance difficult. In addition, a class that is overly dependent on other classes can be difficult to understand and test in isolation. In the context of the work undertaken in this thesis, Cbo is defined for classes and interfaces, constructors and methods. It counts the number of reference types that are used in:

   - field declarations
   - formal parameters and return types
   - throws declarations
   - local variables

   For example, the Cbo for class `ComboBoxEditor` which is illustrated in Figure 2.3 is 3: `Component` the return type for method `getEditorComponent` counts as 1, `Object` is counted as 2 since it is the return type for method `getItem` and it is also the an argument for method `setItem` and `ActionListener` also counts as 1 as i is the argument method `addActionListener`.

```
public interface ComboBoxEditor {
public Component getEditorComponent();
public void setItem(Object anObject);
public Object getItem();
public void selectAll();
public void addActionListener(ActionListener l);
}
```

Figure 2.3 Class ComboBoxEditor

6. Weight Method per Class (Wmc) - the sum of a weighting function over the functions of the module. The *Wmc* uses the nominal weight of 1 for each function, and hence measures the number of functions; the larger the number of methods in a Java class, the more complex the children will be because of inheritance. A high number of methods will lessen the potential for class reuse because the class is likely to become application specific. For example, the Wmc for code fragments contained in class `QueryReportResult`, depicted in Figure 2.4, is 2 for Wmc - constructor `QueryReportResult` and method `getTemplate` are counted as one respectively.

7. Fan-In measures the number of programs that pass information into the current program. For a given program A, the Fan-In is the number of other programs which use A. For example, the number of other programs (known as suppliers) that pass information into the class `ComboBoxEditor` (known as client) in Figure 2.3 is equal to 3 - `Component`, `Object` and `ActionListener`. Three variants of Fan-In are presented: a count restricted to the part of the interface that is externally visible (Fivis), a count that implies that changes to the client must

```
public class QueryReportResult extends VelocityResult {
public QueryReportResult() {
try{
velocityEngine.init();}
catch (Exception e) {
log.error(e); }
}
protected Template getTemplate(OgnlValueStack stack,
VelocityEngine velocity, ActionInvocation invocation, String
location) throws Exception {
Action action = invocation.getAction();
return super.getTemplate(stack, velocity, invocation, location);
}
}
```

Figure 2.4 Class QueryReportResult

be recompiled if the supplier's definition changes (Ficon), and an inclusive count (Fiincl) of Fivis and Ficon.

8. Fan-Out measures the number of programs that accept information from the current program. For a given program A, the Fan-Out is the number of other programs which A uses. Similar to Fan-In, three variants of Fan-Out are presented: a count restricted to the part of the interface that is externally visible (Fovis), a count that implies that changes to the client must be recompiled if the supplier's definition changes (Focon), and an inclusive count (Foincl).

Reusability is the degree to which a component can be reused, and reduces the software development cost by enabling less writing and more assembly. How users can detect which component is the most reusable among several components implementing the same function, and how users can select components with higher reusability are key issues. Therefore, existing studies measure the reusability of components in order to realize the reuse of components effectively [85, 86]. In the work undertaken by Caldiera and Basili [85], domain experts determine components that have reuse potential according to their experience and knowledge. Nevertheless, they paid too much attention to the component function and neglected the quality of components.

In the work undertaken by Lai and Yang [87], they proposed a combination of several metrics to be used by experts in identifying high quality software components. Their approach defined the software component as including design specification, program, and related documentation. Software metrics that are to be used include the McCabe's Cyclomatic complexity (Mvg) [88], Halstead data structure metric [89], nesting level of program construct [90], test coverage [84], and coupling and cohesion metrics [90]. To provide overall measurement of the reusable software component, Lai and Yang [87] combine these metrics using a dynamically weighted linear combination that allows the assignment of different weight values to the same metrics in different situations.

Washizaki and Fukazawa [86] present metric values of the JavaBeans [15] components that are selected by a user from the retrieval list. They include the Depth of Inheritance Tree (Dit) [91] and SCCr [92] in helping a user to decide whether the selected component is suitable to be adopted into the application s/he is working on. Nevertheless, Washizaki and Fukazawa did not demonstrate if the metrics can be used in identifying similarity between a search query and components in the repository.

Based on existing work in software metrics, reusing a software component with high reuse potential and high quality contributes to improve software quality and productivity [87, 86, 90]. With this in mind, we include software metrics, that measures the quality of a component, as structural descriptors of a program. In order to identify the high quality programs from a software repository, discussion of the measurable characteristics of reusable programs and their corresponding metrics is a necessary step. Among the characteristics of a reusable program are the complexity and coupling between objects [87].

Based on existing studies [87, 93, 94, 95], in this research, we include six software metrics to represent quality measurement of a complex program: Mvg, Wmc, Fan-In (Fivis, Ficon) and Fan-Out (Fovis, Focon). The complexity of a program is a measure of the effort required to understand the program and is usually based on the control and data flow of the program. While opinion as to what construes code complexity is quite subjective, over the years the software industry has largely agreed that a highly complex code can be difficult for software developers to understand and therefore is harder to maintain [93, 95]. Moreover, a highly complex code has a high probability of containing defects. Various studies [96, 97] have suggested that a Cyclomatic Complexity (Mvg) value of 10 or higher for a particular program is considered complex.

Wmc is a predictor of how much time and effort is required to develop and maintain a program. The higher the Wmc, the greater amount of testing is required and the amount of maintenance is increased. Rosenberg et al. [93] suggest that an object oriented class should have less than 20 functions, but up to 40 is acceptable. They also claimed that with further analysis, programs with Wmc greater than 40 have a low reliability [93].

The Fan-In and Fan-Out metrics maintain a count of the number of data flows from and into a program plus the number of global data structures that the program updates. The higher the values of these metrics (i.e Ficon, Fivis, Fovis, Focon) in a particular program, then the more complex the program is [84].

In the process of implementing an object oriented application, developers need to ensure that sets of classes are loosely coupled [98]. An application that is loosely coupled implies the number of relationships among all classes in the application has been kept to the minimum. If every object has a reference to every other object, then there is high coupling, and this is undesirable because there is potentially too much information flow between objects. Hence, low Cbo is desirable; this means that objects work more independently of each other. Developers who are searching for examples of programs to be reused, would benefit from retrieving programs with low coupling – low coupling programs minimize the ripple effect where changes in one program cause the necessity for changes in other programs.

2.1.3 Similarity Measurement using Distance Measures

Given a query that contains some desired features, retrieval of components that depict the exact features may sometimes not be possible. Therefore, users are presented with components that come closest (approximate retrieval) to providing these features.

Research have shown that software components and search queries are represented using various representations: text [12], facet [4], graph [99] and formal specification [40]. With such representations, one of the common methods to determine similarity between components and a search query is through the use of distance measure. A distance measure is a function that associates a non-negative numeric value with (a pair of) sequences, with the idea that a short distance means greater similarity. Such an approach expects that the outcome will either be an *exact match* [100] or (failing an exact match) one or more *approximate matches* [100].

In the domain of component retrieval, there have been several approaches [7, 4, 20] of distance measure and this includes the use of linear combinations [101], as in the work undertaken by Girardi and Ibrahim [47], Spanoudakis and Constantopoulos [102], and Sugumaran and Storey [7]. As mentioned in section 2.1.1, Girardi and Ibrahim [47] represent software components in a descriptive manner (i.e frame-based). The distance between a query and a software component is defined by a linear combination of weighted terms, where each term corresponds to a slot of the frame [62]. The term associated to a given slot is the product of two factors: a weight, which reflects the relative importance of the slot in defining the function of the asset; and a similarity index, which reflects to what extent the slot of the query and the slot of a component are similar. The weight is determined by the domain analyst who stores the components in the library, while the similarity index is retrieved from the WordNet [61], the natural language thesauri.

Sugumaran and Storey [7] present a semantic-based solution to component retrieval. The approach employs a domain ontology to provide semantics in refining user queries expressed in natural language and in matching between a user query and components in a repository. In identifying components that are relevant to a given query, a distance measure proposed by Girardi and Ibrahim [62] is employed in the retrieval system.

Spanoudakis and Constantopoulos [102] define a measure of structural distance between queries and assets on the basis of an analysis of their TELOS representations. The distance they introduce is a weighted linear combination of four functions which reflect whether relevant entities in the query and a component are identical and to what extent the query and the component have common attributes via their shared subclasses and their shared super-classes.

Another example of identifying relevant components based on distance measure is the work done by Prieto-Daz [4] and Lucredio et al [20]. In the work undertaken by Prieto-Daz, similarity between a query and software components in the repository is undertaken by measuring closeness of the weighted conceptual graph [4] containing terms described in a facet. Similar to the work by Prieto-Daz [4], Lucredio et. al [20] also represent software component using facets. Nevertheless, they [20] proposed a K-metric function which is based on number of insertions and removals (one substitution counts as one removal followed by one insertion) of keywords that are needed in order to make the keywords sets of the query equal to the keywords sets of a component in the collection.

Vector Model

In addition to existing approaches of using distance measure in component retrieval, we include the discussion on how vector model evaluates the degree of similarity of the program P with regard to the query q using two calculations: Cosine Measure and Euclidean Distance.

The cosine measure proposes to evaluate the degree of similarity of the program P with regard to the query q as the correlation between the vectors $\vec{P}$ and $\vec{q}$. This correlation can be quantified, for instance, by the cosine of the angle between two vectors. That is,

$$sim(P,q) = \frac{\sum_{i=1}^{n}(y_{i,P}) \times (y_{i,q})}{(\sqrt{\sum_{i=1}^{n} y_{i,P}^2}) \times (\sqrt{\sum_{i=1}^{n} y_{i,q}^2})} \tag{2.1}$$

where P is the program, q is the query, y_i is the ith data in the P or q, and n is the number of data in the query. Ranking for cosine measure is done from highest value to the lowest value, i.e. highest cosine measure are placed first. If the angle between the vectors is small they are said to be near each other and a small angle means a high cosine value.

Euclidean distance, or simply *ED*, examines the root of square differences between data of a pair of component and query. In mathematics, the Euclidean distance or Euclidean metric is the distance between the two points that one would measure with a ruler, which can be proven by repeated application of the Pythagorean theorem. By using this formula and symbols defined in equation 2.1, the distance between a program in a repository and a given query can be obtained using the following :

$$distance(P,q) = \sqrt{\sum_{i=1}^{n}(y_{i,P} - y_{i,q})^2} \tag{2.2}$$

For the ED, ranking is done from lowest distance to highest distance, i.e. the program with lowest ED is placed first.

Data Distribution

The degree of similarity of the program P with regard to a given query q can also be identified based on the distribution of data in P and q. In the context of this thesis, data distribution is an information on how data in a software component representation (e.g index) are distributed. An example of data distribution measures is the skewness [103]. In order to determine similarity between two programs, the distance between the data distribution measurement (e.g skewness) is determined. Skewness characterizes the degree of asymmetry of a distribution around its mean [103]. For a set of data containing y_1, y_2, ..., y_n, the formula for skewness is:

$$skewness = \frac{\sum(y_i - \overline{y})^3}{(n-1)s^3} \tag{2.3}$$

where y_i is the ith data in the index, $\overline{y}$ is the mean, s is the standard deviation, and n is the number of data that represents the program. The skewness for a normal distribution is zero, and any symmetric data should have a skewness near zero. Negative values for the skewness indicate data that are skewed left and positive values for the skewness indicate data that are skewed right. By skewed left, we mean that the left tail is heavier than the right tail. Similarly, by skewed right we mean that the right tail is heavier than the left tail.

2.2 Software Classification

Most of the applications stored in the open-source repository systems such as the Sourceforge.net [29] and Freshmeat [30] are classified into various categories (e.g application domain and programming language). If the applications in such sites are correctly classified, retrieval of the required application would be greatly facilitated. In order to reuse program source code, a user may need to manually analyse each of the applications (that may contain more than one program) retrieved by the retrieval system. This is because the applications in these repositories are classified into appropriate domains based on the overall description provided by the developers.

Retrieval of the relevant program source code can be made either by browsing source code that are classified into application domains or by searching through posting a specific search query that includes information on the desired program and application domain. But how are the programs categorized? A developer attempting to organize a collection of programs would most likely categorize the programs based on information in the source code itself (e.g identifier names), some design specifications and the documentation provided with the program. But to understand which application domain the program belongs to, it is very likely the developer would try to gather natural language resources such as comments and *ReadMe* files. Information in natural language are extracted from either external documentation such as manuals and specifications or from internal documentation such as comments and identifier names.

2.2.1 Classifiers

Ugurel et al. [28] classified programs into appropriate application domains and also programming languages using three components, namely, feature extractors, vectorizers and Support Vector Machine (SVM) [104] classifiers. Ugurel et al. [28] demonstrate an SVM based approach to programming language and topic classification of software programs. They trained the classifier with automatically extracted features from the code, comments, and the ReadMe files (i.e. tokens in the code, words, and lexical phrases in the comments and *ReadMe* files). The results imply that large archive collections of mixed software components such as software documentation and program source code can effectively be automatically classified and categorized. Nevertheless, such approach is based on semantic terms extracted from documentation associated with the program. Therefore the approach is only applicable to software that are well-documented. To the knowledge of the researcher, there is no work undertaken in program classification that is based solely on information contained in the program source code. In addition, other than SVM, there is no other work that uses machine learning techniques in classifying program source code into application domain. Examples of these techniques include the C4.5 [105] and K-nearest neighbour (KNN) [106].

The C4.5 [105] deserves special attention due to the fact that it presents the result of research in machine learning that originated from the ID3 system [107]. Therefore, it has always be been the point of comparison for novel approaches in machine learning approaches [108]. C4.5 builds decision trees from a set of training data in the same way as ID3, using the concept of information entropy. The training data is a set $S = s_1, s_2, ...$ of already classified samples. Each sample $s_i = x1, x2, ...$ is a vector where $x1, x2, ...$ represent attributes or features of the sample. The training data is augmented with a vector $C = c1, c2, ...$ where $c1, c2, ...$ represent the class (group) that each sample belongs to. C4.5 uses the fact that each attribute of the data can be used to make a decision that splits the data into smaller subsets. C4.5 examines the normalized Information Gain (difference in entropy) that results from choosing an attribute for splitting the data. The attribute with the highest normalized information gain is the one used to make the decision. The algorithm then recurses on the smaller sublists.

The K-nearest neighbour (KNN) [106] is one of the most popular algorithms for text categorization [109]. Many researchers have found that the KNN algorithm achieves very good performance in their experiments on various data sets [110, 111, 112, 113]. It is an algorithm where the result of new instance query is classified based on majority of K-nearest neighbor category. The purpose of this algorithm is to classify a new object based on attributes and training samples. The classifiers do not use any model to fit and only based on memory. Given a query point, we find K number of objects or (training points) closest to the query point. The classification is using majority vote among the classification of the K objects. K-Nearest neighbor algorithm used neighborhood classification as the prediction value of the new query instance.

On the other hand, the use of statistical analysis in the domain of retrieval have shown promising results [114, 115, 116]. Such approaches include the use of Discriminant function analysis and Linear regression. Discriminant function analysis (DA) is used to determine which attributes in an object, which is under analysis, discriminate between two or more naturally occurring categories. The model is built based on a set of objects (training set) for which the categories are known. Based on the training set, the technique constructs a set of linear functions of the predictors, known as discriminant functions, such that $L = b_1x_1 + b_2x_2 + \quad + b_nx_n + c$, where the b's are discriminant coefficients, the x's are the object attributes and c is a constant. These discriminant functions are used to predict the category of a new object with unknown category. For a k category problem k discriminant functions are constructed. Given a new object, all the k discriminant functions are evaluated and the object is assigned to category i if the ith discriminant function has the highest value.

2.3 Conclusion

Most of the work undertaken in software component retrieval focuses on identifying and employing information from a component to be used as functional descriptors. This is due to the common practice of developers to specify a program's function as the search query. It has been demonstrated by earlier work [4, 10, 7, 45] that the use of information (e.g terms) extracted from the software components (e.g program source code) is beneficial in representing the functionality of the component. Nevertheless, such an approach may not be applicable to software that are not well-documented. Therefore, we need to include additional information, that does not stem from semantic features, as a components' descriptors. Examples of such information is the design patterns and software metrics which can be identified by analyzing structure relationships that exist in a program source code. Even though such information could not illustrate the function of the program, nevertheless similarity between a given query and the programs in a repository can be realized.

Existing structural descriptors for program retrieval (e.g language pattern and programming cliches) are employed to retrieve specific code fragments. This means that a user submits a portion of code that is later mapped to programs in the collection. Programs that contain similar code fragments are presented to the user in the retrieval list. However, since the software developers design their problem solving based on the relationships between objects and methods, there is a need to have a retrieval system that includes such relationships in identifying similarities between a given query and components in a repository. This research differs from those taken in existing studies in that we are interested to identify relevant programs using a combination of functional and structural descriptors. We see the limited use of existing search engines for this particular problem, as code search engines such as the Google code search [23] and Koders search engine [24] provide support only for function-based retrieval. Our intention is to extend the search process supported by such search engines by including structural descriptors to represent programs in a repository. Information on design patterns and software metrics are inferred from structural relationships that exist in a program and later employed as structural descriptors. To represent a program's function, terms extracted from the code and comment statements are employed as functional descriptors.

CHAPTER 3

WEIGHTED TERMS AS FUNCTIONAL DESCRIPTORS

Based on existing studies in software component retrieval [10, 4, 12, 5, 45], one of the common approaches in identifying relevant components (e.g program) from a software repository is using term occurrences -- two components are considered to be similar if they contain a similar set of keywords. In this chapter, we illustrate how relevant terms are extracted from a program and later used as functional descriptors of the program.

3.1 Overview

A developer attempting to understand the function of an application would most likely analyse resources based on the source code itself, some design specifications and the documentation provided with the software. Information written in a natural language can be extracted from either external documentation such as user manuals and design specifications or from internal documentation such as comment statements and file names. This seems reasonable since algorithms depicted in design specifications do not clearly reflect concepts contained in a program but comments and identifiers do [117]. However, applications contained in open-source repositories [29, 30] may not include external documentation which elaborate upon their functionalities, but are always accompanied by program source code. Therefore, the functionality of an application can only be identified based on information that are available in the program.

As discussed in Chapter 2. various approaches have been used to represent the function of a program and this includes the use of term occurrences. A typical example of the term occurrences approach is the `grep` utility used by the UNIX manual system [118]. This utility is used to look for a string pattern in one or more text files, displaying lines that contain the desired pattern. This type of retrieval generates large overheads in the time taken to generate the repository index. If such an approach is employed in a program retrieval system as it is, all of the relevant text (e.g file name and the Java keyword `class` etc.) in each of the programs are included as indices. This generates a very large index, and as the utility is not accompanied by additional mechanism that helps to reduce the search space, searching for a specific term of a particular program context (e.g class name) may generate a list of programs that are irrelevant to the query.

The work described in this chapter is similar to the work undertaken by Maarek et al. [12] and Lindig [3] as we employ uncontrolled vocabulary [12] to represent functionality of a program. Such an approach was undertaken for two reasons, first, the repository contains programs that are written by different developers with various programming background, hence if controlled vocabulary [49] was employed, programs that may be relevant to a given query but do not contain the pre-defined terms will not be retrieved. The second reason was to build a repository index automatically. If a controlled vocabulary was employed, participation of an expert in developing the retrieval system is required to define sets of keywords that best describe or represent concepts relevant to the domain of discourse.

In the context of this thesis, similarity between a given query and programs in a repository is identified based on the existence of the terms defined in a query. For example, if a query contains the term *add*, then programs containing the exact word are considered similar to the query. As a program is written to follow a certain structure, (e.g in Java programming language, a class is defined as consisting of a package, import statements, methods etc.) incorporating information related to a program structure (defined in Definition 5) into the term occurrences approach is believed to contribute to a better program retrieval. By incorporating information on the program structure, the drawback of using uncontrolled vocabulary (i.e unlimited number of terms that generates a broad search space) is overcome.

Definition 5 *A program structure provides the components that make up a particular language (e.g Java) program. This includes package and import statements, a class header and its body, and a method header and its body.*

One of the popular search engines (i.e Google) has introduced a specific search engine, known as Google code search [23], for users to find examples of programs from the web. However, the Google code search is primarily keyword-based, and there is a lack of domain-knowledge associated with its queries. For example, if a user intends to find a method that is able to calculate the sum of two numbers, he may define a query that consist of the phrase *method add* or *method sum*. Upon receiving the user's query, the retrieval system presents the developer with programs that contain both of the terms (*method* and *add*), followed by programs containing one of the terms. Even though the system may present the developer with programs containing both of the terms defined in the search query, these programs may not illustrate the required domain. For example there is no code statement such as *method add* in a

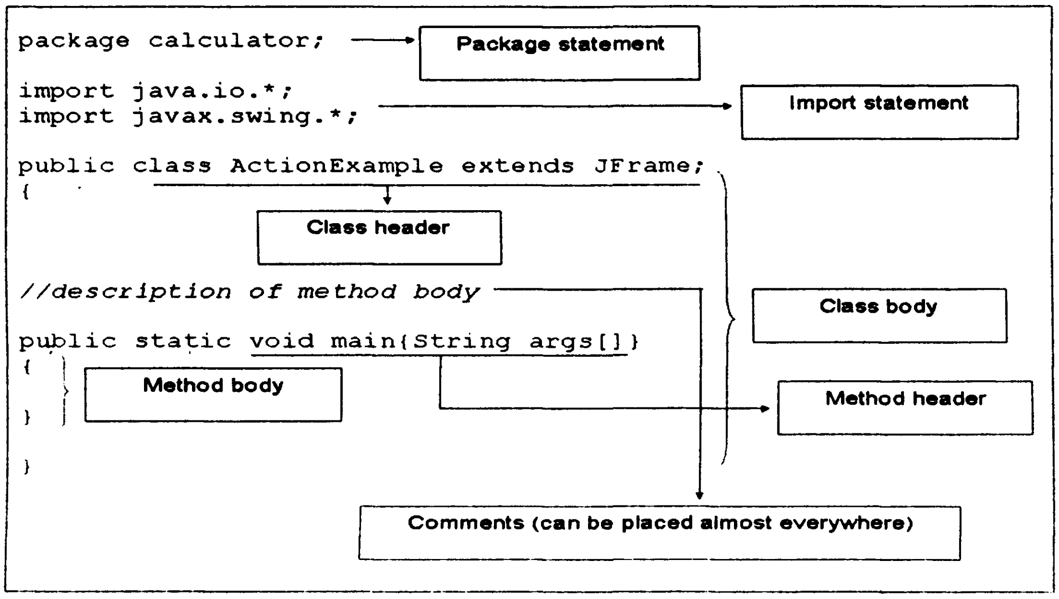
Java program. The method `add` is only reflected by the word *add* and not *method add* as a whole word. With respect to this, terms occurring as a class name, method name, package name or in comments should be treated differently.

The search process utilized in `SourgeForge.net` [29] and Freshmeat [30] also makes use of keywords, and is based on the general descriptions given to each of the applications stored in the repositories. Users of the repositories are presented with applications that contain the terms defined in a search query. Nevertheless, neither of these repositories [29, 30] performed program retrieval based on the different contexts of a program, as elaborated in the example mentioned earlier (i.e *method add*). Our intention is to extend the search process supported by such public domain software repositories [29, 30] and existing code search engines [23, 24]; therefore we propose terms extracted from program structure to be used as functional descriptors of a program.

Similar to the `grep` utility, our retrieval system also works based on string matching; nevertheless, we accompany the extracted terms with relevant weights. The weighting scheme is employed to illustrate the importance of a term in representing the function of a program. Details of the weighting scheme can seen in section 3.3 on page 61. With the assumption that software developers are aware of which program structure the search term refers to, the program retrieval system is able to present the developers with relevant programs.

3.2 Terms as Functional Descriptors

A Java program consists of several components: class header, class body, method header, method body, comments, packages, and import statements. An example of a simple Java class is illustrated in Figure 3.1. Each of these components (illustrated in Figure 3.1) plays a significant role in determining the functionality of a Java program. In the context of this thesis, names extracted from these components (e.g *ActionExample*) are referred as identifier names.



```
package calculator;

import java.io.*;
import javax.swing.*;

public class ActionExample extends JFrame;
{

//description of method body

public static void main(String args[])
{

}

}
```

Figure 3.1 Components of a Java program

The use of a program in part depends on the documenting ability of the names used for its identifiers [119]. Identifiers are the names of any packages, classes, methods and variables defined in a program. Identifier names are one of the important sources of information about program components, as they give an initial idea of the role of each identifier in a program. From the work undertaken by Lindvall and Sandahl [120] and Marcus and Maletic [121], we learned that meaningful identifiers are considered a significant aid to understanding a program. For example, if a developer is analyzing a database program, then a method named *add* contained in the program may indicate the process of adding a new record into a database. Therefore, many of the developers are naming the identifiers according to their function [122].

In designing object oriented applications, developers identify the fundamental objects of the problem domain for a given programming task. These objects are then used as the class name, hence illustrating the functionality of the class; code statements written in the class body are related to the object. Once developers identified the main objects, they next define the internal nature of each object: its attributes (variables) and behaviours (methods). Attributes model the variation that is allowed among different objects and an object maintains a value for each of the defined attributes. All of these pieces of information are represented in identifiers defined in the program(s) of the application, hence reflecting their function.

File names and package names are very useful terms for indexing, as they provide meaningful information about the file or files they represent. In a Java program, the file name provides two types of information. The first is the name of the main class in the file along with the name of its constructors (as these are all the same). This provides a mechanism for determining the difference between a constructor and a method when parsing, as a constructor is a method with the same name as the filename (provided there is only one class per file). The second information the file name provides is some indication of the content of the file, or in a Java program, information about the content of the class or an indication of what the class does. For example, one of the files in a repository is known as `Database.java`. From this, we can determine the name of the class and constructors and deduce that the class possibly involves a connection to a database, or contains operations for data stored in a database. Besides using file names to indicate functionality, package names also serve a similar purpose.

If an application consists of more than one object, similar objects can be grouped together. It is good programming practice to group programs into packages of related classes, with each package in a separate directory [123]. This can be achieved with the use of package statements. Similar to naming Java classes based on their function, the package name also illustrate its function. Packages represent the way we organize programs into different directories according to their functionality and usability, as well as the category they should belong to. An example of packaging is the JDK package from SUN [124]. The idea is that programs in one directory (or package) would have a different functionality from those of another directory. For example, programs in the `java.io` package do something related to input/output, but programs in the `java.net` package give us a way to deal with the Network. In GUI applications, it is quite common for us to see a directory with the name *ui* (user interface), meaning that this directory keeps programs related to the presentation part of the application. On the other hand, if we see a directory called *engine*, this stores programs related to the core functionality of the application instead.

One of the ways a developer can use classes defined in a package is by using the `import` keyword. For example, the statement `import repository.data.Database` allows developers to use the `Database` class, which is defined in the `repository/data` subdirectory. The import statement can be used to infer the functionality of a program as it tells a developer which classes the program is relying on (apart from the standard Java library classes) in order for it to function.

Just as file names and package names contain information about their content, the names of methods and variables often provide information about their content or use. Class methods take on the property of being public, private, or protected. All of these names can be useful as they provide information about the functionality of a class. For example, the file `Database.java` obviously can hold or contain information about a database. We may also be interested in finding out what kind of behaviour an instance of database object can perform. For example, if one of the database methods is `connect`, this could indicate that it establishes a connection to the database. However, even more information may be determined from the fact that the method is private, indicating that this method is only used within the database class and therefore has a very specific purpose within this class.

Variables can also come under different categories. The most common and possibly the most useful are class variables. These are variables contained within a class that are separate from methods, and can therefore be used by any class method, constructor, etc. These often have very descriptive names in order to define effectively the information they hold. For this reason, they are an effective index term. However, besides the name, they also contain other information. Class variables can be private, public, or protected. This can indicate whether the variable is specific to this class or if it is more general and therefore could be used by other classes. There are also local method variables, which can only be used within the method in which they are declared.

The final component to be included in the program structure is program comments. As the work of Nurvitadhi et al. [117] reported a significant difference in program understanding between programs with and without comments, we include program comments as one of the program structure components to infer the functionality of a Java program. Comments are a very useful index term within a program as they quite often adequately explain the functionality of the classes and methods. This causes them to contain words that often would not be in the code itself. For example, a developer may need to ensure that there is only a single creation of a class instance. Therefore, s/he might include a comment statement such as *created only once* in her/his program. There are three types of comments available within a Java program: javadoc comments, method comments and inline comments. Javadoc comments, separated with /** and **/ are structured comments that describe the functionality of a class or method in detail. This causes them to have a very specific structure (and therefore are easier to extract/parse). The second type of comment are method comments. These are more general comments, separated with /* and */. They can span multiple lines or single lines, and they are often used to explain in more detail methods, constructors, variables, or large sections of code. They are more difficult to parse as they are less structured then javadoc comments and in order to index them some assumptions need to be made about the way they are laid out (for example, there is always a space between the /* and the first letter of the comment). The last type of comment is the inline comment. The inline comment is denoted by // and it has no ending symbol; instead, the comment simply ends when the line does. This type of comment is a lot more specific than the first two types of comments and usually describes some small section of code rather than a whole method or constructor. After an analysis of Java programs relating to mathematical operations (programs contained in the `Jama`, `JMP`, `Meditor`, `JNumeric` and `nMath` projects, which were obtained from `Sourceforge.net` [29]) it was found that even though these comments can be very specific to a section, they may contain terms that would be useful to index, so it was decided to index them in the implementation. Common words such as *a*, *the*, *an*, etc. are removed from the comments, which greatly reduces the amount of terms indexed for comments. In addition, object oriented program commenting includes both a class-based comment that provides an overview of a class and a method-based comment that gives information about the content of a method. More specifically, a class-based comment is helpful in developing a high-level knowledge of a program, such as the purpose of the class, what the class does, or the interconnection between classes. On the other hand, a method-based comment provides a more low-level understanding of the program, such as the purpose of the method and implementation technique used.

3.2.1 Extracting Relevant Terms based on Program Structure

In order to extract identifier names from program structure in the programs stored in a software repository, we use the Java parser generator, Java Compiler Compiler [tm] (`JavaCC` [tm]) [81] to create a JavaParser. `JavaCC` generates the following files:

- `JavaCharStream.java` represent the stream of input characters.
- `Token.java` represents a single input token
- `TokenMgrError.java` an error thrown from the token manager.
- `ParseException.java` an exception indicating that the input did not conform to the parsers grammar.
- `JavaParser.java` the parser class.
- `JavaParserTokenManager.java` the token manager class.
- `JavaParserConstants.java` an interface associating token classes with symbolic names.

Instances of objects from all of the files generated by `JavaCC` are created in a program named `ParseFile.java`. This Java program examines a given program, which is under analysis, by using the created instances to parse the program and identifies eleven Java components: Javadoc Comment, Method Comment, Inline Comment, Import statement, Package declaration, Class name, Superclass, Interface class, Method names, Variable names and Filename. To achieve this, firstly an instance of object `JavaCharStream` is created before using it to create the instance of type `JavaParserTokenManager`. Then, an instance of object `Token`, (`t`), is created to hold the value of constants found in the character stream under analysis. For example if the token, `t`, is a kind of `IMPORT` constant, `t.kind == JavaParserConstants.IMPORT`, then the token following `t` is assumed to be class or package name. The parsing includes other token types such as `CLASS`, `PACKAGE`, `ABSTRACT`, `IMPLEMENTS`, `IDENTIFIER`, `SINGLE_LINE_COMMENT`, `FORMAL_COMMENT`, `MULTI_LINE_COMMENT`, `PUBLIC`, `PRIVATE`, `PROTECTED`, `SEMICOLON`, `COMMA`, `DOT`, `LPAREN` and `RPAREN`. This process is undertaken until a constant of type *End of File* (`t.kind == JavaParserConstants.EOF`) is identified. Below are examples of assumptions made when the token is of a particular type:

- `PACKAGE` - token following `t` is assumed to be package name.
- `CLASS` - token following `t` is assumed to be class name.
- `EXTENDS` - token following `t` is assumed to be superclass name.
- `IMPLEMENTS` - token following `t` is assumed to be interface class name.
- `ABSTRACT` - the second token following `t` is assumed to be class name.

The collected information (i.e identifier name) is later used as functional descriptors of the program which is under investigation. Prior to writing the extracted term into an index file, we need to ensure that white space from both ends of the string (term) have been removed. This has to be done to ensure string matching can be performed effectively. It is also necessary to identify if the terms to be used for functional descriptors are not of type Java keywords (e.g throw, int, char, float, abstract, class), and are not of type stopwords. Stopwords are words that may be entered into a search query but cannot be searched for as individual words. For example, if a developer is searching for the string *connect to database*, the word *to* is a stopword. We created a *stopwords.txt* file to include stopwords used by the Google search engine [125] and the Onix Text Retrieval Toolkit [126]. After ensuring that a term is not a Java stop-word nor from the list in stopword.txt, only then the term is included in an *index.txt* file. This file contains terms extracted from the programs in the repository and the relevant information on terms (e.g file name of which the term is extracted from and weight of the term). Related elaboration on the weighting schema is presented in the next section.

3.3 Weighted Functional Descriptors

We make the following assumptions prior to developers submitting their query for programs:

- Developers have some indication of the types of source code in which they are interested. This could be in terms of the keywords they assume to be present within such source code, or the likely method names that such source code could contain. Although not likely to be valid in a general case, we have found this assumption to hold true based on the existing source code archives such as `Sourceforge.net` [29]. Perhaps one reason for this is that developers who offer their source code for use by others often also attempt to describe their data structures or method names with comments that could be relevant for others.

- Developers are familiar with the likely structure of the source code they are trying to find. This may be particularly true for numerical approaches (where nested loops are often used over arrays or similar data structures). Often many programming languages are targeted towards the scientific computing community which provides specialist support for such data structures (examples include OpenMP and High Performance Fortran).

As there are many terms that can be extracted from a program, we prioritize these terms using the concept of *weighted term frequency*, which assigns high weights to terms extracted from a certain component of a program (e.g identifier acting as a class name) and low weights to terms obtained from other components of a program (e.g comment). This is necessary for two reasons. The first is that search terms can appear in a variety of areas in source code (i.e program structure), and depending on where these terms occur, they have different meanings. For example, the same term used for a class and a variable may have completely different meanings in each context. Therefore, in order to provide developers with relevant programs, the context in which it is found needs to be determined and stored. The second is to allow a more advanced form of ranking. For example, a program containing the search term as a class name will be ranked on top of a program containing the term in its comment statements. For this to work, it is necessary to assign a type to each term of the extracted terms – types are derived from program components. Eleven types were determined by myself in the end and this is based on the components of a program that appeared the most in programs that we have stored in our repository. Nevertheless, determining these types was actually a very difficult task as there are so many exceptions to the way a program may be laid out. In order to simplify the process, it was assumed that a program used standard conventions for layout and content, for example, the way a javadoc comment is written. Below is a list of type of terms with their weights (i.e provided in bracket()):

- Javadoc Comment (3): Javadoc Comments are specific to the Java language and provide a means for a programmer to fully document his / her source code as well as providing a means to generate an Application Programmer Interface (API) for the code using the javadoc tool that is bundled with the JDK.

- Method Comment (1): If a comment is going to span across more than one line then a multi-line comment should be used. These are often useful for providing more in-depth information.
- Import statement (1): Import statements point to classes or packages that should be made available for use within the current class. For example, in order to use the `java.applet.Applet` class in a Java file, the class would have to be imported via the `import java.applet.Applet;` or `import java.applet.*;` statement.
- Package declaration (1): If included, the package declaration must be the first statement in the file. The package keyword is followed by a package name. The package name is a series of elements separated by periods. Each period separated element must correspond to a filesystem subdirectory under which the class file is located. For example, if a class was declared to be in the `com.database.gui` package, it would be located in the `com/database/gui/` subdirectory. Only one package declaration is allowed per .java file.
- Class (3): The class name.
- Extends (2): Class in which the existing class inherits (superclass).
- Implements (2): Indicates that a class contains methods for each of the operations specified by the interface.
- Method (2): A Java method is a set of Java statements which can be included inside a Java class. Java methods are similar to functions or procedures in other programming languages.
- Variable (2): Variables are data identifiers. Variables are used to refer to specific values that are generated in a program – values that we want to keep around.

Program data is often easier to understand and manipulate if each data has its own name.

- Filename (3): If a `public` class is present, the class name must match the filename. For example, if a source file contains a definition for a public class `Database`, then the source file must be named Database.java. A source file may contain any number of non-public class definitions.

Based on the above weighting schema, a term `t` found in a filename is more important than the same term found in a variable. This is due to the assumption that the functionality and content of a file is reflected more by the name assigned to the file than the variable.

The terms extracted from a program are known as weighted functional descriptors and are used to represent the functionality of the program. Based on existing work [4, 20, 46, 47, 48, 5], program retrieval performed using the weighted functional descriptors can be considered similar to the descriptive methods (discussed in section 2.1.1 on page 22). This is similar to the practice of extracting relevant terms to be employed in facets as undertaken by Prieto-Diaz [4]. The difference is that, in this work, terms to be extracted are identified based on components of a program (e.g import statements, class header, comments). Prieto-Diaz [4] employed a term as a facet attribute while ignoring which context of a program the term is extracted from.

3.4 Similarity Measurement

To perform a similarity measurement between weighted functional descriptors extracted from a query and programs in a repository, we adopted the Levenshtein distance measure [127]. Levenshtein distance (LD), which was developed in 1965 by Vladimir Iosifovich Levenshtein, is a measure of the similarity between two strings, which we will refer to as the source string **s** and the target string **t**. The greater the Levenshtein distance, the more different the strings are. The distance is the number of deletions, insertions, or substitutions required to transform **s** into **t**. For example,

- If **s** is *test* and **t** is *test*, then $LD(s,t) = 0$, because no transformations are needed. The strings are already identical.
- If **s** is *test* and **t** is *tent*, then $LD(s,t) = 1$, because one substitution (change *s* to *n*) is sufficient to transform **s** into **t**.

The Levenshtein algorithm has been used in order to have a ***flexible*** retrieval system. Comparing this to the similarity measurement employed in existing search engine such as Google code search [23] and in an open-source repository such as SourceForge [29] and Freshmeat [30], which are undertaken based on exact string matching, we expand such an approach by allowing a difference of a pre-defined numbers of letters between the analyzed strings. This is achieved by allowing the users to determine a threshold value which acts as a cutting point in identifying similar string defined in the search query and in a program. For example, if a user defines string **t** in a query and the value 2 as the threshold value, then, only terms contained in source string **s** that require the maximum of two substitutions in order to be transformed into the target string, **t**, are considered to be similar to **t**. With this, the program retrieval system would present users with not only the *exact match* but also with an *approximate match*. The former result is obtained when there is an exact string matching between a term in a query and functional descriptors of a program. On the other hand, an *approximate match* presents users with programs that contain terms that may be similar to the terms defined in the query. By allowing substitution, deletion and/or addition of a number of letters in a term, the presented program retrieval system is able to consider misspelled terms to be similar to the terms defined in a search query.

Our retrieval system works by allowing a user to use a program as the search query. The program may be an existing program that the user is working on based on a given programming task or any other programs that the user feels able to represent his code requirements. In order to identify programs that are relevant to the query program, for each of the programs in the repository, we summed up the weights of the weighted functional descriptors that were similar to the descriptors contained in the query program. Programs from the repository with greater totals of values were ranked at the top of the retrieval list. Later in this thesis, a retrieval list is termed a hit list and is defined as follows:

Definition 6 *A hit list contains a lineup of programs that have been identified as similar to a given query. A program listed on the top of the list is considered to be most similar to a given query.*

To illustrate how similarity between programs is identified, we mapped a query program $P(Q)$ against five programs $(P(1), P(2), P(3), P(4), P(5))$. In $P(Q)$, there are three terms identified as weighted functional descriptors: `database`, `connect` and `display`. In this example, the similarity measure between the weighted functional descriptors in $P(Q)$ and $P(i)$, is undertaken based on *exact match* [100] only. The first search term consists of a filename and class name; the second and third terms are of the method component. Hence, we obtained the value of 10 on summing the weights of the descriptors in $P(Q)$ – both of the terms `database` get the value 3 as they are from the filename and class name and both the term `connect` and `display` are assigned the value 2 as they were identified to be method names. In the below examples, $P(i)$ represents the program under analysis while $W(i)$ represents the summed-up weights of descriptors in a particular program, i.

P(Q) $= \{\textit{database, connect, display}\} \rightarrow W(Q) = \{(3+3) + 2 + 2\} = 10$

P(1) $= \{\textit{database, connect, display}\} \rightarrow W(P1) = \{(3+3) + 2 + 1\} = 9$

P(2) $= \{\textit{display}\} \rightarrow W(P2) = \{(2+1)\} = 3$

P(3) $= \{\textit{connect, display}\} \rightarrow W(P3) = \{2 + 2\} = 4$

P(4) $= \{\} \rightarrow W(P4) = 0$

P(5) $= \{\textit{database}\} \rightarrow W(P5) = \{(3+3)\} = 6$

As the retrieval system ranks programs based on summation values of similar terms, $P(1)$ with $W(P1) = 9$ is presented at the top of the list. This is followed by $P(5)$, $P(3)$, $P(2)$ and $P(4)$.

Below, we illustrate another example, which is undertaken based on *approximate match* [100]. This is achieved by defining the Levenshtein distance as being less or equal to the value three, $LD(s,t) <= 3$. To illustrate how a similarity measurement between programs is undertaken, we mapped the same query from the previous example, $P(Q)$, against five programs $(P(11), P(22), P(33), P(44), P(55))$ from the repository. There are three terms used as weighted descriptors for program $P(Q)$: **`database`**, **`connect`**, and **`display`**. Only terms contained in $P(i)$ that require at the most three deletions, insertions, or substitutions in order to transform an existing string `s` into the weighted functional descriptors for $P(Q)$, `t`, are included as weighted functional descriptors for the program. If $P(i)$ contains several terms that are similar to a term `t`, then the term with the similar weight or with the highest value of weight is identified as an *approximate match* to `t`. For example, the search term **`connect`** can be mapped against **`connected`** and **`connects`**. If **`connected`** is a variable name and **`connects`** is found in Java doc comment, the latter string will be identified as a relevant match to the search term **`connect`**. This is because the string **`connects`** has the weight of 3 while `connected` is identified as 1. In the below program tuples, only the strings identified as being an *exact match* or *approximate match* are included for weighting calculation. For example, a total of ten identifier names (terms) were extracted from program structure $P(11)$. However, only three terms were included for weighting calculation as the remainder of the seven terms require more than three substitutions, insertions or deletions in order to transform them into the required search terms.

P(Q) = {database, connect, display} $\rightarrow W(Q) = \{(3+3) + 2 + 2 = 10\}$

P(11) = {database, connects, displays} $\rightarrow W(P11) = \{(3+3) + 3 + 1 = 10\}$

P(22) = {connected, displayed} $\rightarrow W(P22) = \{1 + 3 = 4\}$

P(33) = {connects, displays} $\rightarrow W(P33) = \{3 + 1 = 4\}$

P(44) = {databases, displayed} $\rightarrow W(P44) = \{ 3 + 3 = 6\}$

P(55) = {databases} $\rightarrow W(P55) = \{3 \}$

Based on the above itemized program tuples, the retrieval mechanism presents the programs in the following descending order: $P(11)$, $P(44)$, $P(22)$, $P(33)$ and $P(55)$. Program $P(11)$ is presented as the most similar program when compared to $P(Q)$. This is followed by $P(44)$ and $P(55)$ which depict only a single term similar to $P(Q)$ and so are ranked on the bottom of the retrieval hit list of five programs.

3.5 Conclusion

In an open-source repository such as `Sourceforge.net` [29], retrieval is performed based on keyword search performed on the description provided for the application and/or the application's name. As an application may contain more than one program, users, whose intention is to reuse code statements, will then need to manually examine the programs in the application to determine whether the programs contain the required code. To help these users, we propose a retrieval system which employs a similar approach. Nevertheless, our retrieval system is undertaken towards a repository of programs and presents users with programs that contain keywords of the required context. The presented work employs a weighting scheme that differentiates the functional descriptors (i.e identifier names) based on the context of the program. With the assumption that the user of the repository is able to refine his search query, for example, to determine which program context (e.g class, method, package) the search term refers to, the retrieval system is able to facilitate users with specific code requirements.

In addition, our program retrieval system provides flexibility in generating queries; allowing the use of program as a query. This expands the capability of expressing search requirements as developers use the existing program developed for a given programming task as the query program. Such an approach delivers context-sensitive information related to both the given programming task and the background knowledge of the user. Relevant terms are extracted from the query program and are later mapped against the weighted functional descriptors of each programs in the repository.

CHAPTER 4

DESIGN PATTERNS AS STRUCTURAL DESCRIPTORS

In this chapter, we demonstrate the identification of design patterns in programs that are obtained from open-source repositories. Information regarding the existence of design patterns in a given program is later used as structural descriptors of the program in a retrieval system.

4.1 Overview

Software developers find design patterns important for a number of reasons. First, they give novice developers access to the best practices of more experienced developers. Second, they allow developers to think of their designs at higher levels of abstraction; for example, instead of focusing on low-level details, such as how to use inheritance, developers can approach complex systems as a collection of design patterns that already make the best use of inheritance. That shift of focus to a higher level of abstraction also provides a common vocabulary when software developers discuss design.

Currently, users who intend to retrieve programs from a source code retrieval system (e.g. Koders search engine [24] and Google code search [23]), would normally create their search query using terms and a combination of Boolean commands such as **AND** or **OR**. However, how do software developers (with the knowledge of design patterns) search for programs that illustrate a particular function through the use of a certain design pattern? For example, we may have a software developer who requires examples of programs that create text rendering objects, which render data obtained from the command line or a program. Furthermore, s/he would like to create the objects by recursively composing similar objects. A fundamental question that arises is how to represent such requirements in a keyword-based search engine such as in the Google code search or Koders. One possibility is by defining a query of *text renderer* AND *command line* AND *program* AND *recursive.* Upon receiving such a query, a keyword-based retrieval system may present the user with programs that contain all of the required terms (*text renderer, command line, program* AND *recursive*). This is followed by programs containing a combination of the required terms and programs that contain either one of the terms. Even though the retrieval system may be able to present users with programs containing terms defined in the search query, the presented programs may not illustrate the required patterns (building objects by recursively composing similar objects) as anticipated by the user.

If programs in a software repository contain explicit information on design patterns in the code or comment statements (e.g *Singleton pattern* in the method comment), then retrieval systems based on term occurrences may be sufficient in presenting users with relevant programs. Otherwise, we need an additional mechanism that does not rely on semantic meanings to identify design patterns embedded in a program. Below, we provide examples of programming tasks that would benefit from having design patterns as program descriptors.

Example 1: *Implement a Logger object that will log events for all subsystems according to the date and time. We cannot have more than one instance of Logger in the application; otherwise, every time we log to a subsystem, a logger will be created, hence generating duplication of information. The logger is accessed by different objects throughout a software system, and therefore requires a global point of access.*

Example 2: *Implement a GUI system that has window objects which can contain various GUI components such as buttons and text areas. The window also contains container objects which can hold other components.*

Example 3: *Implement a weather reporting application, WeatherReporter class, that displays the latest weather data on-screen. The weather is displayed by two classes: GraphicReport (cloud, sun, rain icons) and TextReport (Temperature: 25C, Sunny) . When the weather changes, WeatherReporter sends updates to TextReport object and GraphicReport object.*

4.2 Identification of Design Patterns

Our approach in identifying design patterns embedded in a program is similar to the work undertaken by Keller et al. [72] -- design patterns are detected based on structural relationships that exist in a program. Nevertheless, our approach does not require the code to be represented into an intermediate form prior to the design pattern detection, as undertaken by Keller et al. Our approach is based solely on parsing the code and therefore does not require an additional mechanism (e.g language) prior to detecting the pattern.

There are different ways of categorizing design patterns, depending on what they do and when they should be applied. Gamma et al. [1] classify design patterns into three categories: creational, structural and behavioural. The creational patterns are concerned with the process of object creation, while structural patterns deal with the composition of objects. The behavioural patterns characterize the ways in which objects interact and distribute responsibility. In this work, we include the identification of Singleton, Composite and Observer design patterns representing the three categories respectively. In order to identify these design patterns, we use the same Java parser created for extracting weighted functional descriptors (elaborated in section 3.2.1 on page 59). Relevant information that includes class inheritance, interface and abstract classes, method invocation and method signature that include the arguments and return type, variable declarations and the modification of its value are identified by the parser. This information is later used to determine the existence of a particular design pattern in a program.

4.2.1 Singleton

Creational -- Singleton Pattern: the intention is to ensure a class has only one instance and provides a global point of access to it.

Sometimes it is appropriate to have exactly one instance of a class: window managers, print spoolers, and program systems are prototypical examples. Typically, those types of objects – known as singletons – are accessed by disparate objects throughout a software system, and therefore require a global point of access. The UML diagram of a Singleton pattern is provided in Figure 4.1.

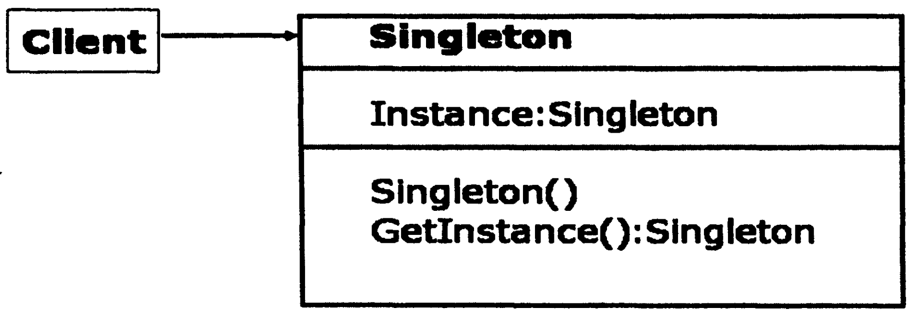


Figure 4.1 UML Diagram of Singleton Pattern

The participants of the Singleton class (as depicted in Figure 4.1 are described as follows:

- Client
  - Clients access any instance of a Singleton only through the Instance method.
- Singleton
  - Defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - Responsible for creating and maintaining its own unique instance.

The working context of Singleton design pattern is illustrated as below:

Suppose we need to write a class that an applet can use to ensure that no more than one audio clip is played at a time. If an applet contains two pieces of code that independently play audio clips, then it is possible for both to be played at the same time. When two audio clips play at the same time, the results depend on the platform. The results may range from confusing, with users hearing both audio clips together, to terrible, with the platform's sound producing mechanism unable to cope with playing two different audio clips at once. To avoid the undesirable situation of two audio clips playing together at the same time, the class developer's class code should stop the previous audio clip before starting the new audio clip. A way to design a class to implement this policy is to ensure that there is only one instance of the class shared by all objects that use that class.

When implemented as the pattern recommends, a class will have direct control over how many instances can be created. Developers ensure that the instance is easily accessible (by many objects) by defining the access modifier for the method accessing the class instance as type public. Algorithm 1 describes how a Singleton pattern is detected.

Based on the pseudo code described in Algorithm 1, we perform Singleton detection by identifying the access modifier for the constructor of a class. To ensure the creation of only a single instance of a class, the constructor should be declared as either private or protected. We then identify the existence of any method with a public access modifier that returns a private member (variable) of the class. Upon identifying these requirements, the particular class is classified as implementing Singleton design pattern.

Algorithm 1 Detection of Singleton Pattern

```
1: Initialize an empty set, $Y = \{\}$
2: For each Java package $Jp$ in the repository
3:  For each Java program $Jf$ in $Jp$
4:   For each class $C$ in $Jf$
5:    Identify constructor $k$ where $(k.type == private) \vee (k.type == protected)$
6:    Identify existence of class variable $v$ in $C$ where $(v.type == private)$
7:    Identify existence of method $m$ where $(m.type == public)$ and has $v$ as the return argument
8: if $k$, $v$ and $m$ exist then
9:    $Y = Y \cup \{Jf\}$
10: end if
```

4.2.2 Composite

Structural Patterns -- Composite Pattern: allows users to treat individual objects, and a composition of objects, uniformly, thereby leading to a recursive composition. The aggregation relationship is typically implemented as a reference from the composite child class to a parent class and has a cardinality of 1-to-N. This is shown in the UML diagram illustrated in Figure 4.2. The key to the Composite pattern is an abstract class that represents both primitives and their containers.

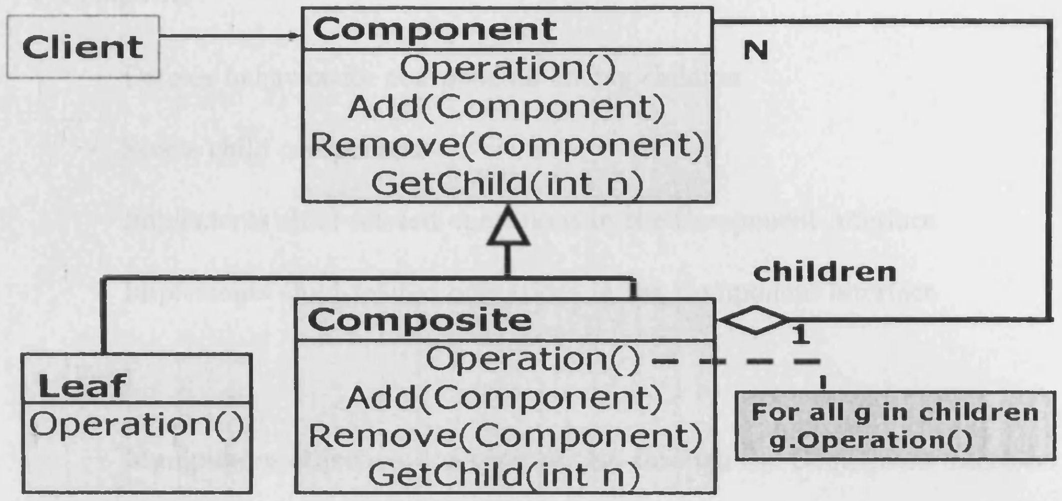


Figure 4.2 UML Diagram of Composite Pattern

Based on Figure 4.2, we identify the following:

- Component
  - Declares the interface for objects in the composition.
  - Implements default behavior for the interface common to all classes, as appropriate.
  - Declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a components parent in the recursive structure, and implements it if that is appropriate
- Leaf
  - Represents leaf objects in the composition and a leaf has no children.

  - Defines behavior for primitive objects in the composition.

- Composite
  - Defines behavior for components having children
  - Stores child components
  - Implements child-related operations in the Component interface
  - Implements child-related operations in the Component interface

- Client
  - Manipulates objects in the composition through the Component interface.

A working context of Composite design pattern is illustrated below:

Suppose that we are writing a document formatting program. It formats characters into lines of text organized into columns that are organized into pages. However, a document may contain other elements. Columns and pages can contain frames that can contain columns. Columns and frames and lines of text can contain images.

In our work, the rules used to detect the existence of Composite design pattern are described in Algorithm 2 and Algorithm 3. We first initialize an empty set to store the final result: pairs of Java program and its interface classes. For every Java package, we identify its Java programs where in each of the programs, we may have at least one Java class. Referring to step 5 in Algorithm 2, the usage of an interface class is to be identified. Within the Java programming language, an interface keyword is used by unrelated objects to interact with each other. Interface class is used to define a protocol of behaviour that can be implemented by any class, anywhere in the class hierarchy. It is useful for capturing similarity among unrelated classes without

Algorithm 2 Detection of Ordinary Composite Pattern

```
1: Initialize an empty set to store tuple of two elements, $Y = \{a, b\}$ where $a$ is a
   Java program and $b$ is an interface class
2: For each package $Jp$ in the repository
3:  For each program $Jf$ in $Jp$
4:   For each class $C$ in $Jf$
5: if $C$ implements an interface class $IC$ then
6:    Identify method $m$ in $C$ that receives argument of type $IC$
7:    if $m$ exists then
8:       $Y = Y \cup \{Jf, IC\}$
9:    end if
10: end if
```

artificially forcing a class relationship, declaring methods that one or more classes are expected to implement, or revealing an object's programming interface without revealing its class. Finally, upon identifying a Java class that implements an interface, we then identify method(s) in which it uses interface as one of its method parameters.

In Algorithm 3, we describe the structural information required in identifying a recursive Composite design pattern. The existence of such a pattern is identified by locating an abstract class that implements an interface class. As depicted in step 6 in Algorithm 3, if there exist a class that inherits the abstract class located earlier, and the class contains at least one method that implements methods defined in the interface class, then the class is considered to be implementing a recursive Composite design pattern.

Algorithm 3 Detection of Recursive Composite Pattern

```
1: Initialize an empty set to store a tuple of three elements, $Y = \{a, b, c\}$ where $a$ is
   a Java program, $b$ is an interface class and $c$ is an abstract class
2: for each package $Jp$ in the repository do
3:   for each program $Jf$ in $Jp$ do
4:     for each class $C$ in $Jf$ do
5:       if $C$ is an abstract class AND $C$ implements an interface class $IC$ then
6:         if subclass of $C$ (i.e $Csb$) exists, and $Csb$ contains methods that imple-
           ments methods in $IC$ then
7:           $Y = Y \cup \{Csb, IF, C\}$
8:         end if
9:       end if
10:    end for
11:  end for
12: end for
```

4.2.3 Observer

Behavioral Patterns -- Observer Pattern: defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. The key objects in the Observer pattern are Subject and Observer. A subject may have any number of dependent observers and all observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subjects state. The Observer pattern can be used in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets developers vary and reuse them independently.
- When a change to one object requires changing others, and developers do not know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

Figure 4.3 illustrates the UML diagram of the Observer pattern. The participants of Observer pattern depict the following criteria:

- Subject
  - Knows its observers and any number of Observer objects may observe a subject.
  - Provides an interface for attaching and detaching Observer Objects.

- Observer
  - Defines an updating interface for objects that should be notified of changes in a subject.
- Concrete Subject
  - Stores a state of interest to Concrete Observer objects.
  - Sends a notification to its observers when its state changes.
- Concrete Observer
  - Maintains a reference to a Concrete Subject object.
  - Stores state that should stay consistent with the subject state.
  - Implements the Observer updating interface to keep its state consistent with the subject state.

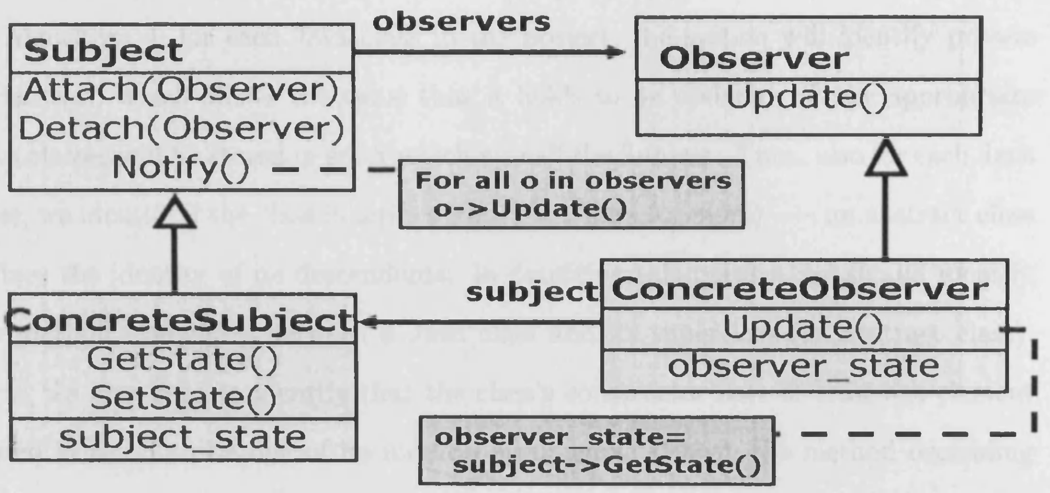


Figure 4.3 UML Diagram of Observer Pattern

A working context of Observer design pattern is illustrated as below:

Suppose we are developing a software application for a company that manufactures smoke detectors, motion sensors and other related security devices. A new line of security devices is introduced that is able to send a signal to a security card that can be installed in most computers. The hope is that companies that make security monitoring systems will integrate these devices and cards with their systems. We are required to write an API that allows future customers to integrate their programs with it easily so their programs will receive notifications from the security card. It must work without forcing the customers to alter the architecture of their existing software.

In detecting the Observer pattern, we have to identify the observers, the object to be observed, and the method(s) used to update any changes that need to be made. In Algorithm 4, for each Java class in the project, the system will identify private variable(s), which allows the value that it holds to be updated. These appropriate Java classes will be stored in set S which we call the Subject. Then, also for each Java class, we identify if the class inherits an abstract class (or more) -- an abstract class defines the identity of its descendants. In detecting this pattern, we should identify the method overriding between a Java class and its superclass (i.e abstract class). Next, we also need to identify that the class's constructor uses at least one element stored in set S to be one of its method arguments. If both the method overriding and the constructor identification succeed, the system stores the related documents in set Y -- Java program, abstract class and subject program.

Algorithm 4 Detection of Observer Pattern

```
1: Initialize three empty sets, $Y = \{\}$, $S = \{\}$ and $IAB = \{\}$
2: For each package $Jp$ in the repository
3:  For each program $Jf$ in $Jp$
4:   For each class $C$ in $Jf$
5:     Identify variable $v$ where ($v.type$ = $private$) and there exists a method that
   updates value of $v$, then store $C$ in set $S$
6:    Identify existence of interface or abstract class and store the class in set $IAB$
7: if class $C$ that implements or extends classes in set $IAB$ exists then
8:    Identify method overriding that exists between $C$ and $IAB_{[i]}$
9:    Identify constructor $k$ in $C$ which accepts element of set $S$ as one of its argument
      types and $C$ is not one of the classes identified in $\{S\}$
10: end if
11: if Rule 8-9 are true then
12:    $Y = Y \cup \{Jf, IAB_{[i]}\}$
13: end if
```

4.2.4 Analysis of Design Pattern Detection in Java Packages

We tested the algorithms defined in the previous section on the Java.awt [128] and JHotDraw [129] packages. These packages were included in the experiment because they have been used in studies related to design pattern detection [130, 78, 77]. Table 4.1 shows the number of classes and programs contained in the packages. We ran our experiment on a Windows XP (2002 Professional edition) personal computer running on a 2.8GHz Intel processor with 1GB RAM.

Table 4.1 Number of Classes and Programs in Java Packages

Package	Number of Classes	Number of Programs
Java.awt	485	345
JHotDraw	464	484

Existing work on design pattern detection includes the *FUJABA* tool suite [78] and the *PTIDEJ* [130]. The developers of *FUJABA* have tested the tool on the Java.awt version 1.3 package and Niere et al. [78] reported only a constellation of classes related to the java.awt.component. Furthermore, it has been reported that *FUJABA* detects a lot of false positives in detecting design patterns in the Java.awt package [131]. On the other hand, Gueheneuc and Jussien [130] neither specified which version of the Java.awt package used in the test nor illustrated any detection accuracy or performance results for *PTIDEJ*.

The recent work done in identifying design patterns in Java programs is discussed by Shi and Olsson in [77, 132]. They proposed a tool named *PINOT* that is claimed to be able to identify design patterns based on patterns described in Gamma et al.[1]. Based on the detection results obtained using *PINOT* (reported in [77, 132]), we learned that the tool has not been able to identify the recursive composite design pattern, that is, composites with cycles. Our retrieval system identifies subclasses of any class that implements design pattern to be relevant to the detection of design patterns. For example, similar to *PINOT*, in the JHotDraw package, we identify the `AbstractFigure.java` as implementing Composite pattern. Nevertheless, we also identify `AttributeFigure.java` as implementing the same pattern as it inherits the `AbstractFigure` class. Also, in the JHotDraw package, as the system identified `AbstractTool` as implementing Observer pattern, 13 other classes that inherit `AbstractTool` were also identified as implementing the same pattern. As our purpose of having design pattern detection in a retrieval system is to help users to retrieve programs that illustrate the required design patterns, the identification of Java classes (including subclasses) that employ a particular design pattern through the use inheritance would also be relevant to the users. This is because these classes may contain similar function as defined in a query program.

Table 4.2 shows the number of programs implementing a particular design pattern identified using the algorithms defined in section 4.2 and the results obtained using *PINOT* [77, 132].

Table 4.2 Detection of Design Patterns in Java Packages

Pattern	Java.awt		JHotDraw	
	Our approach	PINOT	Our approach	PINOT
Singleton	13	6	3	0
Composite	9	6	9	6
Observer	8	2	23	9

Based on literature which includes the work by Rijsbergen [133], Baeza-Yates and Ribeiro-Neto [13], precision and recall for program retrieval are defined as in Definitions 7 and 8. In the context of the present work, the results of design pattern detection are represented using the precision score.

Definition 7 *Precision is the ratio of the number of relevant programs retrieved to the total number of all (irrelevant and relevant) programs presented in a hit list (see Definition 6 on page 66).*

Definition 8 *Recall is the ratio of the number of relevant programs retrieved to the total number of relevant programs in a repository.*

Both precision and recall have a fixed range: 0.0 to 1.0 (or 0% to 100%). Our test results were verified against a pattern discussion board [134] and by manual means. The precision scores of the design pattern detection are presented in Table 4.3. More than half of the programs identified to implement the three design patterns are relevant. The precision score for detecting Singleton pattern in the JHotDraw package was 100% while approximately 90% of the programs identified by the detector to implement Composite and Observer pattern were relevant. A similar result was also obtained when the design pattern detection was performed on the Java.awt package, as the precision scores were all greater than 50%.

Table 4.3 Precision Scores for Design Pattern Detection in Java Packages

Pattern	Java.awt	JHotDraw
Singleton	0.667	1
Composite	0.77	0.889
Observer	0.625	0.87

4.2.5 Analysis of Design Pattern Identification in Open-Source Applications

In this experiment, we used five Java applications obtained from `Sourceforge.net` [29], representing different domains that include database, entertainment, scheduling and communications. The applications included in the experiment are as follows:

- *Borg*, a calendar and task tracking system written with the purpose of providing a month view, month-print, email reminders, popup reminders and a to do list. The task tracker manages issues through various states of time.
- *FreeGuide*, a program used to develop a TV guide. It is able to download TV listings from the Internet, view them off-line, create a personalised TV guide, and allow users to choose their favorites programmes.
- *Jtds*, an open-source JDBC driver for Microsoft SQL Server and Sybase. It is based on the work of the FreeTDS project and is currently the fastest complete JDBC driver for SQL Server and Sybase.
- *Kafenio*, a WYSIWYG Editor for HTML Browsers that supports Java 1.3. Kafenio is partly based on Howard Kistlers' Ekit Editor.
- *TvBrowser* is a Java-based TV guide which is easily extendible using plugins. It is designed to look like a paper based TV guide.

Using the algorithms defined in section 4.2, our design pattern detector identifies 69 out of 288 programs to have implemented design patterns. And, out of the 69 programs, 44 of them employed a Composite design pattern and 14 implemented the Observer design pattern, while the remainder (i.e 11) was identified as employing the Singleton pattern. A percentage ratio of the relevant programs is shown in Figure 4.4.

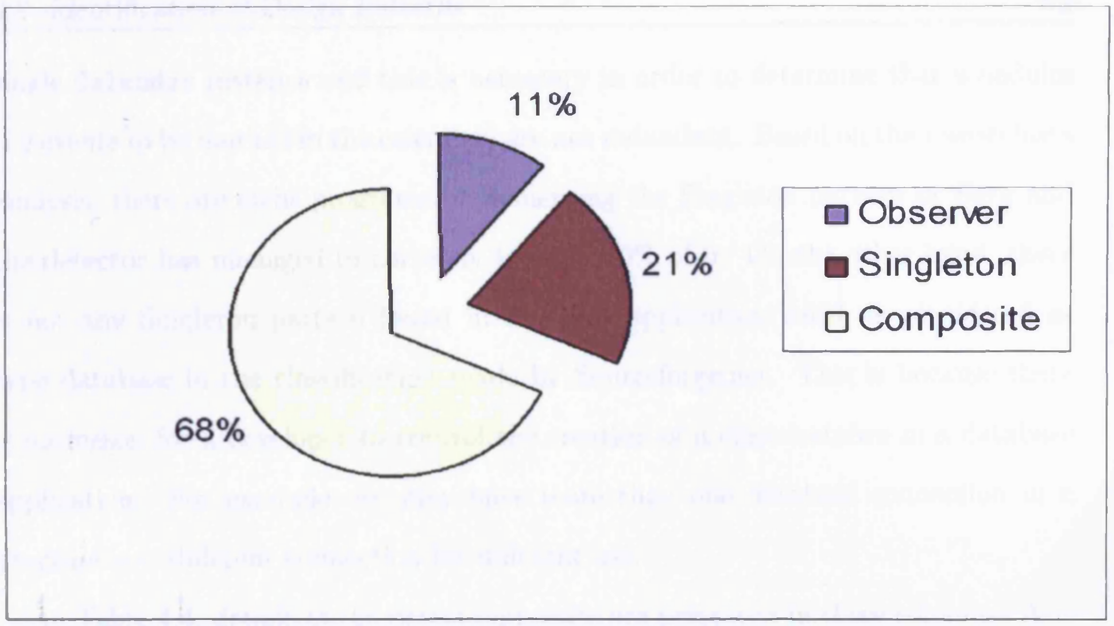


Figure 4.4 Ratio of Detected Design Patterns

Based on data depicted in Figure 4.4, it is suggested that it is more common to find the Composite design pattern rather than the Singleton or Observer in open-source applications. This supports the idea that Composite design pattern is the core abstraction behind successful recurring frameworks [135]. Based on our analysis on the programs used in this experiment, software developers tend to develop an application by treating combinations of objects uniformly. For example, in the *FreeGuide* package, the developer created a catalogue of a TV programme by combining related catalogues obtained from the internet. Our design pattern detector has identified 20% of the programs in this package to have employed the Composite design pattern.

The Singleton design pattern is found in the scheduling application, *Borg*. *Borg* is classified under the scheduling domain as it a task tracking system that helps users to organize their activities. In the *Borg* application, for example, there exist only a single `Calendar` instance and this is necessary in order to determine that schedules (i.e events to be marked in the calendar) are not redundant. Based on the researcher's analysis, there are eight programs implementing the Singleton pattern in *Borg* and the detector has managed to correctly identify 50% of it. On the other hand, there is not any Singleton pattern found in the *Jtds* application which is considered as type database in the classification made by Sourceforge.net. This is because there is no reason for a developer to control the creation of a class instance in a database application. For example, we may have more than one database connection in a program –– different connection for different use.

In Table 4.4, details of the detection results are presented in three columns: **Answer Set (*Ans*)**, **Retrieval Set (*Ret*)** and **Relevant Retrieval (*AnsRet*)**. The *Ans* column contains the number of programs that have been identified (manually) to implement a particular design pattern while the *Ret* column includes the number of programs identified by the proposed detector. Upon obtaining the two sets of programs, we then determine programs that are listed in both sets and include the information (number of programs) in the *AnsRet* column.

Table 4.4 Detection of Design Patterns in Open-Source Applications

Application	# Programs	Singleton			Composite			Observer		
		Ans	Ret	AnsRet	Ans	Ret	AnsRet	Ans	Ret	AnsRet
Borg	46	8	5	4	7	4	4	5	6	2
FreeGuide	72	1	1	1	17	14	14	0	0	0
Kafenio	54	3	2	2	14	10	9	5	5	5
TvBrowser	49	7	6	6	14	9	9	0	0	0
Jtds	67	0	0	0	10	7	6	0	0	0
Total	288	19	14	13	62	44	42	10	11	7

The precision and recall scores for design pattern detection in open-source applications are presented in Table 4.5. These scores were generated based on data depicted in Table 4.4. The empty column marked by – indicates that there is not any precision or recall for the particular application since the application does not employ any design patterns. Based on the analysis, our design pattern detector has correctly identified 67% (i.e 44/62) of the Composite pattern embedded in the open-source applications, hence generating an average recall of 0.65 which is approximate to the recall (average) for detecting the Observer pattern, 0.7. On the other hand, the average recall for detecting Singleton pattern was 0.71. We also discovered that developers of the open-source applications (e.g *FreeGuide* and *Kafenio*) have not used interface classes in employing the Composite design pattern, but instead they incorporate the pattern using inheritance relationships. Most of the methods inherited from the superclass were override to include different behaviours of the class instance.

Table 4.5 Precision and Recall for Design Patterns Detection in Open-Source Applications

Application	Singleton		Composite		Observer	
	Precision	Recall	Precision	Recall	Precision	Recall
Borg	0.8	0.5	1	0.57	0.33	0.4
FreeGuide	1	1	1	0.82	-	-
Kafenio	1	0.67	0.9	0.64	1	1
TvBrowser	1	0.86	1	0.64	-	-
Jtds	-	-	0.85	0.6	-	-
average	0.95	0.71	0.95	0.65	0.67	0.7

4.3 Design Patterns in Program Retrieval

We proposed the use of design patterns as structural descriptors of programs contained in a repository. Our program retrieval system make use of information on the existence of design patterns in a program in identifying similarity between a query and programs in the repository. Similar to work described in the previous chapter, users of our retrieval system are allowed to use program as the search query. By doing so, users are able to express their search requirements precisely; functional descriptors based on a program context and structural descriptors based on structural relationships. Our retrieval system assists users to develop applications that can be reused in the future; function-based and/or structure-based. This is achieved by presenting the users with programs that illustrate the required function (based on functional descriptors elaborated in Chapter 3) and structure (i.e design patterns) which can be used as programming examples. As one of the best ways of learning how to program is by examining programming examples [136], users can modify the programs presented by our retrieval system to fulfill the requirements in a given programming task.

In the process of designing an object oriented application, developers generate details of how objects should be created and managed, and how they should behave -- structural relationships between objects. This can be done using various modelling tool which includes the UML [37] that provides different diagrams to model different design requirements. Once this information has been identified, a developer may use his/her existing program which was created based on a particular diagram (e.g Entity Relationship Diagram (ERD)) as a search query to retrieve similar programs from a software repository.

When retrieving programs that may contain a Singleton design pattern, developers are presented with code statements that control the creation of class instance.

Sometimes, it is important for some classes to have exactly one instance, for example, program system and print spooler. Java developers require the Composite pattern because, often developers manipulate composites exactly the same way they manipulate primitive objects. For example, graphic primitives such as lines or text must be drawn, moved, and resized. However, developers also want to perform the same operation on composites, such as drawings, that are composed of those primitives. By retrieving programs that implement a Composite design pattern, developers could use these programs as examples for them to develop groups of objects with the same composition. On the other hand, programs that were implemented based on an Observer design pattern illustrate a one-to-many dependency between objects -- when one object changes state, all of its dependents are notified and updated automatically. Such programs may be useful to developers creating an application that requires broadcasting utilities. For example, an instance of class `Engine` that allows another instance, `EngineMonitorTemperature`, to monitor its operating temperature. The `EngineMonitorTemperature` is responsible for monitoring the Engine's operating temperature and act appropriately when the temperature exceeds the maximum allowed temperature. It will do this without the `Engine` needing to do or know anything about it.

In order to utilize design patterns as structural descriptors in a program retrieval system, we use information on the structural relationships that illustrate a pattern as a program index. This means that each of the programs under analysis is given a tuple containing three indices: the first represents the Singleton, the second corresponds to the Composite while the last indicates the Observer design pattern. These indices represent numbers of structural relationships that illustrate a particular design pattern. The structural relationships were the rules presented in Algorithms 1, 2, 3 and 4 in section 4.2. For example, if P99 illustrates the Singleton design pattern by obeying one of the three rules defined in Algorithm 1 (that is either step 5, 6 or 7 in the algorithm) then the program tuple for P99 would be $\{1, 0, 0\}$.

4.4 Conclusion

Design patterns are used as one of the program descriptors in order to assist developers in retrieving programs that employ the required design patterns. As existing program retrieval systems such as the Google code search [23] and Koders search engine [24] are solely based on keyword match, fulfilling search requirements that includes design patterns would be difficult. This is due to the difficulties in identifying code statements that explicitly depict existence of design patterns in a program (e.g *This class implements Observer pattern*). However, by analyzing the structural relationships contained in a program, we are able to infer the existence of a particular design pattern in the program. Such information is later used as program descriptors and contributes to identifying programs that illustrate the required function and structure as defined in a query program.

In this chapter, we have demonstrated the identification of programs that implement a particular design pattern (i.e Singleton, Composite and Observer). As the software repository contains open-source applications that may require different execution environment, the use of structural analysis in detecting design patterns in a program is more practicable than the work undertaken based on dynamic analysis. This is because dynamic analysis requires the application under analysis to be executed and since open-source applications may require specific environments prior to execution, such an approach is less practicable. In addition, compared to existing work of design patterns detection [72, 78, 77], we include the identification of a recursive Composite design pattern. Such detection identifies sub classes of any Java class (super class) that implements a particular design pattern to be relevant to the design pattern detection. This is an advantage since the sub classes may illustrate a similar function as required by the user, while the super class does not. Each of the identified design patterns in this work represents a different phase of managing classes and object instance(s). Starting from how objects are created, followed by how they are related and how they behave, a retrieval system that includes design patterns as program descriptors can assist users in retrieving programs that implement the required design pattern. Even though the presented work does not include the identification of all design patterns described by Gamma et al. [1], we see it as a starting point in developing a program retrieval system that combines functional and structural descriptors.

CHAPTER 5

SOFTWARE METRICS AS STRUCTURAL DESCRIPTORS

In this chapter, we describe the use of software metrics as structural descriptors in retrieving relevant programs from a repository. Based on software metrics that are automatically extracted from a program, the program is classified into either database or graphics application domain.

5.1 Overview

Software reuse does not only mean use of existing codes, it also involves the organization and use of conceptual information [137]. This includes organizing programs into the programming language used, execution platform and application domain. Nevertheless, much of the work [29, 30] undertaken in organizing the programs is performed manually and/or is based on natural language. For example, in `SourceForge.net` [29], classification of an application into the appropriate domain (e.g database, multimedia, games and financial) is undertaken based on the description provided by its developer. If such an approach is adopted into a program retrieval system, classification of programs in the system could not be automated if relevant descriptions are not available. In addition, program classification that is performed based on textual analysis of code statements [28] is not applicable if the programs are not well-documented (the definition of well-documented is provided in Definition 2 on page 4). Therefore, there is a need of methods that do not rely on textual analysis in classifying a program into an application domain. There are two objectives in the work described in this chapter: 1) to classify a program into an application domain (i.e database or graphics) using structural information extracted from the program, and 2) to investigate whether program classification contributes to a better program retrieval. We extend the approach used in `SourceForge.net` [29] and `Freshmeat.net` [30] by using software metrics extracted from a program to determine the appropriate application domain for the program.

We used programs obtained from `SourceForge.net`, which are then automatically classified into application domains using the predictions made by Discriminant Analysis (DA) [138], C4.5 decision tree [105] and k-nearest neighbour (KNN) [106]. Related discussion on these classifiers have been presented in section 2.2.1 on page 46. In order to determine if program classification contributes to a better program retrieval, the term occurrences approach which is undertaken based on the context of a program, as described in Chapter 3, is employed to represent the function of the program. By using DA [138] which is based on statistical analysis, we can determine how software metrics may be combined into a mathematical equation to predict the most likely application domain of a program. On the other hand, the C4.5 [105] is included in the experiment as it has always been the point of comparison for novel approaches in machine learning [108]. In addition, the work undertaken by Lim et al. [139] and Ganti et al. [140] showed that the C4.5 algorithm generates good classification accuracy and is the fastest among the compared algorithms (i.e Neural network and k-nearest neighbor). As for the K-nearest neighbour (KNN) [106], it is chosen to be included in the experiment due to its algorithm that classifies an object (i.e program) based on the classification of the object's neighbourhood. With this, we can identify if programs from the same domain contains similar metric values. Furthermore, as our repository contains programs that were developed by different developers with different styles of writing source code, the use of KNN which is robust to noisy data (i.e outlier) and is an effective classifier for large data sets [141, 112, 111] would be suitable.

We use the freeware C and C++ Code Counter (CCCC) [142] to extract software metrics from a program. CCCC is a source code analyzer tool that analyses C++ and Java programs and generates a report on various metrics of the code. At the time of our research, CCCC extracted a total of 19 software metrics, nevertheless, only twelve metrics were used in the experiment undertaken. The selection is made based on the strength of relationship that exists between the metrics and the application domain categorized in `Sourceforge.net` [29]. Such relationships are identified using the Pearson correlation analysis [103] which describes the strength and direction of a linear relationship between two continuous variables. According to Pallant [103], a correlation of 0 indicates no relationship at all, a correlation of 1.0 indicates a perfect positive correlation, and a value of -1.0 indicates a perfect negative correlation. Regardless of the direction of the relationship (positive or negative), Cohen [143] suggested that the value of Pearson correlation, r, is considered to be large (i.e strong) if it is between 0.5 and 1.0. If r is in the range of 0.3 and 0.49, then it is considered as medium, and, if r is equal or less than 0.29, then the correlation is considered to be small. Based on this suggestion, only software metrics that depict a correlation as low as 0.5 are chosen to be used as the independent variables in determining the dependent variable (i.e application domain). The selected metrics are listed below and details of these metrics can be found in section 2.1.2 on page 33.

1. Number of Modules (Nom)
2. Lines of Code (Loc)
3. McCabe's Cyclomatic Complexity (Mvg)
4. Depth of Inheritance Tree (Dit)
5. Coupling Between Object Classes (Cbo)
6. Weight Method per Class (Wmc)
7. Ficon
8. Fivis
9. Fiincl
10. Focon
11. Fovis
12. Foincl

In this work, we are also making the assumption that program classification is better undertaken using a small number of metrics. This is because such an approach requires less information extraction. Therefore, from the twelve metrics, we exclude metrics that depict $r < 0.7$. Such a value (i.e 0.7) was used as the cut-point, as Owen and Jones [144] suggested that a strong relationship between two variables is depicted by a correlation value higher than 0.7. Based on the suggestion, we include a second experiment that uses metrics Nom, Mvg, Cbo, Wmc, Dit, Fivis, Ficon and Fiincl to differentiate a database from graphics programs.

5.2 Program Classification into Application Domains

In this work, programs from two application domains as classified in SourceForge.net [29] were used: database and graphics. These programs were chosen as they depict a similar function, that is data organization (e.g add, remove), but yet operate on different types of data. Database programs organize text-based information such as employee details in a personnel database system, while graphics programs focus on the organization of graphical objects, for example, photo images. A program is classified under the category of database if the functionality of the program is related to querying, storing, or managing (updating) information (i.e text-based) in a database. In addition, programs that illustrates the process of connecting to a database system are also categorized under the domain of database.

On the other hand, programs under the category of graphics should illustrate the function of creating, modifying and/or storing images. These may include programs that, have collaboration tools (e.g whiteboard in Netmeeting), add captions and descriptions to digital photograph collections, and/or provide utilities for the manipulation of graphic images (e.g resize, crop). In addition, programs that visualize images contained in a given directory are also considered as graphics programs. It is the intention of the experiment in this study to see how well the classifiers (i.e DA, C4.5 and KNN) would do when trained on programs of distinct domains; database programs deal with structured objects while graphics programs handle unstructured objects.

To obtain the classifier models, we performed experiments on two data sets; training and testing. The training data set includes a total of 584 Java programs (371 database and 213 graphics) while the testing data set contains 236 Java programs.

The classifier models obtained during training process were later verified using the testing data set. We used classifiers C4.5 and KNN which are provided by *WEKA* [145] and DA was obtained using *SPSS* [146]. *WEKA* is a collection of machine learning algorithms for data mining tasks developed at the University of Waikato and is an open-source software issued under GNU general Public license. In *WEKA*, the C4.5 decision tree is known as J48 while the KNN is known as an instance based neighbour (IBk). Therefore, in this thesis, the results are reported using these names (C4.5 and IBk). On the other hand, *SPSS* was used to generate a DA classifier model. In this chapter, the experiment undertaken attempts to find evidence that the software metrics (i.e independent variables) contained in a program can be used to determine the application domain (dependent variable) of the program. Furthermore, we intend to learn if program classification is better undertaken using twelve metrics rather than 8 metrics.

Data presented in Figure 5.1 shows that the training models generated using twelve independent variables produced a higher classification accuracy compared to using eight variables. The classification accuracy is calculated based on the number of correctly classified programs when compared to the classification undertaken in `Sourceforge.net` [29]. In this experiment, a combination of 12 independent variables has helped the classifiers to differentiate better between the two categories of programs. Based on the analysis performed on the programs, we learned that values for certain metrics in the database and graphics programs are in the same range. For example, one third (30%) of the programs in the repository contained Fivis that are in the range of 5 to 10, and 40% of the programs contain the same Nom values, which is 2. Therefore, the classifiers need a larger number of independent variables in discriminating between the two types of programs. As the classifier models which were generated using twelve metrics produced a higher classification accuracy compared to

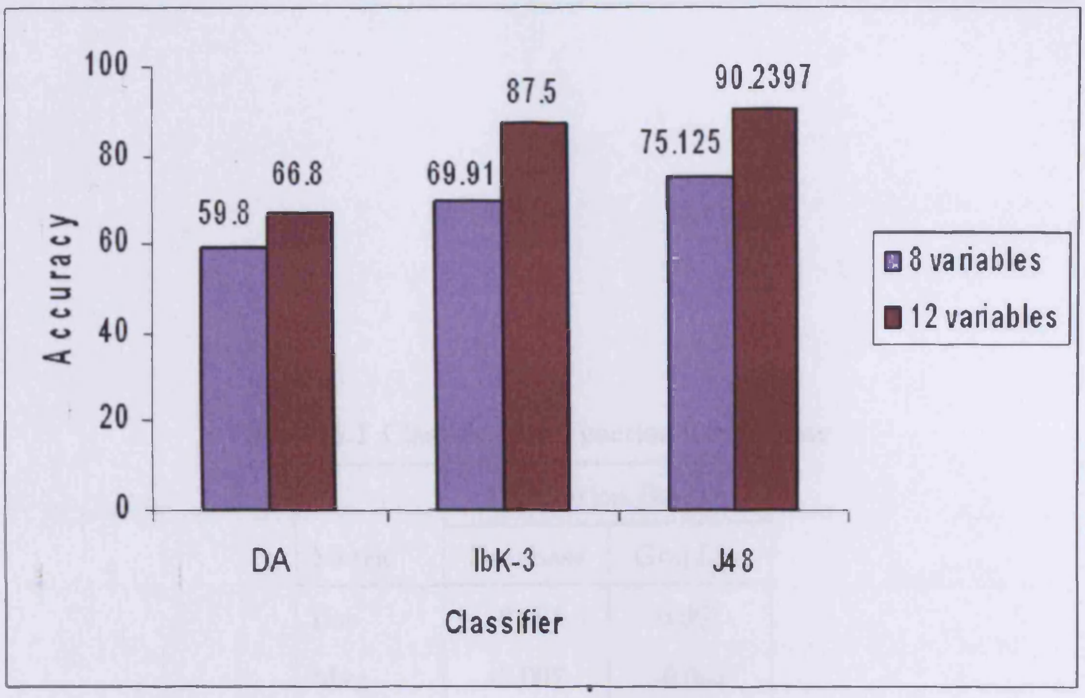


Figure 5.1 Classification Accuracy using 8 and 12 Independent Variables

the models based on eight metrics, further experiments undertaken in this work were based on the twelve metrics models.

There were two sets of classification function coefficients used in developing a DA classifier model and they are depicted in table 5.1. The DA model does not include data from metrics Fiincl and Foincl due to similarities of values with metrics Fivis and Fovis. Using this model, the classification of a program into an application domain can be made using equations 5.1 and 5.2 which were generated based on the coefficients depicted in Table 5.1. These equations generate classification scores for the program.

Table 5.1 Classification Function Coefficients

	Application Domain	
Metric	Database	Graphics
Loc	-0.001	0.001
Mvg	-0.007	-0.002
Nom	0.007	-0.007
Wmc	-0.002	0.044
Dit	0.758	0.199
Cbo	0.044	0.031
Fivis	0.112	0.066
Ficon	-0.053	0.258
Fovis	-1.375E-05	-6.952E-06
Focon	-0.001	-0.001
Constant	-1.299	-1.190

$$\begin{aligned} f\{database\} &= (-0.001 * Loc) - (0.007 * Mvg) + (0.007 * Nom) \\ &\quad -(0.002 * Wmc) + (0.758 * Dit) + (0.044 * Cbo) + (0.112 * Fivis) \\ &\quad -(0.053 * Ficon) - ((1.375E - 05) * Fovis) - (0.001 * Focon) - 1.299 \end{aligned} \tag{5.1}$$

$$\begin{aligned} f\{graphics\} &= (0.001 * Loc) - (0.002 * Mvg) - (0.007 * Nom) \\ &\quad +(0.044 * Wmc) + (0.199 * Dit) + (0.031 * Cbo) + (0.066 * Fivis) \\ &\quad +(0.258 * Ficon) - ((6.952E - 06) * Fovis) - (0.001 * Focon) - 1.190 \end{aligned} \tag{5.2}$$

Once the classification scores for a program have computed, the program is classified belonging to the domain for which it has the highest classification score. For example, the classification scores for program P1 which contains metric values {55, 8, 2, 6, 1, 3, 3, 2, 2, 2 } are -0.29003 and -0.42501 for $f\{database\}$ and $f\{graphics\}$ respectively. Since the value of $f\{database\}$ is greater than $f\{graphics\}$, the DA model considers P1 as a database program.

In order to get the optimum classification result using a KNN model, we need to choose the appropriate value of k, which is the number of nearest neighbour to a program which is under analysis (query point). The choice of k is regarded as one of the most important factors of the model that can strongly influence the classification result [111]. A small value of k will lead to a large variance in predictions while setting k to a large value may lead to a large model bias. Thus, k should be set to a value large enough to minimize the probability of misclassification and small enough (with respect to the number of cases in the sample) so that the k nearest points are close enough to the query point. Thus, there is an optimal value for **k** that achieves the right trade-off between the bias and the variance of the model. In our work, **k** is given the value of 3 resulting from the process of estimating **k** using an algorithm known as cross-validation [147]. Prior to obtaining the value, the KNN model has been trained using value **k** ranging from 1 to 13. The undertaken experiment (based on cross-validation) showed that by using **k=3**, we obtained the highest classification accuracy.

A decision tree generated based on the J48 analysis is provided in *Appendix A: Decision Tree - J48*. In measuring the effectiveness of the three classifiers, the question is, *have we classified the programs into their application domain or have we misclassified some of the application domain?*. A number of these measures are derivatives of measurements from the information retrieval domain.

- **falsePositives** The number of incorrect classifications a category contains.
- **Precision** The ratio of the number of relevant programs classified to the total number of irrelevant and relevant programs being classified.
- **Recall** The ratio of the number of relevant programs classified to the total number of relevant programs in the category.

Tables 5.2(a) and 5.2(b) reveal the falsePositives, precision and recall scores which were calculated after the program classification experiment was completed. Data in Table 5.2(a) shows that J48 and IbK-3 have similar capabilities in identifying database programs. This is shown by the recall scores depicted in table 5.2(a) which shows that there is only a difference of 0.1 between the two classifiers. However, J48 outperforms both DA and IbK-3 by 10% in the precision scores. Ninety two percent (92%) of programs classified into the database domain, using the J48 model,

Table 5.2 Classification Analysis based on 12 Independent Variables

(a) Database

MEASUREMENT	J48	IbK-3	DA
Precision	0.929	0.882	0.837
Recall	0.916	0.927	0.593
falsePositives	0.122	0.216	0.202

(b) Graphics

MEASUREMENT	J48	IbK-3	DA
Precision	0.858	0.861	0.53
Recall	0.878	0.784	0.732
falsePositives	0.084	0.073	0.407

are actually relevant, hence generating the lowest falsePositives score, i.e 0.122.

Data in Table 5.2(b) shows that there is a reduction in the precision and recall scores for J48 and IbK-3 when compared to the scores obtained for database programs. On the other hand, the DA model generates a better classification then the one made for database programs. An increment of approximately 15% in recall was achieved by DA in classifying graphics programs compared to the database programs.

Based on the data depicted in Table 5.2 and in Figure 5.2, we learned that machine learning classifiers outperformed the statistical-based classifier in classifying programs contained in our repository. The recall scores depicted in Table 5.2 show that the IbK-3 and J48 outperformed DA in classifying the programs. Such a result can be accounted for by the similar metric values depicted in programs used in the undertaken experiment. In this context, two metric values are considered similar if they are the same (exact) or depicting a difference of two (the most). For example, if the Ficon metric values in P1 and P2 are 3 and 2 respectively, then the programs are considered to have the similar metric values. The distance between programs with similar metric values (when mapped into a vector space) is small, hence suggesting the IbK model, which is based on neighbourhood classification, to classify the programs into a same domain.

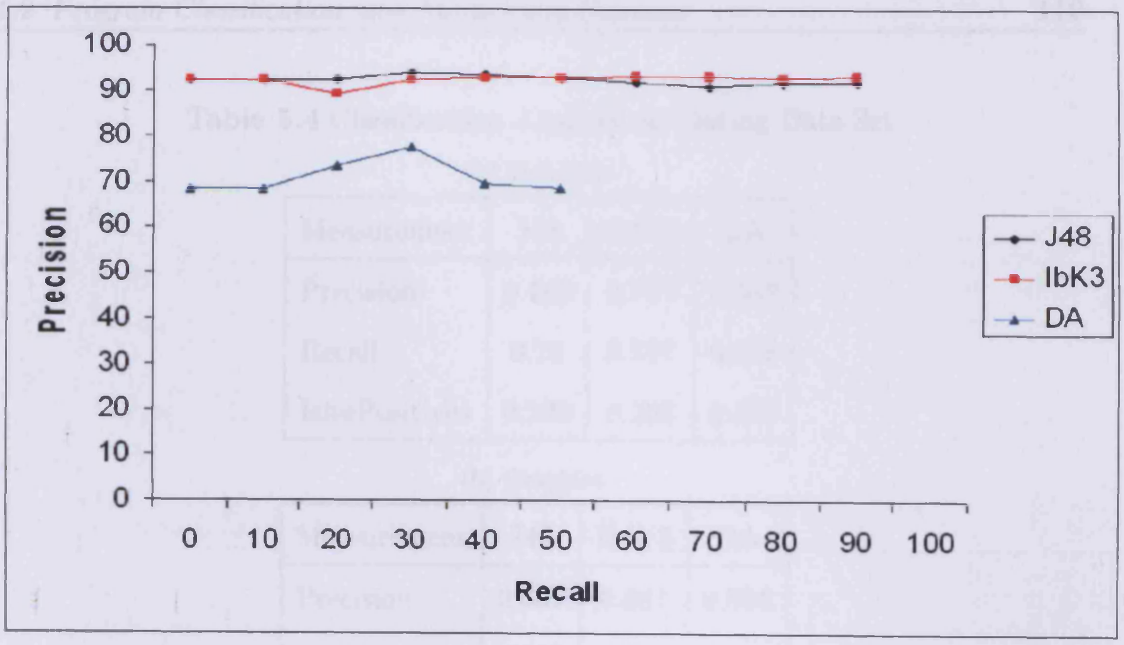
5.2.1 Program Classification using Testing Data Set

The classifier models from the previous experiment were then used to classify a new set of programs (testing data set) containing 136 Java programs. This data set contains 64 database and 72 graphics programs which were also obtained from `Sourceforge.net` [29]. From the data depicted in table 5.3, it is noted that the classification accuracy obtained using the three classifiers are in the range of 57% to 73%. Out of 136 programs, the IbK-3 has correctly classified 98 programs while 94 programs classified by the J48 model was relevant. On the other hand, the DA has correctly classified 57% of the programs (i.e 78 out of 136).

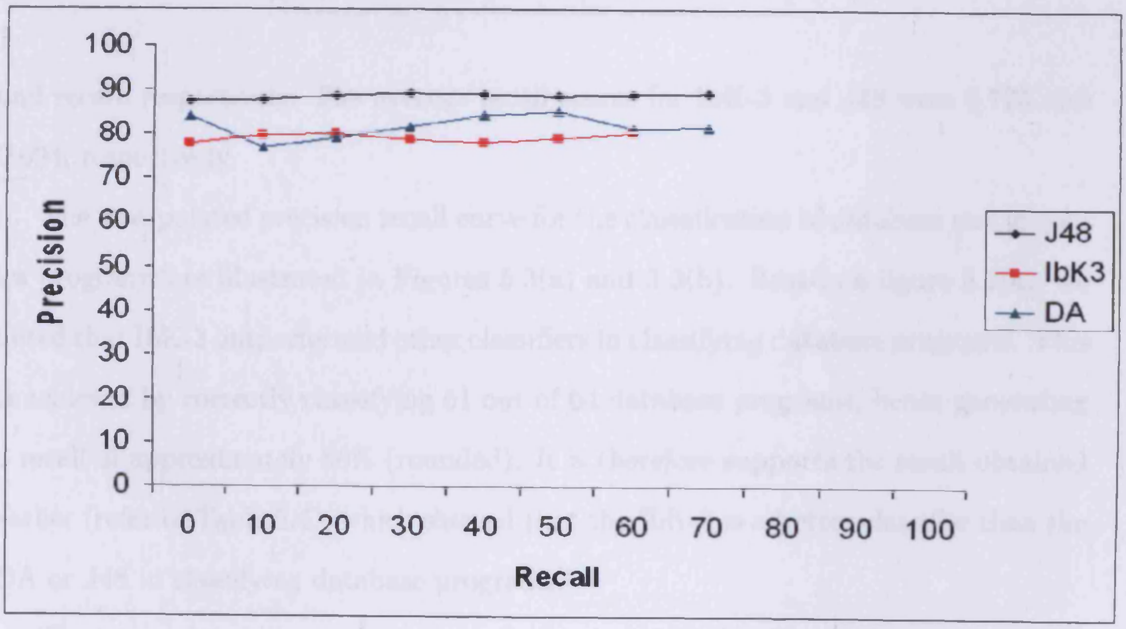
Table 5.3 Classification Accuracy: Testing Data Set

Measurement	**J48**	**IbK-3**	**DA**
Testing Data Set	70.149	73.134	57.46

In Tables 5.4(a) and 5.4(b), the precision, recall and falsePositives scores obtained after completing the program classification experiment are provided. As expected (based on results obtained in the previous experiment), the IbK-3 model has outperformed the DA and J48 in classifying programs into the application domains. The average precision score for IbK-3 was 0.739 while J48 generated an average of 0.714. On the other hand DA generated an average of 0.581 and 0.567 for precision



(a) Database



(b) Graphics

Figure 5.2 Interpolated Precision at 11 Standard Recall Levels relative to Program Classification into Application Domain

Table 5.4 Classification Analysis on Testing Data Set

(a) Database

Measurement	J48	IbK-3	DA
Precision	0.762	0.797	0.607
Recall	0.75	0.797	0.578
falsePositives	0.239	0.203	0.393

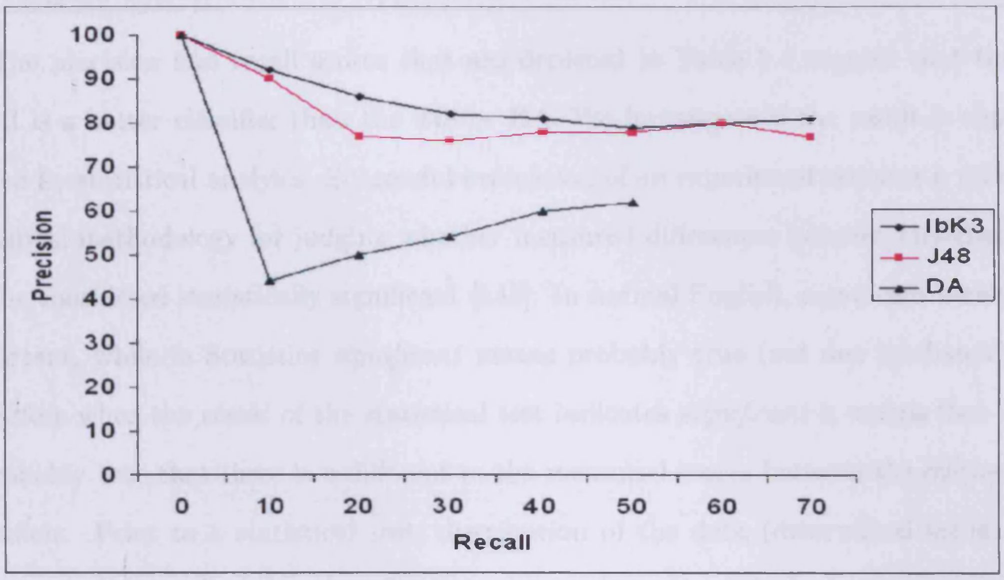
(b) Graphics

Measurement	J48	IbK-3	DA
Precision	0.667	0.681	0.556
Recall	0.639	0.653	0.557
falsePositives	0.406	0.347	0.444

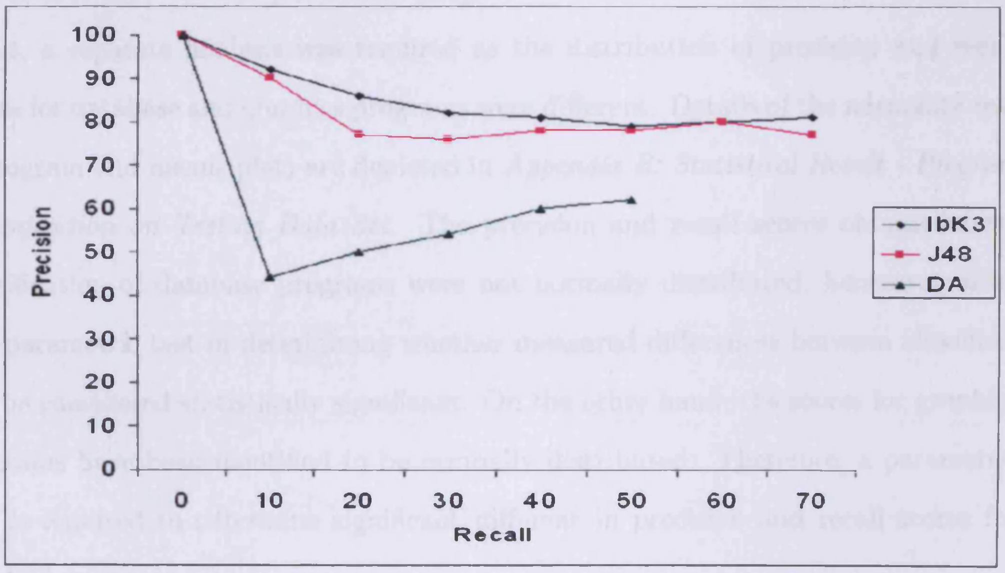
and recall, respectively. The average recall scores for IbK-3 and J48 were 0.725 and 0.694, respectively.

The interpolated precision recall curve for the classification of database and graphics programs are illustrated in Figures 5.3(a) and 5.3(b). Based on figure 5.3(a), we noted that IbK-3 outperformed other classifiers in classifying database programs. This is achieved by correctly classifying 51 out of 64 database programs, hence generating a recall of approximately 80% (rounded). It is therefore supports the result obtained earlier (refer to Table 5.4), which showed that the IbK-3 is a better classifier than the DA or J48 in classifying database programs.

The interpolated precision recall curve that is depicted in Figure 5.3(b), illustrates that IbK-3 has also out performed DA and J48 in classifying graphics programs. The IbK-3 has correctly classified 65% of the graphics program compared to J48 which correctly classified 64% of programs from the same domain.



(a) Database



(b) Graphics

Figure 5.3 Interpolated Precision Recall Curve for Program Classification using Testing Data Set

The precision and recall scores that are depicted in Table 5.4 suggest that the IbK-3 is a better classifier than the J48 or DA. We investigate if the result is supported by statistical analysis. Successful evaluation of an experiment requires a valid statistical methodology for judging whether measured differences between classifiers can be considered statistically significant [148]. In normal English, ***significant*** means important, while in Statistics ***significant*** means probably true (not due to chance). Therefore when the result of the statistical test indicates ***significant*** it means that it is probably true that there is a different in the measured scores between the distinct classifiers. Prior to a statistical test, distribution of the data (determined using a normality test) needs to be identified in order to determine which significance test (parametric or non-parametric) is most suitable for the given set of data [103].

A normality test was performed using *SPSS* version 11.5 and based on the test result, a separate analysis was required as the distribution of precision and recall scores for database and graphics programs were different. Details of the normality test (histogram and means plot) are depicted in *Appendix B: Statistical Result - Program Classification on Testing Data Set.* The precision and recall scores obtained from classification of database programs were not normally distributed, hence requiring non-parametric test in determining whether measured differences between classifiers can be considered statistically significant. On the other hand, the scores for graphics programs have been identified to be normally distributed. Therefore, a parametric test is required to determine significant different in precision and recall scores for graphics programs.

The non-parametric Kruskal-Wallis test [103] was conducted to determine whether there is significant different in precision and recall scores for database programs. Data in Table 5.5(b) reveals that there is a significant different at $\alpha = 0.05$ in the precision and recall scores across the three classifiers. Such α value was used in the test as it is one of the standard values used in a statistical test [103]. Both of the `Asymp.Sig.` values for the precision and recall scores are 0.000 which are less than the α value. This indicates that there is a different in precision and recall scores across the J48, DA and IbK-3.

When the data under analysis are not normally distributed, and the measurements at best contain rank order information, then inspection of the mean ranks would reveal which classifier is better than the others [103]. The mean ranks depicted in table 5.5(a) reveals that the IbK-3, with 63.52 and 68.37 had the highest precision and recall scores, with the DA reporting the lowest. Such a result indicates that the IbK-3 is a better classifier compared to DA and J48.

Table 5.5 Kruskal-Wallis Test Result Relative to Program Classification into Database Domain

(a) Mean Rank

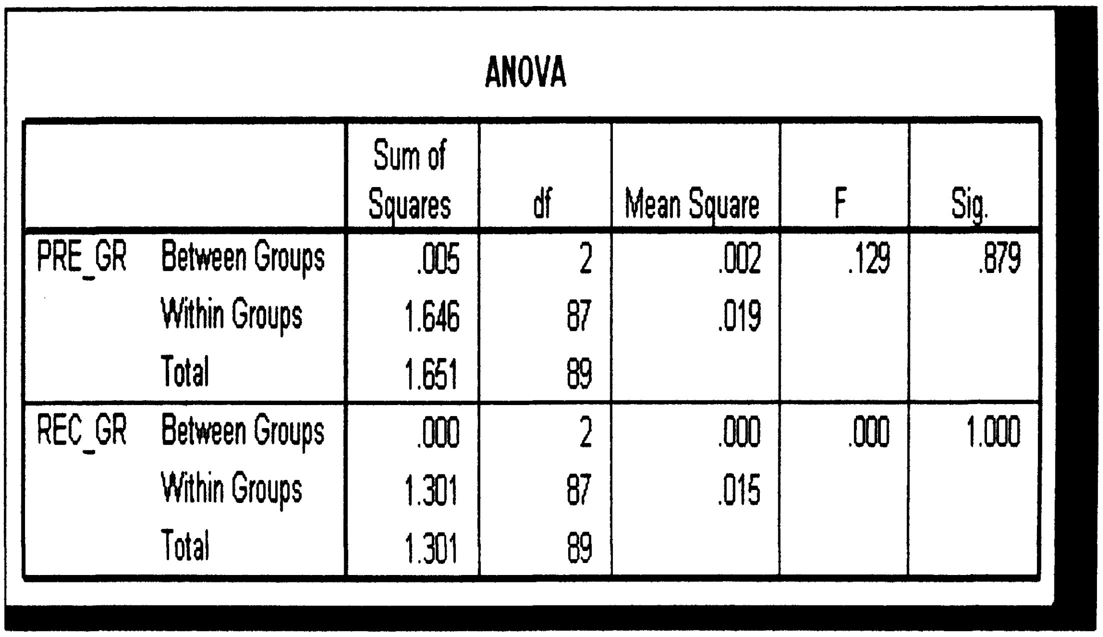
Scores	Classifier	Mean Rank
Precision	J48	50.22
	IbK-3	63.52
	DA	22.77
Recall	J48	45.50
	IbK-3	68.37
	DA	22.63

(b) Test Statistics

	Precision	Recall
Chi-Square	38.146	46.006
df	2	2
Asymp.Sig.	0.000	0.000

In order to identify whether there is a significant different between classifiers in precision and recall scores for graphics programs, we conducted a one-way between-groups analysis of variance (one way ANOVA [103]). The statistical results depicted in Table 5.6 indicate that the difference in precision and recall scores, obtained using DA, J48 and IbK-3 are less significant at $\alpha = 0.05$. This is depicted by `Sig.` values of 0.879 for precision and 1.00 for recall which were greater than the α value used in the test, that is 0.05. Other information (i.e descriptive statistics) related to the test is included in *Appendix B: Statistical Result - Program Classification on Testing Data Set.*

Table 5.6 ANOVA Test Results of Precision and Recall Scores



ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
PRE_GR	Between Groups	.005	2	.002	.129	.879
	Within Groups	1.646	87	.019		
	Total	1.651	89			
REC_GR	Between Groups	.000	2	.000	.000	1.000
	Within Groups	1.301	87	.015		
	Total	1.301	89			

Programs of Combined Domain

In addition to the testing data set, the classifier models have also been used to classify six programs that the researcher believes belong to both domains. This is because the programs illustrate the characteristics of both domains. For example, programs obtained from the Data Crow 2.12 project [29] organize data of type movie, book, images and software (games and program). As they deal with different types of data (structured and unstructured), the programs can be classified as database and graphics programs. These programs were also obtained from `Sourceforge.net` [29].

Table 5.7 reveals the application domain of the programs which were predicted using classifier models J48, IbK-3 and DA. In addition, the table also include classification made by `Sourceforge.net`. Classifier J48 and IbK-3 have similarly classified half of the programs into the graphics domain and the other half as type database. Even though classifier DA has also classified three programs into the graphics domain, nevertheless one of them is not the same program as classified by J48 and IbK-3. Both classifiers J48 and IbK-3 categorize P3 into the database domain while DA classified it as type graphics. On the other hand, DA classified P4 to be a database program while J48 and IbK-3 classified it as a graphics program.

5.3 Program Retrieval based on Terms and Application Domain

In this section, we investigate whether program retrieval can be improved if programs are classified into application domains prior to retrieval. Given a query, retrieval of relevant programs is performed using four mechanisms. The first mechanism involves retrieving programs based on term occurrences as elaborated upon in section

Table 5.7 Program Classification: Programs of Combined Domain

Program	J48	IbK-3	DA	`Sourceforge.net`
P1	database	database	database	database
P2	database	database	database	database
P3	database	database	graphics	graphics
P4	graphics	graphics	database	internet
P5	graphics	graphics	graphics	graphics
P6	graphics	graphics	graphics	graphics

3.4 on page 64. Similar to the first mechanism, the second, third and fourth retrieval mechanisms are also based on term similarities, but prior to retrieval, programs are classified into application domains. The second retrieval mechanism classifies programs using the IbK-3 model, the third uses the J48 classifier model while the fourth employs the DA model. The retrieval was performed on a repository containing 584 Java programs, which is the same set of programs used in the experiment elaborated upon in section 5.2.

Given a query, we compare the precision and recall scores (defined as in Definitions 7 and 8 on page 88) calculated based on the results obtained using the four mechanisms. We later investigate whether there is a significant different in the measured scores between the mechanisms, also, we determined if classifying programs into application domain improves the precision and/or recall.

A total of ten queries was used in the experiment and retrieval analysis was undertaken for the first 25 programs presented in the hit list (refer to Definition 6 on page 66 for the definition of a hit list). Examples of queries posted to the retrieval system are as defined in Table 5.8. Each query includes three types of information: (1) term (e.g add), (2) context of the term in a program (e.g method) and (3) application domain of the required program (e.g database). The expected outcome from the retrieval is a list of programs that contain the desired term, which is used in the required context, and the programs are suitable to be used in the requested application domain.

Table 5.8 Examples of Queries

Query	**Term**	**Program Component**	**Application Domain**
1	retrieve	method	database
2	thumbnail	identifier	graphics
3	photo	identifier	graphics
4	table	method	database
5	move	method	graphics
6	add	method	database
7	display	method	graphics
8	list	method	graphics
9	connect	class	database
10	get	method	database

Table 5.9 contains the precision and recall scores for program retrieval undertaken using the different retrieval mechanisms. In this table, precision and recall scores for the first mechanism is represented as `Unclassified`, the second by `IbK-3`, the third by `J48` while the fourth mechanism appears as `DA`. Based on data depicted in the table, we can see that by classifying programs into application domains prior to retrieval, we obtained a higher precision. The increase in precision can be seen when Queries 1, 5, 6, 7 and 10 were used in the experiment. This result can be explained by the selection of terms used in defining a query, that is, if the query contains a term that is common in both domains, then it is better to classify the programs prior to retrieval. This is because if information on the application domain of the

programs is not available, users are presented with programs containing the searched term but may not illustrate the required context. For example, the term add defined in Query 5 is a verb that can be seen as method names in both types of application domains, database and graphics. We may have code statements of adding records in a database program and statements of adding an image in a graphics program. Therefore by classifying programs into application domain prior to program retrieval would filter out programs that are from the irrelevant application domain.

Table 5.9 Precision and Recall Scores for Queries 1 to 10

	Precision				Recall			
Query	Unclassified	IbK-3	J48	DA	Unclassified	J48	IbK-3	DA
1	0.72	1	1	0.88	0.45	0.63	0.63	0.55
2	0.76	1	1	0.88	0.61	0.81	0.81	0.71
3	0.84	0.8	0.88	0.72	0.44	0.42	0.46	0.38
4	1	1	0.96	0.76	0.48	0.48	0.46	0.37
5	0.72	1	1	0.88	0.36	0.51	0.51	0.45
6	0.64	1	1	1	0.32	0.51	0.51	0.51
7	0.68	0.68	0.8	0.76	0.59	0.57	0.69	0.66
8	0.76	1	1	0.88	0.39	0.51	0.51	0.45
9	0.92	0.88	0.88	0.8	0.49	0.47	0.47	0.43
10	0.64	0.64	0.76	0.56	0.31	0.3	0.37	0.27

Based on the data depicted in Table 5.9, we present the average precision and recall scores in Table 5.10. Data in table 5.10 reveals that retrieval undertaken based on program classification performed using the IbK-3 model generates the highest precision and recall. This is followed by classification made using the J48 and DA models. On the other hand, retrieval performed without program classification generates the lowest average precision and recall scores.

Table 5.10 Average of Precision and Recall Scores for Four Retrieval Mechanisms

Measurements	Unclassified	IbK-3	J48	DA
Precision	0.7680	0.9280	0.900	0.8120
Recall	0.4440	0.5420	0.5210	0.4780

In order to identify whether there is statistically significant different in precision and recall scores between different retrieval mechanisms, we performed the Kruskal-Wallis test [103] since the scores were not normally distributed. Results of normality test is included in *Appendix C: Statistical Result - Program Retrieval using Classified Programs.* Based on the Kruskal-Wallis test performed using data depicted in Table 5.9, it is suggested that there is a significant different in precision scores across the four retrieval mechanisms. This is shown by the data (output from the test) presented in Table 5.11(b) which suggest that the significance level (`Asymp.Sig.`) for precision scores was 0.02 (rounded). This is less than the α=0.05, which is normally used in statistical test [103]. To investigate which of the retrieval mechanisms had the highest overall precision, the mean rank is taken into consideration. Based on data in Table 5.11(a), it is suggested that retrieval undertaken using classification made by IbK-3 had the highest precision, with the DA reposting the lowest. Details of the statistical test are also included in *Appendix C.*

On the other hand, data depicted in Table 5.11(b) suggests that at α=0.05, difference in recall scores across the four retrieval mechanisms is less significant. This is shown by the `Asymp.Sig` value for recall scores (i.e 0.250) which is greater than the α value used in the experiment.

Existing work suggest that the success of a classification method can be deter-

Table 5.11 Kruskal-Wallis Test Result Relative to Program Retrieval

(a) Mean Rank

	Classifier	Mean Rank
Precision	Unclassified	12.95
	J48	25.05
	IbK-3	26.90
	DA	17.10
Recall	Unclassified	15.45
	J48	23.40
	IbK-3	24.70
	DA	18.45

(b) Test Statistics

	Precision	Recall
Chi-Square	9.952	4.105
df	3	3
Asymp.Sig.	0.019	0.250

mined by the proportion of programs that are correctly classified out of all relevant programs [106, 113, 28]. Since the statistical results reveal that the difference in recall scores between DA, J48 and IbK-3 is less significant at $\alpha = 0.05$), we conclude that the capabilities of the classifiers in classifying programs into the relevant application domains are similar. Therefore, we decided to use classifications made by all three classifiers (J48, IbK-3 and DA) in our retrieval system. This means that, given a program, there will be three classification results; one from each classifier. However, we represent the classification results by indicating the portion of classifiers that classify the program into a particular domain. In our program retrieval system, information of the application domain of a program is represented as a tuple of two indices. The first index represents the database domain while the second corresponds to the graphics domain. If we have a program, P99, which has been classified into the database domain by two out three classifiers, than the information on the programs application domain would be $\{0.66, 0.33\}$. If all three classifiers have classified the program as a database program then P99 would be represented as $\{1, 0\}$.

5.4 Conclusion

We have demonstrated the use of software metrics in determining the application domain of a program. Such an approach is beneficial in automating program classification performed on applications that are not well-documented – definition for well-documented is provided in Definition 2 on page 4. In the experiments undertaken, the use of more metrics as the independent variables generates a better classification result. Since the programs used in the undertaken experiment illustrate similar functionalities (e.g add and delete), the classifiers require more metrics to differentiate between a database and graphics programs. In addition, we have also demonstrated that by classifying programs into application domains prior to program retrieval can improve the precision and recall scores.

CHAPTER 6

COMBINING FUNCTIONAL AND STRUCTURAL DESCRIPTORS

In this chapter, we demonstrate how to integrate data obtained from Chapters 3, 4, and 5, into a single index. This index known as a compound index is used to represent a search query and programs in a software repository. In the context of this thesis, a compound index is defined as follows:

Definition 9 *A compound index is an index containing several indices which includes functional and structural descriptors. The elements of this index are represented using continuous values (e.g 1, 4, 10, etc.) which indicate the importance of functional descriptors, existence of design patterns and software metrics contained in a program, and application domain of a program.*

Based on the work undertaken by Mili et al. [100] and literature discussed in section 2.1.3 on page 41, relevant components are retrieved by identifying components that minimize some measure of distance to a user query. Such an approach expects that the outcome will either be an *exact match* [100] or (failing an exact match) one or more *approximate matches* [100]. As many studies [58, 149, 5, 150] have demonstrated the effectiveness of using vector model in presenting users with *exact* and *approximate matches*, we investigate the use of this model in retrieving programs that are similar to a given query. In this work, the vector model evaluates the degree of similarity of a component P with regard to a query q using two calculations: Cosine Measure and Euclidean Distance. Relevant formula used as the similarity measurements can be seen in equations 2.1 and 2.2 described in section 2.1.3 on page 43. We also investigate the use of information on data distribution as the similarity measurement for a program retrieval system. The idea is to identify whether programs with similar data distribution illustrate similar function and structure. An elaboration on similarity measurement based on data distribution can be seen in section 2.1.3 on page 44.

6.1 Model of Program Retrieval using a Combination Approach

Figure 6.1 illustrates the model of program indexing and retrieval used in this work. There are three types of information extraction performed on a program: function, design patterns and software metrics.

The first extractor, namely `Descriptor` (as illustrated in Figure 6.1), creates a functional index file consisting of a program-term matrix (refer to Definition 10). This index file is updated whenever the repository receives new applications to be stored or when there is a request to withdraw a particular application.

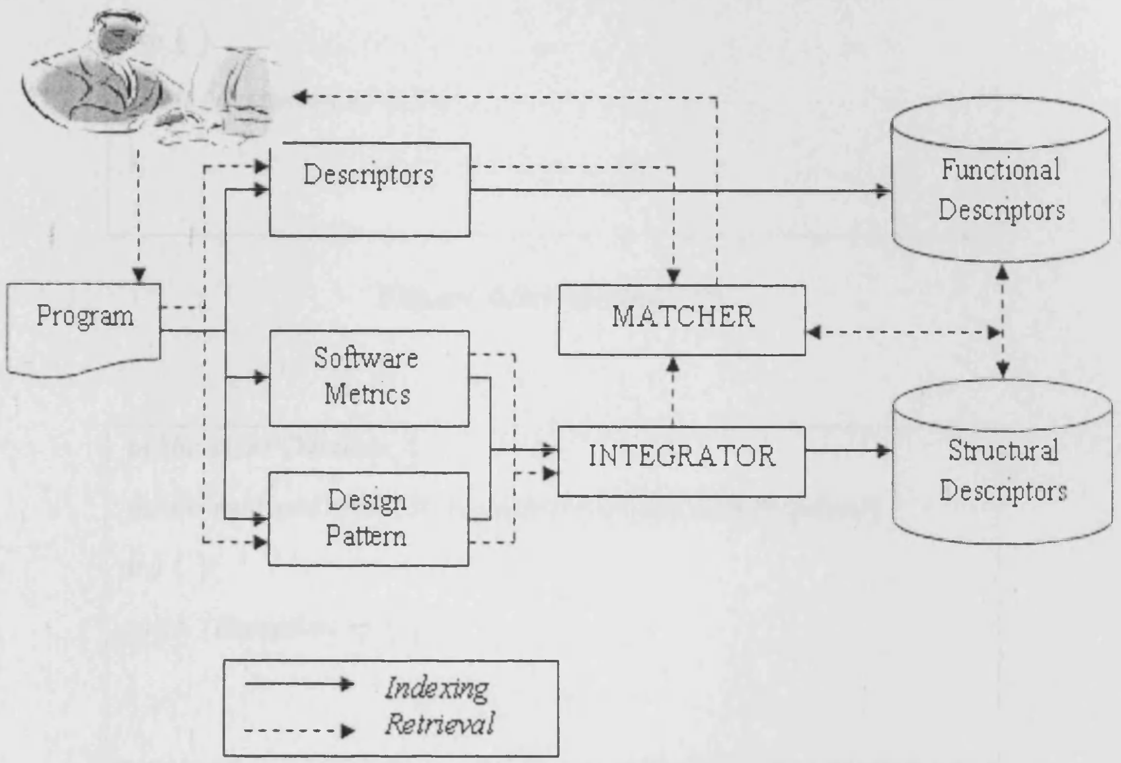


Figure 6.1 Model of Program Indexing and Retrieval using a Combination Approach

Definition 10 *A program-term matrix contains rows corresponding to the programs and columns corresponding to the weighted terms (refer discussion on page 61). For instance if we have two (simple) programs, P1 (Figure 6.2) and P2 (Figure 6.3):*

```
public class Database {
public void connect(String connectString) throws DBException {
try { }
catch (Exception e) { } }
}
```

Figure 6.2 Database - P1

```
public class Database {
public void setDriver(String driver) throws DBException {
try { }
catch (Exception e) { } }
}
```

Figure 6.3 Database - P2

then a program-term matrix would be as follows:

The second extractor that is illustrated in Figure 6.1 is the `Design Pattern`, and it is used to identify the existence of three design patterns in a program (as elaborated upon in Chapter 4). The `Software Metrics` extractor, extracts twelve software metrics (as demonstrated in Chapter 5) from a program and uses these

Table 6.1 Program-term Matrix

	database	connect	connectString	setDriver	driver
P1	3	2	2	0	0
P2	3	0	0	2	2

metrics to classify a program into the appropriate application domain (i.e database or graphics). Information on the program's application domain and seven software metrics that represent program reusability (as discussed in section 2.1.2 on page 33) are submitted to an `Integrator`. The `Integrator` will then combine this information with that obtained from the `Design Pattern` and generate the following:

1. confidence level of the existence of design patterns - (p_1, p_2, p_3). The higher the value of p_x in a program, and in this case the value of x is 1 to 3, the more the program is believed (through the existence of structural relationships) to employ a particular design pattern.

2. confidence of classification of a program into appropriate application domain - (d_1, d_2). The greater the value of d_1 or d_2 , the more a program is believed (based on classification made by the classifiers) to be in a particular application domain.

3. software metrics used in determining program complexity and reusability - $(m_1, m_2, m_3, m_4, m_5, m_6, m_7)$. The fewer the value of m_x in a program, the more reusable the program is.

Variable p_x comprises data on the existence of three design patterns, namely Singleton(p_1), Composite(p_2) and Observer(p_3). The second variable, d_x, represents information generated by application domain classifiers (i.e J48, IbK-3 and DA) that currently classify a given program into either database (d_1) or graphics (d_2). Finally, seven software metrics, m_x where x is the value of 1 to 7, are used to represent the complexity and reusability of a program. Details on these metrics have been discussed in section 2.1.2 on page 33. All of this information is mapped into a program-structural matrix which is later stored as a structural index file. In the context of this thesis, such a matrix is defined as Definition 11. Similar to the functional index file, a structural index file is only created once and is updated whenever the repository receives new applications to be stored or when there is a request to withdraw any particular application.

Definition 11 *A program-structural matrix contains rows corresponding to the programs and columns corresponding to the structural descriptors. An example of such a matrix is as follows:*

Table 6.2 Program-structural Matrix

	p_1	p_2	p_3	d_1	d_2	m_1	m_2	m_3	m_4	m_5	m_6	m_7
P3	2	0	3	1	0	0	8	7	0	0	7	1
P4	0	2	0	0	1	5	2	5	1	1	2	1

A program that is submitted to the retrieval system as a search query will undergo the same process: the `Descriptor` in Figure 6.1 extracts relevant terms from the program and generates a program-term matrix for the query program. The `Integrator` integrates information on design patterns, application domain and software metrics, which were identified by the `Design Pattern` and `Software Metrics`, of the program. This information is then sent to the `Matcher` which combines them into a compound index. This index is later known as a `Query Compound Index` (Qci).

Definition 12 *A query compound index is a compound index (refer to Definition 9) that is created by analyzing a program submitted by a user to be used as the search query.*

For example: given a program, Q as the query, the `Integrator` produced the following compound index $Qci = 3, 2, 2, 2, 0, 3, 1, 0, 0, 8, 7, 0, 0, 7, 1$. This index is built upon the following:

Weighted terms (w): **{3, 2, 2 }** These values represent the three terms (e.g database, connect, connectString) extracted from program Q.

Design pattern (p): **{2, 0, 3 }** The three values in the tuple represent the existence of three design patterns in a program. For example, value 2 illustrates that two of Singleton's rules have been fulfilled while the value 3 indicates that there are three structural relationships in the program that illustrate existence of Observer design pattern.

Topic Classification (d): **{ 1, 0 }** The values in this tuple depict how many of the topic classifiers (elaborated upon in 5) have classified the program into a particular application domain. The first indice in this tuple represents the database domain while the second indice represents the graphics domain. Value 1 as indice number four of the compound index indicates that all three classifiers have categorized the query as being in the database domain. If the value is 0.33, then it indicates that only one out of three (i.e 1/3) classifiers have identified the program as a database program.

Program Reusability (m): **{ 0, 8, 7, 0, 0, 7, 1 }** The first two indices represent the complexity of a program while the rest of the values are used in determining the reusability of a given program. The first value represents Wmc and this is followed by Mvg. The third metrics is the Cbo and is followed by Fivis, Ficon, Fovis and Focon. Details of these metrics can be seen in section 2.1.2 on 33.

Upon creating a Qci to represent a search query, the **Matcher** (illustrated in Figure 6.1) creates another program-term matrix. The matrix contains rows corresponding to the programs in the repository and columns corresponding to the weighted terms (e.g database, connect, connectString) found in the Qci. This matrix only includes programs that contain term(s) as defined in the query program. For each of the programs in the matrix, the **Matcher** combines the weighted terms, based on data in the functional index file (i.e generated by the **Descriptor**), with structural descriptors defined in the structural index file, into a Program Compound Index (Pci). Similarity between a Qci and each of the Pci, (Pci_n), is determined in order to present a user with relevant programs.

6.1.1 Requirements of the Combination Approach

In order to realize the proposed source code retrieval system (i.e combination approach), we are assuming that the search query is presented in the format of a program. This means that a user may use his/her existing program, which is currently developed for a programming task, as the search query. This partially completed program (query program) acts as a template since the user's search requirements (function and structure) are depicted in the program. Such an approach is beneficial to developers who have identified the required objects and/or how they interact for a given programming task. The retrieval system will then identify programs (from the repository) containing similar function and structure and include them in the retrieval hit list.

As illustrated in Figure 6.1, in order to identify functional and structural descriptors depicted in a program or a query program, the retrieval system extracts three types of data: terms, design patterns and software metrics. The first two data can be obtained using parsers that extract terms based on program structure (as defined in Definition 5 on page 52) and identify relationships that exist between properties of the program (e.g method invocations and inheritance). On the other hand, software metrics that are used to determine the program's application domain and its reusability are identified using the C and C++ Code Counter (CCCC) [142].

The identification of functional and structural descriptors as elaborated in Chapters 3, 4 and 5 can be extended to cater larger number of programs which may contain various design patterns and originate from several applications domains. Our approach of using weighted terms as the functional descriptors of a program can easily be integrated with existing methods of identifying semantic meanings. For example, the mechanism can be integrated with WordNet [61] to obtain other the synonyms of the extracted terms. This would increase the performance of the system as similarity measurement is not restricted to string matching. Furthermore, if a developer intends to have a domain-based repository, that is a collection of programs for a specific domain (e.g medical, finance), then the functional descriptors can be identified based on a relevant ontology [151, 7].

Currently, the work described in this thesis focuses on identifying three design patterns; Singleton, Composite and Observer. As the identification is made based on structural relationships, it can later be extended to include other design patterns. For example, in a Decorator pattern, we learn that there are additional responsibilities attached dynamically to the Component object. Such a relationships can be used as an indicator of the existence of Decorator pattern in a Java program. Furthermore, our source code retrieval model can also be integrated with other design pattern detection tools (e.g *PINOT* [77]). Information on design patterns, contained in a program, that have been detected by the tool can later be incorporated into the compound index representing the program.

Based on the work discussed in Chapter 5, if programs from other domains (besides database and graphics) are to be included in the repository, it is believed that KNN would also able to generate an acceptable result (classification accuracy). This assumption is made based on existing work [141] which has employed KNN to classify objects from a large number of categories. The average classification accuracy obtained in the experiments performed using objects from 267 categories is approximately 71% [141]. As most of the existing open-source repositories [29, 30] categorize their applications into less than 100 domains, the use of KNN in this context would generate promising result. Furthermore, if programs from different domains are identified to illustrate different metric trends, than the use of KNN that groups together objects having similar data is applicable.

6.2 Experiments

As discussed in Chapters 3, 4 and 5, we proposed to retrieve relevant programs using functional and structural descriptors of a program: weighted terms, design patterns and software metrics. In this section, similarity measurement between compound indexes is performed using calculations undertaken based on vector model and data distribution which have been explained in section 2.1.3 on page 41. All of the experiments were performed on a repository that consists of 584 Java programs. Experiments of program retrieval using distinct similarity measurements were conducted based on:

- a set of queries
- a single query

The first type of experiment requires a set of queries to be submitted to the retrieval system. A total of ten programs were used as the search queries in the experiment. These programs contain different amount of functional descriptors and different software metric values. In addition, the query programs also illustrate different design patterns confidence levels of the existence of design patterns. Half of the programs are from a database domain and the other half represent the graphics domain.

To investigate whether one of the retrieval techniques (i.e program retrieval undertaken using different similarity measurements) outperforms the other for the set of queries, an analysis of individual query is included in the experiment. The relevance of programs retrieved by the system and a list of programs that are relevant to a given query (i.e answer set) were determined by the researcher, (i.e identifying programs that the researcher thought to be relevant to a given query). The answer set of each of the query used in the experiment is identified by determining similarity in terms of (1) function, (2) design patterns and (3) application domain. As discussed in section 3.2 on page 53, identifier names contained in a program represent the functionality of the program. If a program contains at least half of the functional descriptors identified in a query program, and the program also employs a similar design pattern(s) as in the query program, than the program is considered to be relevant to the query. In addition, the program should also comes from the same application domain as the query program.

6.2.1 Evaluation of Program Retrieval

In the experiments undertaken, performance of the retrieval system using different similarity measurements was measured using the precision (refer to Definition 7) and recall (refer to Definition 8) scores. The purpose of computing the recall and precision scores of the program retrieval system is to compare the three similarity measurements to find the best one that can be used as the default similarity measurement in order to automate program retrieval. The scores should not be taken as an absolutely objective measurement of the effectiveness of similarity measurements (skewness, ED and cosine measure) in a retrieval system because the scores were calculated based on program retrieval performed on the existing repository.

The recall and precision are measures for the entire hit list (a hit list is defined as Definition 6 on page 66). They do not account for the quality of ranking the programs in the hit list. Developers want the retrieved programs to be ranked according to their relevance to the query instead of just being returned as a set. The most relevant programs must be in the top few programs returned for a query. Relevance ranking can be measured by computing precision at different cut-off points (i.e *precision at n*) [13]. Therefore, in the experiments undertaken, the results of program retrieval are based on precision at a fixed document cut-off value (DCV).

Definition 13 *A retrieval system employing a DCV examines only a fixed number of programs (e.g n=10) for a given query and uses this information to compute a precision and/or recall scores.*

For example, if the top 10 programs are all relevant to the query and the next ten are all non relevant, we have 100% precision at a cut off of 10 documents but a 50% precision at a cut off of 20 documents. Relevance ranking in this hit list is good since all relevant programs are above all the non relevant ones.

Table 6.3 Precision and Recall Scores for the Top 10 Programs

Query	# Relevant	Skewness		Euclidean Distance		Cosine Measure	
	Programs	Precision	Recall	Precision	Recall	Precision	Recall
Q1	67	0.3	0.045	0.6	0.090	0.6	0.090
Q2	194	0.2	0.010	0.4	0.031	0.5	0.026
Q3	15	0.2	0.133	0.3	0.2	0.3	0.2
Q4	20	0.3	0.150	0.4	0.2	0.4	0.2
Q5	28	0.4	0.143	0.6	0.214	0.5	0.179
Q6	45	0.5	0.111	0.7	0.156	0.7	0.156
Q7	105	0.5	0.048	0.8	0.076	0.8	0.076
Q8	88	0.4	0.045	0.6	0.068	0.6	0.068
Q9	275	0.5	0.018	0.7	0.025	0.7	0.025
Q10	144	0.6	0.042	0.8	0.056	0.7	0.049

6.2.2 Analysis of Program Retrieval Undertaken using Different Similarity Measurements

The precision and recall values obtained from the experiments undertaken are depicted in Table 6.3. The precision and recall were obtained using DCV=10, and the calculation were based on number of relevant programs which were determined by the researcher.

From the data depicted in Table 6.3, it can be seen that 9 out of 10 queries undertaken using ED as the similarity measurement generated a precision greater or at least similar to using the cosine measure. The highest precision (i.e 0.8) for program retrieval was obtained when the ED and cosine measure were used as the similarity measurements for Q7. In addition, by using ED as the similarity measurement, we

Table 6.4 Precision and Recall Scores for the Top 20 Programs

Query	Answer Set	Skewness		Euclidean Distance		Cosine Measure	
		Precision	Recall	Precision	Recall	Precision	Recall
Q1	67	0.35	0.104	0.75	0.224	0.7	0.209
Q2	194	0.4	0.041	0.5	0.052	0.6	0.062
Q3	15	0.25	0.333	0.35	0.467	0.4	0.533
Q4	20	0.25	0.250	0.6	0.6	0.6	0.6
Q5	28	0.4	0.286	0.75	0.536	0.75	0.536
Q6	45	0.6	0.267	0.8	0.356	0.7	0.311
Q7	105	0.65	0.124	0.9	0.171	0.85	0.162
Q8	88	0.5	0.114	0.8	0.182	0.75	0.170
Q9	275	0.65	0.047	0.8	0.058	0.8	0.058
Q10	144	0.7	0.097	0.85	0.118	0.8	0.111

also obtained the highest recall (i.e 0.214) in the experiment, and this is revealed by the recall scores for Q5. The low scores in recall can be accounted for the small value of DCV used in the experiment. If the DCV is smaller than the number of relevant programs, it is difficult to obtain a recall score of one. For example, if only five programs are examined and 50 relevant programs exist for a given query, then the recall is only 0.1 (10%) even if all the programs examined are relevant. Hence, this makes the search methods (i.e similarity measurements) appear much worse than they actually are.

In order to determine if the precision and recall scores increase if a larger number of retrieved programs is analyzed, the researcher included a retrieval analysis for the top 20 programs (i.e DCV=20). The precision and recall scores for DCV=20 are shown in Table 6.4.

The average scores of precision and recall obtained in the analysis (i.e DCV=20), together with the average scores for data depicted in Table 6.3 (i.e DCV=10) are depicted in Table 6.5. The average precision obtained using ED has increased 20.3% when DCV=20 was employed, while the retrieval undertaken using cosine measure generated an improvement of 19.8% in its precision scores. Program retrieval performed using the skewness as similarity measurement also shown an improvement. Data in Table 6.5 also reveals that the average recall scores have increased more than double when larger cut-off value is employed. The average recall when skewness, ED and cosine measure were employed as the similarity measurements has increased 123.2%, 147.6% and 157.8% individually.

Table 6.5 Average of Precision and Recall Scores for the Top 10 and 20 Programs

	Skewness		Euclidean Distance		Cosine Measure	
	Precision	Recall	Precision	Recall	Precision	Recall
DCV=10	0.390	0.075	0.590	0.112	0.580	0.107
DCV=20	0.475	0.166	0.710	0.276	0.695	0.275

Data in Table 6.5 reveals that there is a small difference in the average precision and recall scores between ED and the cosine measure. For DCV=20, the difference in precision is 0.015 and for DCV=10, the difference is 0.01. As for the average recall scores, the difference is 0.001 when DCV=20 and 0.005 when DCV=10. Even though the difference is small, based on the experiment undertaken, the use of ED in measuring similarity between a query and programs in the repository is shown to be better than using skewness and cosine measure. Based on data depicted in Table 6.5, the researcher concludes that it is better to use ED as the similarity measurement in order to automate the proposed program retrieval system. This is because by using

ED, we obtain a better precision and recall for the top 10 and 20 programs in the hit list.

The decision of choosing ED rather than cosine measure or skewness as the similarity measurement in the program retrieval system is also supported by the statistical analysis. We need to verify the decision with statistical test since we may obtain the same mean scores for the three retrieval techniques (i.e program retrieval performed using different similarity measurements) if different query programs were used in the experiment. If the mean scores were to be the same, we could not determine which classifier generates a better result than the other. For example, if we have technique A with the following data {0.7, 0.1, 0.1, 0.1} and technique B with {0.5, 0.3, 0.1, 0.1}, the mean scores (average) for these techniques are the same, that is 0.25. With such values, we could not determine which retrieval techniques generates a better result. Nevertheless, this can done by performing statistical test which includes measurements performed on ranked data – measurement observations are converted to their ranks in the overall data set: the smallest value gets a rank of 1, the next smallest gets a rank of 2, and so on with tied ranks included where appropriate. Such an approach would represent method A as {4, 1.2, 1.2, 1.2) and B as {3, 2, 1.2, 1.2}. The mean ranks for technique A and B would be 1.9 and 1.85 respectively. By using mean ranks as the point of comparison, then the decision of which is a better similarity measurements can be made.

As suggested by Hull [148], there is a need of a statistical methodology to determine whether measured differences between the retrieval methods can be considered statistically significant. Prior to a statistical test, distribution of the data needs to be identified in order to determine which significance test is most suitable for a given set of precision and recall scores. This is done by performing a normality test on the measured data. The normality test result as shown in Table 6.6 is obtained based on the preçision and recall scores for DCV=20. The `Sig.` values 0.274, 0.082 and 0.189, and 0.146, 0.239 and 0.069, from the Shapiro-Wilk test [103] of normality are greater than the α avlue used in this test, which is 0.05. Such results imply that it is acceptable to assume that the precision and recall distributions for the skewness, ED and cosine measure are normal, hence suggesting parametric test to be employed in determining significant different in the scores across the similarity measurements. In Table 6.6, the skewness, ED and cosine measure are represented as Class 1, 2 and 3.

Table 6.6 Test of Normality for Precision and Recall Scores for the Top 20 Programs

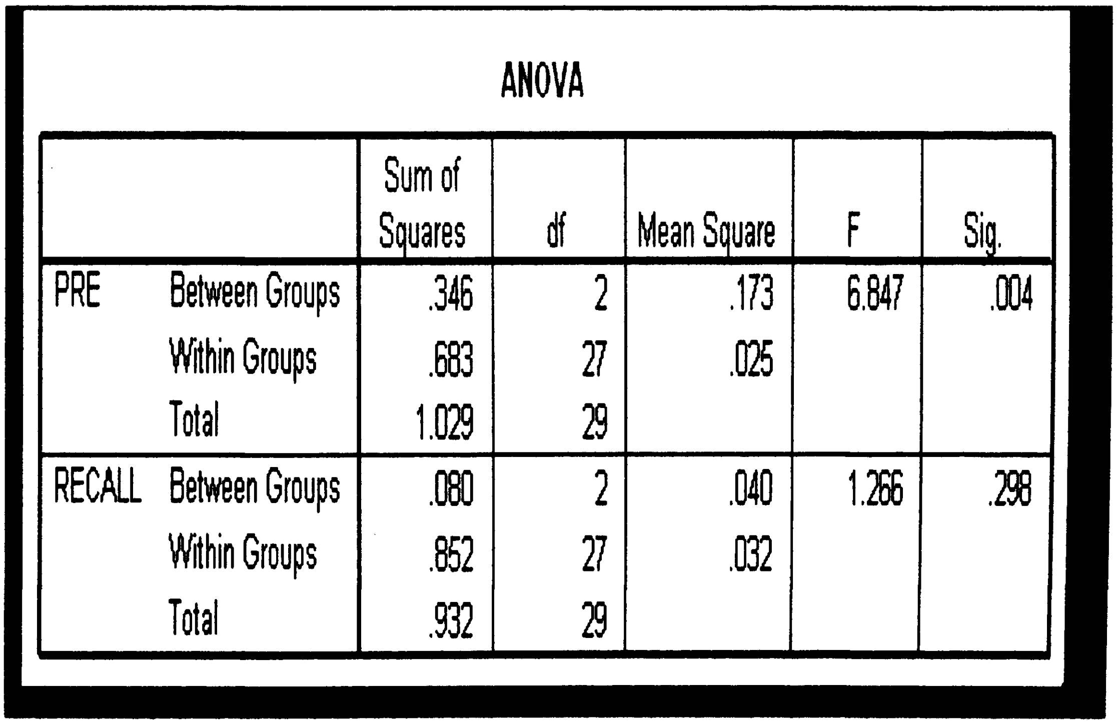
		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	CLASS	Statistic	df	Sig.	Statistic	df	Sig.
PRE	1	.172	10	.200*	.909	10	.274
	2	.291	10	.016	.863	10	.082
	3	.215	10	.200*	.894	10	.189
RECALL	1	.254	10	.066	.884	10	.146
	2	.203	10	.200*	.903	10	.239
	3	.225	10	.164	.856	10	.069

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

The researcher then performed the ANOVA test [103] to detect significant difference in the retrieval scores (i.e normally distributed) across multiple similarity measurements. In implementing the test, our null hypothesis, H_0, was that all the similarity measurements being tested were equivalent in terms of precision and recall. If the `p-value` obtained in the test is less then the identified significance level, α, we could conclude that the similarity measurement were significantly different. In this test, the α value is set to be 0.05 which is an acceptable value in any statistical test [103]. Data depicted in Table 6.7 reveals that there was a statistically significant difference in precision scores across the three similarity measurements as the `p-value` which is represented as `Sig.` was 0.004. However, data in Table 6.7 does not reveal which similarity measurement is different from which other similarity measurement. The statistical significance of the differences between each pair of similarity measurements is identified through a Post-Hoc test [103], the result of which is provided in Table 6.8.

Table 6.7 ANOVA Test Result of Precision and Recall Scores

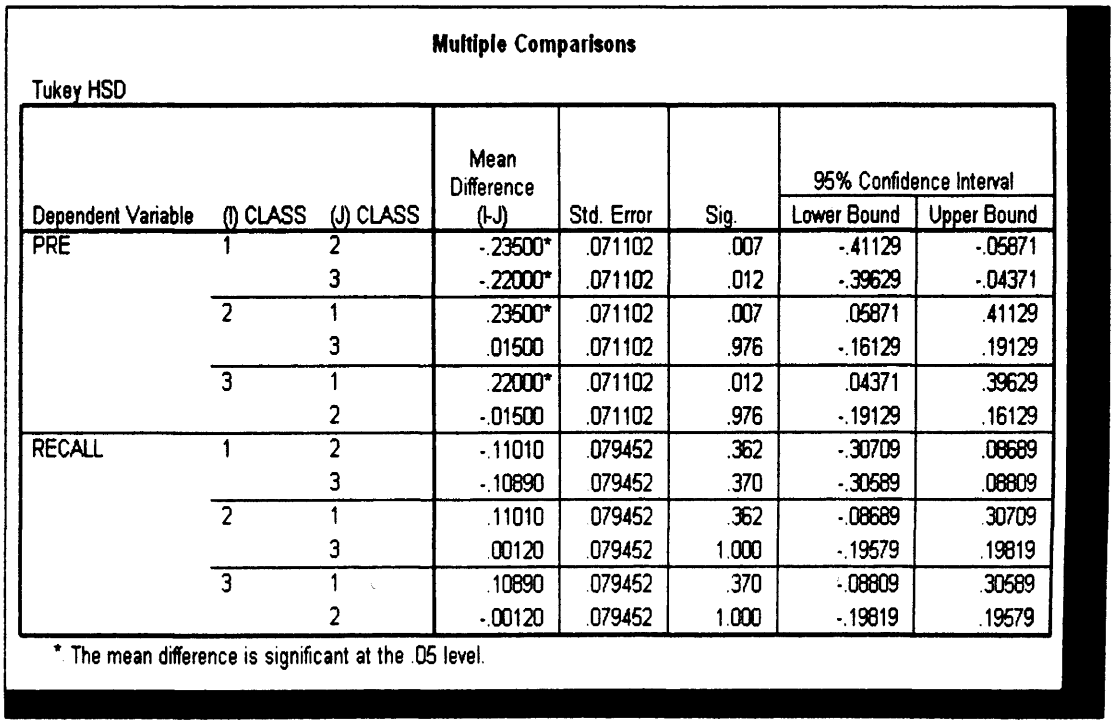


ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
PRE	Between Groups	.346	2	.173	6.847	.004
	Within Groups	.683	27	.025		
	Total	1.029	29			
RECALL	Between Groups	.080	2	.040	1.266	.298
	Within Groups	.852	27	.032		
	Total	.932	29			

The symbol asterisks (∗) next to the values depicted in the column `Mean Difference` in Table 6.8, indicates that the three similarity measurements being compared are significantly different from one another at the p<0.05. The exact significance value is given in the column labelled `Sig`. In this table, the similarity measurements are represented as Class 1 for skewness, 2 for ED and 3 for cosine measure. For the precision scores, only class 1 and class 2, and class 1 and class 3 are identified as statistically significant different from one another. This means that there is a significant difference in the scores between skewness and ED and between skewness and cosine measure. Nevertheless, the difference in precision and recall scores between ED and cosine measure is less significant at $\alpha = 0.05$.

Table 6.8 Post-Hoc Test for Multiple Comparisons



Multiple Comparisons

Tukey HSD

Dependent Variable	(I) CLASS	(J) CLASS	Mean Difference (I-J)	Std. Error	Sig.	95% Confidence Interval	
						Lower Bound	Upper Bound
PRE	1	2	-.23500*	.071102	.007	-.41129	-.05871
		3	-.22000*	.071102	.012	-.39629	-.04371
	2	1	.23500*	.071102	.007	.05871	.41129
		3	.01500	.071102	.976	-.16129	.19129
	3	1	.22000*	.071102	.012	.04371	.39629
		2	-.01500	.071102	.976	-.19129	.16129
RECALL	1	2	-.11010	.079452	.362	-.30709	.08689
		3	-.10890	.079452	.370	-.30589	.08809
	2	1	.11010	.079452	.362	-.08689	.30709
		3	.00120	.079452	1.000	-.19579	.19819
	3	1	.10890	.079452	.370	-.08809	.30589
		2	-.00120	.079452	1.000	-.19819	.19579

* The mean difference is significant at the .05 level.

On the other hand, data in Tables 6.7 and 6.8 reveal that difference in recall scores across the three similarity measurements is less significant at $\alpha = 0.05$. As we are assuming that the users of a program retrieval system may only examine the top n programs presented in the hit list, significant different in recall scores between ED, skewness and cosine measure is less useful compared to precision scores in selection of similarity measurement.

6.3 Conclusion

Based on the results illustrated in Tables 6.3 and 6.5, it is suggested that it is better to use Euclidean distance (ED) compared to the cosine measure and skewness in identifying similarities between a query program and programs in the repository. In addition, the statistical analysis result presented in Table 6.7 reveals that there is a significant different in precision scores between the similarity measurements. The result is somehow unexpected because it was assumed that similarity measurement using the cosine measure would generate better retrieval results. This is because many studies [58, 152, 149] have reported promising results when the cosine measures was employed in identifying relevant objects (e.g text documents and software components). On the other hand, our statistical analysis (refer to Table 6.8) supports the work undertaken by Qian et al. [153] that reported the cosine measure works no worse than ED in a retrieval system when recall is taken into consideration. Nevertheless, in order to automate the program retrieval system, the ED has been chosen as the similarity measurement. This is based on the fact that it generated the highest precision and recall scores in the experiments undertaken using 10 and 20 as the document cut-point value. In addition, the `Sig` value for precision and recall scores between ED and skewness is smaller than the `Sig` value between cosine measure and skewness. Such a result indicates that ED is better than cosine measure in classifying programs into application domains.

CHAPTER 7

EVALUATION AND DISCUSSION

This chapter presents the results of two types of evaluation conducted on the program retrieval system. The first evaluation analyses the results obtained upon submitting two types of search query, which involves database and graphics application domains. The second evaluation studies empirically how well our system supports program retrieval through field experiments with developers. The purpose of the empirical studies of our retrieval system was not to analyze the quality of programs used as search queries, but to analyze whether a combination of functional and structural descriptors could increase the retrieval effectiveness, compared to using functional descriptors on their own. The empirical studies attempted to answer the following questions:

- Are developers able to reuse unknown programs with the support of our retrieval system?
- Is the compound index (refer to Definition 9 on page 123) that is built upon a combination of functional and structural descriptors capable of retrieving programs relevant to the query for source code?
- Does a developer's programming experience contribute to the identification of

relevant programs in a hit list that has been generated by the retrieval system upon receiving a query? (hit list is described as Definition 6 on page 66.) For example, *is there any relationship between programming experience and precision of retrieval?*

7.1 Comparison with Other Tools

Our retrieval approach concerns the use of functional and structural descriptors (refer to Definitions 3 and 4 on page 7) in representing a program. It is similar to other work that is related to source code retrieval such as Google code search [23], Koders search engine [24] and *SCRUPLE* [25]. However, they [23, 24, 25] focus on using functional and structural descriptors on their own, hence programs presented to the users only contain either the required function or structure. In addition, comparing our approach to [23, 24, 25] which involves searching for code samples that match a specified regular expression [24], our approach identifies information that may not be explicitly available in a program (e.g design patterns and software metrics).

We consider the work undertaken in this research similar to the Koders search engine [24], as the same mechanism is used to identify functional descriptors. Identifier names extracted from a program are used as functional descriptors of the program and two terms are considered to be similar if their syntax matches (as elaborated upon in section 3.4 on page 64) and are from the same context in a program (e.g class name, method name, etc). Below are examples of search queries handled in the Koders search engine which can also be used for retrieving programs using our retrieval system:

- Search for classes whose name contains <*search term*>.
- Search for methods whose name contains <*search term*>.

- Search for interfaces whose name contains <*search term*>.

- Search for files whose name contains <*filename*>.

- Search for classes named <*Default*> with a method named <*PageInit*>.

In addition to the above search queries, our approach includes the use of structural descriptors (refer to Definition 4 on 7) in identifying similarity between a query program and programs in a repository. The importance of including such information has been elaborated upon in section 1.2.1 on page 8, and is further supported by the undertaken experiments which are elaborated upon in section 7.2 and 7.3. In order to determine whether the combination of structural and functional descriptors increase the effectiveness of program retrieval, precision and retrieval evaluation is performed based on two hit lists. The first list contains programs that have been retrieved based on functional descriptors (functional approach), while the second list contains programs retrieved based on functional and structural descriptors (combination approach). The retrieval mechanism which is similar to the one used in Koders search engine [24] is employed as the functional approach, and our approach of combining functional and structural descriptors represents the combination approach. The experiments undertaken were used to investigate if program retrieval is better undertaken using the combination approach rather than the functional approach.

7.2 Objective Evaluations of the Retrieval System

This section describes the analysis performed by the retrieval system based on a search query from the user. Our experimentation platform consists of a PC (Pentium 4, 1.00GB memory) running under a Windows XP Professional environment (version 2002) and during the experiment there were no other tasks running on the PC. Based on the elaboration made in section 6.1 on page 124, the combination approach uses a compound index to combine the functional and structural descriptors into a single index. It took a total of 6 hours and 43 minutes to build the functional and structural index files (refer to section 6.1 for elaboration on contents of the files). All of the experiments were performed on a repository of 90.6MB (size on disk), that consists of 9320 Java programs obtained from `Sourceforge.net` [29] – 4670 programs were from the database category and 4650 programs were from the graphics category.

While measuring recall (defined in Definition 8 on page 88), requires assessing the entire repository to determine a set of relevant programs when given a query, precision which is defined in Definition 7 on page 88, requires assessment of a fixed number of programs only. Since our software repository was not pre-assessed, that is, we have not determined programs that are relevant to a given query, we decided to focus on precision scores rather than recall. Result analysis (i.e precision) is undertaken based on a document cut-off value (DCV) –– DCV is described in Definition 13 on page 134. Based on the given definition, only the first n programs that were presented in the hit list were analyzed. Such an approach is employed since a user of a software repository might be willing to examine only a fixed number of programs presented in the hit list [148]. Similar to existing work [154, 155, 156], a DCV=10 is employed in the retrieval analysis.

In Table 7.1, we present the precision (refer to Definition 7 on page 88) scores that are calculated based on the top 10 programs presented in the retrieval hit list. In addition, data in Figure 7.1 also reveals the processing times for the two queries submitted to the retrieval system during objective evaluation. In the context of this chapter, processing time is the duration (measured in milliseconds (ms)) of the retrieval system to generate a hit list upon receiving a source code query.

Referring to the data depicted in Table 7.1, there are two precision scores for

Table 7.1 Precision Scores and Processing Time

	Functional approach		*Combination approach*	
Query	Precision	Time(ms)	Precision	Time(ms)
MySQLDatabase.java	0.1	835	0.4	1141
PickPhotosPanel.java	0.2	880	0.3	1188

each query. One is obtained from retrieval undertaken based on functional descriptors (functional approach), and the other is based on program retrieval performed using a combination of functional and structural descriptors (combination approach). Elaboration on the results is presented in sections 7.2.1 and 7.2.2.

7.2.1 Retrieval from Database Application Domain

The first query submitted to the system requires programs that illustrate instances of the Observer design pattern. As for the functional criteria, the query emphasizes how to connect to a SQL database. The required programs must also determine whether or not the user has the appropriate JDBC driver loader, and whether the database connection could be authorized. A program known as `MySQLDatabase.java` is used as the search query (included in *Appendix D: MySQLDatabase.java*).

Data in Table 7.1 shows that there is an improvement in precision when the retrieval system employed the combination approach. The precision has doubled, that is from 0.1 to 0.4. Even though it took a longer time to generate a hit list when the combination approach was employed, the extra time was caused by the need of identifying structural descriptors in the query, and creating the appropriate compound indices (i.e Qci and Pci) as elaborated upon in section 6.1 on page 124.

Using functional descriptors as the only representation of a program, retrieval for a query that contains functional and structural requirements is rather difficult. This can be accounted for the lack of code statements and/or comment statements in a program that explicitly depict the existence of a particular design pattern (e.g variable named Observer or comment statement such as *This class implements the Observer pattern*). In addition, based on existing work on design pattern detection [132, 78, 76], textual analysis on code and comment statements is not applicable as software developers only describe the function of a program or method in these statements and not the design patterns used in the program. Nevertheless, design patterns can be identified by analyzing the structural relationships (e.g method dependencies and class hierarchies) that exist in a program [72]. As our retrieval system includes information on the existence of design patterns in a program (refer to section 4.2 on page 4.2), along with the functional descriptors of the program, retrieval for a query that contains functional and structural requirements generated a better precision compared to the functional approach.

Given `MySQLDatabase.java` as the search query, four out of ten programs in the hit list generated using the combination approach depict the required function and structure. In Figure 7.1, the programs represented as P2, P4, P5 and P7 were identified as being the most relevant programs that implement the Observer design pattern in creating a JDBC database connection. The graph in this figure is plotted based on metric values of the programs and details of the metrics can be found in section 2.1.2 on page 33.

In Table 7.2, we classified all of the ten programs presented in the hit list (combination approach) into three categories. The first category, "`Good`", contained programs that fulfilled both types of requirements; structural and functional. The second category, "`Relevant`", contained programs that fulfilled either the functional or structural requirements while the "`Bad`"contained programs that did not match the search

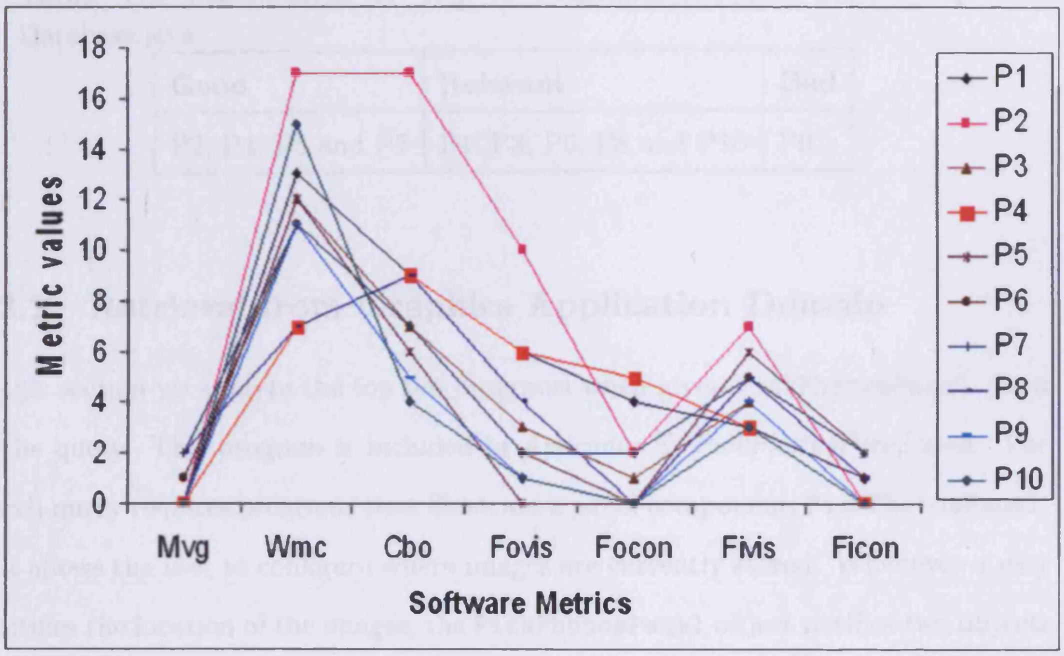


Figure 7.1 Software Metrics in Programs Retrieved using the Combination Approach for MySQLDatabase.java

requirements at all. Program P9 was identified as irrelevant as it led an error in reading data from a database, which did not relate to the required function. Furthermore, based on the researchers' analysis, there was no design pattern employed in the program.

Table 7.2 Classification of Top 10 Programs Retrieved for MySQLDatabase.java

Good	Relevant	Bad
P2, P4, P5 and P7	P1, P3, P6, P8 and P10	P9

7.2.2 Retrieval from Graphics Application Domain

In this section we analyze the top ten programs when given `PickPhotosPanel.java` as the query. This program is included in *Appendix E: PickPhotosPanel.java.* The search query requires programs that illustrate a panel component, `PickPhotosPanel`, that allows the user to configure where images are currently stored. Whenever a user identifies the location of the images, the `PickPhotosPanel` object notifies two objects (`publishManager` and `photoSource`) that use information of the location. Therefore, the Java class to be developed should define an implementation of a one-to-many dependency relationship, that is between `PickPhotosPanel` and `publishManager`, and with `photoSource`, so that when the panel component changes state, the other objects that rely on the panel component are notified and updated automatically.

Similar to the result obtained from the first query (`MySQLDatabase.java`), data in Table 7.1 also reveals that precision increased when the system employed the combination approach. It took 1188 milliseconds for a hit list to be generated using the combination approach, while only 880 milliseconds when the functional approach was employed. Nevertheless, the precision was improved by 50% when program retrieval was undertaken using the combination approach. Three out of ten programs in the combination approach hit list depict similar function and structure as identified in the search query. Similar to the classification of programs as undertaken in the previous section (section 7.2.1), programs retrieved for `PickPhotosPanel.java` are also classified into three categories: "`Good`", "`Relevant`"and "`Bad`".

Table 7.3 Classification of Top 10 Programs Retrieved for PickPhotosPanel.java

Good	**Relevant**	**Bad**
P1, P3 and P5	P2, P4, P6, P7 and P10	P8 and P9

Table 7.3 reveals that `P1`, `P3` and `P5` are identified to be relevant to the query, both in terms of function and structure. Program P6 and P10 illustrate the required function while P2, P4 and P7 illustrate the required structure. On the other hand, program P8 and P9 are not relevant to graphics applications as they both illustrate a text reporting function which does not relate to organization of images. Structural descriptors (i.e software metrics) of the programs presented in the hit list are shown in Figure 7.2.

7.2.3 Analysis of Software Metrics in Programs Retrieved in the Experiment

Using the metric values in programs depicted in Figures 7.1 and 7.2, we illustrate the average values of software metrics contained in the programs in Figures 7.3 and 7.4. Elaboration on the metrics that were used to plot the graphs can be found in section 2.1.2 on page 33. Comparing the average values of software metrics in programs retrieved using different approaches (combination approach and functional approach), we learned that by including structural descriptors (i.e in the combination

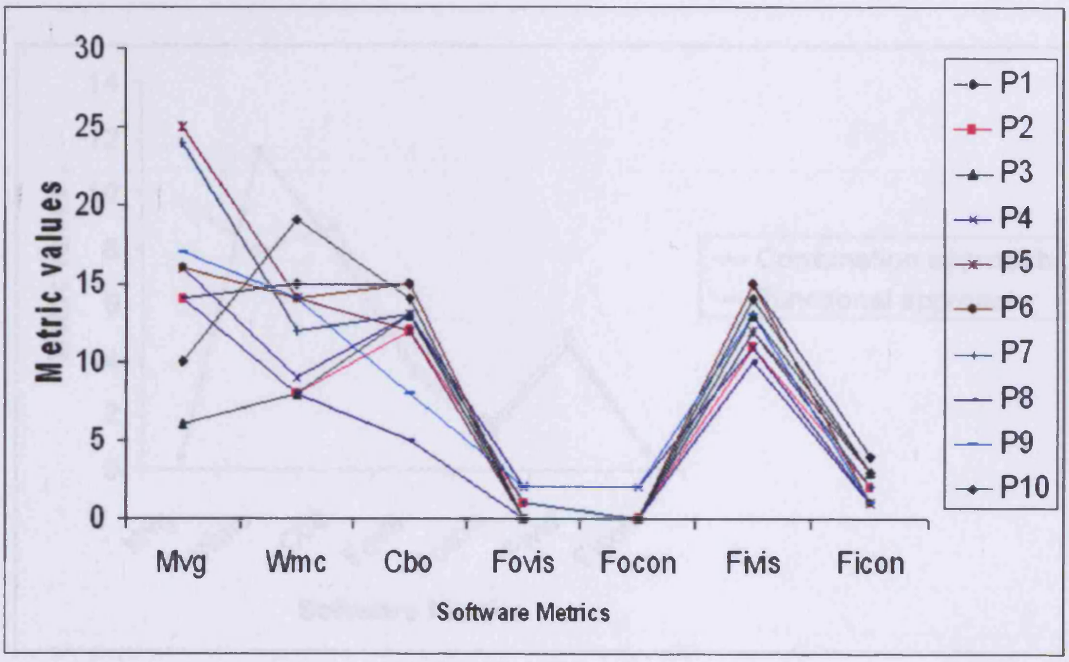


Figure 7.2 Software Metrics in Programs Retrieved using the Combination Approach for PickPhotosPanel.java

approach) to represent programs, developers are presented with programs that depict fewer couplings (i.e less Cbo) and less complexity (i.e less Mvg). As noted in other studies [87, 86], retrieving programs that are reusable and of high quality is beneficial for software developers. Therefore, as Cbo and Mvg indicate the reusability and quality of a program [84, 91, 87, 86], it is better for developers to (re)use programs with fewer coupling (i.e Cbo) and less complex (i.e Mvg).

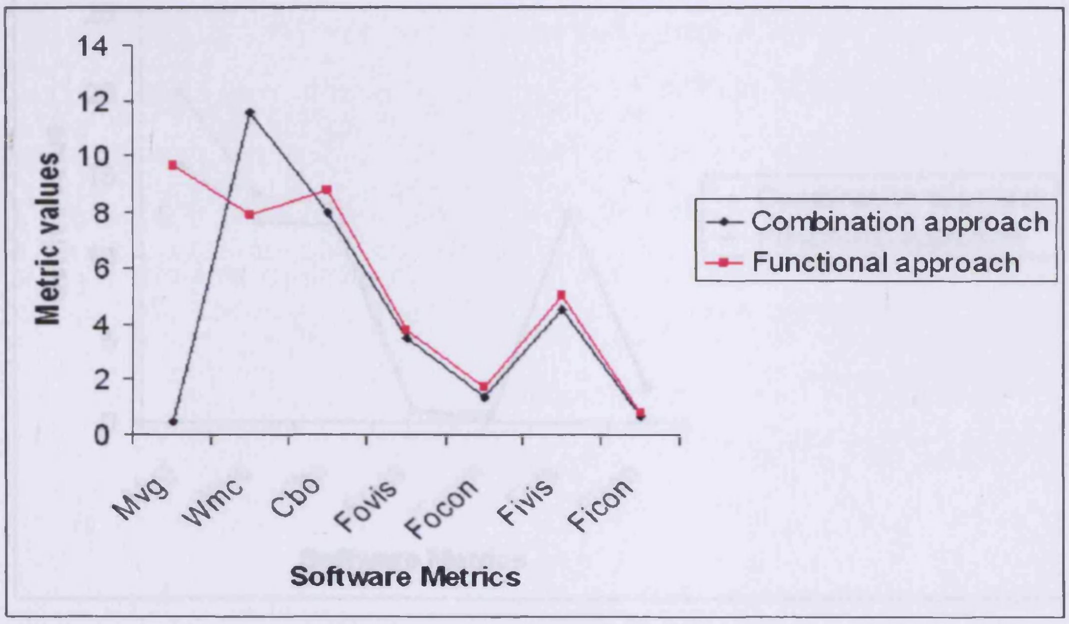


Figure 7.3 Average Values of Software Metrics in Programs Retrieved for MySQLDatabase.java

The graphs in Figure 7.3 and 7.4 are generated based on software metrics of all the programs retrieved using either the functional or the combination approach. Since a user may only (re)use programs that s/he identified to be most relevant to the given query, in Figure 7.5, we illustrate the software metrics of programs that have been classified into "Good"category for query `MySQLDatabase.java`. A

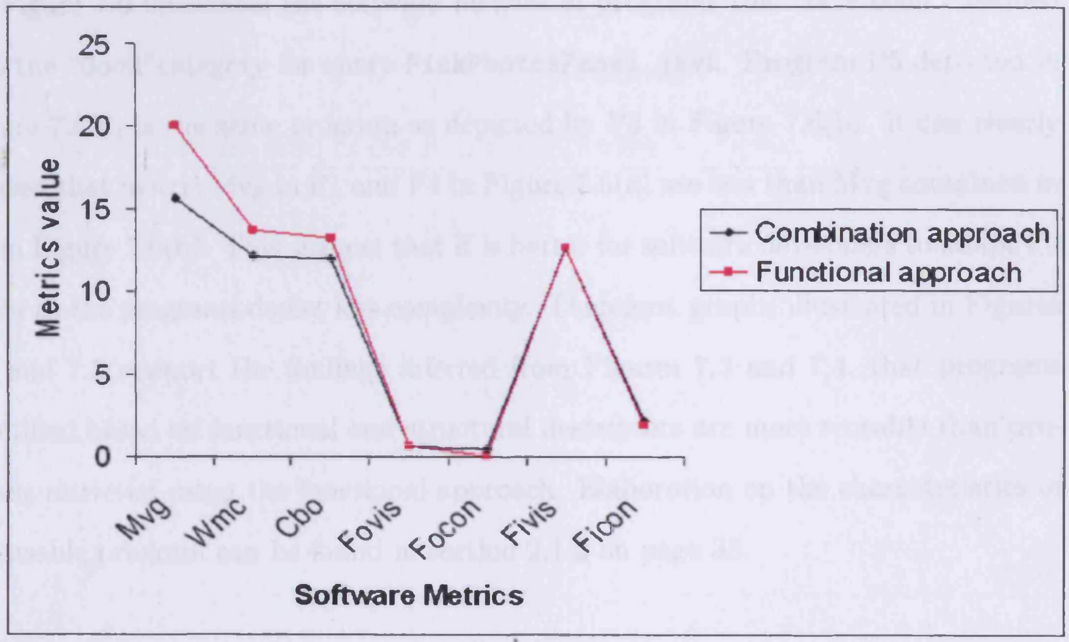


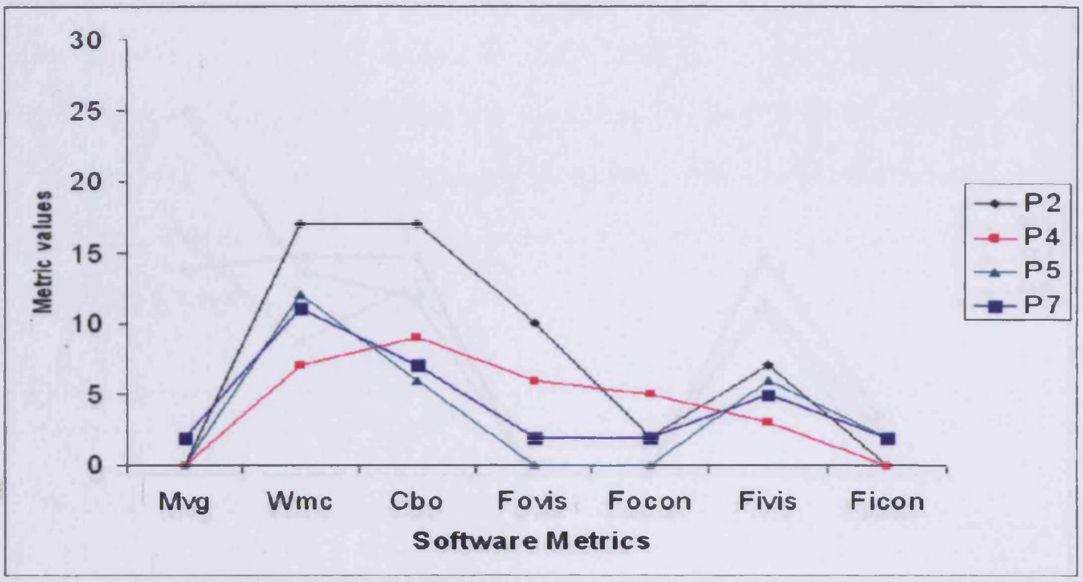
Figure 7.4 Average Values of Software Metrics in Programs Retrieved for PickPhotosPanel.java

comparison between Figures 7.5(a) and 7.5(b) shows that programs retrieved using the combination approach contain software metrics with lower values, when compared to programs retrieved using the functional approach. This is shown in the metrics Wmc, Cbo, Fovis and Fivis. However, Program P2 depicted in both Figure 7.5(a) and 7.5(b) are the same program.

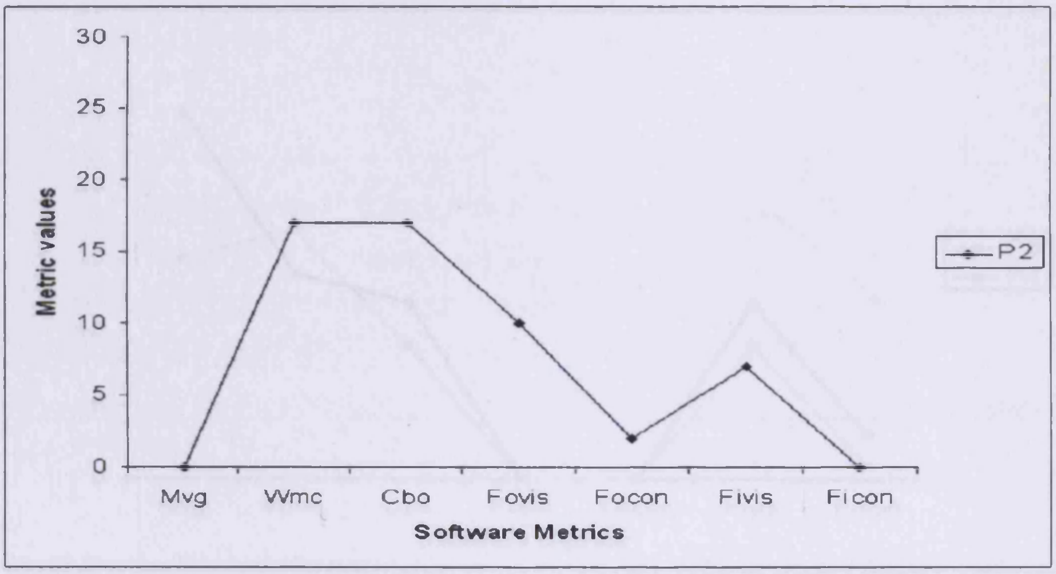
Figure 7.6 illustrates the software metrics of programs that have been classified into the "Good"category for query `PickPhotosPanel.java`. Program P5 depicted in Figure 7.6(a) is the same program as depicted by P3 in Figure 7.6(b). It can clearly be seen that metric Mvg in Pl and P4 in Figure 7.6(a) are less than Mvg contained in P3 in Figure 7.6(b). This suggest that it is better for software developers to adapt P3 or P4 as the programs depict less complexity. Therefore, graphs illustrated in Figures 7.6 and 7.5 support the findings inferred from Figures 7.3 and 7.4, that programs identified based on functional and structural descriptors are more reusable than programs retrieved using the functional approach. Elaboration on the characteristics of a reusable program can be found in section 2.1.2 on page 33.

Software Metrics in Different Application Domains

Another lesson learned from using structural descriptors in the program retrieval system is the use of software metrics in discriminating programs from a different application domain (i.e database and graphics). The graph in Figure 7.7 shows that programs of database and graphics application domains illustrate different trends in their metric values (Mvg, Wmc and Cbo). In this analysis, comparison is undertaken as the increment or reduction of values when two metrics are compared between each other. For example, if metric X is 20 and metric Y is 25, then it is suggested that there is an increment of 25% (i.e (25-20)/20 * 100) in metric values when X is compared against Y, and there is a reduction of 20% (i.e (20-25)/25 * 100) in metric values if

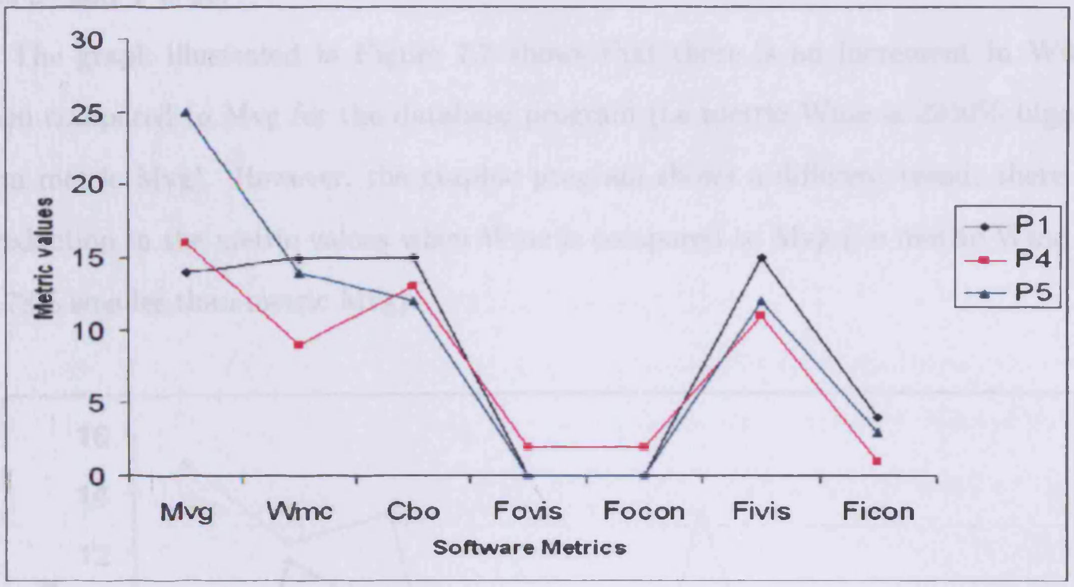


(a) Combination approach

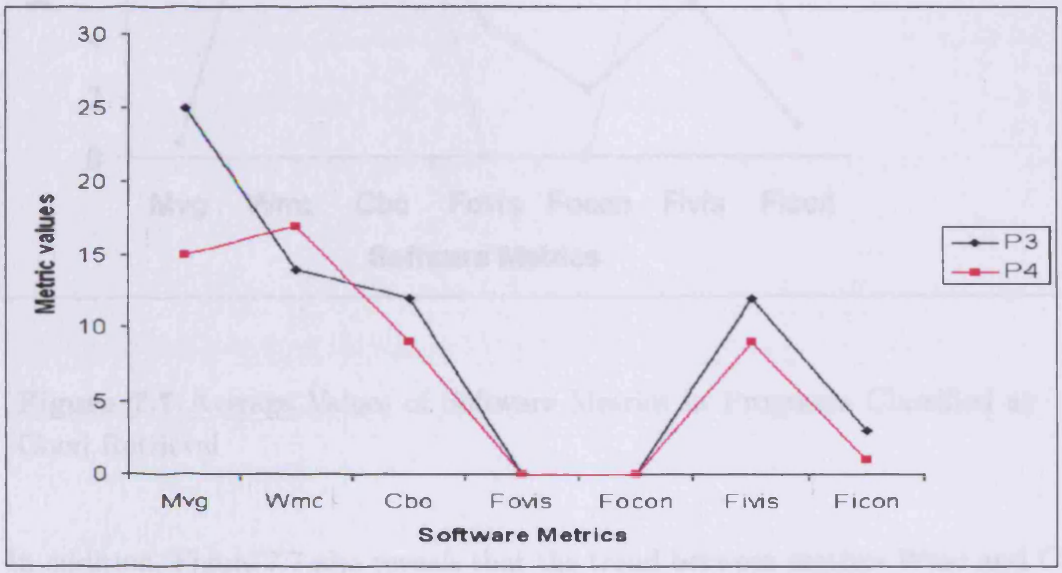


(b) Functional approach

Figure 7.5 Software Metrics in Programs Classified as Good Retrieval for MySQLDatabase.java



(a) Combination approach



(b) Functional approach

Figure 7.6 Software Metrics in Programs Classified as Good Retrieval for PickPhotosPanel.java

X is 25 and Y is 20.

The graph illustrated in Figure 7.7 shows that there is an increment in Wmc when compared to Mvg for the database program (i.e metric Wmc is 2250% bigger than metric Mvg). However, the graphic program shows a different trend; there is a reduction in the metric values when Wmc is compared to Mvg (i.e metric Wmc is 17.78% smaller than metric Mvg).

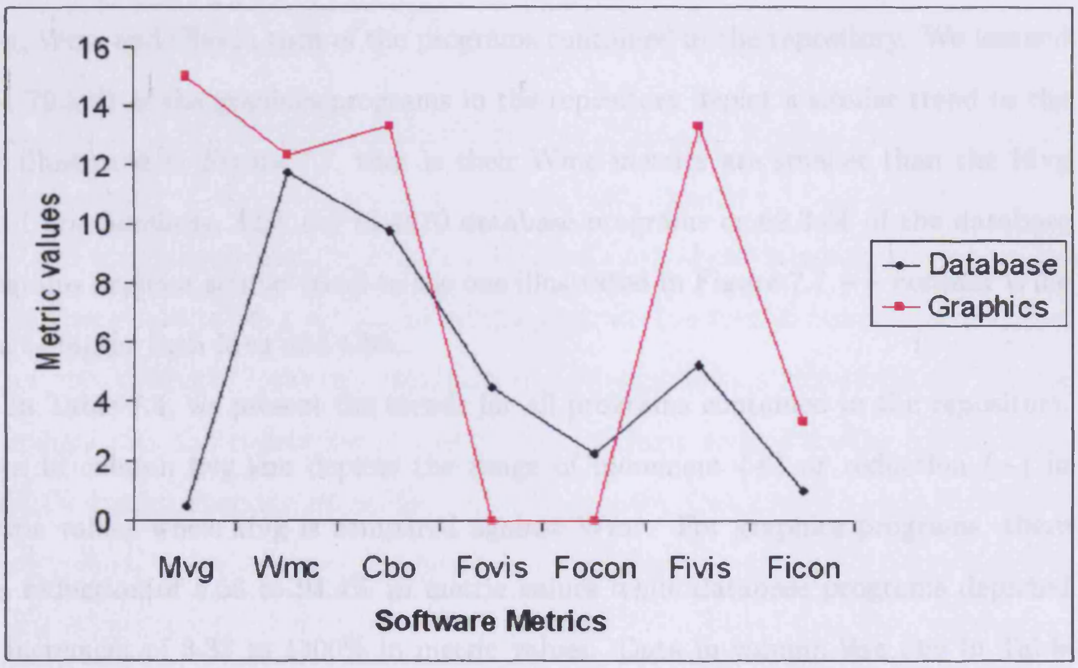


Figure 7.7 Average Values of Software Metrics in Programs Classified as Good Retrieval

In addition, Figure 7.7 also reveals that the trend between metrics Wmc and Cbo differ for programs from different application domains (database and graphics). The graphic program contains Cbo that is 8.11% bigger than Wmc, while the database program contains Cbo that is 17% smaller than Wmc.

Referring back to Figures 7.1 and 7.2, they also show that database and graphics programs illustrate different trends in their metric values. All of the programs in Figure 7.1 contain Wmc that is bigger than Mvg. In addition, nine out of ten programs contain Cbo that is smaller than Wmc. On the other hand, graphics programs illustrated in Figure 7.2 depict a different trend. Seven out of ten programs illustrated in the figure contain Wmc that is smaller than Mvg and Cbo.

In order to identify if such trends exist in all of the programs, we analyzed metrics Mvg, Wmc and Cbo in each of the programs contained in the repository. We learned that 79.50% of the graphics programs in the repository depict a similar trend to the one illustrated in Figure 7.7, that is their Wmc metrics are smaller than the Mvg and Cbo. Similarly, 3238 out of 4670 database programs or 69.34% of the database programs depict a similar trend to the one illustrated in Figure 7.7 –– contain Wmc that is bigger than Mvg and Cbo.

In Table 7.4, we present the trends for all programs contained in the repository. Data in column `Mvg_Wmc` depicts the range of increment (+) or reduction (−) in metric values when Mvg is compared against Wmc. For graphics programs, there is a reduction of 5.56 to 94.4% in metric values while database programs depicted an increment of 3.33 to 1300% in metric values. Data in column `Wmc_Cbo` in Table 7.4 represents the range of increment or reduction in metric values when Wmc is compared against Cbo. For graphics programs, there is an increment of 7.14 to 400% while for database programs there is a reduction of 4 to 86.67% in the metric values. For example, there is a graphic program in the metrics, we learned that there is a reduction of 50% (i.e (4-8)/8 * 100) when Mvg is compared against Wmc, and an increment of 150% (i.e (10-4)/4 * 100) when Wmc is compared against Cbo.

The other example, which is from the database domain, is a program with 20, 23 and 14 for Mvg, Wmc and Cbo respectively. Based on these metrics, we can see that there is an increment of 15% (i.e (23-20)/20 * 100) when Wmc is compared against

Mvg. On the other hand, there is a reduction of 39.13% (i.e (14-23)/23 * 100) in the metric values when Cbo is compared against Wmc.

Table 7.4 Percentage of Increment and Reduction of Metric Values

Application Domain	Mvg_Wmc	Wmc_Cbo
Database	(+) 3.33 - 1300%	(−) 4 - 86.67%
Graphics	(−) 5.56 - 94.4%	(+) 7.14 - 400%

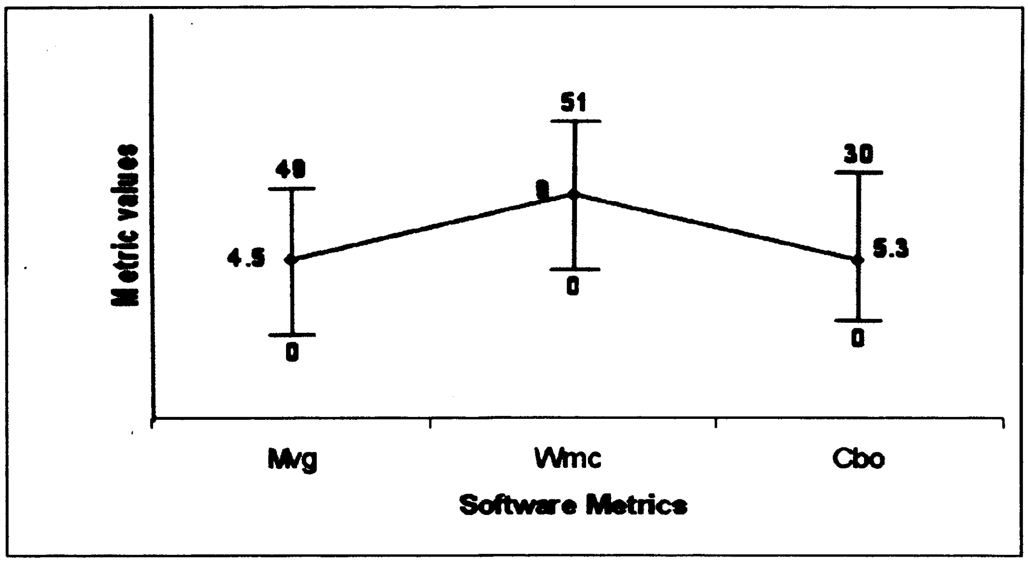
Based on the analysis of the trends and the graphs in Figures 7.1, 7.2 and 7.7, we noticed that the database and graphics programs illustrate different shapes of line graphs. We represent the graphs in those figures as the one presented in Figure 7.8.

Software metrics (i.e Mvg, Wmc and Cbo) contained in a database program would illustrate a similar shape of line graph as the graph shown in Figure 7.8(a). On the other hand, a graphic program would illustrate a graph that is similar to the graph in Figure 7.8(b). Based on these figures, it is suggested that metric trends that exists between Mvg and Wmc, and between Wmc and Cbo contribute to discriminate a database from graphics programs. With this, we obtained the following classification function:

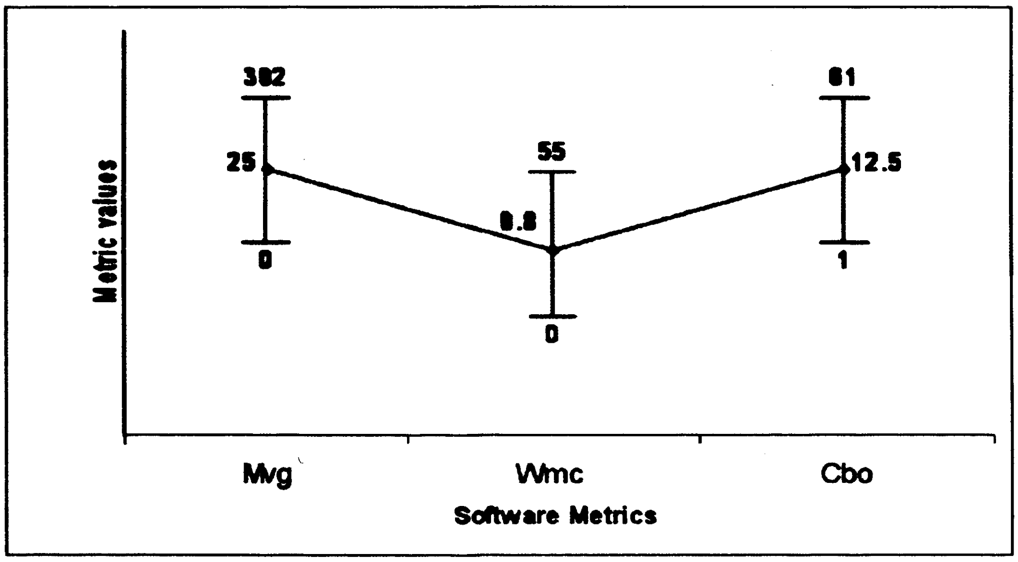
$$f(x,y,z) = \begin{cases} 1 & \text{if } 0 \leq x \leq m_1, 0 \leq z \leq m_2, max(x,z) \leq y \leq m_3 \\ 2 & \text{if } 0 \leq x \leq n_1, 1 \leq z \leq n_2, y \leq min(x,z), 0 \leq y \leq n_3 \end{cases} \tag{7.1}$$

where 1 is database, 2 is graphics, x is Mvg, y is Wmc and z is Cbo. In addition, based on the programs in our repository, $m_1 = 49$, $m_2 = 30$, $m_3 = 51$, $n_1 = 392$, $n_2 = 61$ and $n_3 = 55$.

In addition to the analysis of trends in a program, descriptive statistics (i.e maximum, mean and minimum values) of metrics Mvg, Wmc and Cbo for programs



(a) Database



(b) Graphics

Figure 7.8 Shapes of Graphs with Descriptive Statistics for Database and Graphics Programs

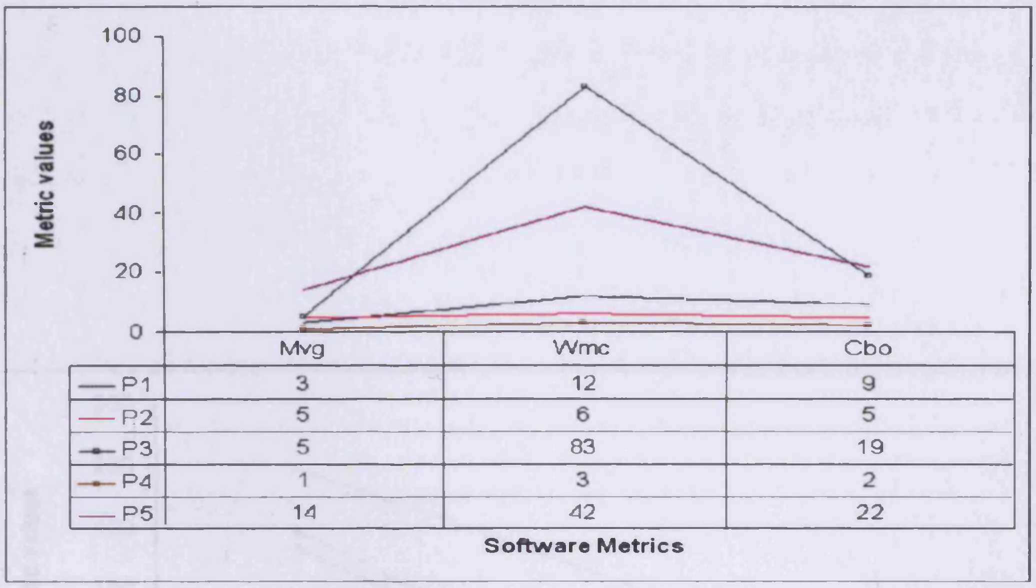
contained in the repository (depicted in Figure 7.8) reveal that graphics programs are more complex than database programs. As mentioned in section 2.1.2, program complexity can be measured using Mvg. The mean value of Mvg (i.e 25) for graphics programs is six times bigger than the mean for database programs (i.e 4.5). In addition, the maximum value of Mvg (i.e 392) for graphics programs is eight times the maximum value of database programs (i.e 49). Figure 7.8 also reveals that graphics programs are tightly coupled compared to database programs – the mean value of Cbo in graphics programs (i.e 12.5) is doubled the value in database programs (i.e 5.3).

In order to validate the use of Equation 7.1 in program classification, we extract the relevant metrics from a set of new programs. This new data set contains ten programs; five of each domain (database and graphics). These programs obtained from the **Sourceforge.net** [29] were not initially included in the repository. For reference, examples of the programs are included in *Appendix F: Test Programs used for the Identification of Metric Trends.* A graph depicting metric values of the programs is illustrated in Figure 7.9. As the metric values of the programs (except P3 in Figure 7.9(a)) are all included in the range as defined for database and graphics programs (Equation 7.1), the plotted graphs are similar to the one illustrated in Figure 7.8. For example, based on the classification function, P5 in Figure 7.9(b) with Mvg=12, Wmc=5 and Cbo=11 fulfilled the requirements of a graphics programs. The Mvg value in the program is smaller than 392, the value of Cbo is between 1 and 61 and the value of Wmc is smaller than the minimum value between Mvg and Cbo (i.e $min(Mvg, Cbo)$) and it is between 0 and 55. P1 in Figure 7.9(a) is considered as a database program, similar to the classification undertaken by **Sourceforge.net** [29]. This is because it's Wmc value is larger than the maximum value between Mvg and Cbo (i.e 9), the Mvg is smaller than 49 and the Cbo is smaller than 30. Program

P3 depicted in Figure 7.9(a) contains Wmc that is larger than the value suggested in the classification function, that is 51. However, the values of metric Mvg and Cbo are in the range defined in the function. Based on the researchers' analysis, the program contains a large number of functions, hence resulting Wmc=83. As software developers may include as many functions as they like in a program, such practice is not common since it would caused difficulties in maintaining and/or understanding the program. Nevertheless, the graph plotted based on the metric values of the program is similar to the one illustrated in Figure 7.8(a), hence suggesting that it is a database program.

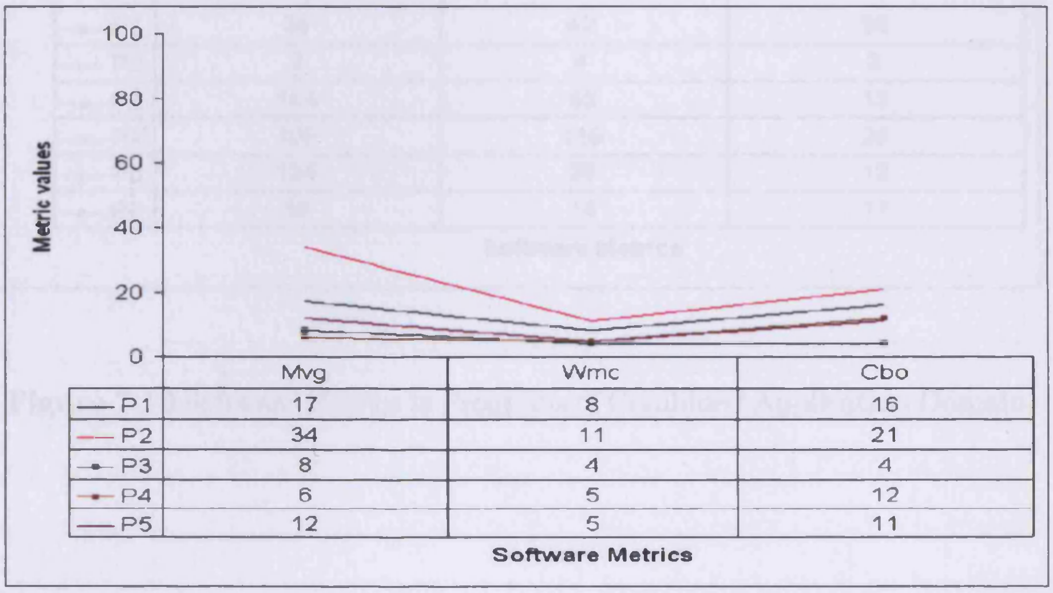
In addition, we also include programs which can be classified into both application domains. These are the same programs used in experiment undertaken in section 5.2.1 on page 115. The programs depict the function of organizing various data type (images, audio, XML documents and text) in a database. For reference, we include the programs in *Appendix G: Test Programs of Combined Domain.* Figure 7.10 illustrates the graphs plotted using metrics Mvg, Wmc and Cbo of the programs. Based on Equation 7.1, program P2 in the figure fulfilled the requirements of a database program while P6 is considered as a graphic program. Based on the analysis, P2 illustrates the function of a keyword search performed in an image database while P6 illustrates the function of connecting to a database and managing the objects in the database. Objects in this database is represented in a tree representation, hence suggesting it to be considered as type graphics. Nevertheless, it is also acceptable to classify the program into the database domain as it illustrates the function of database connection. In addition, the graphs of P2 depict a similar pattern to the graph in Figure 7.8(a) while P6 is similar to the graph illustrated in Figure 7.8(b).

As for P1, even though the plotted graph is similar to the one presented in Figure 7.8(a), its Cbo value is greater than the maximum value suggested for a database



	Mvg	Wmc	Cbo
P1	3	12	9
P2	5	6	5
P3	5	83	19
P4	1	3	2
P5	14	42	22

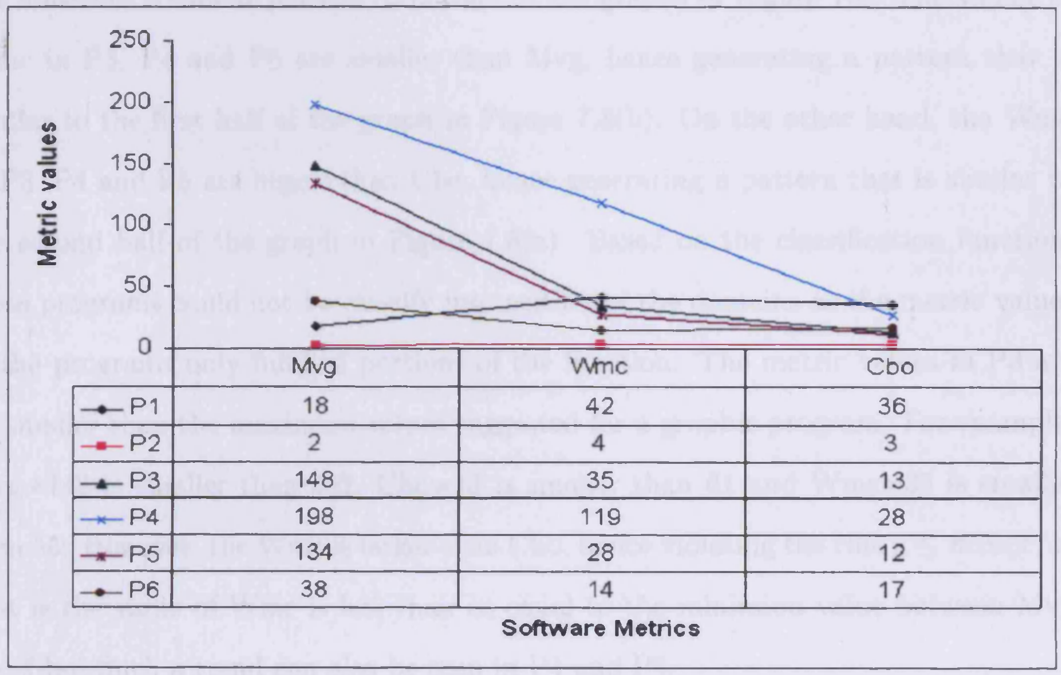
(a) Database



	Mvg	Wmc	Cbo
P1	17	8	16
P2	34	11	21
P3	8	4	4
P4	6	5	12
P5	12	5	11

(b) Graphics

Figure 7.9 Software Metrics in Database and Graphics Programs: Testing Data Set



	Mvg	Wmc	Cbo
P1	18	42	36
P2	2	4	3
P3	148	35	13
P4	198	119	28
P5	134	28	12
P6	38	14	17

Figure 7.10 Software Metrics in Programs of Combined Application Domain

program (in the classification function). Such a trend is accounted for the programs' function that uses two types of information, textual and images, contained in a database. Hence, resulting a larger value of CBO as the object to be created (i.e tshirt) relies on various other objects (image, textual description).

On the other hand, the graphs illustrated by P3, P4 and P5 in Figure 7.10 do not illustrate a similar pattern to neither of the graphs in Figure 7.8. The values of Wmc in P3, P4 and P5 are smaller than Mvg, hence generating a pattern that is similar to the first half of the graph in Figure 7.8(b). On the other hand, the Wmc in P3, P4 and P5 are bigger than Cbo, hence generating a pattern that is similar to the second half of the graph in Figure 7.8(a). Based on the classification function, these programs could not be classify into neither of the domains as the metric values in the programs only fulfilled portions of the function. The metric values in P3 are all smaller than the maximum values suggested for a graphic program. For example, Mvg=148 is smaller than 392, Cbo=13 is smaller than 61 and Wmc=35 is smaller than 55. However, the Wmc is larger than Cbo, hence violating the rule $y \leq min(x, z)$ that is the value of Wmc is less than or equal to the minimum value between Mvg and Cbo. Such a trend can also be seen in P4 and P5.

However, based on the analysis made by the researcher, P3 is considered to be a graphic program as it contains a larger number of functions (i.e methods) dealing with images and audio than functions handling textual information. On the other hand, P4 that illustrates the function of managing XML documents can be considered as type database as its methods focus on textual information rather visual images. Similarly, P5 is also considered by the researcher to be a database program. This is because the program focuses on string matching in creating a query for an image database. Each of the images stored in the database is provided with a textual description, and in P5 the images are retrieved based on keyword match performed on these descriptions.

7.3 Empirical Evaluations of the Retrieval System

To identify the effectiveness of our retrieval system in supporting program retrieval and program reuse, subjective evaluation experiments were conducted. The objective of the evaluation was two-fold: (1) to verify that a combination of functional and structural descriptors generates better retrieval than using functional descriptors on their own, and, (2) to identify whether the retrieved programs can be (re)used to develop the required program (based on a given programming task). The objectives were achieved through the use of an information retrieval measurement, that is, precision. The identified measurement indicates which retrieval mechanism is better than the other, and represents the possibility of the retrieved programs to be adapted into the working context (i.e a given programming task). The higher the precision, the better a retrieval mechanism is, and there is a bigger chance that the user would reuse the program(s) in developing a program for the given task. The structure of the experiments is described in the following section, which is then followed by the findings.

7.3.1 Subjects of Experiments

Subjects were recruited from post-graduate students from the School of Computer Science, covering Year 1 to Year 3. Because the goal of the retrieval system was to present programs to developers based on a particular task, only students who already had programming knowledge and experience were recruited as subjects. Furthermore, because the system is a prototype that at the moment only handles Java programs, a basic knowledge of Java programming language was also required so that subjects could easily create a suitable query based on the task given and later would be better able to evaluate programs in the hit list.

Ten subjects voluntarily participated in the evaluation experiments. This number doubles the number of subjects involved in evaluating a retrieval system as undertaken by Ye [156]. All but three subjects were using another language as their main programming language. Nevertheless, they had been writing programs using the Java language. The subjects' expertise in Java programming varied, ranging from beginner to expert level. All of them knew the syntax of Java well; the difference of their expertise came from the range of reusable components they knew (classes and methods in API libraries). Table 7.5 summarizes their background knowledge about programming in general and Java in particular.

Table 7.5 Programming Knowledge and Expertise of Subjects

(a) Subject 1 to 5

Subject	**S1**	**S2**	**S3**	**S4**	**S5**
Years of general programming	15	7	10	10	6
Current major programming language	Java	C++	Java	Java	Java
Years of Java programming	2	1	6	6	4
Frequency in Java programming	daily	daily	daily	daily	daily
Frequency in acting as system analyst(1-never, 5-always)	5	1	4	5	5
Self evaluation of Java expertise(1-beginner, 5-master)	2	4	4	3	3
Self evaluation of knowledge in designing problem solving	1	3	1	2	1
Self evaluation of knowledge in software metrics	2	1	1	1	2

(b) Subject 6 to 10

Subject	**S6**	**S7**	**S8**	**S9**	**S10**
Years of general programming	7	2	13	8	4
Current major programming language	Java	C++	Java	C++	C++
Years of Java programming	6	3	5	7	3
Frequency in Java programming	daily	daily	daily	daily	daily
Frequency in acting as system analyst(1-never, 5-always)	4	3	5	3	1
Self evaluation of Java expertise(1-beginner, 5-master)	4	2	4	4	3
Self evaluation of knowledge in designing problem solving	3	1	2	2	2
Self evaluation of knowledge in software metrics	2	3	2	3	3

7.3.2 Structure of Experiments

Programming Tasks

Because the subjects were volunteers, large and time-consuming tasks were not very suitable. The experiments used programming tasks similar to the typical assignments of a programming language course, which could be implemented with several methods in about 20 to 60 minutes. The following tasks were used in the experiments.

Task 1: Write a class for formatting text to be used in database applications. The class should include the creation of a single instance that uses a string that has been passed to it as the text to be formatted. Upon receiving the string, it must notify other objects that rely on the changes it has made towards the string.

Task 2: Write a class to retrieve data from a database. The data are to be nested into radio or checkbox button in a database application. To motivate reuse, minimum couplings between objects should be considered in the coding.

Task 3: Write a class that creates connection to an SQL database. Once the connection has been made, the object should notify object `ReadData` to retrieve data from the database.

Task 4: Write a program that depicts GUI utility methods to be used in graphics applications. This should include 1) creation of a thumbnail version of the given `BufferedImage`, 2) reading and saving an image from/to a file. Also ensure that you restrict instantiation of a class to one object.

Task 5: Write a panel component that allows the user to configure where images are currently stored. The class defines an implementation of a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.

Task 6: The class to be written should be able to organize and manage photos in a collection. Such an organization should manipulate composite objects in exactly the same way primitive objects are manipulated.

All of the tasks could be implemented with different combinations of program structures. Therefore, these tasks allowed us to observe how the delivery of the system matched the given tasks. For each programming task, we provide a program template as a guideline in developing the program. The subjects later modify the program based on their programming experience and use it as the search query. In *Appendix H: Program Templates Used as Search Queries*, we include examples of program templates (for the programming tasks) used in this experiment.

7.3.3 Results

In this section, we present findings obtained from the empirical experiments undertaken. A group of 10 people were asked to use the system and retrieval effectiveness was measured using precision that was obtained at DCV=10 – DCV is described in Definition 13 on page 134. The analysis of the precision required each subject to determine if each of the retrieved programs were relevant to the search query. Recall measurement was not undertaken in the retrieval analysis since in order to build an answer set, subjects were required to analyze each of the programs in the collection and determine whether it fulfilled the task requirements. Considering that the subjects were volunteers, such an activity would be time consuming.

Eight of the subjects were given two tasks, one from each application domain (Database and Graphics). The other two subjects had only one task each. In total, there were 18 queries submitted to the retrieval system during the experiment -- nine queries of database tasks (Task 1, 2 and 3) and nine queries of graphics tasks (Task 4, 5 and 6). Upon submitting a query, the user would receive two hit lists; the first contains programs identified based only on functional descriptors while the second contains programs identified using a combination of functional and structural descriptors.

Table 7.6 Average of Precision Scores and Processing Time

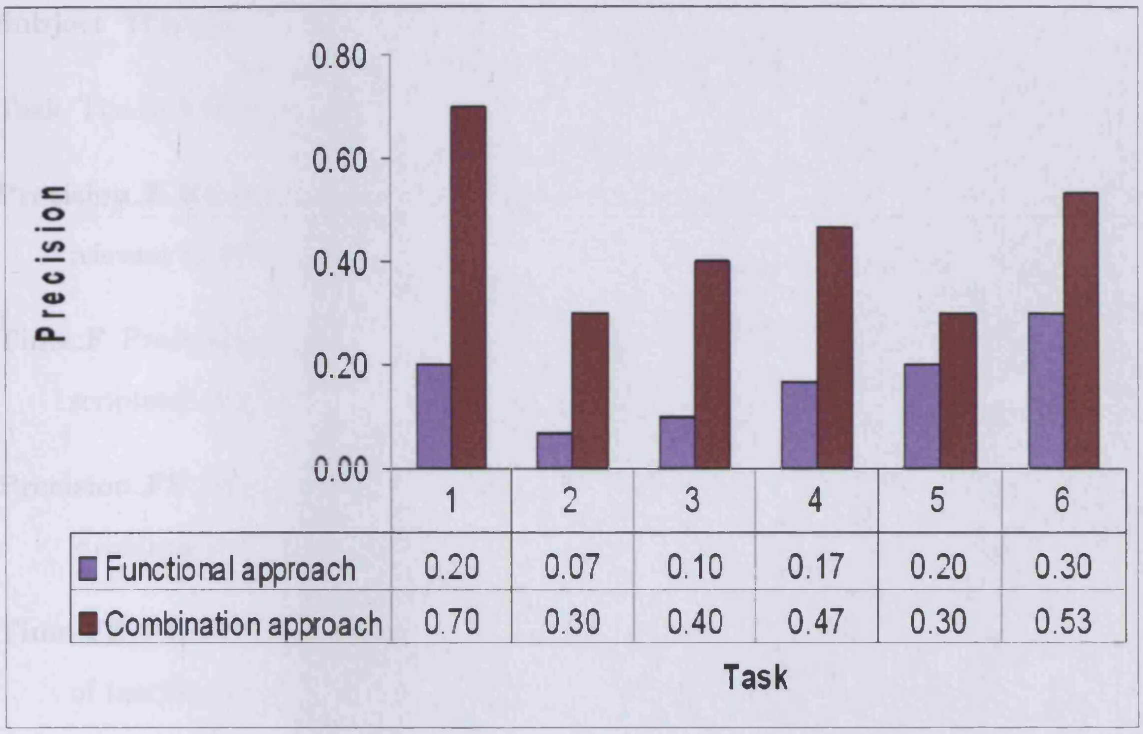
Application Domain	Functional approach		Combination approach	
	Precision	Time(ms)	Precision	Time(ms)
All queries	0.172	890	0.450	1190
Database queries	0.122	879	0.467	1184
Graphics queries	0.222	900	0.433	1195

Table 7.6 presents a summary of the results obtained from the experimental evaluation. Data in the table includes average values of precision and processing time for both types of queries, Database and Graphics. In addition, data in the table also reveals the average values of precision and processing time for all of the queries submitted to the system. It took an average of 1190 milliseconds to generate a single hit list using the combination of functional and structural descriptors. However, less times is taken to generate a hit list using functional descriptors. Even though the approach of combining functional and structural descriptors requires more processing time, it is well worth as the average precision for a single query is increased from 0.172 to 0.450. In a similar to such improvement, an average precision for a single query of type graphics was raised from 0.122 to 0.467 while for a query of type database, the precision was up to 0.467 from as low as 0.122. Even though these precisions are less than 0.7 which we assumed to be successful, the retrieved programs are shown

to have the required structure as identified in the programming task. Nevertheless, such a result (precision ≤ 0.7) is partly caused by the approach we took in identifying similarity between functional descriptors of a query and programs in the repository. As the Levenshtein (refer to section 3.4 on page 64) threshold value has been set to 2, functional descriptors of programs in the repository that can be identified as similar to functional descriptors of the query are restricted. For example, referring to Task 3 as in section 7.3.2, the identifier `dbConnectionName` presence as a class variable in `DbGetConnection.java` is not identified to be relevant to one of the terms identified in the search query (i.e `Get Connection`) as the distance between these strings is more than 2. Nevertheless, when the programs were analyzed manually, these identifiers were shown to illustrate similar functions.

Based on data depicted in Table 7.6, we also learned that queries originating from the graphics domain requires longer processing time. When the functional approach was employed, the average processing time for a database query is 879 milliseconds while a graphic query requires additional of 21 milliseconds. Similar to the pattern, retrieval for a graphic query performed using the combination approach requires 1195 milliseconds compared to the database query which only needed 1184 milliseconds. This can be reasoned by the complexity of the programs used as the search query. The graphics programs contain a larger number of methods and they also illustrate various structural relationships (e.g method invocations).

An average precision for each of the six tasks that were given to the subjects are depicted in Figure 7.11. The biggest difference of precision was obtained when the subjects were given Task 1 as the programming task. By using a combination of functional and structural descriptors, the precision was increased from 0.2 to 0.7, which is more than triple. Based on data depicted in Figure 7.11 and Table 7.6, it was learned that such an approach benefits tasks of Database more than of Graphics.



Task	1	2	3	4	5	6
Functional approach	0.20	0.07	0.10	0.17	0.20	0.30
Combination approach	0.70	0.30	0.40	0.47	0.30	0.53

Figure 7.11 Average of Precision Scores for Six Tasks

Table 7.7 details the findings of the experiment. Values in each column of the table are defined as follows while the post-experiment questions and an example of subject feedback form for the undertaken experiment are provided in *Appendix I: Subjects Feedback*

Subject The subjects who participated in the experiment.

Task The task that was given to the subject.

Precision_F Ratio of retrieved programs based on functional descriptors, that are relevant to the subject's query program.

Time_F Processing time taken before a list of programs (based on functional descriptors) is presented to the user (measured in milliseconds)

Precision_FS Ratio of retrieved programs based on a combination of functional and structural descriptors, that are relevant to the subject's query program.

Time_FS Processing time taken before a list of programs (based on a combination of functional and structural descriptors) is presented to the user (measured in milliseconds)

Satisfaction_FS Subject's satisfaction towards the retrieved programs (based on a combination of functional and structural descriptors) using the 5-point Likert-scale [157] (1-not satisfied at all to 5-very satisfied)

Based on the data depicted in Table 7.5 on page 170 and the data of using a combination of functional and structural descriptors in Table 7.7, we present findings that are related to the subject's background. This finding was supported by statistical analysis which was undertaken using data depicted in Table 7.5 on page 170 and in Table 7.7. The Pearson correlation coefficient [103], r, was employed to determine

Table 7.7 Overall Result of Field Experiment on 10 Subjects

Task	**Subject**	**Precision_F**	**Time_F**	**Precision_FS**	**Time_FS**	**Satisfaction_FS**
1	S1	0.3	875	0.8	1178	3
1	S5	0.3	865	0.8	1180	4
1	S9	0	871	0.5	1175	3
2	S3	0.1	924	0.2	1225	4
2	S4	0	924	0.3	1224	3
2	S8	0.1	923	0.4	1225	4
3	S6	0	832	0.5	1142	3
3	S10	0.3	835	0.5	1141	3
3	S2	0	868	0.2	1170	2
4	S1	0.3	860	0.8	1163	4
4	S5	0.2	879	0.5	1163	3
4	S8	0	864	0.1	1165	4
5	S3	0.2	880	0.3	1188	4
5	S4	0.2	881	0.3	1188	2
5	S9	0.2	880	0.3	1188	4
6	S6	0	950	0.6	1235	3
6	S7	0.8	965	0.8	1239	3
6	S10	0.1	949	0.2	1231	3

the direction and strength of the linear relationship between two continuous variables (e.g Precision_FS and Java expertise) and we obtained the following:

1. There was a strong negative correlation (r=-0.638) between precision of programs delivered (as measured by Precision_FS) and Java skill (as measured by Java expertise), [p<0.1], with high levels of precision associated with lower levels of Java skill. The result is obtained using $\alpha = 0.1$. The higher Java skill a subject has, the less precision is obtained in retrieving relevant programs. One possible explanation is that developers with a better knowledge of Java programming skill tend to be more specific in determining what is relevant when given a task. Therefore, they easily disregard programs that they believe are not significant to the problem context.

2. There was a medium negative correlation (r=-0.422) between precision of programs delivered (as measured by Precision_FS) and years of Java programming (as measured by years of Java programming), with medium levels of precision associated with medium levels of years of Java programming. The statistical test is based on $\alpha = 0.1$. The less experience (measured using number of years) a subject has in using Java as a programming language, the more a medium precision is obtained in retrieving relevant programs. This may be due to the practice of having a higher number of programs as programming examples if a subject is new to a programming language. Therefore, these subjects easily identify programs to be relevant to the given programming task.

3. There was a medium correlation between frequency in analyzing a problem prior to writing the code (measured as frequency in acting as the system analyst) and satisfaction with the delivered programs (measured as Satisfaction_FS). Such a finding is illustrated in the correlation analysis between the two variables.

Statistical analysis has revealed a correlation of r=0.410 at $\alpha = 0.1$ between the two variables. We learned that a subject that has moderate experience in problem solving tended to be moderately satisfied by the retrieval system. One possible reason is that with experience in problem solving, a developer can identify programs that are suitable to be adapted into the application at-hand.

7.4 Conclusion

In this chapter, we have presented evaluations performed in order to evaluate the proposed retrieval system. Based on the experiments undertaken, it is noted that the proposed approach of combining functional and structural descriptors is better than the approach of using functional descriptors on their own. Furthermore, the combination approach has been able to present users with programs that illustrate the required function and structure. We obtained approximately 50% precision in both the objective and subjective evaluations, and based on existing studies [45, 7] the precision can be made better if additional mechanisms (e.g Latent Semantic Indexing [45] and ontology-based [7]) are employed to identify functional descriptors of a program.

Software metrics are intended to measure software quality characteristics quantitatively. Among several quality characteristics, reusability is particularly important when reusing software components. The findings of the objective evaluation which is described in section 7.2.3 indicates that the presented programs of the combination approach illustrate less complexity. Program complexity is contributed to by various factors and they include the number of linearly independent routes through a directed acyclic graph that maps the flow of control of a subprogram (Mvg), the number of methods defined in a program (Wmc) and the measure of interdependence between methods in a program (Cbo). By using programs that illustrate the required function, and contain small values of Mvg, Wmc and Cbo, as programming examples, developers are able to develop their own application that depict better quality.

In addition, we also learned that there is a difference of trends (shown by the different shapes of graphs) in software metrics for database and graphics programs. Based on the experiments undertaken, it is noted that database programs contain greater value of Wmc when compared to metrics Mvg and Cbo. On the other hand, metric Wmc for graphics programs are smaller than metrics Mvg and Cbo. For programs that depict the features of database and graphics domain, they illustrate a combination trend in the metric values.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In the introduction to this thesis, we posed a number of questions concerning the use of a combination of functional and structural descriptors that support program retrieval from a software repository. In this chapter, we will reflect on these questions, describe how the various chapters contribute to answering each questions and draw some conclusions.

8.1 Conclusion

Reusing existing software components is one way of building software. By reuse, developers can avoid repetitive work and focus on the unique features of the new system. However, the problem is how software developers can know that they are doing something that others have done many times before. As one developer said [158]:

I could be creating a method that does exactly the same thing somebody else does...even though we have access to each others code. We might call them different names and we might have a bit different way of doing it, but were still doing the same thing.

This has been the central question investigated in this thesis. The main contributions of this thesis include: (1) a combination of functional and structural descriptors in identifying relevant programs from a software repository, (2) a mechanism (i.e compound index) that can be used to integrate descriptors of a program, (3) the use of software metrics in classifying a program into an application domain, and (4) the detection and use of design patterns as structural descriptors in a program retrieval system.

8.1.1 Information Extraction

Question 1: *How can we extract information from a program that can be used as functional or structural descriptors in a program retrieval system?*

One of the first steps in a software retrieval system is indexing components in the repository: the automated extraction of information from program source code. In Chapter 1, we argue that this step is hindered by the typical irregularities that occur in program source code (e.g identifier names and design patterns) which make it hard to parse the program for its function and structure. Hence, we suggested to provide additional information for the identifier names extracted from the program and use the structural relationships that are contained in a program to infer existence of design patterns in the program. In Chapter 3, we illustrated how relevant terms were extracted from the different contexts of a program and used as functional descriptors. Each descriptor were given appropriate weight to indicate its importance in the program. In order to make use of information that is not explicitly available in a program, in Chapters 4 and 5, we elaborated on how to use structural relationships that exist in a program as program descriptors. In Chapter 4, this information has been used to identify existence of design patterns Singleton, Composite and Observer in a program, while in Chapter 5 we employed the information to classify the program into an application domain.

Another contribution of this thesis that relates to information extraction, is extending the retrieval system as an active component repository [156]. In general, information systems that just offer software artefacts (e.g program, software documentation) to a user are of little use because they ignore the users' working context. The working context consists of the task acted upon and the user acting. The challenge of implementing an active information system is to deliver context-sensitive information related to both the task at hand and the background knowledge of the user. Needs for reusable programs are not determined before programming starts, as most current component retrieval repository systems have assumed; they arise in the middle of the programming process [159]. In as much as developers are using keywords and/or phrases to represent search requirements, for example as in Google code search [23], it has been shown in this thesis that it is possible for component repository systems to capture the requirements by utilizing information available in existing programs that are developed for a given task. In Chapters 6 and 7, the experiments undertaken were all based on programs as the search queries.

8.1.2 Creating New Representation

Question 2: *How can we combine functional and structural descriptors of a program to represent a query and programs in a repository?*

In Chapter 6, we introduced a new mechanism that integrates information that was extracted from Chapter 3, 4 and 5. We have demonstrated the use of a compound index to integrate information on the function and structure of a program. As the functional descriptors represent what a component does, structural descriptors symbolize structural relationships that exist in achieving the function. Therefore, an integrated index of functional and structural descriptors represents a component better, compared to using either the functional or structural descriptors on their own. The compound index is flexible, as the number of functional and structural descriptors used to represent a query and/or a component is not fixed. It can be expanded to include other information that relates to a component, such as the software architecture (structural descriptors) and the sample of input/output for the component (functional descriptors). In addition, the index is generated automatically by the retrieval system and therefore is economical, as it does not require the involvement of a human (expert) such as in a faceted approach [160].

8.1.3 Supporting Program Retrieval

Question 3: *How can we use the information obtained in the first two questions to support and improve program retrieval?*

An issue that arises when combining functional and structural descriptors is the significant of using structural descriptors in identifying relevant program retrieval. One of the benefits that has been gained by using structural descriptors is the automation of program classification into an appropriate application domain. The work undertaken has been demonstrated in Chapter 5 and it includes the classification of programs into a database or graphics domain based on software metrics contained in the program. Details on these metrics can seen in section 2.1.2 on page 33. Results of the experiments undertaken in Chapter 5 have shown that by classifying programs into application domains, retrieval of relevant programs can be improved. In addition, in Chapter 7, we learned that the database and graphics programs illustrate different shapes of graphs (line graphs). Based on the existing programs in the repository, we obtained two classifications functions that can be used to determine the application domain of a program. These functions (Equation 7.1) were elaborated upon in section 7.2.3 on page 156.

The main contribution of this research is the use of functional and structural descriptors in presenting software developers with relevant programs. It has been identified in Chapter 7 (through objective and subjective analysis) that by using the combination approach, precision in program retrieval has been increased when compared to using functional descriptors on their own. Furthermore, by combining functional and structural descriptors, developers are presented with programs that are simple (less complex) but yet illustrating the required function. As mentioned in much software maintenance literature such as Scotto et al. [161] and Lehman et al. [162], it is easier to understand and maintain applications that show less complexity. Thus, by using programs with smaller values of Mvg, Wmc and Cbo as programming examples, developers are able to create an application of their own with the required function, and at the same time the application is also easy to maintain.

Question 4: *How is similarity measurement undertaken between a query and the program in a software repository?*

Upon proposing the use of a compound index to integrate functional and structural descriptors of a program, there is a need to identify how similarity measurement between a compound index of a query and of a program from the repository is to be performed. In Chapter 6, we demonstrate experiments involving two types of similarity measurements: vector model and data distribution. The use of the former was to investigate if programs having similar data distribution are to illustrate similar function and structure. That is, we would like to learn whether programs having the same order of indices can be considered similar. On the other hand, the latter measurement was used to discover if similarity between two programs are identified better by treating them as vectors. It is shown in Chapter 6 that by using Euclidean distance (ED) to compute the degree of similarity between programs, higher precision and recall have been obtained, when compared to using the cosine measure and skewness.

8.2 Future Work

While this research proved that it is better to retrieve programs by using the combination of functional and structural descriptors rather than focusing on functional descriptors only, many additional areas that need to be resolved have been identified. These can be broadly classified into the areas of tool improvements, domain expansion, and suitable similarity measurements.

8.2.1 Extending to a Higher Scale

The identification of reusable programs in object-oriented open-source applications can be automated to a high degree. Through careful attention to knowledge extraction and metrics analysis, tools can be built that will present sufficient information to the user to enable the user to make an intelligent reuse choice. The assumption that a combination of functional and structural descriptors of a program can provide sufficient information to support the selection of programs from a repository has proven valid. While many other application domains are yet to be explored, the domain used in this research was acceptable to give a degree of confidence in extendability to other domains. Furthermore, other software metrics could be extracted from a program, which can later be used in classifying programs into application domains other than database and graphics. We would then be able to identify other metric trends that may exist in programs from different application domains (e.g financial, scheduling etc.). In addition, with a greater number of software metrics extracted from a program, we would be better able to estimate the quality of the retrieved programs. The gathering of information about potential reusable software components is quite doable. However, the metrics used to assess software reusability are currently very subjective and much work needs to be done to quantify and improve this assessment process.

Elements of a compound index that act as query and program descriptors in the combination retrieval system can be easily expanded by including more structural data that is extracted from search queries and programs in a repository. For example, besides identifying Singleton, Composite and Observer design patterns in a program, several other design patterns as described by Gamma et al. [1] can be identified and incorporated into the compound index. The trade-off between flexibility and complexity that arises from generality in using design patterns is another issue that clearly needs to be explored more fully. Perhaps, information regarding the cost of adapting a program into the application in context would also be valuable information that would help users of the repository in selecting relevant programs.

Furthermore, other promising means of functional descriptors can be employed. These include incorporating a thesaurus (e.g Wordnet [61]) that identifies synonyms of terms extracted from a program. With this, the function of a program is better represented as several terms can be used to represent a single function.

8.2.2 Supporting More Complicated Indexing and Retrieval Mechanisms

How well a software component retrieval system such as the proposed combination based retrieval system can deliver relevant programs depends on how well it can capture the programming task from existing information provided by the user. The proposed system has tried to capture the task based on the existing program written by the developer. The structure relationships that exist in a program are revealed through design patterns and software metrics. However, there may be other information that can be utilized, for example the software architecture [163, 164].

As a greater number of indices are incorporated into the compound index, a similarity measurement that includes weighting schema can also be introduced. This is to help users to rank the importance of descriptors used in identifying the most relevant programs. If a developer requires examples of programs that illustrate a particular design pattern but does not emphasize the function of the program, then higher weight can be given to the indices representing a design pattern. With such an approach, the retrieval system will be more flexible as it can easily be modified to represent a users' search requirements and preferences.

The program retrieval system can also explore the affinity of programs to deliver relevant programs. The affinity of two programs is the likelihood that two programs will be used in the same application. The repository system can deliver programs that have a high affinity with the programs used by programmers in their current application. For example, if a developer shows his interest towards a particular program that has been retrieved from a search, the retrieval system will then present him with program(s) that are closely related to it. There are two possible approaches to computing the affinity of two programs: coupling-based or statistics-based. The coupling-based approach looks at how tightly two programs are structurally connected. Two components are more likely to appear together if they both access common data, or if the data type output by one program is the same as the data type input by another. The statistics based approach looks at how often two programs appear together in a single application. Such co-occurrence information could be obtained in a way similar to the automatic thesaurus construction in information retrieval systems by treating programs as documents.

BIBLIOGRAPHY

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[2] Carma McClure. *The Three Rs Of Software Automation: Reengineering Repository Reusability.* Prentice Hall, 1992.

[3] Christian Lindig. Concept-based component retrieval. In J. Kohler, F. Giunchiglia, C. Green, and C. Walther, editors, *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 21–25, 1995.

[4] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of ACM*, 34(5):88–97, May 1991.

[5] Yong Yang, Weishi Zhang, Xiuguo Zhang, and Jinyu Shi. A weighted ranking algorithm for facet-based component retrieval system. In *ACST'06: Proceedings of the 2nd IASTED International Conference on Advances in Computer Science and Technology*, pages 274–279, Anaheim, CA, USA, 2006. ACTA Press.

[6] Hongjian Niu and Young Park. An execution-based retrieval of object-oriented components. In *ACM-SE 37: Proceedings of the 37th Annual Southeast Regional Conference*, page 18, New York, NY, USA, 1999. ACM Press.

[7] Vijayan Sugumaran and Veda C. Storey. A semantic-based approach to component retrieval. *The DATA BASE for Advances in Information Systems*, 34(3):8–24, 2003.

[8] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE 2002: Proceedings of the 24th International Conference on Software Engineering*, pages 513–523, New York, NY, USA, 2002. ACM Press.

[9] Tower of hanoi. http://www.cut-the-knot.org/recurrence/hanoi.shtml. last accessed on September 12, 2007.

[10] William B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

[11] Gerhard Fischer, Scott Henninger, and David Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th International Conference on Software Engineering*, pages 318–328, Austin, Texas, United States, 1991.

[12] Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. Guru: Information retrieval for reuse. In P.Hall, editor, *Landmark Contributions in Software Reuse and Reverse Engineering, Uni-com Seminars.* 1994.

[13] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, January 1999.

[14] Microsoft. Com: Component object model technologies. http://www.microsoft.com/com/default.mspx. last accessed on September 12, 2007.

[15] Java beans. http://java.sun.com/products/javabeans/. last accessed on September 12, 2007.

[16] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions Programming Language Systems*, 19(2):292–333, 1997.

[17] Urs Holzle. Integrating independently-developed components in object-oriented languages. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 36–56, London, UK, 1993. Springer-Verlag.

[18] TIOBE Software. Tiobe programming community index for march 2007. http://www.tiobe.com/tpci.htm, 2007. last accessed on September 12, 2007.

[19] Rosario Girardi. *Classification and Retrieval of Software through their Description in Natural Language.* Phd thesis, Computer Science Department, University of Geneva, 1995.

[20] Daniel Lucredio, Alan Gavioli, Antonio F. do Prado, and Mauro Biajiz. Component retrieval using metric indexing. In *Proceedings IEEE International Conference on Information Reuse and Integration (IRI 2004)*, pages 79–84, Las Vegas, November 8-10 2004. IEEE Systems, man and Cybernetics Society.

[21] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[22] Andrian Marcus. *Semantic Driven Program Analysis.* PhD thesis, School of Computer Science, Kent State University, August 2003.

[23] Google code search. http://www.google.com/codesearch. last accessed on September 12, 2007.

[24] Koders search engine. http://www.koders.com/. last accessed on September 12, 2007.

[25] Paul Santanul and Prakash Atul. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[26] Santanu Paul. Scruple: a reengineer's tool for source code search. In *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 329–346. IBM Press, 1992.

[27] Linda Mary Wills. Automated program recognition by graph parsing. Master's thesis, Massachusetts Institute of Technology, 1992.

[28] Secil Ugurel, Robert Krovetz, and C. Lee Giles. What's the code?: Automatic classification of source code archives. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 632–638. ACM Press, 2002.

[29] Sourceforge. http://sourceforge.net/. last accessed on September 12, 2007.

[30] Freshmeat. http://freshmeat.net/. last accessed on September 12, 2007.

[31] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 357–370, New York, NY, USA, 2000. ACM Press.

[32] Yuhanis Yusof and Omer F. Rana. Template mining in source code digital libraries. In *Proceedings of the International Workshop on Mining Software*

Repositories, 26th International Conference on Software Engineering, pages 122–126, Edinburgh, UK, 2004.

[33] Yuhanis Yusof and Omer F. Rana. Supporting program indexing and querying in source code digital libraries. In Brian Henderson-Sellers and Michael Winikoff, editors, *Proceedings of the 7th International Workshop on Agent-Oriented Information Systems, co-located with the Fourth International Conference on Autonomous Agents and Multi-Agent Systems(AAMAS05)*, pages 34–41, Utrecht, The Netherlands, 2005.

[34] Yuhanis Yusof and Omer F. Rana. Supporting program indexing and querying in source code digital libraries, revised selected papers. In Manuel Kolp, Paolo Bresciani, Brian Henderson-Sellers, and Michael Winikoff, editors, *Agent-Oriented Information Systems III*, volume 3529 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2006.

[35] Yuhanis Yusof and Omer F. Rana. Integration of descriptors for software component retrieval. In *2nd International Conference on Knowledge Science, Engineering and Management (accepted)*, Lecture Notes in Computer Science. Springer, November 2007.

[36] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[37] Uml documentation. http://umlcenter.visual-paradigm.com/. last accessed on September 12, 2007.

[38] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *The Experience Factory*, pages 469–476. John Wiley and Sons, Inc, 1994.

[39] Eduardo Ostertag, James Hendler, Rubn Prieto Daz, and Christine Braun. Computing similarity in a reuse system: An al-based approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(3):205–228, 1992.

[40] Johann Schumann and Bernd Fischer. Nora/hammr: Making deduction-based software component retrieval practical. In *Proceedings of the 1997 International Conference on Automated Software Engineering(ASE'97)*, pages 246–254, Lake Tahoe, CA, 1997.

[41] John Penix and Perry Alexander. Using formal specifications for component retrieval and reuse. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, pages 356–365, 1998.

[42] Bernd Fischer. *Deduction-Based Software Component Retrieval.* PhD thesis, Faculty of Mathematics and Informatics, University of Passau, Germany, June 2001.

[43] Sathit Nakkrasae and Peraphon Sophatsathit. A formal approach for specification and classification of software components. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 773–780. ACM Press, New York, 2002.

[44] Hafedh Mili, Estelle Ah-Ki, Robert Godin, and Hamid Mcheick. An experiment in software component retrieval. *Information and Software Technology*, 45:633–649, 2003.

[45] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineer-*

ing (WCRE'04), pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[46] R. T. Mittermeir and W. Rossak. Software bases and software archives: alternatives to support software reuse. In *ACM '87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow*, pages 21–28, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[47] M. R. Girardi and B. Ibrahim. Automatic indexing of software artifacts. In *Proceedings of 3rd. International Conference on Software Reuse*, pages 24–32, Rio de Janeiro, Brazil, 1994.

[48] Mingyang Gu, Agnar Aamodt, and Xin Tong. Component retrieval using conversational case-based reasoning. pages 259–271, 2005.

[49] James C. French, Allison L. Powell, Fredric C. Gey, and Natalia Perelman. Exploiting a controlled vocabulary to improve collection selection and retrieval effectiveness. In *CIKM '01: Proceedings of the 10th International Conference on Information and Knowledge Management*, pages 199–206, 2001.

[50] M. R. Girardi and B. Ibrahim. Using english to retrieve software. *Journal of System Software*, 30(3):249–270, 1995.

[51] Lamia Labed Jilani, Jules Desharnais, Marc Frappier, Rym Mili, and Ali Mili. Retrieving software components that minimize adaptation effort. In *Automated Software Engineering*, pages 255–277, 1997.

[52] Steven Atkinson and Roger Duke. A methodology for behavioural retrieval from class libraries. Technical Report 94-28, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Australia, September 1994.

[53] Andy Podgurski and Lynn Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions Software Engineering Methodology*, 2(3):286–303, 1993.

[54] Dewayne E. Perry and Steven S. Popovitch. Inquire: Predicate-based use and reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 144–151, September 1993.

[55] Ali Mili, Rym Mili, and Roland T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445–460, July 1994.

[56] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A key to reuse. In *Proceedings of SIGSOFT*, pages 7–10, Los Angeles, California, 1993.

[57] Luqi and Jiang Guo. Toward automated retrieval for a software component repository. In *Proceedings of ECBS IEEE Conference and Workshop Engineering of Computer-Based Systems*, pages 99–105, March 1999.

[58] Gerard Salton and M. E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM*, 15(1):8–36, 1968.

[59] Rudolf Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, 1982.

[60] Scott Deerwester, Susan T. Dumais, George W. Furnas, and Thomas K. Landauer. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, pages 391–407, 1990.

[61] Christiane Fellbaum, editor. *WordNet. An electronic lexical database.* Cambridge, MA: MIT Press;, 1998.

[62] M. R. Girardi and Bertrand Ibrahim. A similarity measure for retrieving software artifacts. In *The 6th International Conference on Software Engineering and Knowledge Engineering (SEKE'94)*, pages 478–485, Jurmala, Latvia, 1994.

[63] A. Aamodt. A knowledge representation system for integration of general and casespecific knowledge. In *Proceedings from IEEE TAI-94, International Conference on Tools with Artificial Intelligence*, pages 836–839, 1994.

[64] Richard C. Waters. The programmer's apprentice: a session with kbemacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, 1985.

[65] Charles Rich. Inspection methods in programming: Cliches and plans. Technical report, Cambridge, MA, USA, 1987.

[66] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988.

[67] Richard C. Waters and Yang Meng Tan. Toward a design apprentice: supporting reuse and evolution in software design. *SIGSOFT Software Engineering Notes*, 16(2):33–34, 1991.

[68] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[69] Steven John Metsker. *Design Patterns Java Workbook.* Addison-Wesley, 2002.

[70] Lutz Prechelt. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–882, 1998.

[71] Jagdish Bansiya. Automating design-pattern identification: DP++ is a tool for C++ programs. *Dr. Dobb's Journal*, 1998.

[72] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering*, pages 226–235, Los Angeles, California, 1999.

[73] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*, pages 29–32. IEEE Computer Society Press, May 2003.

[74] D.M. Shawky, S.K. Abd-El-Hafiz, and A. L. El-Sedeek. A dynamic approach for the identification of object-oriented design patterns. In Peter Kokol, editor, *Software Engineering*, Innsbruck, Austria, February 2005.

[75] Niklas Pettersson. Measuring precision for static and dynamic design pattern recognition as a function of coverage. In *WODA '05: Proceedings of the 3rd International Workshop on Dynamic Analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[76] Jason McC. Smith and David Stotts. Spqr: Flexible automated design pattern extraction from source code. In *Proceedings 18th IEEE International Conference on Automated Software Engineering*, pages 215–224, October 2003.

[77] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns for high performance computing. In *Workshop on Patterns in High Performance Computing*, University of Illinois at Urbana-Champaign, 2005.

[78] Jorg Niere, Wilhelm Schfer, Jrg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering*, pages 338–348. ACM Press, 2002.

[79] Gcc. Gcc, the gnu compiler collection. http://gcc.gnu.org/. last accessed on September 12, 2007.

[80] William W.McCune. *OTTER 3.0 Reference Manual and Guide*, 1994.

[81] SUN. java.net. http://javacc.dev.java.net/. last accessed on September 12, 2007.

[82] Premkumar T. Devanbu. Genoa: a customizable language- and front-end independent code analyzer. In *ICSE '92: Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, New York, NY, USA, 1992. ACM Press.

[83] Oscar Nierstrasz, Sander Tichelaar, and Serge Demeyer. Cdif as the interchange format between reengineering tools. In *OOPSLA'98 Workshop*, Vancouver, October 1998.

[84] Norman E Fenton. *Software Metrics: A Rigorous Approach.* Chapman & Hall, 1994.

[85] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, 1991.

[86] Hironori Washizaki and Yoshiaki Fukazawa. *Software Reuse: Methods, Techniques and Tools*, chapter Component-Extraction-Based Search System for Object-Oriented Programs, pages 254–263. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004.

[87] Sen-Tarng Lai and Chien-Chiao Yang. A software metric combination model for software reuse. In *APSEC '98: Proceedings of the 5th Asia Pacific Software Engineering Conference*, pages 70–77, Washington, DC, USA, 1998. IEEE Computer Society.

[88] Thomas J. McCabe. A complexity measure. *IEEE Transactions Software Eng.*, 2(4):308–320, 1976.

[89] Halstead M.H. *Elements of Software Science.* Elsevier, New York, 1977.

[90] Richard E. Fairley. *Software Engineering Concepts.* McGraw-Hill, Inc., New York, NY, USA, 1986.

[91] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[92] Hironori Washizaki, Hirokazu Yamamoto, and Yoshiaki Fukazawa. A metrics suite for measuring reusability of software components. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, pages 211–223, Washington, DC, USA, 2003. IEEE Computer Society.

[93] Linda Rosenberg, Ted Hammer, and Jack Shaw. Software metrics and reliability. In *Proceedings of the 9th International Symposium*, Germany, 1998.

[94] Octavian Paul Rotaru and Marian Dobre. Reusability metrics for software components. In *The third ACS/IEEE International Conference on Computer System and applications*, pages 24–31, Cairo,Egypt, 2005.

[95] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, January 1996.

[96] Andrew Glover. Code improvement through cyclomatic complexity. http://www.onjava.com/pub/a/onjava/2004/06/16/ccunittest.html. last accessed on September 12, 2007.

[97] Apache Maven Project. Metrics. http://maven.apache.org/reference/metrics.html. last accessed on September 12, 2007.

[98] Arthur J. Riel. *Object-Oriented Deisgn Heuristics.* Addison-Wesley Longman Publishing, 1996.

[99] Khaled M. Hammouda and Mohamed S. Kamel. Phrase-based document similarity based on an index graph model. In *2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 203–210, Maebashi City, Japan, December 2002.

[100] A. Mili, R. Mili, and R. T. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349–414, 1998.

[101] Wikipedia encyclopedia. Linear combination. http://en.wikipedia.org/wiki/Linear_combination. last accessed on September 12, 2007.

[102] George Spanoudakis and Panos Constantopoulos. Measuring similarity between software artifacts. In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, pages 387–394, Skokie, 1994.

[103] Julie Pallant. *SPSS Survival Manual*, chapter 6, pages 53–54. Open University Press, 2004.

[104] DTREG Software For Predictive Modeling and Forecasting. Svm - support vector machines. http://www.dtreg.com/svm.htm. last accessed on September 12, 2007.

[105] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[106] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, IT-13(1):21–27, 1967.

[107] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106.

[108] Salvatore Ruggieri. Efficient c4.5. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):438-444, 2002.

[109] Christopher D. Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing.* The MIT Press, 1st edition edition, June 1999.

[110] Yiming Yang and Xin Liu. A re-examination of text categorization methods. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 42–49, New York, NY, USA, 1999. ACM Press.

[111] Wei Li and Harry Delugach. Software metrics and application domain complexity. In *Fourth Asia-Pasific Software Engineering and International Computer Science Conference*, pages 513–514, 1997.

[112] Oh-Woog Kwon and Jong-Hyeok Lee. Text categorization based on k-nearest neighbor approach for web site classification. *Information Processing Management*, 39(1):25–44, 2003.

[113] Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar. Text categorization using weight adjusted k -nearest neighbor classification. *Lecture Notes in Computer Science*, 2035:53–59, 2001.

[114] Qi Tian, Ying Wu, and Thomas S. Huang. Incorporate discriminant analysis with EM algorithm in image retrieval. In *IEEE International Conference on Multimedia and Expo(I)*, pages 299–302, 2000.

[115] Daniel L. Swets and Juyang Weng. Hierarchical discriminant analysis for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(5):386–401, 1999.

[116] Kien-Ping Chung and Chun Che Fun. A hierarchical nonparametric discriminant analysis approach for a content-based image retrieval system. In *ICEBE '05: Proceedings of the IEEE International Conference on e-Business Engineering*, pages 346–351, Washington, DC, USA, 2005. IEEE Computer Society.

[117] Eriko Nurvitadhi, Wing Wah Leung, and Curtis Cook. Do class comments aid java program understanding. In *33rd ASEE/IEEE Frontiers in Education Conference*, pages 13–17, November 5-8 2003.

[118] UNIX. Unix programmer's manual. http://cm.bell-labs.com/cm/cs/who/dmr/1stEdman.html. last accessed on September 12, 2007.

[119] Bruno Caprile and Paolo Tonella. Nomen est omen:analyzing the language of function identifiers. In *Proceedings of the Working Conference on Reverse Engineering, WCRE'99*, pages 112–122, Atlanta, Georgia, USA, October 1999.

[120] Mikael Lindvall and Kristian Sandahl. Practical implications of traceability. *Software Practice Experience*, 26(10):1161–1180, 1996.

[121] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[122] Harold Rodriguez. Tips for linux. http://www.codecoffee.com/tipsforlinux/articles2/043.html. last accessed on September 12, 2007.

[123] Bruce Eckel. *Thinking in Java*. Prentice Hall, 2002.

[124] SUN. Jdk. http://java.sun.com/javase/downloads/index.jsp. last accessed on September 12, 2007.

[125] Google. http://www.google.com. last accessed on September 12, 2007.

[126] Stopword list. http://www.lextek.com/manuals/onix/stopwords1.html. last accessed on September 12, 2007.

[127] Thierry Lecroq. Sequence comparison. http://www-igm.univ-mlv.fr/~lecroq/seqcomp/. last accessed on September 12, 2007.

[128] SUN Microsystems. Awt the sun java abstract window toolkit. http://java.sun.com/products/jdk/awt. last accessed on September 12, 2007.

[129] Jhotdraw as open-source project. http://www.jhotdraw.org/. last accessed on September 12, 2007.

[130] Yann-Gael Gueheneuc and Narendra Jussien. Using explanations for design patterns identification. In *Proceedings of IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, pages 57–64, August 2001.

[131] Claudia Raibulet and Francesca Arcelli. Program comprehension and design pattern detection: An experience report. In *The international workshop on Object-Oriented Reengineering*, Nantes, France, July 2006.

[132] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *ASE '06: Proceedings of the 21st IEEE International Con-*

ference on Automated Software Engineering (ASE'06), pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[133] C.J. Van Rijsbergen. *Information Retrieval*, chapter 6, pages 65–132. Butterworths, 1975.

[134] Pattern stories:java awt. http://wiki.cs.uiuc.edu/PatternStories/JavaAWT. last accessed on September 12, 2007.

[135] Dirk Riehle. Composite design patterns. In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 218–228, 1997.

[136] John R. Anderson, Robert Farell, and Ron Sauers. Learning to program in lisp. *Cognitive Science*, 8:87–130, 1984.

[137] Richard Creps, Mark A. Simos, and Ruben Prieto-Diaz. The stars conceptual framework for reuse processes, software technology for adaptable, reliable systems (stars). Technical report, DARPA, 1992.

[138] William R. Klecka. *Discriminant Analysis*. Sage Publications, first edition edition, 1980.

[139] Tjen-Sien Lim, Wei-Yin Loh, and Yu-Shan Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, 40(3):203–228, 2000.

[140] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. Mining very large databases. *Computer*, 32(8):38–45, 1999.

[141] Hans-Peter Kriegel, Alexey Pryakhin, and Matthias Schubert. *Database Systems for Advanced Applications*, chapter Multi-represented kNN-Classification

for Large Class Sets, pages 511–522. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, May 2005.

[142] Tim Littlefair. CCCC – C and C++ code counter. http://cccc.sourceforge.net/. last accessed on September 12, 2007.

[143] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Hillsdale, NJ. 2nd edition edition, 1998.

[144] Frank Owen and Ron Jones. *Statistics*. Trans-Atlantic Publications, 4th edition edition, 1994.

[145] University of Waikato. Weka. http://www.cs.waikato.ac.nz/ml/weka. last accessed on September 12, 2007.

[146] Spss. http://www.spss.com. last accessed on September 12, 2007.

[147] Christophet M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, USA, 1996.

[148] David Hull. Using statistical testing in the evaluation of retrieval experiments. In *SIGIR '93: Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 329–338, New York, NY, USA, 1993. ACM Press.

[149] C. T. Yu and Gerard Salton. Precision weighting - an effective automatic indexing method. *Journal of the ACM*, 23(1):76–88, January 1976.

[150] Holger Billhardt, Daniel Borrajo, and Victor Maojo. A context vector model for information retrieval. *Journal of American Society Information Science Technology*, 53(3):236–249, 2002.

[151] John F. Sowa. Ontology, metadata, and semiotics. In *ICCS '00: Proceedings of the Linguistic on Conceptual Structures*, pages 55–81, London, UK, 2000. Springer-Verlag.

[152] Gerard Salton and C. S. Yang. on the specification of term values in automatic indexing. *Journal of Documentation*, 29:351–372, 1973.

[153] Gang Qian, Shamik Sural, Yuelong Gu, and Sakti Pramanik. Similarity between euclidean and cosine angle distance for nearest neighbor queries. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1232–1237, New York, NY, USA, 2004. ACM Press.

[154] Rabia Gulcin Demirci, Vildan Kismir, and Yiltan Bitirim. An evaluation of popular search engines on finding turkish documents. In *Proceedings of the 2nd International Conference on Internet and Web Applications and Services*, pages 61–70, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[155] Ian Soboroff, Charles Nicholas, and Patrick Cahan. Ranking retrieval systems without relevance judgments. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 66–73, New York, NY, USA, 2001. ACM Press.

[156] Yunwen Ye. *Supporting Component-Based Software Development with Active Component Repository Systems.* Phd thesis, Department of Computer Science, University of Colorado, 2001.

[157] Stanley L. Sclove. Notes on liker scales. http://www.uic.edu/classes/idsc/ids270sls/likert.htm, 2001. last accessed on September 12, 2007.

[158] Robert G. Fichman and Chris F. Kemerer. Object technology and reuse: Lessons from early adopters. *Computer*, 30(10):47–59, 1997.

[159] Arun Sen. The role of opportunism in the software design reuse process. *IEEE Transactions on Software Engineering*, 23(7):418–436, 1997.

[160] Ruben Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, April 1990.

[161] Marco Scotto, Alberto Sillitti, Giancarlo Succi, and Tullio Vernazza. A relational approach to software metrics. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1536–1540, New York, NY, USA, 2004. ACM Press.

[162] M.M. Lehman, D.E. Perry, and J.F. Ramil. Implications of evolution metrics on software maintenance. *International conference on software maintenance*, pages 208–217, 1998.

[163] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1:1–40, 1993.

[164] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions Softw. Eng.*, 26(1):70–93, 2000.

Appendix A

J48 Decision Tree

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     metricsDatabaseGraphicsWeka12attr_NO_LOC2
Instances:    584
Attributes:   13
              LOC
              MVG
              COM
              WMC1
              DIT
              CBO
              Fivis
              Ficon
              Fiincl
              vis
              con
              inc
              CLASS
Test mode:    evaluate on training data

=== Classifier model (full training set) ===

J48 pruned tree
------------------

DIT <= 2
|   COM <= 3
|   |   Ficon <= 1
|   |   |   Fivis <= 1
|   |   |   |   COM <= 0
|   |   |   |   |   WMC1 <= 1: 1 (3.0)
|   |   |   |   |   WMC1 > 1: 2 (2.0)
|   |   |   |   COM > 0: 1 (2.0)
|   |   |   Fivis > 1
|   |   |   |   LOC <= 29
|   |   |   |   |   Fivis <= 2: 2 (41.0/1.0)
|   |   |   |   |   Fivis > 2
|   |   |   |   |   |   COM <= 1
|   |   |   |   |   |   |   WMC1 <= 2: 1 (5.0/1.0)
|   |   |   |   |   |   |   WMC1 > 2: 2 (2.0)
|   |   |   |   |   |   COM > 1: 2 (2.0)
|   |   |   |   LOC > 29
|   |   |   |   |   DIT <= 1
|   |   |   |   |   |   COM <= 1: 2 (11.0/1.0)
|   |   |   |   |   |   COM > 1
|   |   |   |   |   |   |   MVG <= 3: 1 (8.0/1.0)
|   |   |   |   |   |   |   MVG > 3: 2 (4.0)
|   |   |   |   |   DIT > 1: 1 (4.0)
|   |   Ficon > 1: 1 (8.0/1.0)
|   COM > 3
|   |   DIT <= 1
|   |   |   COM <= 129
|   |   |   |   con <= 0
|   |   |   |   |   MVG <= 3
|   |   |   |   |   |   Ficon <= 2
|   |   |   |   |   |   |   COM <= 20
|   |   |   |   |   |   |   |   vis <= 4
|   |   |   |   |   |   |   |   |   Fivis <= 2
|   |   |   |   |   |   |   |   |   |   DIT <= 0
|   |   |   |   |   |   |   |   |   |   |   WMC1 <= 2
|   |   |   |   |   |   |   |   |   |   |   |   LOC <= 25: 1 (7.0)
|   |   |   |   |   |   |   |   |   |   |   |   LOC > 25: 2 (3.0)
|   |   |   |   |   |   |   |   |   |   |   WMC1 > 2: 2 (3.0/1.0)
|   |   |   |   |   |   |   |   |   |   DIT > 0: 2 (17.0/3.0)
|   |   |   |   |   |   |   |   |   Fivis > 2
|   |   |   |   |   |   |   |   |   |   WMC1 <= 2: 1 (23.0/1.0)
|   |   |   |   |   |   |   |   |   |   WMC1 > 2
|   |   |   |   |   |   |   |   |   |   |   CBO <= 4: 2 (2.0)
|   |   |   |   |   |   |   |   |   |   |   CBO > 4
|   |   |   |   |   |   |   |   |   |   |   |   COM <= 13
|   |   |   |   |   |   |   |   |   |   |   |   |   MVG <= 1: 1 (3.0)
|   |   |   |   |   |   |   |   |   |   |   |   |   MVG > 1
|   |   |   |   |   |   |   |   |   |   |   |   |   |   WMC1 <= 4: 1 (3.0)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   WMC1 > 4: 2 (2.0)
|   |   |   |   |   |   |   |   |   |   |   |   COM > 13: 2 (4.0/1.0)
|   |   |   |   |   |   |   |   vis > 4: 2 (13.0/2.0)
|   |   |   |   |   |   |   COM > 20
|   |   |   |   |   |   |   |   WMC1 <= 4: 1 (19.0)
|   |   |   |   |   |   |   |   WMC1 > 4
|   |   |   |   |   |   |   |   |   LOC <= 153
|   |   |   |   |   |   |   |   |   |   LOC <= 14: 2 (2.0)
|   |   |   |   |   |   |   |   |   |   LOC > 14: 1 (46.0/7.0)
|   |   |   |   |   |   |   |   |   LOC > 153: 2 (5.0/1.0)
|   |   |   |   |   |   Ficon > 2: 1 (8.0)
|   |   |   |   |   MVG > 3
|   |   |   |   |   |   DIT <= 0
|   |   |   |   |   |   |   CBO <= 5: 2 (29.0/12.0)
|   |   |   |   |   |   |   CBO > 5: 1 (34.0/7.0)
|   |   |   |   |   |   DIT > 0
|   |   |   |   |   |   |   Fivis <= 7
|   |   |   |   |   |   |   |   Ficon <= 2
|   |   |   |   |   |   |   |   |   COM <= 24
|   |   |   |   |   |   |   |   |   |   LOC <= 130: 2 (16.0/1.0)
|   |   |   |   |   |   |   |   |   |   LOC > 130: 1 (4.0/1.0)
|   |   |   |   |   |   |   |   |   COM > 24
|   |   |   |   |   |   |   |   |   |   WMC1 <= 3: 1 (5.0)
|   |   |   |   |   |   |   |   |   |   WMC1 > 3
|   |   |   |   |   |   |   |   |   |   |   Ficon <= 1
|   |   |   |   |   |   |   |   |   |   |   |   Fivis <= 5
|   |   |   |   |   |   |   |   |   |   |   |   |   WMC1 <= 7: 2 (4.0)
|   |   |   |   |   |   |   |   |   |   |   |   |   WMC1 > 7
|   |   |   |   |   |   |   |   |   |   |   |   |   |   vis <= 4: 1 (3.0)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   vis > 4: 2 (3.0)
|   |   |   |   |   |   |   |   |   |   |   |   Fivis > 5: 1 (9.0/2.0)
|   |   |   |   |   |   |   |   |   |   |   Ficon > 1
|   |   |   |   |   |   |   |   |   |   |   |   Fivis <= 6: 1 (2.0)
|   |   |   |   |   |   |   |   |   |   |   |   Fivis > 6: 2 (2.0)
|   |   |   |   |   |   |   |   Ficon > 2: 1 (3.0)
|   |   |   |   |   |   |   Fivis > 7: 2 (40.0/7.0)
|   |   |   |   con > 0
|   |   |   |   |   WMC1 <= 5: 1 (8.0)
|   |   |   |   |   WMC1 > 5
|   |   |   |   |   |   LOC <= 56: 2 (4.0)
|   |   |   |   |   |   LOC > 56: 1 (7.0/1.0)
|   |   |   COM > 129
|   |   |   |   Ficon <= 1: 1 (23.0)
|   |   |   |   Ficon > 1
|   |   |   |   |   COM <= 225: 2 (2.0)
|   |   |   |   |   COM > 225: 1 (2.0)
|   |   DIT > 1
|   |   |   WMC1 <= 13: 1 (63.0/4.0)
|   |   |   WMC1 > 13
|   |   |   |   vis <= 256: 2 (5.0/1.0)
|   |   |   |   vis > 256: 1 (4.0)
DIT > 2: 1 (60.0)

Number of Leaves  :     51

Size of the tree :      101
```

Appendix B

Statistical Result - Program Classification using Testing Data Set

xplore

LASS_DB

Case Processing Summary

		Cases					
		Valid		Missing		Total	
	CLASS_DB	N	Percent	N	Percent	N	Percent
PRE_DB	1	48	100.0%	0	.0%	48	100.0%
	2	51	100.0%	0	.0%	51	100.0%
	3	37	100.0%	0	.0%	37	100.0%
REC_DB	1	48	100.0%	0	.0%	48	100.0%
	2	51	100.0%	0	.0%	51	100.0%
	3	37	100.0%	0	.0%	37	100.0%

Descriptives

	CLASS_DB			Statistic	Std. Error
PRE_DB	1	Mean		.8066267	.01172527
		95% Confidence Interval for Mean	Lower Bound	.7830385	
			Upper Bound	.8302149	
		5% Trimmed Mean		.8004831	
		Median		.7777800	
		Variance		.007	
		Std. Deviation		.08123507	
		Minimum		.71429	
		Maximum		1.00000	
		Range		.28571	
		Interquartile Range		.0363775	
		Skewness		1.715	.343
		Kurtosis		1.665	.674
	2	Mean		.8356439	.00997456
		95% Confidence Interval for Mean	Lower Bound	.8156094	
			Upper Bound	.8556784	
		5% Trimmed Mean		.8303594	
		Median		.8055600	
		Variance		.005	
		Std. Deviation		.07123258	
		Minimum		.76190	
		Maximum		1.00000	
		Range		.23810	
		Interquartile Range		.0730200	
		Skewness		1.478	.333
		Kurtosis		.930	.656

Descriptives

	CLASS_DB			Statistic	Std. Error
PRE_DB	3	Mean		.5642641	.02809635
		95% Confidence Interval for Mean	Lower Bound	.5072820	
			Upper Bound	.6212461	
		5% Trimmed Mean		.5521125	
		Median		.5227300	
		Variance		.029	
		Std. Deviation		.17090340	
		Minimum		.33333	
		Maximum		1.00000	
		Range		.66667	
		Interquartile Range		.1374550	
		Skewness		1.700	.388
		Kurtosis		2.571	.759
REC_DB	1	Mean		.3828150	.03157383
		95% Confidence Interval for Mean	Lower Bound	.3192966	
			Upper Bound	.4463334	
		5% Trimmed Mean		.3828150	
		Median		.3828150	
		Variance		.048	
		Std. Deviation		.21874991	
		Minimum		.01563	
		Maximum		.75000	
		Range		.73437	
		Interquartile Range		.3828150	
		Skewness		.000	.343
		Kurtosis		-1.200	.674
	2	Mean		.4062525	.03252603
		95% Confidence Interval for Mean	Lower Bound	.3409221	
			Upper Bound	.4715830	
		5% Trimmed Mean		.4062525	
		Median		.4062500	
		Variance		.054	
		Std. Deviation		.23228232	
		Minimum		.01563	
		Maximum		.79688	
		Range		.78125	
		Interquartile Range		.4062500	
		Skewness		.000	.333
		Kurtosis		-1.200	.656

Descriptives

	CLASS_DB			Statistic	Std. Error
REC_DB	3	Mean		.2968750	.02780489
		95% Confidence Interval for Mean	Lower Bound	.2404841	
			Upper Bound	.3532659	
		5% Trimmed Mean		.2968750	
		Median		.2968750	
		Variance		.029	
		Std. Deviation		.16913055	
		Minimum		.01563	
		Maximum		.57813	
		Range		.56250	
		Interquartile Range		.2968750	
		Skewness		.000	.388
		Kurtosis		-1.200	.759

Tests of Normality

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	CLASS_DB	Statistic	df	Sig.	Statistic	df	Sig.
PRE_DB	1	.326	48	.000	.695	48	.000
	2	.282	51	.000	.746	51	.000
	3	.253	37	.000	.778	37	.000
REC_DB	1	.065	48	.200*	.956	48	.068
	2	.065	51	.200*	.956	51	.054
	3	.067	37	.200*	.956	37	.156

*. This is a lower bound of the true significance.

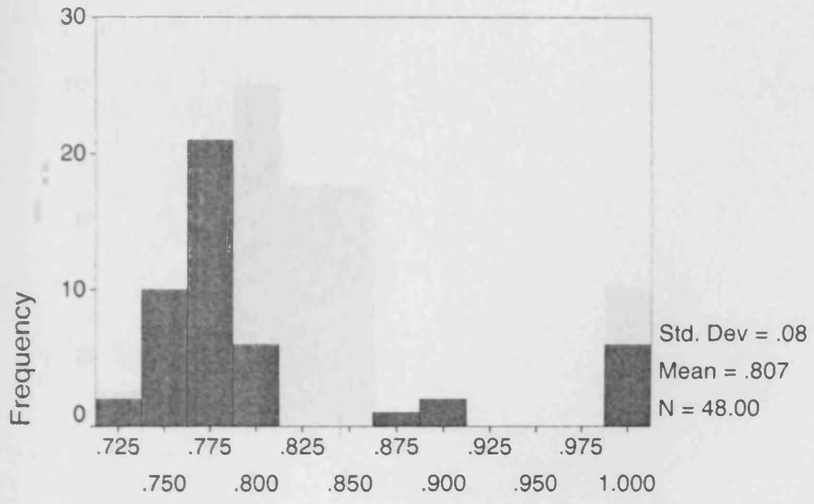
a. Lilliefors Significance Correction

PRE_DB

Histograms

Histogram

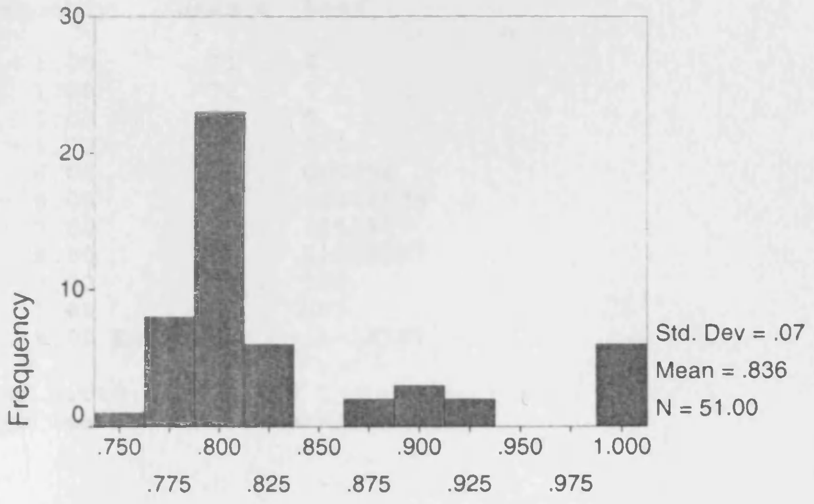
For CLASS_DB= 1



PRE_DB

Histogram

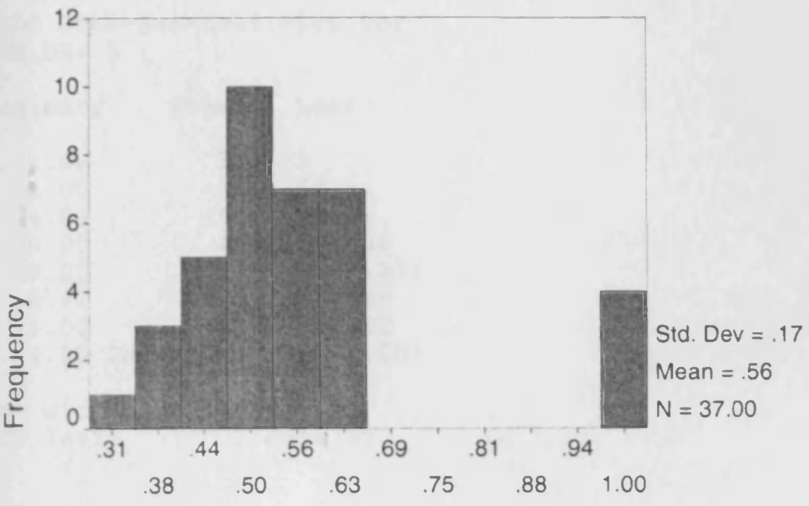
For CLASS_DB= 2



PRE_DB

Histogram

For CLASS_DB= 3



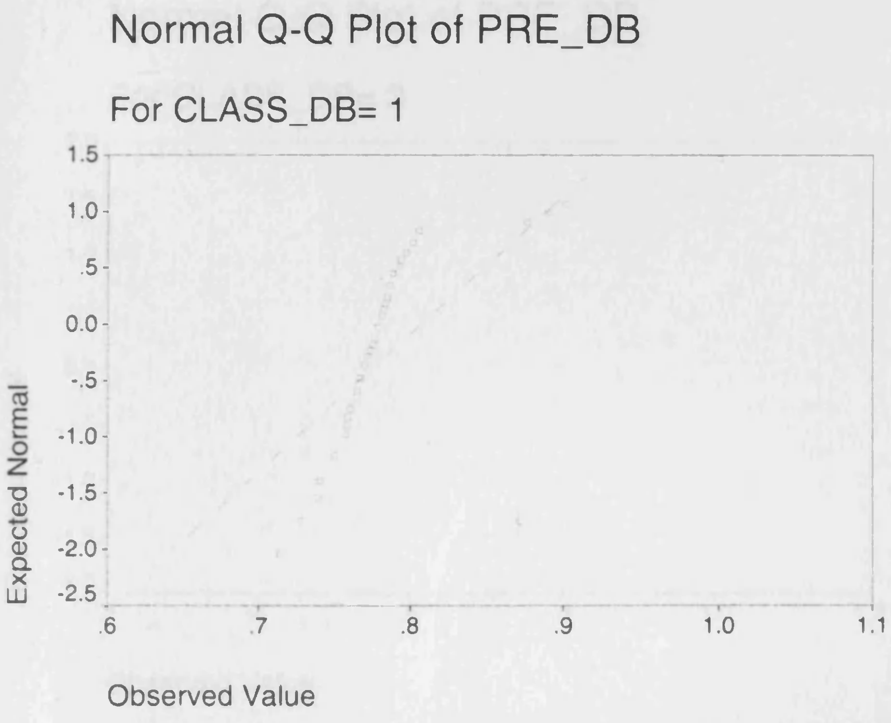
PRE_DB

Stem-and-Leaf Plots

```
PRE_DB Stem-and-Leaf Plot for
CLASS_DB= 1

 Frequency     Stem &  Leaf

     1.00        71 .  4
     1.00        72 .  7
     1.00        73 .  9
     1.00        74 .  0
     6.00        75 .  000788
     8.00        76 .  01446699
     7.00        77 .  1255777
     8.00        78 .  01235557
     3.00        79 .  125
     3.00        80 .  003
     9.00 Extremes     (>=.875)

 Stem width:     .01000
 Each leaf:        1 case(s)


PRE_DB Stem-and-Leaf Plot for
CLASS_DB= 2

 Frequency     Stem &  Leaf

    18.00         7 .  667778888999999999
    20.00         8 .  00000000000111112223
     3.00         8 .  678
     4.00         9 .  0012
     6.00 Extremes     (>=1.00)
```

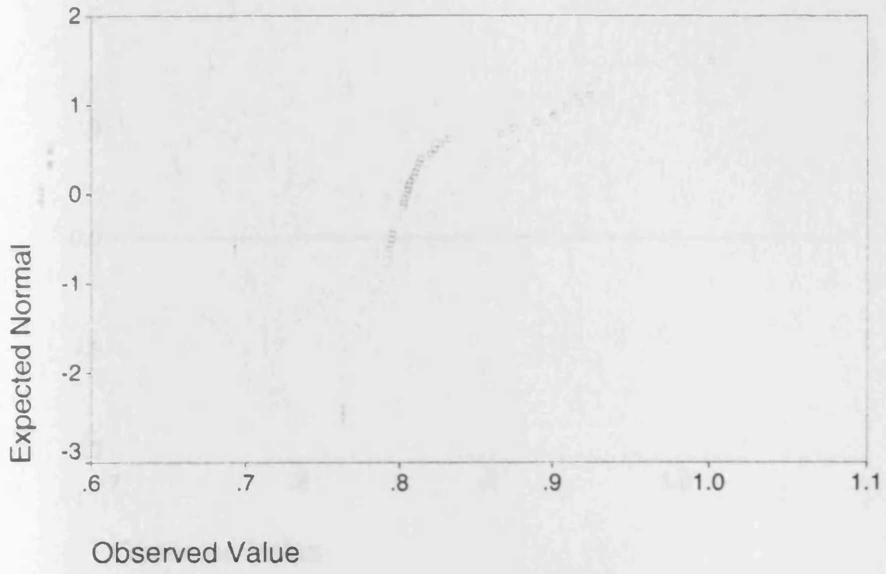
```
Stem width:     .10000
Each leaf:        1 case(s)


PRE_DB Stem-and-Leaf Plot for
CLASS_DB= 3

 Frequency    Stem &  Leaf

     1.00        3 .  3
     2.00        3 .  69
     4.00        4 .  0224
     6.00        4 .  666788
     8.00        5 .  00001234
     6.00        5 .  567889
     6.00        6 .  001122
     4.00 Extremes    (>=1.00)

 Stem width:     .10000
 Each leaf:        1 case(s)
```

Normal Q-Q Plots



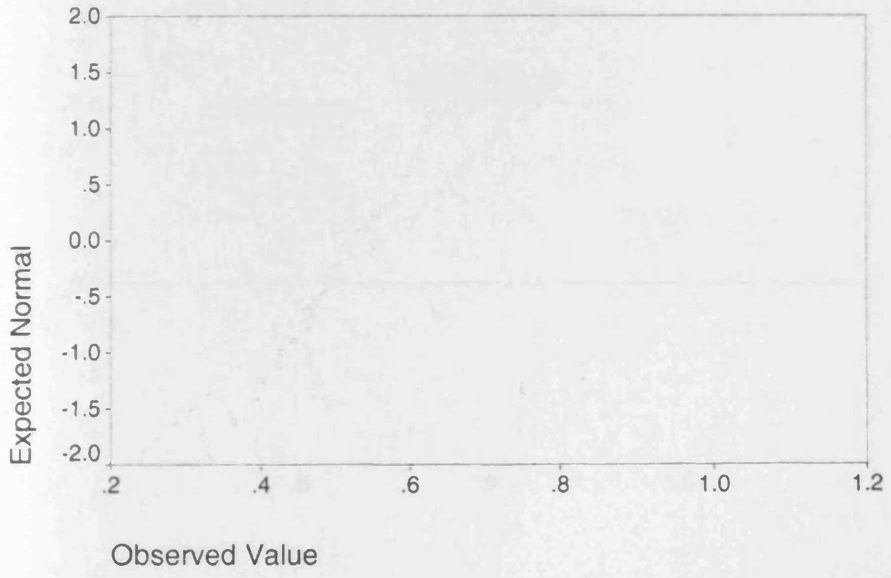
Normal Q-Q Plot of PRE_DB

For CLASS_DB= 2



Normal Q-Q Plot of PRE_DB

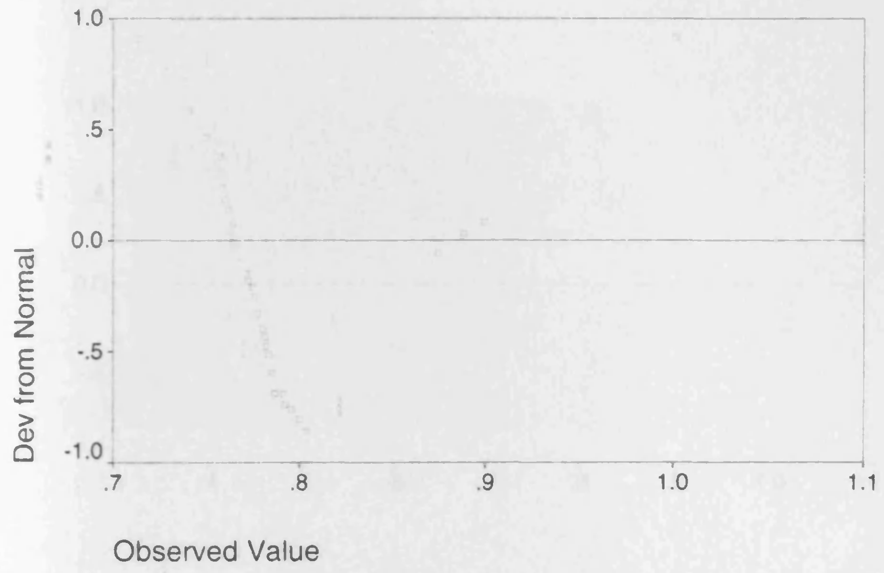
For CLASS_DB= 3



Detrended Normal Q-Q Plots

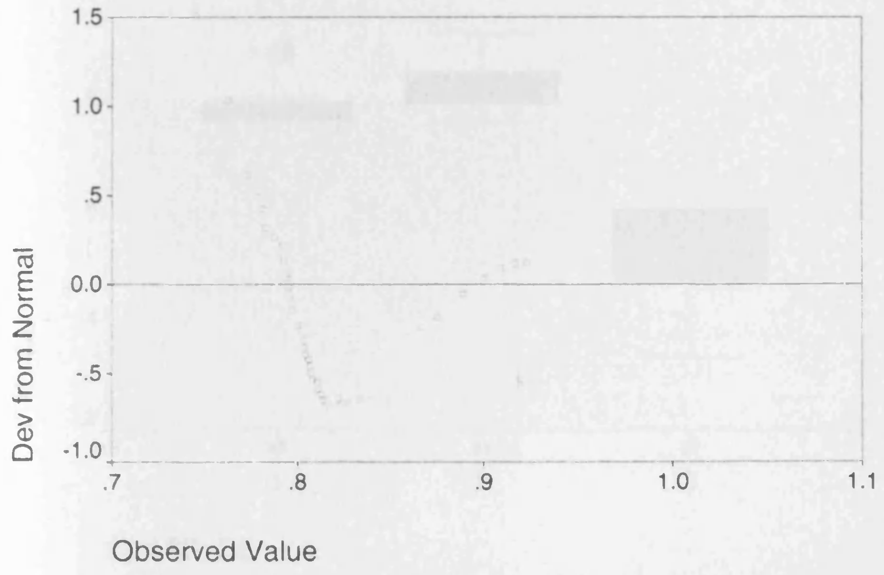
Detrended Normal Q-Q Plot of PRE_DB

For CLASS_DB= 1



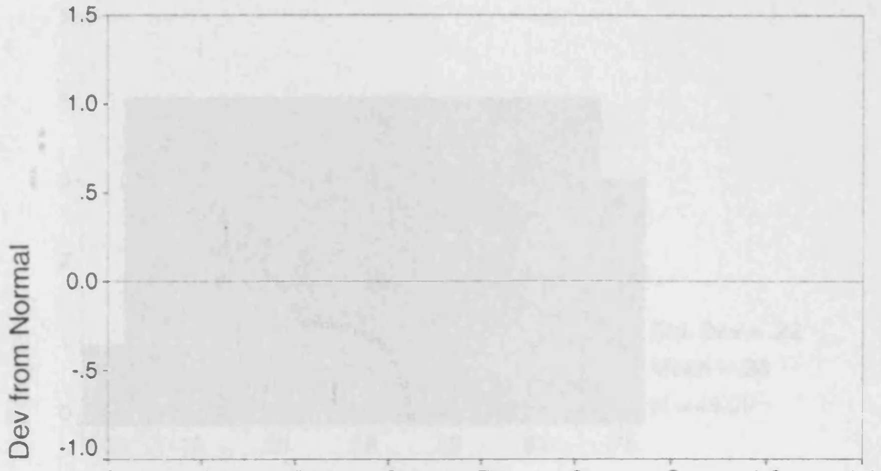
Detrended Normal Q-Q Plot of PRE_DB

For CLASS_DB= 2

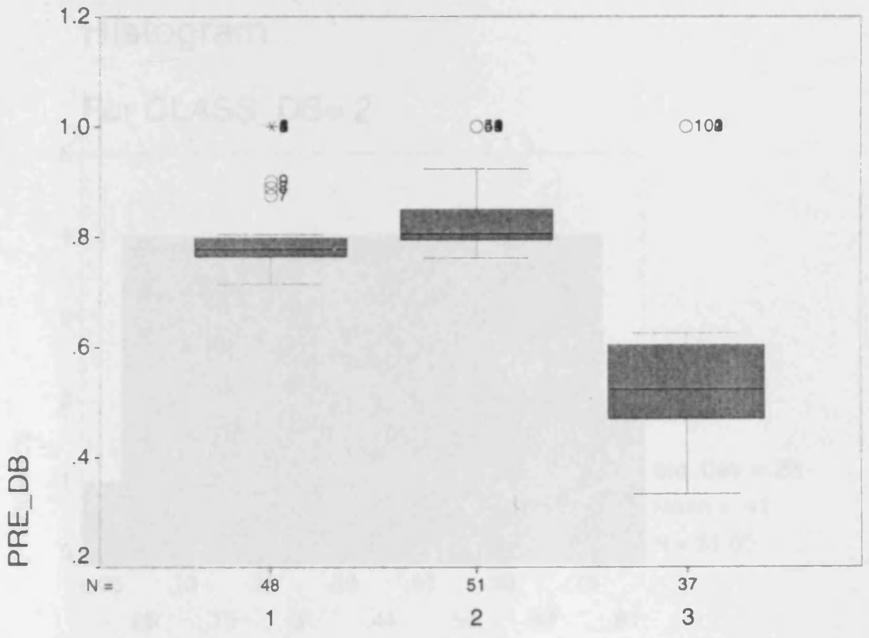


Detrended Normal Q-Q Plot of PRE_DB

For CLASS_DB= 3



Observed Value



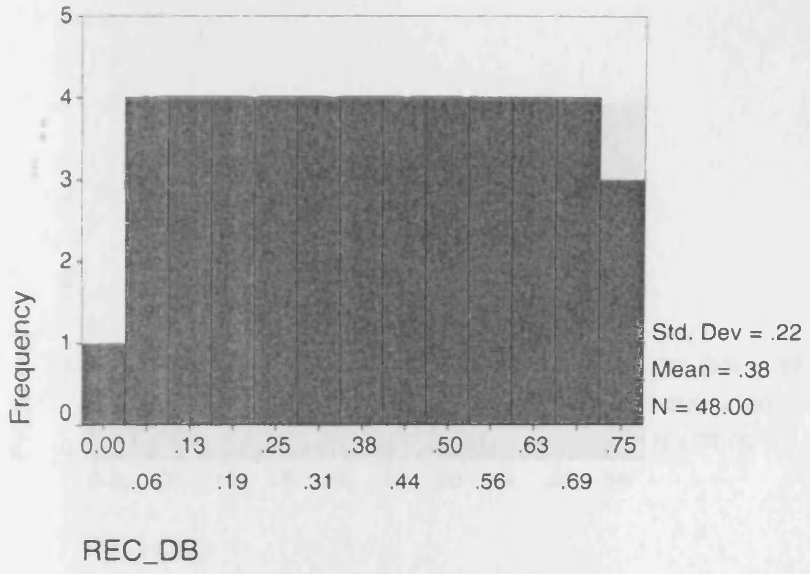
CLASS_DB

REC_DB

Histograms

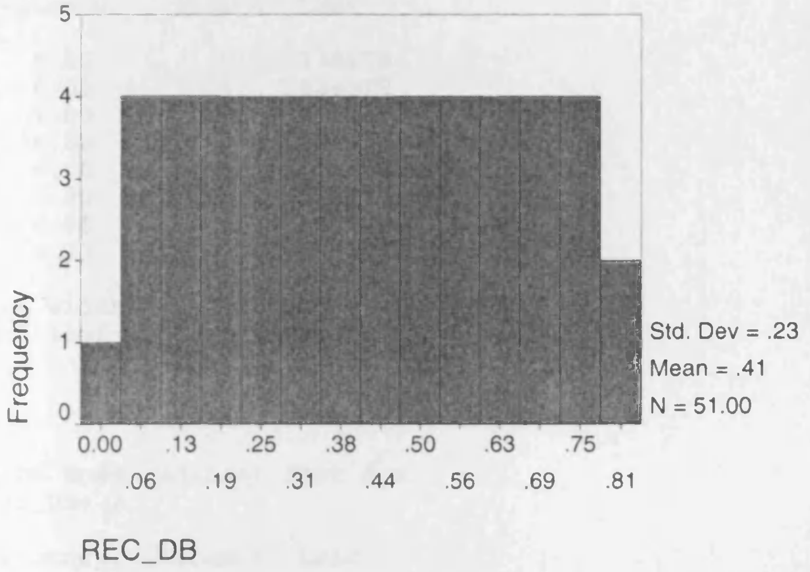
Histogram

For CLASS_DB= 1



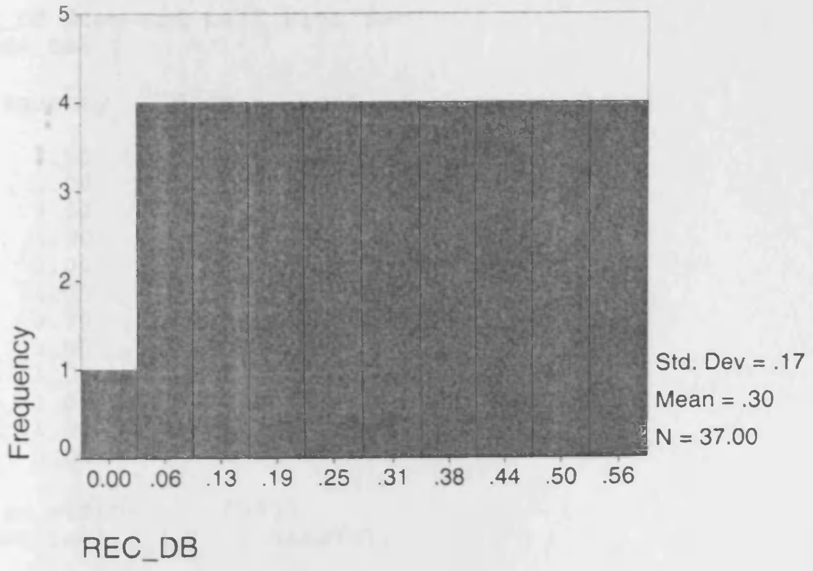
Histogram

For CLASS_DB= 2



Histogram

For CLASS_DB= 3



Stem-and-Leaf Plots

```
REC_DB Stem-and-Leaf Plot for
CLASS_DB= 1

 Frequency    Stem &  Leaf

     6.00        0 .  134679
     6.00        1 .  024578
     7.00        2 .  0135689
     6.00        3 .  124579
     6.00        4 .  023568
     7.00        5 .  0134679
     6.00        6 .  024578
     4.00        7 .  0135

 Stem width:     .10000
 Each leaf:        1 case(s)
```

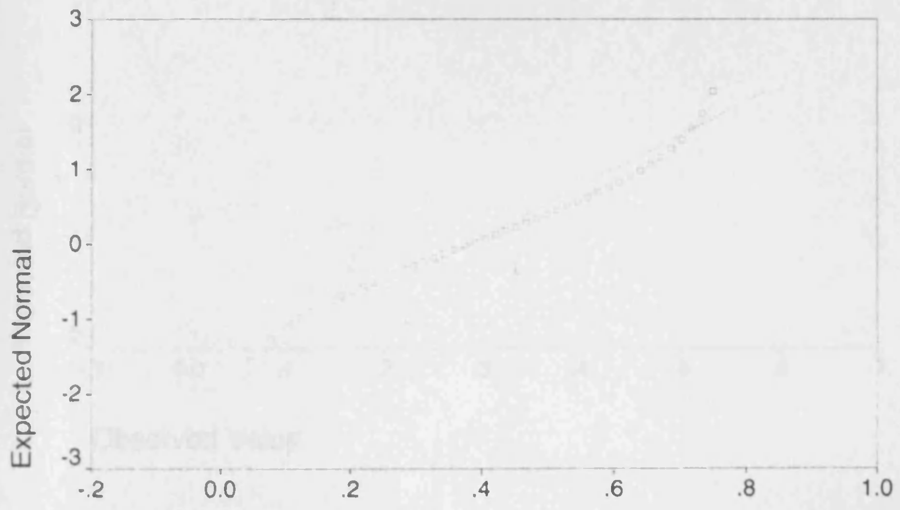
```
REC_DB Stem-and-Leaf Plot for
CLASS_DB= 2

 Frequency    Stem &  Leaf

     6.00        0 .  134679
     6.00        1 .  024578
     7.00        2 .  0135689
     6.00        3 .  124579
     6.00        4 .  023568
     7.00        5 .  0134679
     6.00        6 .  024578
     7.00        7 .  0135689
```

```
Stem width:     .10000
Each leaf:        1 case(s)


REC_DB Stem-and-Leaf Plot for
CLASS_DB= 3

 Frequency     Stem &  Leaf

     3.00        0 .  134
     3.00        0 .  679
     3.00        1 .  024
     3.00        1 .  578
     3.00        2 .  013
     4.00        2 .  5689
     3.00        3 .  124
     3.00        3 .  579
     3.00        4 .  023
     3.00        4 .  568
     4.00        5 .  0134
     2.00        5 .  67

 Stem width:     .10000
 Each leaf:        1 case(s)
```

Normal Q-Q Plots

Normal Q-Q Plot of REC_DB

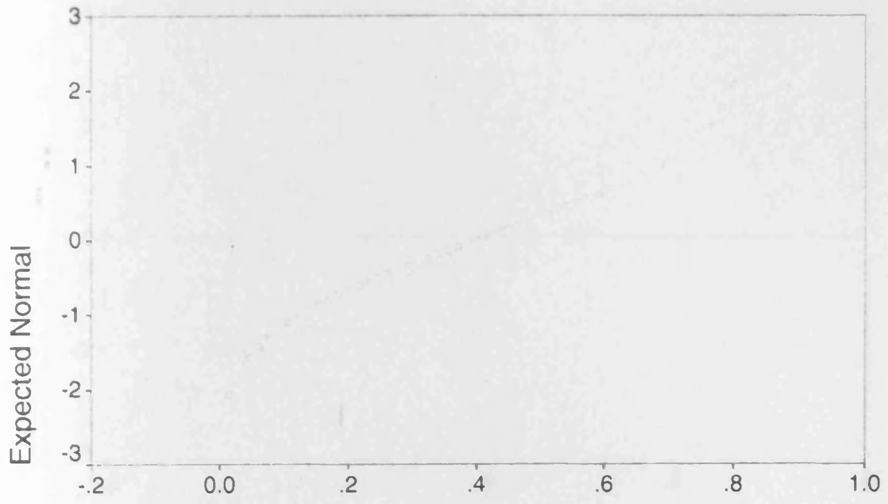
For CLASS_DB= 1



Observed Value

Normal Q-Q Plot of REC_DB

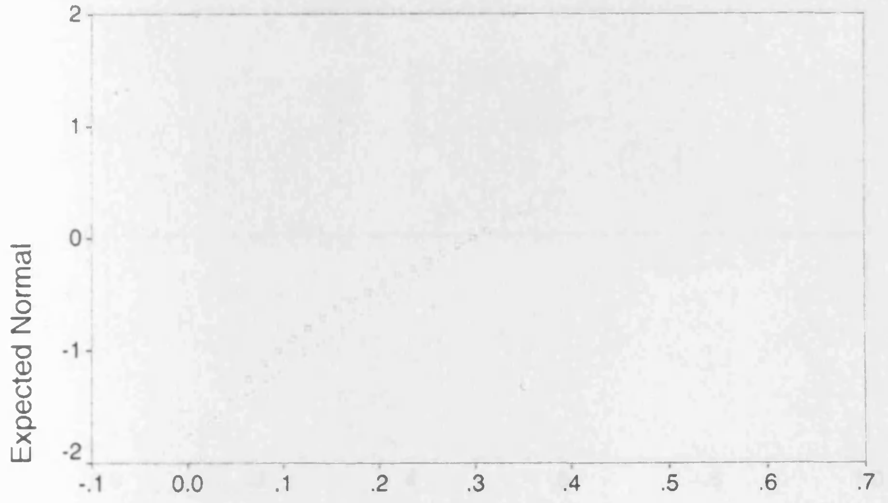
For CLASS_DB= 2



Observed Value

Normal Q-Q Plot of REC_DB

For CLASS_DB= 3

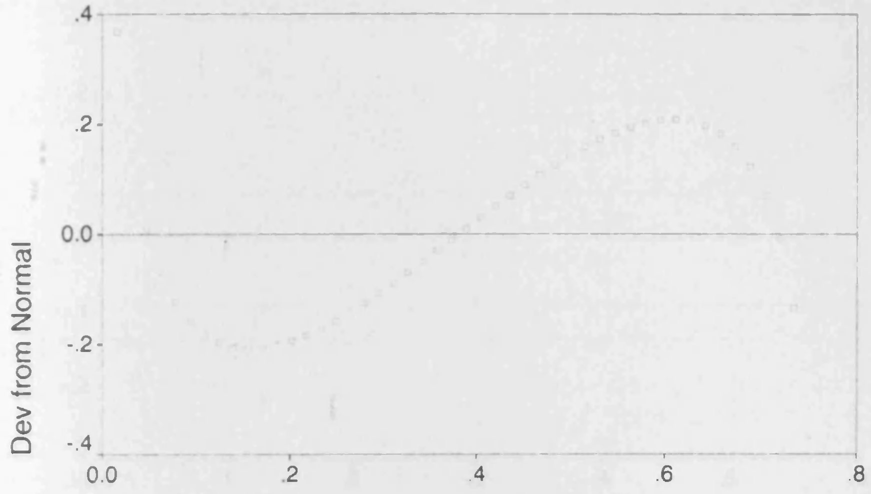


Observed Value

Detrended Normal Q-Q Plots

Detrended Normal Q-Q Plot of REC_DB

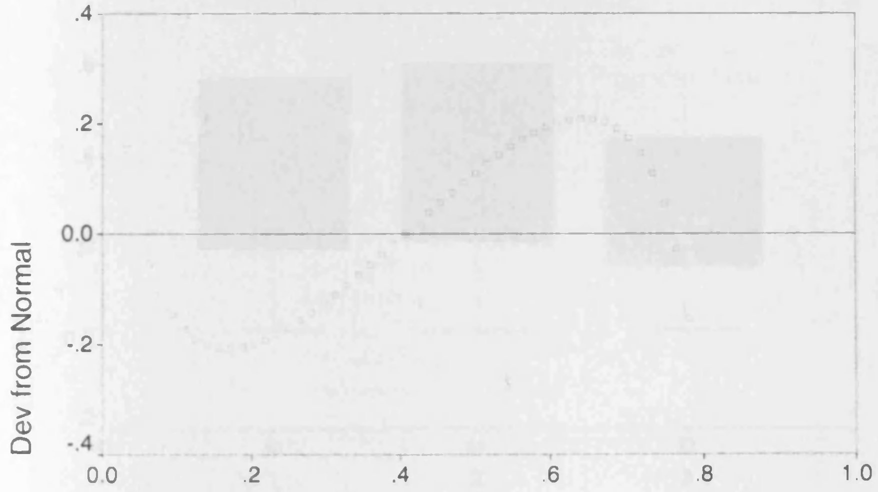
For CLASS_DB= 1



Observed Value

Detrended Normal Q-Q Plot of REC_DB

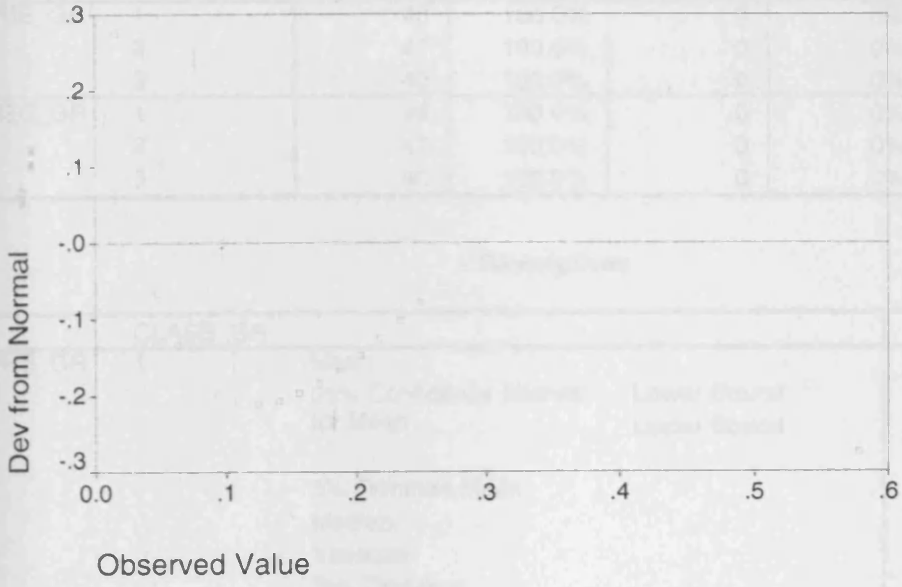
For CLASS_DB= 2

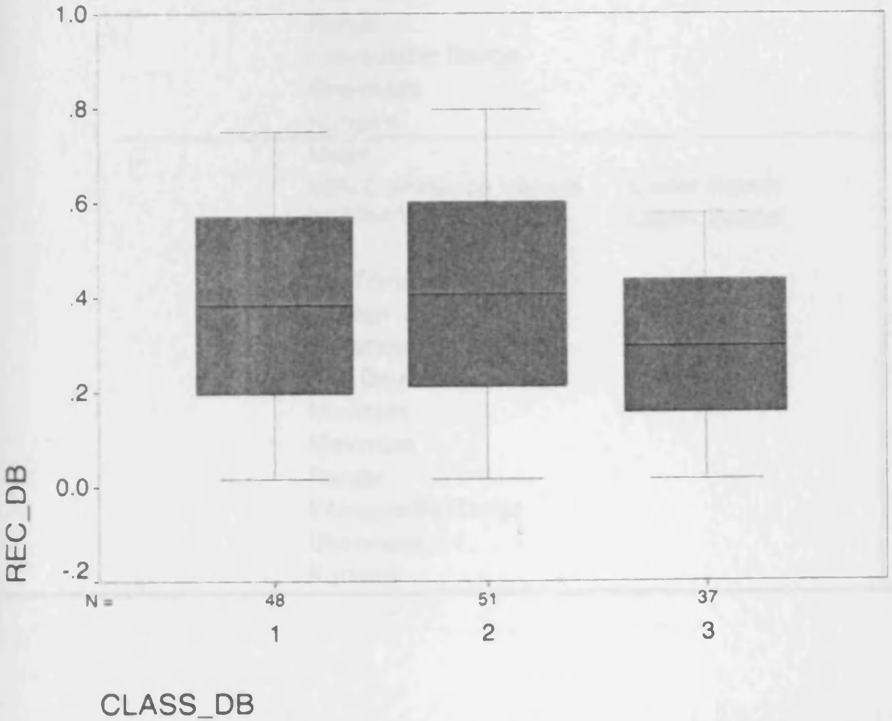


Observed Value

Detrended Normal Q-Q Plot of REC_DB

For CLASS_DB= 3





Explore

CLASS_GR

Case Processing Summary

		Cases					
		Valid		Missing		Total	
	CLASS_GR	N	Percent	N	Percent	N	Percent
PRE_GR	1	46	100.0%	0	.0%	46	100.0%
	2	47	100.0%	0	.0%	47	100.0%
	3	40	100.0%	0	.0%	40	100.0%
REC_GR	1	46	100.0%	0	.0%	46	100.0%
	2	47	100.0%	0	.0%	47	100.0%
	3	40	100.0%	0	.0%	40	100.0%

Descriptives

	CLASS_GR			Statistic	Std. Error
PRE_GR	1	Mean		.6132896	.00908953
		95% Confidence Interval for Mean	Lower Bound	.5949823	
			Upper Bound	.6315968	
		5% Trimmed Mean		.6208150	
		Median		.6339700	
		Variance		.004	
		Std. Deviation		.06164821	
		Minimum		.37500	
		Maximum		.67347	
		Range		.29847	
		Interquartile Range		.0721100	
		Skewness		-2.003	.350
		Kurtosis		4.714	.688
	2	Mean		.6868857	.00906457
		95% Confidence Interval for Mean	Lower Bound	.6686397	
			Upper Bound	.7051318	
		5% Trimmed Mean		.6932214	
		Median		.6969700	
		Variance		.004	
		Std. Deviation		.06214357	
		Minimum		.50000	
		Maximum		.76271	
		Range		.26271	
		Interquartile Range		.0434000	
		Skewness		-1.920	.347
		Kurtosis		3.708	.681

Descriptives

	CLASS_GR			Statistic	Std. Error
PRE_GR	3	Mean		.5229163	.01197952
		95% Confidence Interval for Mean	Lower Bound	.4986854	
			Upper Bound	.5471471	
		5% Trimmed Mean		.5327069	
		Median		.5513700	
		Variance		.006	
		Std. Deviation		.07576512	
		Minimum		.25000	
		Maximum		.57895	
		Range		.32895	
		Interquartile Range		.0294400	
		Skewness		-2.358	.374
		Kurtosis		4.903	.733
REC_GR	1	Mean		.3263889	.02748687
		95% Confidence Interval for Mean	Lower Bound	.2710275	
			Upper Bound	.3817503	
		5% Trimmed Mean		.3263888	
		Median		.3263850	
		Variance		.035	
		Std. Deviation		.18642501	
		Minimum		.01389	
		Maximum		.63889	
		Range		.62500	
		Interquartile Range		.3263850	
		Skewness		.000	.350
		Kurtosis		-1.200	.688
	2	Mean		.3333334	.02777776
		95% Confidence Interval for Mean	Lower Bound	.2774197	
			Upper Bound	.3892471	
		5% Trimmed Mean		.3333332	
		Median		.3333300	
		Variance		.036	
		Std. Deviation		.19043471	
		Minimum		.01389	
		Maximum		.65278	
		Range		.63889	
		Interquartile Range		.3333300	
		Skewness		.000	.347
		Kurtosis		-1.200	.681

Descriptives

	CLASS_GR			Statistic	Std. Error
REC_GR	3	Mean		.2847222	.02567254
		95% Confidence Interval for Mean	Lower Bound	.2327946	
			Upper Bound	.3366498	
		5% Trimmed Mean		.2847222	
		Median		.2847222	
		Variance		.026	
		Std. Deviation		.16236739	
		Minimum		.01389	
		Maximum		.55556	
		Range		.54167	
		Interquartile Range		.2847222	
		Skewness		.000	.374
		Kurtosis		-1.200	.733

Tests of Normality

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	CLASS_GR	Statistic	df	Sig.	Statistic	df	Sig.
PRE_GR	1	.181	46	.001	.793	46	.000
	2	.245	47	.000	.776	47	.000
	3	.334	40	.000	.621	40	.000
REC_GR	1	.066	46	.200*	.956	46	.079
	2	.065	47	.200*	.956	47	.073
	3	.067	40	.200*	.956	40	.124

*. This is a lower bound of the true significance.

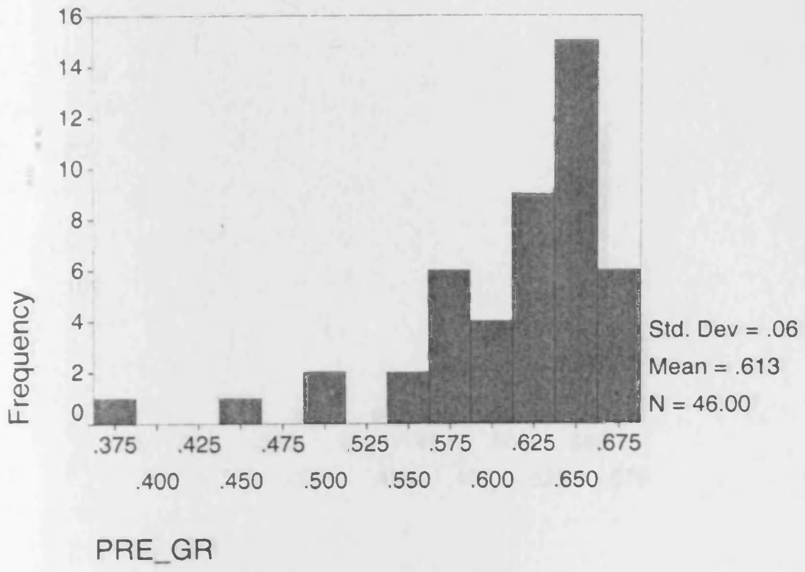
a. Lilliefors Significance Correction

PRE_GR

Histograms

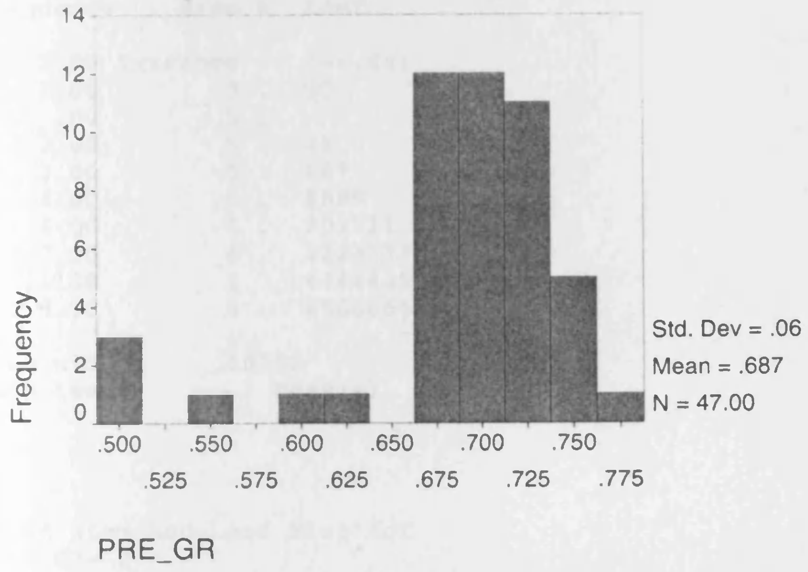
Histogram

For CLASS_GR= 1



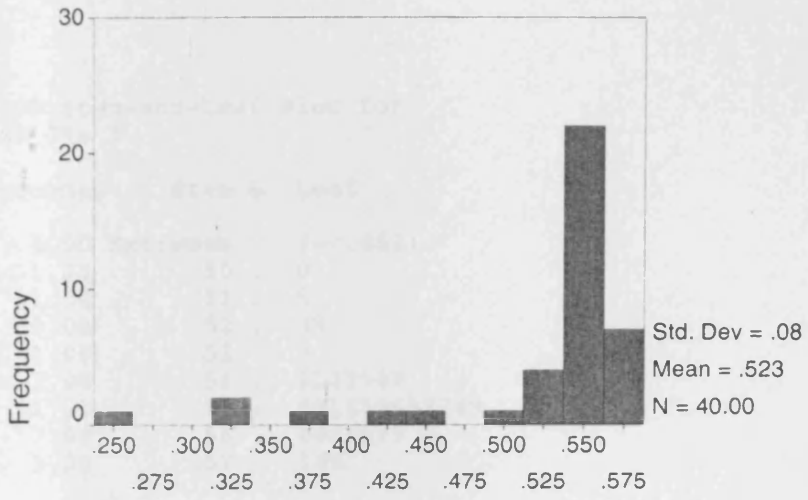
Histogram

For CLASS_GR= 2



Histogram

For CLASS_GR= 3



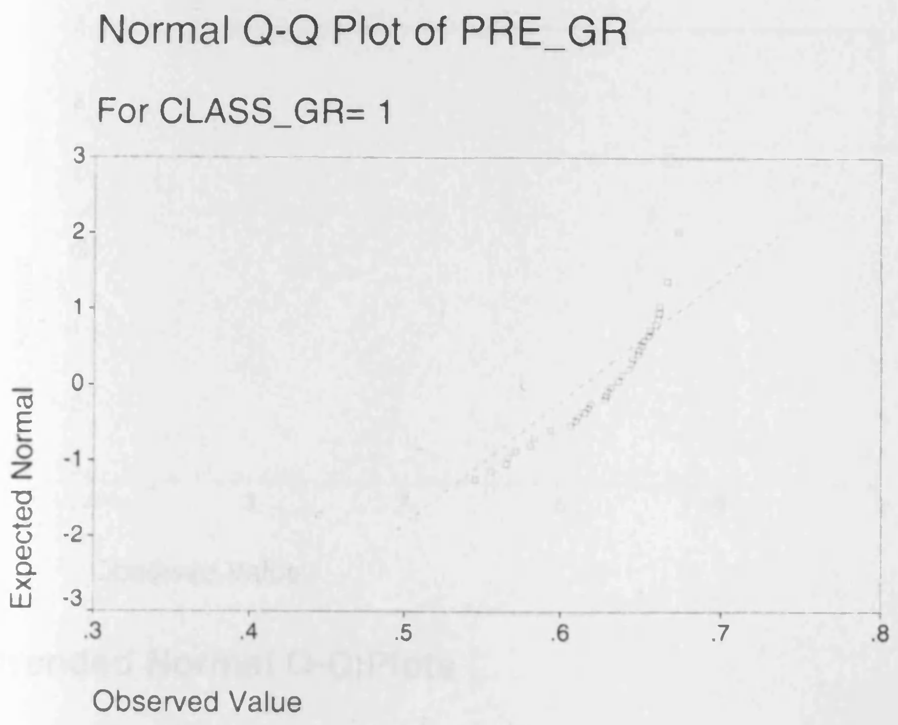
PRE_GR

Stem-and-Leaf Plots

```
PRE_GR Stem-and-Leaf Plot for
CLASS_GR= 1

 Frequency     Stem &  Leaf

     2.00 Extremes    (=<.44)
     2.00        5 .  00
      .00        5 .
     2.00        5 .  45
     3.00        5 .  667
     4.00        5 .  8889
     6.00        6 .  001111
     7.00        6 .  2223333
    11.00        6 .  44444455555
     9.00        6 .  666666667

 Stem width:     .10000
 Each leaf:        1 case(s)
```

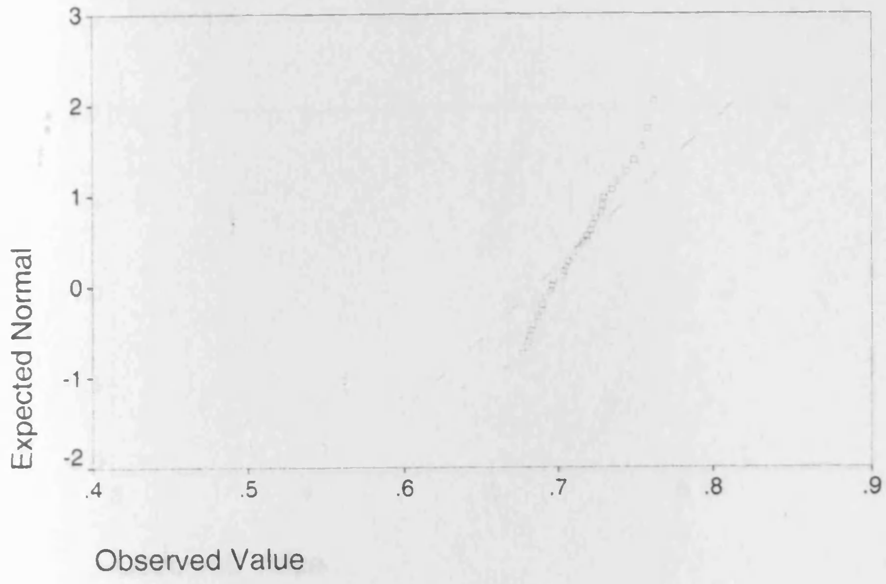
```
PRE_GR Stem-and-Leaf Plot for
CLASS_GR= 2

 Frequency     Stem &  Leaf

     5.00 Extremes    (=<.60)
     1.00        6 .  3
      .00        6 .
     5.00        6 .  66667
    14.00        6 .  88888888999999
     8.00        7 .  00001111
     8.00        7 .  22222233
     5.00        7 .  44555
     1.00        7 .  6

 Stem width:     .10000
 Each leaf:        1 case(s)



PRE_GR Stem-and-Leaf Plot for
CLASS_GR= 3

 Frequency     Stem &  Leaf

     6.00 Extremes     (=<.462)
     1.00       50 .  0
     1.00       51 .  5
     2.00       52 .  99
     1.00       53 .  3
     7.00       54 .  1233589
    12.00       55 .  011155555789
     7.00       56 .  0025579
     3.00       57 .  168

 Stem width:     .01000
 Each leaf:        1 case(s)
```

Normal Q-Q Plots



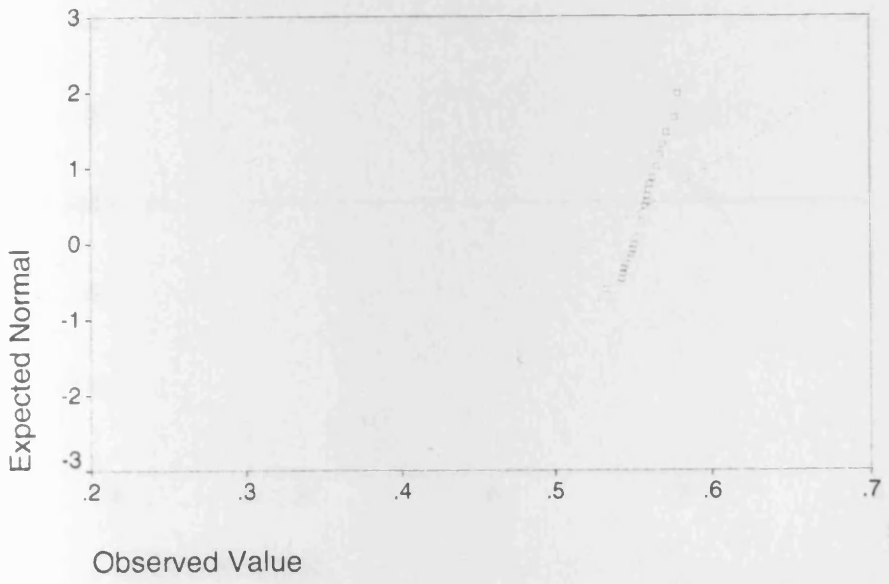
Normal Q-Q Plot of PRE_GR

For CLASS_GR= 2



Normal Q-Q Plot of PRE_GR

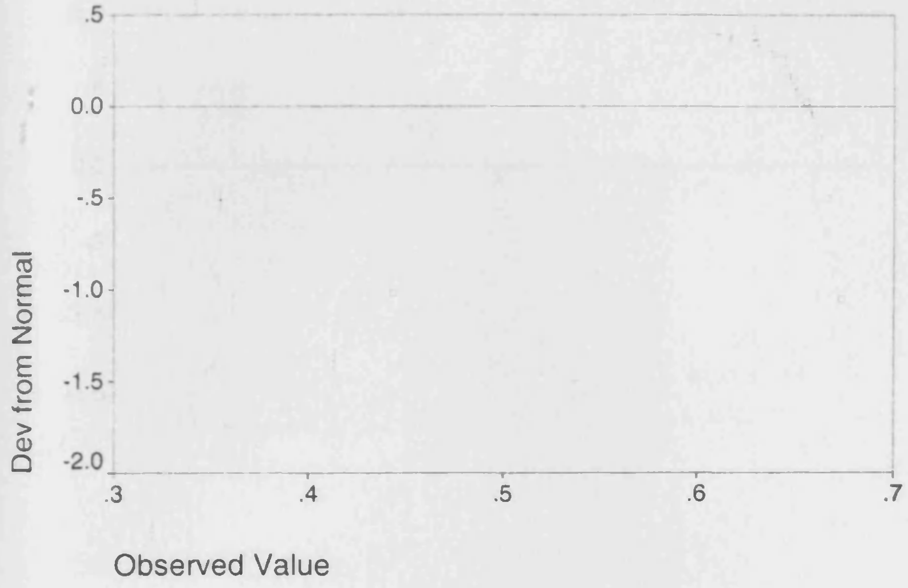
For CLASS_GR= 3



Detrended Normal Q-Q Plots

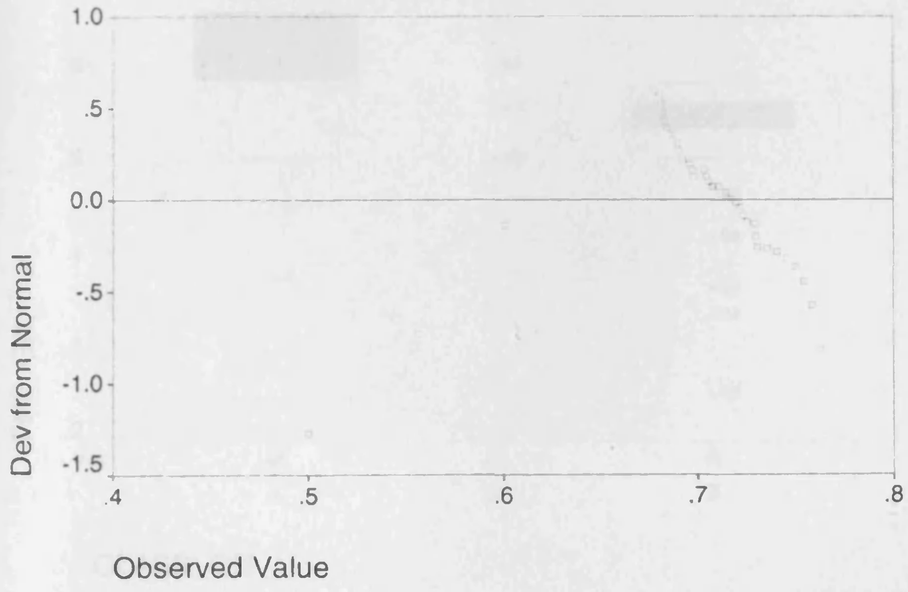
Detrended Normal Q-Q Plot of PRE_GR

For CLASS_GR= 1



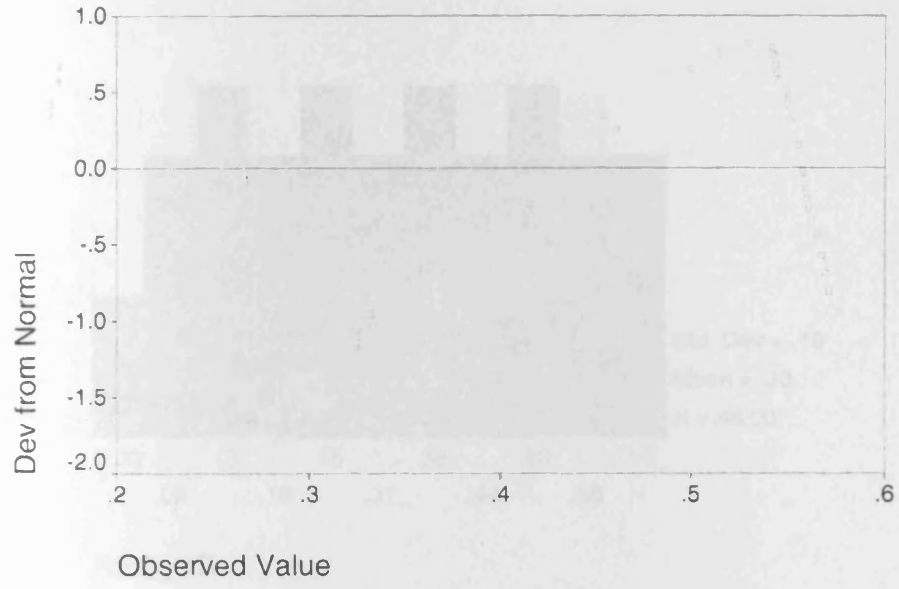
Detrended Normal Q-Q Plot of PRE_GR

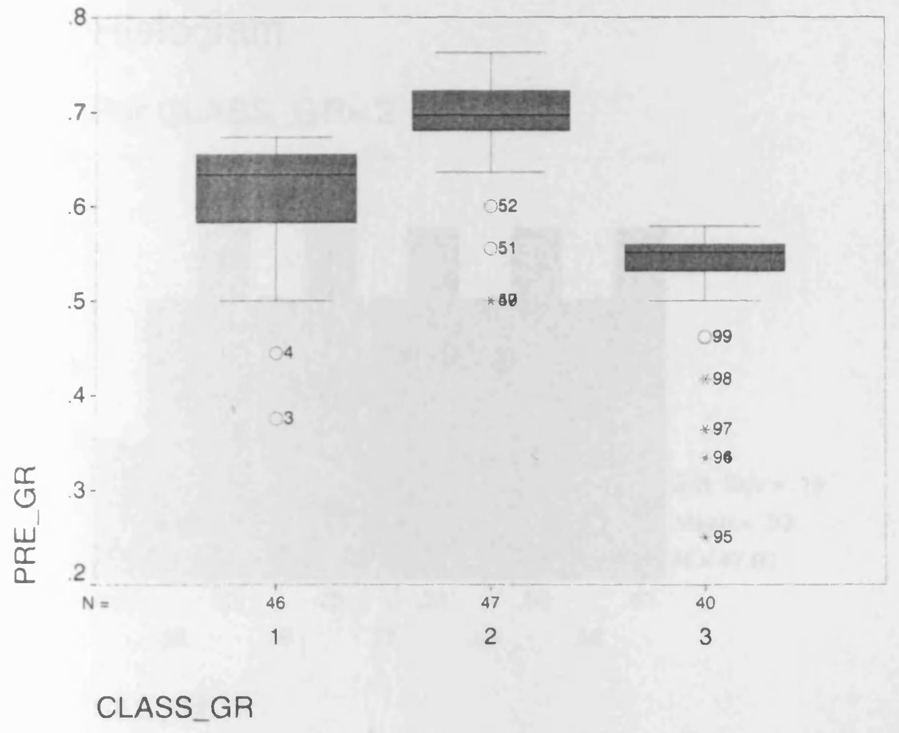
For CLASS_GR= 2



Detrended Normal Q-Q Plot of PRE_GR

For CLASS_GR= 3



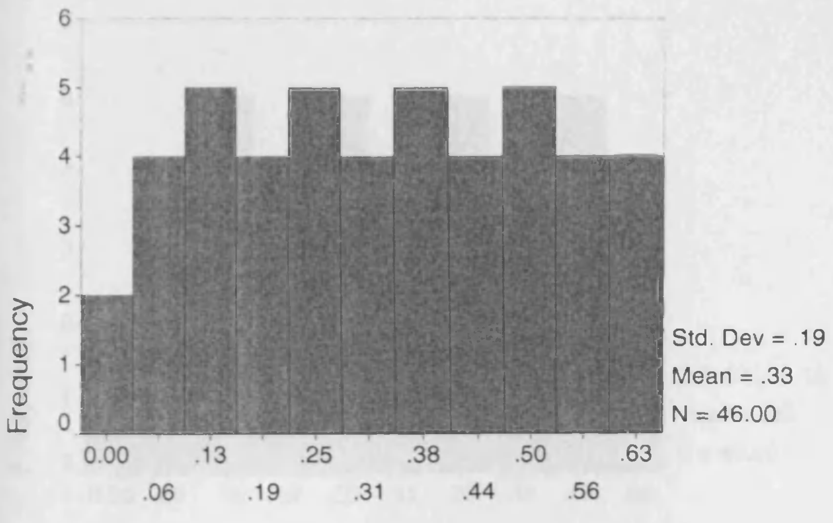


REC_GR

Histograms

Histogram

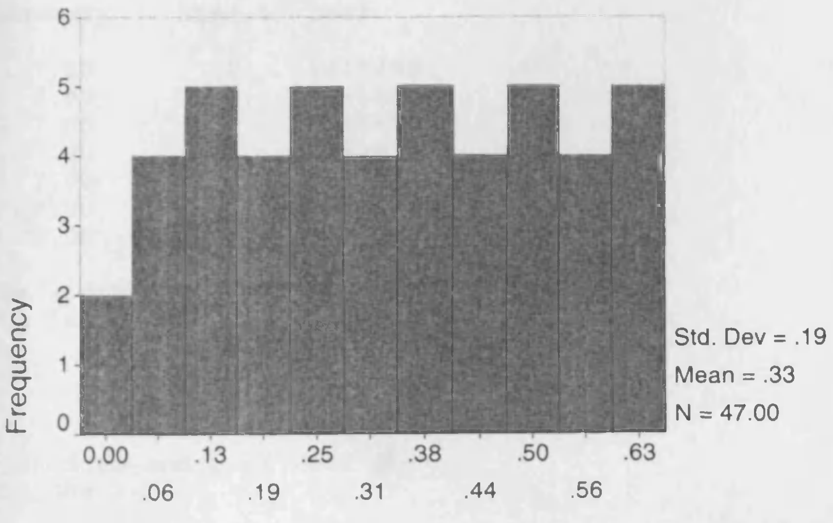
For CLASS_GR= 1



REC_GR

Histogram

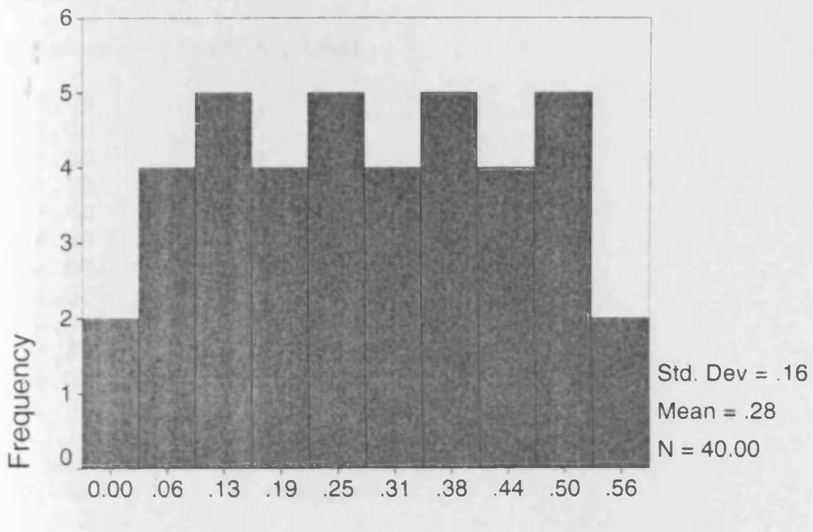
For CLASS_GR= 2



REC_GR

Histogram

For CLASS_GR= 3



REC_GR

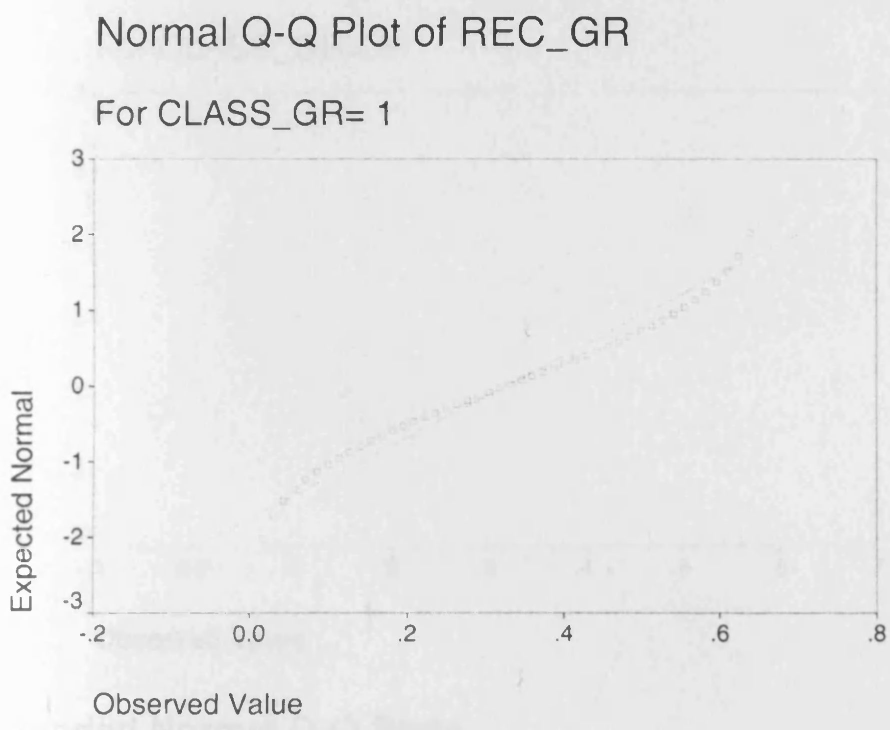
Stem-and-Leaf Plots

```
REC_GR Stem-and-Leaf Plot for
CLASS_GR= 1

 Frequency    Stem &  Leaf

     7.00        0 .  1245689
     7.00        1 .  1235689
     7.00        2 .  0235679
     7.00        3 .  0134678
     7.00        4 .  0134578
     8.00        5 .  01245689
     3.00        6 .  123

 Stem width:    .10000
 Each leaf:       1 case(s)
```

```
REC_GR Stem-and-Leaf Plot for
CLASS_GR= 2

 Frequency    Stem &  Leaf

     7.00        0 .  1245689
     7.00        1 .  1235689
     7.00        2 .  0235679
     7.00        3 .  0134678
     7.00        4 .  0134578
     8.00        5 .  01245689
     4.00        6 .  1235

 Stem width:    .10000
 Each leaf:       1 case(s)
```

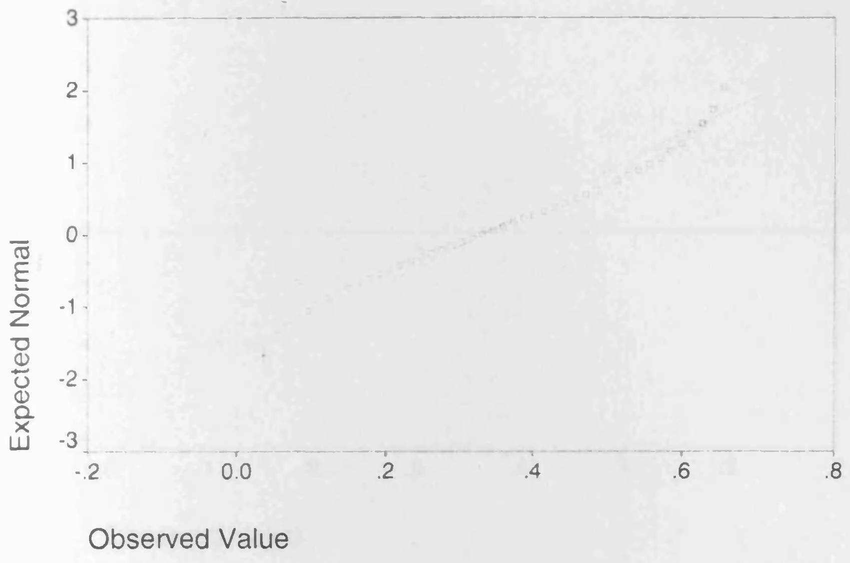
```
REC_GR Stem-and-Leaf Plot for
CLASS_GR= 3

 Frequency     Stem &  Leaf

     3.00        0 .  124
     4.00        0 .  5689
     3.00        1 .  123
     4.00        1 .  5689
     3.00        2 .  023
     4.00        2 .  5679
     4.00        3 .  0134
     3.00        3 .  678
     4.00        4 .  0134
     3.00        4 .  578
     4.00        5 .  0124
     1.00        5 .  5

 Stem width:     .10000
 Each leaf:        1 case(s)
```

Normal Q-Q Plots



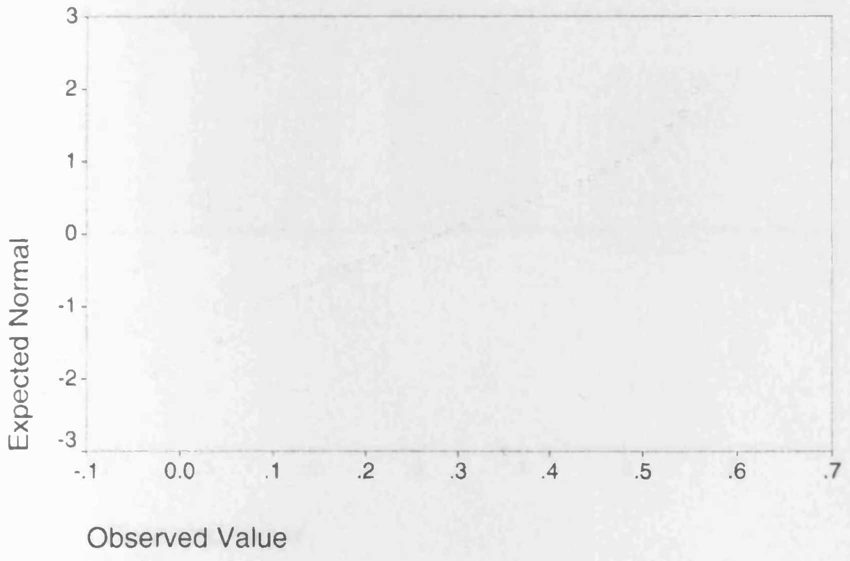
Normal Q-Q Plot of REC_GR

For CLASS_GR= 2



Normal Q-Q Plot of REC_GR

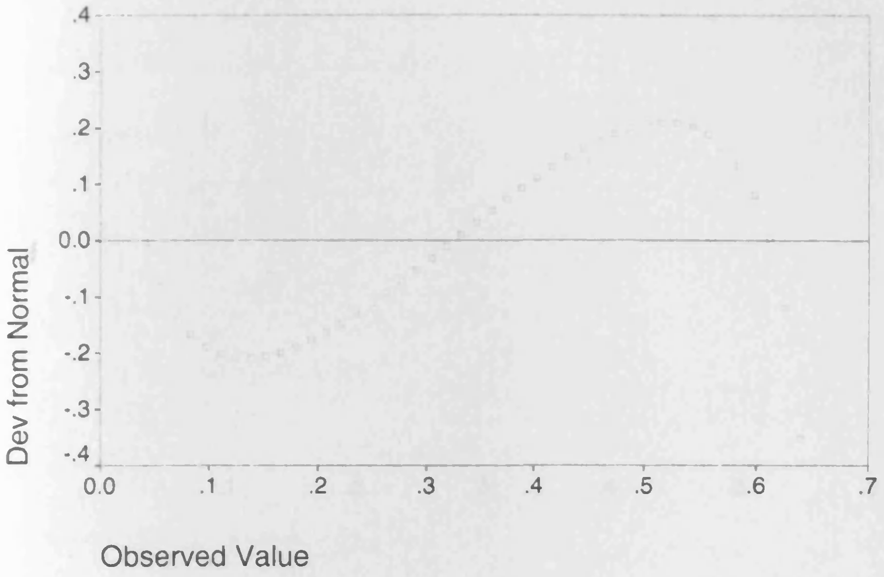
For CLASS_GR= 3



Detrended Normal Q-Q Plots

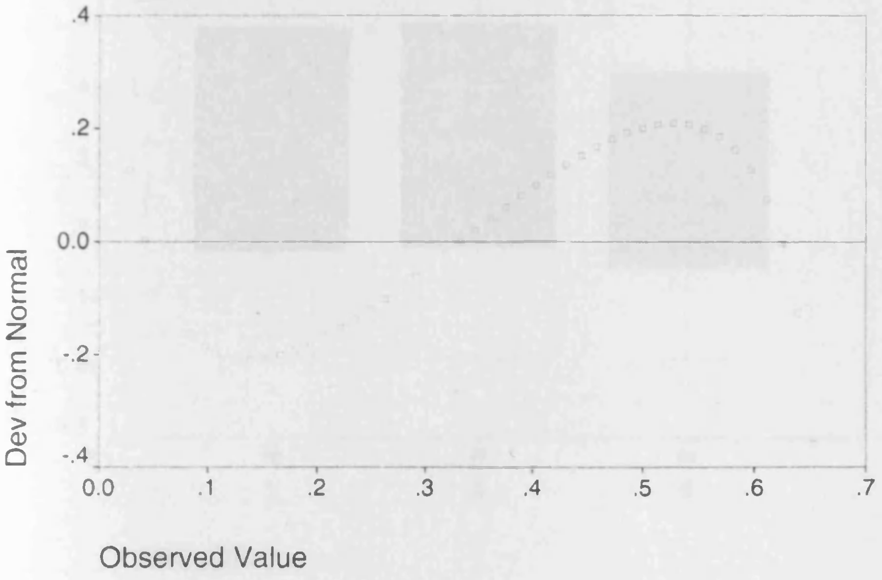
Detrended Normal Q-Q Plot of REC_GR

For CLASS_GR= 1



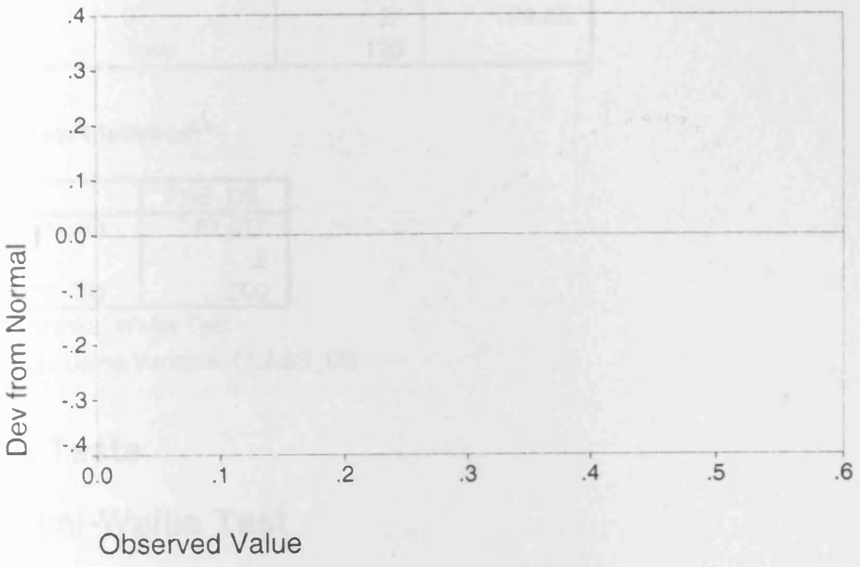
Detrended Normal Q-Q Plot of REC_GR

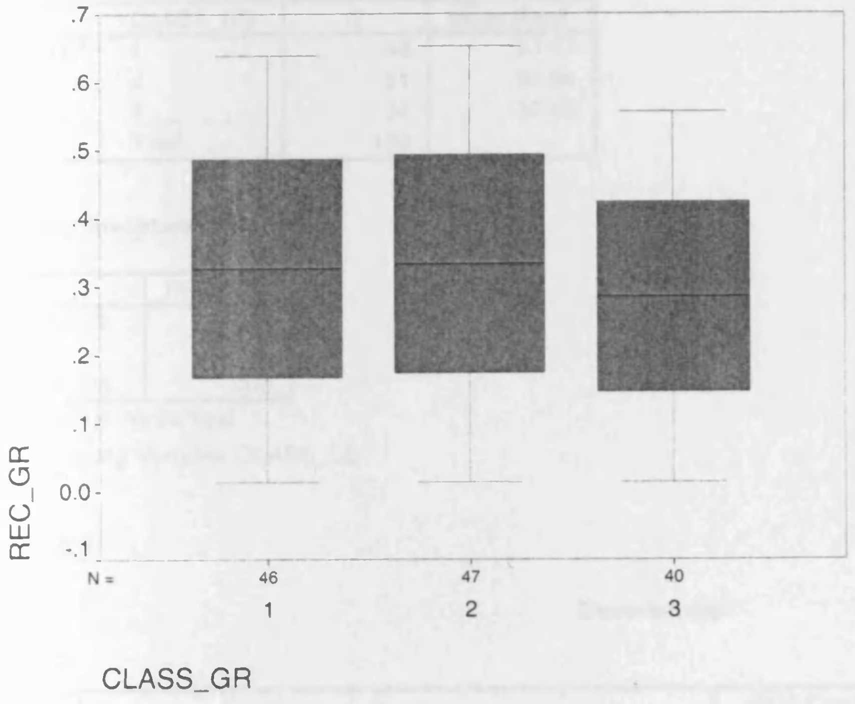
For CLASS_GR= 2



Detrended Normal Q-Q Plot of REC_GR

For CLASS_GR= 3





NPar Tests

Kruskal-Wallis Test

Ranks

	CLASS_DB	N	Mean Rank
PRE_DB	1	48	70.35
	2	51	95.37
	3	37	29.05
	Total	136	

Test Statistics[a,b]

	PRE_DB
Chi-Square	61.017
df	2
Asymp. Sig.	.000

a. Kruskal Wallis Test

b. Grouping Variable: CLASS_DB

NPar Tests

Kruskal-Wallis Test

Ranks

	CLASS_DB	N	Mean Rank
PRE_GR	1	48	63.97
	2	51	92.88
	3	34	32.46
	Total	133	

Test Statistics[a,b]

	PRE_GR
Chi-Square	50.649
df	2
Asymp. Sig.	.000

a. Kruskal Wallis Test

b. Grouping Variable: CLASS_DB

Oneway

Descriptives

REC_DB

					95% Confidence Interval for Mean	
	N	Mean	Std. Deviation	Std. Error	Lower Bound	Upper Bound
1	48	.3828150	.21874991	.03157383	.3192966	.4463334
2	51	.4062525	.23228232	.03252603	.3409221	.4715830
3	37	.2968750	.16913055	.02780489	.2404841	.3532659
Total	136	.3682233	.21514663	.01844867	.3317376	.4047091

Descriptives

REC_DB

	Minimum	Maximum
1	.01563	.75000
2	.01563	.79688
3	.01563	.57813
Total	.01563	.79688

Test of Homogeneity of Variances

REC_DB

Levene Statistic	df1	df2	Sig.
3.035	2	133	.051

ANOVA

REC_DB

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	.272	2	.136	3.030	.052
Within Groups	5.977	133	.045		
Total	6.249	135			

Post Hoc Tests

Multiple Comparisons

Dependent Variable: REC_DB
Tukey HSD

					95% Confidence Interval	
(I) CLASS_DB	(J) CLASS_DB	Mean Difference (I-J)	Std. Error	Sig.	Lower Bound	Upper Bound
1	2	-.0234375	.04262965	.847	-.1244802	.0776051
	3	.0859400	.04637542	.157	-.0239810	.1958610
2	1	.0234375	.04262965	.847	-.0776051	.1244802
	3	.1093775*	.04577783	.048	.0008730	.2178821
3	1	-.0859400	.04637542	.157	-.1958610	.0239810
	2	-.1093775*	.04577783	.048	-.2178821	-.0008730

*. The mean difference is significant at the .05 level.

Homogeneous Subsets

REC_DB

Tukey HSD[a,b]

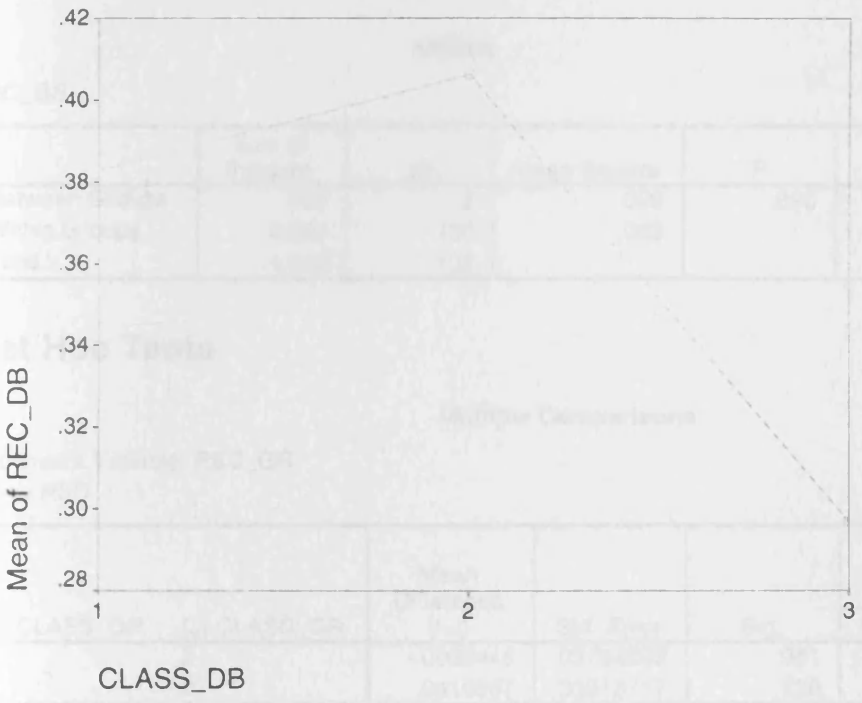
CLASS_DB	N	Subset for alpha = .05 1	2
3	37	.2968750	
1	48	.3828150	.3828150
2	51		.4062525
Sig.		.139	.861

Means for groups in homogeneous subsets are displayed.

a. Uses Harmonic Mean Sample Size = 44.465.

b. The group sizes are unequal. The harmonic mean of the group sizes is used. Type I error levels are not guaranteed.

Means Plots



Oneway

Descriptives

REC_GR

	N	Mean	Std. Deviation	Std. Error	95% Confidence Interval for Mean Lower Bound	Upper Bound
1	46	.3263889	.18642501	.02748687	.2710275	.3817503
2	47	.3333334	.19043471	.02777776	.2774197	.3892471
3	40	.2847222	.16236739	.02567254	.2327946	.3366498
Total	133	.3163116	.18087579	.01568392	.2852873	.3473360

Descriptives

REC_GR

	Minimum	Maximum
1	.01389	.63889
2	.01389	.65278
3	.01389	.55556
Total	.01389	.65278

Test of Homogeneity of Variances

REC_GR

Levene Statistic	df1	df2	Sig.
.885	2	130	.415

ANOVA

REC_GR

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	.058	2	.029	.888	.414
Within Groups	4.260	130	.033		
Total	4.319	132			

Post Hoc Tests

Multiple Comparisons

Dependent Variable: REC_GR
Tukey HSD

(I) CLASS_GR	(J) CLASS_GR	Mean Difference (I-J)	Std. Error	Sig.	95% Confidence Interval	
					Lower Bound	Upper Bound
1	2	-.0069445	.03754592	.981	-.0959607	.0820717
	3	.0416667	.03913717	.538	-.0511221	.1344555
2	1	.0069445	.03754592	.981	-.0820717	.0959607
	3	.0486112	.03894304	.427	-.0437174	.1409397
3	1	-.0416667	.03913717	.538	-.1344555	.0511221
	2	-.0486112	.03894304	.427	-.1409397	.0437174

Homogeneous Subsets

REC_GR

Tukey HSD[a,b]

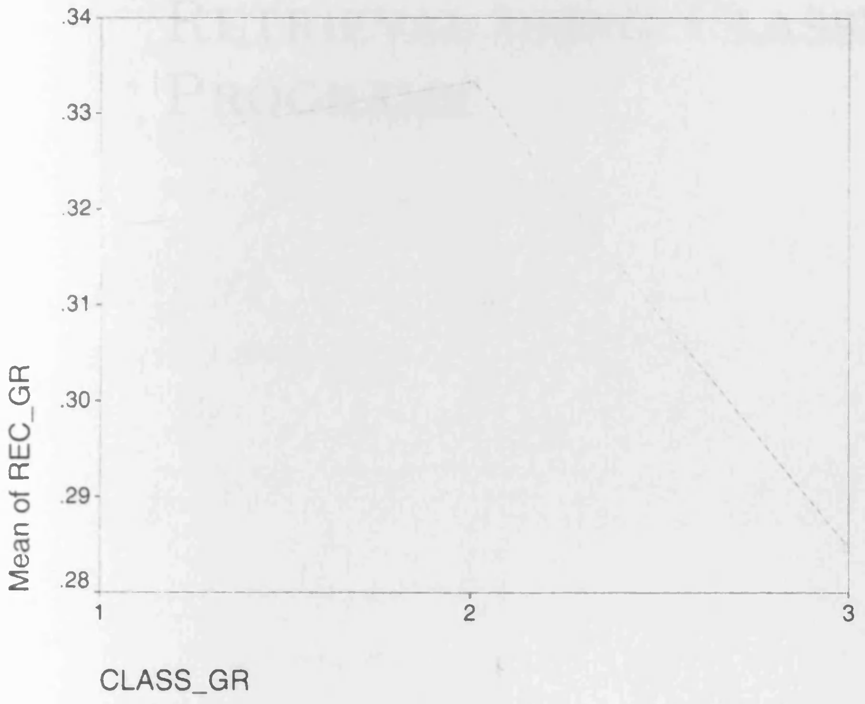
CLASS_GR	N	Subset for alpha = .05
		1
3	40	.2847222
1	46	.3263889
2	47	.3333334
Sig.		.420

Means for groups in homogeneous subsets are displayed.

a. Uses Harmonic Mean Sample Size = 44.107.

b. The group sizes are unequal. The harmonic mean of the group sizes is used. Type I error levels are not guaranteed.

Means Plots



APPENDIX C

STATISTICAL RESULT - PROGRAM RETRIEVAL USING CLASSIFIED PROGRAMS

Appendix C : Statistical Result - Program Retrieval using Classified Programs

NPar Tests

Kruskal-Wallis Test

Ranks

	CLASS	N	Mean Rank
PRE	1	10	12.95
	2	10	25.05
	3	10	26.90
	4	10	17.10
	Total	40	

Test Statistics[a,b]

	PRE
Chi-Square	9.952
df	3
Asymp. Sig.	.019

a. Kruskal Wallis Test

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
PRE	1	10	7.95	79.50
	2	10	13.05	130.50
	Total	20		

Test Statistics[b]

	PRE
Mann-Whitney U	24.500
Wilcoxon W	79.500
Z	-1.975
Asymp. Sig. (2-tailed)	.048
Exact Sig. [2*(1-tailed Sig.)]	.052[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
PRE	1	10	6.95	69.50
	3	10	14.05	140.50
	Total	20		

Test Statistics[b]

	PRE
Mann-Whitney U	14.500
Wilcoxon W	69.500
Z	-2.727
Asymp. Sig. (2-tailed)	.006
Exact Sig. [2*(1-tailed Sig.)]	.005[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
PRE	1	10	9.05	90.50
	4	10	11.95	119.50
	Total	20		

Test Statistics[b]

	PRE
Mann-Whitney U	35.500
Wilcoxon W	90.500
Z	-1.107
Asymp. Sig. (2-tailed)	.268
Exact Sig. [2*(1-tailed Sig.)]	.280[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
PRE	2	10	10.45	104.50
	3	10	10.55	105.50
	Total	20		

Test Statistics[b]

	PRE
Mann-Whitney U	49.500
Wilcoxon W	104.500
Z	-.041
Asymp. Sig. (2-tailed)	.967
Exact Sig. [2*(1-tailed Sig.)]	.971[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
PRE	2	10	12.55	125.50
	4	10	8.45	84.50
	Total	20		

Test Statistics[b]

	PRE
Mann-Whitney U	29.500
Wilcoxon W	84.500
Z	-1.597
Asymp. Sig. (2-tailed)	.110
Exact Sig. [2*(1-tailed Sig.)]	.123[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
PRE	3	10	13.30	133.00
	4	10	7.70	77.00
	Total	20		

Test Statistics[b]

	PRE
Mann-Whitney U	22.000
Wilcoxon W	77.000
Z	-2.179
Asymp. Sig. (2-tailed)	.029
Exact Sig. [2*(1-tailed Sig.)]	.035[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Kruskal-Wallis Test

Ranks

	CLASS	N	Mean Rank
PRE	1	10	12.95
	2	10	25.05
	3	10	26.90
	4	10	17.10
	Total	40	
RECALL	1	10	15.45
	2	10	23.40
	3	10	24.70
	4	10	18.45
	Total	40	

Test Statistics[a,b]

	PRE	RECALL
Chi-Square	9.952	4.105
df	3	3
Asymp. Sig.	.019	.250

a. Kruskal Wallis Test

b. Grouping Variable: CLASS

NPar Tests

Kruskal-Wallis Test

Ranks

	CLASS	N	Mean Rank
RECALL	1	10	15.45
	2	10	23.40
	3	10	24.70
	4	10	18.45
	Total	40	

Test Statistics[a,b]

	RECALL
Chi-Square	4.105
df	3
Asymp. Sig.	.250

a. Kruskal Wallis Test

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
RECALL	1	10	8.65	86.50
	2	10	12.35	123.50
	Total	20		

Test Statistics[b]

	RECALL
Mann-Whitney U	31.500
Wilcoxon W	86.500
Z	-1.401
Asymp. Sig. (2-tailed)	.161
Exact Sig. [2*(1-tailed Sig.)]	.165[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
RECALL	1	10	8.00	80.00
	3	10	13.00	130.00
	Total	20		

Test Statistics[b]

	RECALL
Mann-Whitney U	25.000
Wilcoxon W	80.000
Z	-1.893
Asymp. Sig. (2-tailed)	.058
Exact Sig. [2*(1-tailed Sig.)]	.063[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
RECALL	1	10	9.80	98.00
	4	10	11.20	112.00
	Total	20		

Test Statistics[b]

	RECALL
Mann-Whitney U	43.000
Wilcoxon W	98.000
Z	-.530
Asymp. Sig. (2-tailed)	.596
Exact Sig. [2*(1-tailed Sig.)]	.631[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
RECALL	2	10	10.40	104.00
	3	10	10.60	106.00
	Total	20		

Test Statistics[b]

	RECALL
Mann-Whitney U	49.000
Wilcoxon W	104.000
Z	-.077
Asymp. Sig. (2-tailed)	.939
Exact Sig. [2*(1-tailed Sig.)]	.971[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
RECALL	2	10	11.65	116.50
	4	10	9.35	93.50
	Total	20		

Test Statistics[b]

	RECALL
Mann-Whitney U	38.500
Wilcoxon W	93.500
Z	-.873
Asymp. Sig. (2-tailed)	.383
Exact Sig. [2*(1-tailed Sig.)]	.393[a]

.a. Not corrected for ties.

b. Grouping Variable: CLASS

NPar Tests

Mann-Whitney Test

Ranks

	CLASS	N	Mean Rank	Sum of Ranks
RECALL	3	10	12.10	121.00
	4	10	8.90	89.00
	Total	20		

Test Statistics[b]

	RECALL
Mann-Whitney U	34.000
Wilcoxon W	89.000
Z	-1.215
Asymp. Sig. (2-tailed)	.224
Exact Sig. [2*(1-tailed Sig.)]	.247[a]

a. Not corrected for ties.

b. Grouping Variable: CLASS

Appendix D

MySQLDatabase.java

```
    public String getUserID()
    {
      return user;
    }



    /**
     * @return does this account have dba granted to it?
    public boolean isDBA()
    {
      boolean DBARole = false;

      try
      {
        ResultSet rs = getStatement().executeQuery
          ("select GRANTED_ROLE from USER_ROLE_PRIVS");
        // where USERNAME = '"+user.toUpperCase()+"'");
        while (rs.next())
        {
          String role = rs.getString("GRANTED_ROLE").toUpperCase();
          if (role.equals("DBA"))
            DBARole = true;
        }
      }
      catch (Exception e)
      {
        DBARole = false;
      }
      return DBARole;
    }
     */

    //////////////////////////////////////////////////////////////////////////
    // Private Methods
    //////////////////////////////////////////////////////////////////////////



    //////////////////////////////////////////////////////////////////////////
    //
    //////////////////////////////////////////////////////////////////////////


    // the various mysql-specific objects
    public static final int ROOT           = 0;
    public static final int SCHEMA         = 1;
    public static final int COLUMN         = 9;

    public static final int TABLEGROUP     = 30;
    public static final int TABLE          = 31;
    public static final int TABLEINFO      = 32;
    public static final int TABLEDATA      = 33;
    public static final int TABLECOLS      = 34;

    protected String driver = "org.gjt.mm.mysql.Driver";
    protected Table table;

    private String user;
    private String password;
    private String dbname;
    private String host;
    private String port;
  }

```

Appendix E

PickPhotosPanel.java

```

import java.util.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PickPhotosPanel
    extends javax.swing.JPanel
    implements WizardPanel, ItemListener
{

    /** Creates new form PickFilesPanel */
    public PickPhotosPanel(PublishManager publishManager) {
        this.publishManager = publishManager;
        initComponents ();
        ((FileSelector)pnlFileSelector).addItemListener( this );
        spScroll.getHorizontalScrollBar().setUnitIncrement( 16 );
        spScroll.getVerticalScrollBar().setUnitIncrement( 16 );
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the FormEditor.
     */
    private void initComponents() {//GEN-BEGIN:initComponents
        lblTitle = new javax.swing.JLabel();
        pnlContents = new javax.swing.JPanel();
        tpInstructions = new javax.swing.JTextPane();
        pnlCenter = new javax.swing.JPanel();
        spScroll = new javax.swing.JScrollPane();
        pnlFileSelector = new FileSelector( true );

        pnlControls = new javax.swing.JPanel();
        btnRefresh = new javax.swing.JButton();
        btnBrowse = new javax.swing.JButton();
        tpChoiceDescription = new javax.swing.JTextPane();

        setLayout(new java.awt.BorderLayout());

        setBackground(java.awt.Color.white);
        lblTitle.setText("Pick Photos");
        lblTitle.setForeground(new java.awt.Color(0, 153, 153));
        lblTitle.setFont(new java.awt.Font("SansSerif", 1, 14));
        lblTitle.setBorder(new javax.swing.border.EmptyBorder(new
java.awt.Insets(3, 3, 3, 3)));
        add(lblTitle, java.awt.BorderLayout.NORTH);

        pnlContents.setLayout(new java.awt.GridBagLayout());
        java.awt.GridBagConstraints gridBagConstraints1;

        pnlContents.setBackground(java.awt.Color.white);
        tpInstructions.setEditable(false);
        tpInstructions.setFont(new java.awt.Font("SansSerif", 0, 12));
        tpInstructions.setText("Choose the folder containing all the pictures you
want to publish.  To expand a folder, double-click on it.");
        gridBagConstraints1 = new java.awt.GridBagConstraints();
        gridBagConstraints1.gridwidth = java.awt.GridBagConstraints.REMAINDER;
        gridBagConstraints1.fill = java.awt.GridBagConstraints.HORIZONTAL;
        gridBagConstraints1.anchor = java.awt.GridBagConstraints.NORTHWEST;
        gridBagConstraints1.weightx = 1.0;
        pnlContents.add(tpInstructions, gridBagConstraints1);

        pnlCenter.setLayout(new java.awt.BorderLayout());

        spScroll.addComponentListener(new java.awt.event.ComponentAdapter() {
            public void componentResized(java.awt.event.ComponentEvent evt) {
                spScrollComponentResized(evt);
            }
        });

        pnlFileSelector.setBackground(java.awt.Color.white);
        spScroll.setViewportView(pnlFileSelector);

        pnlCenter.add(spScroll, java.awt.BorderLayout.CENTER);

        gridBagConstraints1 = new java.awt.GridBagConstraints();
        gridBagConstraints1.gridwidth = java.awt.GridBagConstraints.REMAINDER;
        gridBagConstraints1.fill = java.awt.GridBagConstraints.BOTH;
        gridBagConstraints1.insets = new java.awt.Insets(20, 40, 20, 40);
        gridBagConstraints1.anchor = java.awt.GridBagConstraints.NORTHWEST;
        gridBagConstraints1.weighty = 1.0;
        pnlContents.add(pnlCenter, gridBagConstraints1);

        pnlControls.setBackground(java.awt.Color.white);
        btnRefresh.setToolTipText("Refresh the contents of the selected folder.");
        btnRefresh.setFont(new java.awt.Font("SansSerif", 0, 12));
        btnRefresh.setText("Refresh Folder");
        btnRefresh.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnRefreshActionPerformed(evt);
            }
        });

        pnlControls.add(btnRefresh);

        btnBrowse.setToolTipText("Browse for folders");
        btnBrowse.setFont(new java.awt.Font("SansSerif", 0, 12));
        btnBrowse.setText("Browse...");
        btnBrowse.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnBrowseActionPerformed(evt);
            }
        });

        pnlControls.add(btnBrowse);

        gridBagConstraints1 = new java.awt.GridBagConstraints();
        gridBagConstraints1.gridwidth = java.awt.GridBagConstraints.REMAINDER;
        gridBagConstraints1.fill = java.awt.GridBagConstraints.HORIZONTAL;
        gridBagConstraints1.weightx = 1.0;
        pnlContents.add(pnlControls, gridBagConstraints1);

        tpChoiceDescription.setEditable(false);
        tpChoiceDescription.setFont(new java.awt.Font("SansSerif", 0, 12));
        tpChoiceDescription.setText("Selection: None\n\n");
        gridBagConstraints1 = new java.awt.GridBagConstraints();
        gridBagConstraints1.gridwidth = java.awt.GridBagConstraints.REMAINDER;
        gridBagConstraints1.fill = java.awt.GridBagConstraints.HORIZONTAL;
        gridBagConstraints1.insets = new java.awt.Insets(15, 0, 15, 0);
        gridBagConstraints1.anchor = java.awt.GridBagConstraints.NORTHWEST;
        pnlContents.add(tpChoiceDescription, gridBagConstraints1);

        add(pnlContents, java.awt.BorderLayout.CENTER);

    }//GEN-END:initComponents

    private void btnBrowseActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_btnBrowseActionPerformed
        // Get the parent frame:
        Container parent = this;
        while( !((parent = parent.getParent()) instanceof JFrame) );

        FileDialog fileDialog = new FileDialog( (Frame)parent,
            "Browse for Photos Folder", FileDialog.LOAD );
        // If the user has a directory selected, use that directory.
        // If not, then try to use the directory from the photo source.
        // Otherwise, let it use whatever directory it sees fit.
        if( selectedDirectory != null ) {
            fileDialog.setDirectory( selectedDirectory.getAbsolutePath() );
        }
        else {
            File sourceDir = publishManager.getPhotoSource().getSourceDir();
            if( sourceDir != null ) {
                fileDialog.setDirectory( sourceDir.getAbsolutePath() );
            }
        }
        fileDialog.setModal( true );
        fileDialog.setVisible( true );

        String directory = fileDialog.getDirectory();
        String filename = fileDialog.getFile();
        if( filename != null ) {
            File result = new File( directory, filename );
            if( !result.isDirectory() ) {
                result = result.getParentFile();
            }
            // First, set selectedDirectory.  Then, try to set it in the
            // tree.  In case the tree setting fails, make sure we still
            // have the desired directory selected.
            tryDirectory( result );
            ((FileSelector)pnlFileSelector).setSelectedFile( result );
            tryDirectory( result );
        }
    }//GEN-LAST:event_btnBrowseActionPerformed

    private void btnRefreshActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_btnRefreshActionPerformed
        ((FileSelector)pnlFileSelector).refreshSelection();
    }//GEN-LAST:event_btnRefreshActionPerformed

    private void spScrollComponentResized(java.awt.event.ComponentEvent evt)
{//GEN-FIRST:event_spScrollComponentResized
        ((FileSelector)pnlFileSelector).scrollToSelection();
    }//GEN-LAST:event_spScrollComponentResized


    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JLabel lblTitle;
    private javax.swing.JPanel pnlContents;
    private javax.swing.JTextPane tpInstructions;
    private javax.swing.JPanel pnlCenter;
    private javax.swing.JScrollPane spScroll;
    private javax.swing.JPanel pnlFileSelector;
    private javax.swing.JPanel pnlControls;
    private javax.swing.JButton btnRefresh;
    private javax.swing.JButton btnBrowse;
    private javax.swing.JTextPane tpChoiceDescription;
    // End of variables declaration//GEN-END:variables

    private PublishManager publishManager;
    private File selectedDirectory;
    private PhotoSource tempPhotoSource = new PhotoSource();

    // We remember this value in case the user clicks back and
    // changes the directory.  We want to warn that captions may be lost.
    private File lastSelectedDirectory = null;

    /**
     * Final directory selection
     */
    private void selectDirectory( File dir ) {
        selectedDirectory = dir;
        PhotoSource source = publishManager.getPhotoSource();
        source.scanDirectory( selectedDirectory );
        Settings.getInstance().setProperty(
            Constants.DEFAULT_INPUT_DIR,
            dir.getAbsolutePath() );
    }

    /**
     * Try this directory, before officially selecting it
     */
    private void tryDirectory( File dir ) {
        selectedDirectory = dir;
        tempPhotoSource.scanDirectory( selectedDirectory );
        updateGUI();
    }

    /** Returns true if all required data was filled in for this panel.
     */
    public boolean isSatisfied() {
        int numPhotos = tempPhotoSource.getPhotos().size();
        return numPhotos > 0;
    }
    /** Called when the panel is shown to the user
     */
    public void showPanel() {
        File dir = publishManager.getPhotoSource().getSourceDir();
        ((FileSelector)pnlFileSelector).setSelectedFile( dir );
```

```java
            tryDirectory( dir );
    }
    /** Called when the panel is hidden from the user
     */
    public void hidePanel(boolean forwardDirection) throws CannotChangePanelException
    {
        if( (lastSelectedDirectory != null) &&
            !lastSelectedDirectory.equals( selectedDirectory ) )
        {
            java.awt.Toolkit.getDefaultToolkit().beep();
            int result = JOptionPane.showConfirmDialog( this,
                "By changing the selected directory, you will lose\n" +
                "any captions or other changes made before.\n\n" +
                "Are you sure you wish to change your selection?\n",
                "Warning", JOptionPane.YES_NO_OPTION,
                JOptionPane.WARNING_MESSAGE );
            switch( result ) {
                case JOptionPane.YES_OPTION:
                    // everything is okay - continue.
                    break;
                case JOptionPane.NO_OPTION:
                    ((FileSelector)pnlFileSelector).setSelectedFile(
                        lastSelectedDirectory );
                    throw new CannotChangePanelException();
            }
        }
        if( forwardDirection ) {
            lastSelectedDirectory = selectedDirectory;
        }
        selectDirectory( selectedDirectory );
    }

    public void itemStateChanged(java.awt.event.ItemEvent itemEvent) {
        tryDirectory( (File)itemEvent.getItem() );
    }

    private void updateGUI() {
        if( selectedDirectory == null ) {
            tpChoiceDescription.setText( "Selection: None\n\n" );
            btnRefresh.setEnabled( false );
        }
        else {
            btnRefresh.setEnabled( true );
            // Update UI:
            int numPhotos = tempPhotoSource.getPhotos().size();
            tpChoiceDescription.setText(
                "Selection: " + selectedDirectory.getAbsolutePath() + "\n\n" +
                "Found " + numPhotos +
                    " photo" + ((numPhotos != 1) ? "s" : "") +
                    " in this folder." );
        }
    }
}
```

Appendix F

Test Programs used for the Identification of Metric Trends

```

package voji.db;
import java.lang.*;
import java.util.*;
import java.io.*;
import java.sql.*;
import voji.utils.*;

/*
 * CLASS Database
 */
public class Database
{
    /*
     * vector of all database listeners
     */
    private static Vector listeners=new Vector();

    /*
     * add a database listener
     */
    public static void addDatabaseListener(DatabaseListener listener)
    {
    /* add listener to vector of listeners */
    listeners.add(listener);

    /* initialize listener if necessary */
    if (getConnection()!=null)
        {
        try { listener.databaseChanged(); }
        catch (Exception ex) {}
        }
    }

    /*
     * let all listeners know that the database has been changed
     */
    protected static void fireDatabaseChanged()
    {
    /* call databaseChanged() of all listeners */
    for (Iterator i=listeners.iterator();i.hasNext();)
        {
        try { ((DatabaseListener)i.next()).databaseChanged(); }
        catch (Exception ex) {}
        }
    }

    /*
     * current connection
     */
    private static Connection connection;

    /*
     * current statement
     */
    private static Statement statement;

    /*
     * connect to another database
     */
    public static void connect(Connection newConnection) throws SQLException
    {
    /* set connection */
    connection=newConnection;

    /* create statement */
    statement=createStatement();

    /* let all listeners know that the database has been changed */
    fireDatabaseChanged();
    }

    /*
     * connect to another database
     */
    public static void connect(String driver,String url,String user,
                    String password)
	throws ClassNotFoundException,SQLException
	{
	/* load driver */
	if (driver!=null) Class.forName(driver);

	/* set connection */
	connect(DriverManager.getConnection(url,user,password));
	}

	/*
	 * connect to another database
	 */
	public static void connect(String url,String user,String password)
	throws SQLException
	{
	/* set connection */
	connect(DriverManager.getConnection(url,user,password));
	}

	/*
	 * connect to another database
	 */
	public static void connect(Properties info)
	throws ClassNotFoundException,SQLException
	{
	/* set connection */
	connect(info.getProperty("voji.db.driver"),
	    info.getProperty("voji.db.url"),
	    info.getProperty("voji.db.user"),
	    info.getProperty("voji.db.password"));
	}

	/*
	 * get current connection
	 */
	public static Connection getConnection()
	{
	/* return current connection */
	return connection;
	}

	/*
	 * create statement
	 */
	public static Statement createStatement() throws SQLException
	{
	/* create statement from current connection */
	return getConnection().createStatement();
	}

	/*
	 * prepare statement
	 */
	public static PreparedStatement prepareStatement(String sql)
	throws SQLException
	{
	/* prepare statement from current connection */
	return getConnection().prepareStatement(sql);
	}

	/*
	 * prepare call
	 */
	public static CallableStatement prepareCall(String sql) throws SQLException
	{
	/* prepare call from current connection */
	return getConnection().prepareCall(sql);
	}

	/*
	 * execute query
	 */
	public static ResultSet executeQuery(String sql) throws SQLException
	{
	/* execute query at created statement */
	return statement.executeQuery(sql);
	}
```

```

      /*
       * execute update
       */
      public static int executeUpdate(String sql) throws SQLException
      {
      /* execute update at created statement */
      return statement.executeUpdate(sql);
      }
  }

```

```java

package smallsql.database;


class SQLToken{
    int value;
    int offset;  // start offset des tokens im SQL
    int length;  // Länge des Tokens
    String name;

    SQLToken (int value, int tokenStart, int tokenEnd){
        this.value  = value;
        this.offset = tokenStart;
        this.length = tokenEnd-tokenStart;
    }

    /**
     * Constructor used for quoted strings
     */
    SQLToken (String name, int value, int tokenStart, int tokenEnd){
        this.value  = value;
        this.offset = tokenStart;
        this.length = tokenEnd-tokenStart;
        this.name   = name;
    }

    String getName(char[] sql){
        if(name != null) return name;
        return new String( sql, offset, length );
    }
}

```

```

import java.awt.Color;

import jcckit.graphic.ClippingShape;
import jcckit.graphic.GraphPoint;
import jcckit.graphic.GraphicalComposite;
import jcckit.graphic.GraphicalElement;
import jcckit.graphic.LineAttributes;
import jcckit.graphic.Polygon;
import jcckit.graphic.ShapeAttributes;
import jcckit.util.ConfigParameters;
import jcckit.util.Factory;

/**
 *  A simple curve is the basic implementation of the {@link Curve} interface.
 *
 *  @author Franz-Josef Elmer
 */
public class SimpleCurve implements Curve {
  /** Configuration parameter key. */
  public static final String SYMBOL_FACTORY_KEY = "symbolFactory",
                             WITH_LINE_KEY = "withLine",
                             SOFT_CLIPPING_KEY = "softClipping",
                             LINE_ATTRIBUTES_KEY = "lineAttributes",
                             INITIAL_HINT_FOR_NEXT_POINT_KEY
                                      = "initialHintForNextPoint";
  private final ClippingShape _clippingShape;
  private final SymbolFactory _symbolFactory;
  private final GraphicalComposite _symbols;
  private final GraphicalComposite _completeCurve;
  private final GraphicalElement _legendSymbol;
  private final Hint _initialHintForNextPoint;
  private final Polygon _curve;
  private final boolean _softClipping;
  private Hint _hintForNextPoint;

  /**
   * Creates a new curve. The parameter <tt>config</tt> contains:
   * <table border=1 cellpadding=5>
   * <tr><th>Key &amp; Default Value</th><th>Type</th><th>Mandatory</th>
   *     <th>Description</th></tr>
   * <tr><td><tt>initialHintForNextPoint = null</tt></td>
   *     <td><tt>ConfigParameters</tt></td><td>no</td>
   *     <td>Definition of an initial {@link Hint} for the first curve point.
   *         </td></tr>
   * <tr><td><tt>lineAttributes = </tt>a {@link ShapeAttributes}
   *         instances with default values and line colors based on
   *         the formula <tt>Color.getHSBColor(curveIndex/6,1,0.8)</tt></td>
   *     <td><tt>ConfigParameters</tt></td><td>no</td>
   *     <td>Configuration parameters of an instances of
   *         {@link jcckit.graphic.GraphicAttributes} for the
   *         {@link Polygon Polygons} connecting curve points.</td></tr>
   * <tr><td><tt>symbolFactory = null</tt></td>
   *     <td><tt>ConfigParameters</tt></td><td>no</td>
   *     <td>Configuration parameters defining an instances of
   *         {@link SymbolFactory} for the {@link Symbol Symbols}
   *         decorating curve points.</td></tr>
   * <tr><td><tt>softClipping = true</tt></td>
   *     <td><tt>boolean</tt></td><td>no</td>
   *     <td>If <tt>true</tt> no explicit clipping takes
   *         place but the symbol is not drawn if the corresponding curve
   *         point is outside the axis box.<br>
   *         If <tt>false</tt> the symbol is
   *         drawn in any case but it may be clipped by the axis box.
   *         Soft-clipping should be set to <tt>false</tt> if the
   *         symbols are not located around the curve point (like for bars).
   *         </td></tr>
   * <tr><td><tt>withLine = true</tt></td>
   *     <td><tt>boolean</tt></td><td>no</td>
   *     <td>If <tt>true</tt> curve points are connected by a
   *         {@link jcckit.graphic.Polygon}.</td></tr>
   * </table>
   * @param config Configuration parameters described above.
   * @param curveIndex Index of this curve in the collection of curves
   *        defining a {@link Plot}.
   * @param numberOfCurves Number of curves in this collection.
   * @param clippingShape Clipping shape. Can be <tt>null</tt>.
   * @param legend Legend. Will be used to calculate the legend symbol.
   * @throws IllegalArgumentException if <tt>symbolFactory == null</tt> and
   *         <tt>withLine == false</tt>.
   *
   */
  public SimpleCurve(ConfigParameters config, int curveIndex,
                     int numberOfCurves, ClippingShape clippingShape,
                     Legend legend) {
    _symbolFactory = (SymbolFactory) Factory.createOrGet(
        config.getNode(SYMBOL_FACTORY_KEY), null);
    boolean withLine = config.getBoolean(WITH_LINE_KEY, true);
    LineAttributes lineAttributes = (LineAttributes) Factory.createOrGet(
        config.getNode(LINE_ATTRIBUTES_KEY),
        new ShapeAttributes(null, Color.getHSBColor((curveIndex % 6) / 6f,
                                                     1f, 0.8f),
                            0, null));
    if (_symbolFactory != null || withLine) {
      _clippingShape = clippingShape;
      _completeCurve = new GraphicalComposite(null);
      if (withLine) {
        GraphicalComposite container = new GraphicalComposite(clippingShape);
        _curve = new Polygon(lineAttributes, false);
        container.addElement(_curve);
       _completeCurve.addElement(container);
      } else {
        _curve = null;
      }
      _softClipping = config.getBoolean(SOFT_CLIPPING_KEY, true);
      if (_symbolFactory != null) {
        _symbols = new GraphicalComposite(_softClipping ? null
                                                        : clippingShape);
        _completeCurve.addElement(_symbols);
      } else {
        _symbols = null;
      }
    } else {
      throw new IllegalArgumentException(
          "Either a SymbolFactory must exist or withLines == true.");
    }
    _hintForNextPoint = _initialHintForNextPoint
        = (Hint) Factory.createOrGet(
            config.getNode(INITIAL_HINT_FOR_NEXT_POINT_KEY), null);
    _legendSymbol = legend.createSymbol(curveIndex, numberOfCurves,
                                        _symbolFactory, withLine,
                                        lineAttributes);
  }

  /**
   * Returns the graphical representation of a curve.
   * @return always the same instance.
   */
  public GraphicalElement getView() {
    return _completeCurve;
  }

  /** Returns the legend symbol. */
  public GraphicalElement getLegendSymbol() {
    return _legendSymbol;
  }

  /** Appends a new point to the curve if inside the clipping shape. */
  public Hint addPoint(GraphPoint point, Hint hintFromPreviousCurve) {
    if (_curve != null) {
      _curve.addPoint(point);
    }
    Hint hintForNextCurve = hintFromPreviousCurve;
    if (_symbolFactory != null) {
      Symbol symbol = _symbolFactory.createSymbol(point, _hintForNextPoint,
                                                  hintFromPreviousCurve);
      if (_clippingShape == null || !_softClipping
          || _clippingShape.isInside(point)) {
        _symbols.addElement(symbol.getSymbol());
      }
      _hintForNextPoint = symbol.getHintForNextPoint();
      hintForNextCurve = symbol.getHintForNextCurve();
    }
    return hintForNextCurve;
    }

    public void removeAllPoints() {
      if (_curve != null) {
        _curve.removeAllPoints();
      }
      if (_symbols != null) {
        _symbols.removeAllElements();
      }
      _hintForNextPoint = _initialHintForNextPoint;
    }
  }

```

Appendix G

Test Programs - Combined Domain

```
package net.sf.dc.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import net.sf.dc.console.dialogs.MessageBox;
import net.sf.dc.core.DataCrow;
import net.sf.dc.core.Repository;
import net.sf.dc.core.data.DataFilter;
import net.sf.dc.core.data.DataFilterOptions;
import net.sf.dc.core.data.DataManager;
import net.sf.dc.core.modules.IChildModule;
import net.sf.dc.core.objects.DcObject;
import net.sf.dc.messages.Messages;
import net.sf.dc.settings.DcSettings;
import net.sf.dc.wf.WorkFlow;
import net.sf.dc.wf.requests.IRequest;
import net.sf.dc.wf.requests.RequestCollection;
import net.sf.dc.wf.requests.SynchronizeWithManagerRequest;

public class DatabaseManager {

    public static DcDatabase db = null;
    public static boolean isServerClientMode = false;

    public static void initialize() {
        db = new DcDatabase();

        Connection connection = null;
        while (connection == null) {
            connection = getConnection();
        }

        try {
            db.initiliaze(connection);
        } catch (Exception exp) {
            Messages.add("Could not find or connect to the database!",
Messages._ERROR);
            Messages.add(exp);
            new MessageBox("Could not find or connect to the database!",
MessageBox._ERROR);
        }
    }

    public static int getQueueSize() {
        return db.getQueueSize();
    }

    public static void applySettings() {
        db.setDbProperies(getConnection());
    }

    public static void closeDatabases(boolean compact) {
        try {
            if (db != null) {
                Connection connection = getConnection();
                Statement stmt = connection.createStatement();

                if (!isServerClientMode) {
                    if (compact)
                        stmt.executeUpdate("SHUTDOWN COMPACT");
                    else
                        stmt.executeUpdate("SHUTDOWN");
                }

                connection.close();
            }
        } catch (Exception exp) {
            Messages.add(exp);
        }
    }
```

```
        try {
            String name = db.getName();
            Class.forName(DcSettings.getValueAsString(Repository.settings.stDatabaseDriver));

            String address;
            if (name.startsWith("//") || name.startsWith("\\\\")) {
                isServerClientMode = true;
                address = "jdbc:hsqldb:hsql:" + name;
            } else {
                isServerClientMode = false;
                address = "jdbc:hsqldb:" + DataCrow.baseDir +  "data/" + name;
            }

            Connection connection = DriverManager.getConnection(address, "SA", "");
            connection.setAutoCommit(true);

            return connection;
        } catch (Exception exp) {
            Messages.add(exp);
            new MessageBox(exp.getMessage(), MessageBox._ERROR);
        }

        System.exit(1);
        return null;
    }

    public static ResultSet runQueryDirectUnclosed(String sQuery, boolean silent) throws Exception {
        if (!silent)
            Messages.add(sQuery, Messages._QUERY);

        Connection connection = getConnection();
        Statement stmt = connection.createStatement();
        return stmt.executeQuery(sQuery);
    }

    public static Collection runQueryDirect(String sQuery, boolean catchErrors, boolean logQuery) throws Exception {
        Collection data = null;
        try {
            Connection connection = getConnection();
            Statement stmt = connection.createStatement();
            ResultSet result = stmt.executeQuery(sQuery);
            data = new WorkFlow().convertToDCObjects(result);

            result.close();
            stmt.close();
            connection.close();
        } catch (SQLException sqlExp) {
            if (!sqlExp.getMessage().equals("No ResultSet was produced")) {
                if (catchErrors)
                    Messages.add(sqlExp);
                else
                    throw sqlExp;
            }
        }

        if (logQuery)
            Messages.add(sQuery, Messages._QUERY);

        return data;
    }

    public static Collection runQueryDirect(String sQuery, boolean logQuery) {
        Collection objects = null;

        try {
            objects = runQueryDirect(sQuery, true, logQuery);
        } catch (Exception exp) {
            Messages.add(exp);
        }

        return objects;
    }

    public static Collection runQueryDirect(DcObject dco, boolean logQuery) {
        Query query = new Query(Query._SELECT, dco, null, null);
        Messages.add(query.getQueries()[0], Messages._QUERY);
        return runQueryDirect(query.getQueries()[0], logQuery);
    }

    public static  Collection runQueryDirect(Query query, boolean logQuery) {
        Collection data = new ArrayList();
        for (int i = 0; i < query.getQueries().length; i++) {
            String qry = query.getQueries()[i];
            if (qry != null) {
                Collection c = runQueryDirect(qry, logQuery);
                if (c != null) data.addAll(c);
            }
        }

        RequestCollection rc = query.getRequestors();
        if (rc != null) {
            IRequest[] requests = rc.getRequests();
            for (int i = 0; i < requests.length; i++) {
                requests[i].execute(data);
            }
        }

        return data;
    }

    public static void updateValues(DcObject dco) {
        boolean isChanged = dco.isChanged();
        if (isChanged) {
            Query query = new Query( Query._UPDATE, dco, null,
dco.getRequests());

            if (dco.isBatch())
                query.setBatch(dco.isEndOfBatch());

            query.setSilence(dco.getSilence());
            db.addQuery(query);
        }

        Collection c = dco.getChildren();
        Collection children = new ArrayList();
        children.addAll(c);
        int counter = 1;
        for (Iterator iter = children.iterator(); iter.hasNext(); counter++) {
            DcObject child = (DcObject) iter.next();
            if (child.isChanged()) {

                child.addRequest(new SynchronizeWithManagerRequest(
                        SynchronizeWithManagerRequest._UPDATE, child));

                boolean exists = false;
                if (child.getID() != null && child.getID().length() > 0) {
                    DcObject childTest = child.getModule().getDcObject();
                    childTest.setValue(DcObject._ID, child.getID());
                    Collection objects = runQueryDirect(childTest, true);
                    exists = objects.size() > 0;
                }

                Query query;
                if (!exists)
                    query = new Query(Query._INSERT, child, null,
child.getRequests());
                else
                    query = new Query(Query._UPDATE, child, null,
child.getRequests());

                if (!isChanged) {
                    query.setBatch(counter == children.size());
                    query.setSilence(counter != children.size());
                } else {
                    query.setSilence(true);
                }
                db.addQuery(query);
            }
        }
    }

    public static  void insertValues(DcObject dco) {

        Query query = new Query(Query._INSERT, dco, null, dco.getRequests());
        if (dco.isBatch())
            query.setBatch(dco.isEndOfBatch());

        query.setSilence(dco.getSilence());
        db.addQuery(query);

        Collection children = dco.getChildren();
        for (Iterator iter = children.iterator(); iter.hasNext(); ) {
            DcObject child = (DcObject) iter.next();

            child.addRequest(new SynchronizeWithManagerRequest(
                    SynchronizeWithManagerRequest._ADD, child));

            child.setValue(child.getParentReferenceFieldIndex(), dco.getID());
            query = new Query(Query._INSERT, child, null, child.getRequests());
            query.setSilence(true);
            db.addQuery(query);
        }
    }

    public static  void deleteValues(DcObject dco) {
        Query query = new Query(Query._DELETE, dco, null, dco.getRequests());
        if (dco.isBatch())
            query.setBatch(dco.isEndOfBatch());

        if (dco.getModule().getChildModule() != null) {

        }

        query.setSilence(dco.getSilence());
        db.addQuery(query);
    }


    public static boolean uniqueValues(DcObject o, boolean isUpdate) {
        if (o.hasPrimaryKey() && !(o.getModule() instanceof IChildModule)) {
            boolean hasRequiredFields = false;
            DcObject dco = o.getModule().getDcObject();

            int[] fields = o.getFieldIndices();
            for (int i = 0 ; i < fields.length; i++) {
                int field = fields[i];
                if (o.isRequired(field)) {
                    hasRequiredFields = true;
                    dco.setValue(field, o.getValue(field));
                }
            }

            if (hasRequiredFields) {
                DataFilterOptions dfo = new DataFilterOptions(null, true, true);
                DataFilter df = new DataFilter(o.getModule().getIndex(), dco,
dfo);
                DcObject[] objects = DataManager.get(o.getModule().getIndex(),
df);

                int count = 0;
                for (int i = 0; i < objects.length; i++) {
                    count = !isUpdate || !objects[i].getID().equals(o.getID()) ?
count + 1 : count;
                }

                if (count > 0)
                    return false;
            }
        }
        return true;
    }
}


```

```java
package au.com.lastweekend.jim;

import java.util.Set;

import au.com.lastweekend.jim.imagebase.ImageBase;
import au.com.lastweekend.jim.imagebase.KeywordProvider;
import au.com.lastweekend.jim.imagebase.query.CombinationCondition;
import au.com.lastweekend.jim.ui.ImageSearchResultsModel;
import au.com.lastweekend.jim.ui.SearchKeywordProvider;
import au.com.lastweekend.jim.ui.SearchProvider;

/**
 * @author grant@lastweekend.com.au
 * @version $Id: JimSearchProvider.java,v 1.3 2006/03/01 09:52:49 ggardner Exp $
 */
public class JimSearchProvider implements SearchProvider {

    private KeywordProvider _keywordProvider;
    private JimContext _jimContext;

    public JimSearchProvider(JimContext jimContext) {

        _jimContext = jimContext;
    }

    /*
     * @see au.com.lastweekend.jim.ui.SearchProvider#getKeywordProvider()
     */
    public KeywordProvider getKeywordProvider() {

        if (_keywordProvider == null) {
            _keywordProvider = new
SearchKeywordProvider(_jimContext.getImageBase());

        }
        return _keywordProvider;
    }

    /*
     * @see au.com.lastweekend.jim.ui.SearchProvider#doSearch(boolean,
java.util.Set)
     */
    public void doSearch(boolean andMatch, Set<String> selectedKeywords) {

        ImageBase imageBase = _jimContext.getImageBase();
        CombinationCondition condition =
imageBase.getCombinationCondition(andMatch);
        for (String keyword : selectedKeywords) {
            condition.add(imageBase.getEqualsCondition("keywords",keyword));
        }

        _jimContext.getJimPort().addContactSheet(new
ImageSearchResultsModel(condition, _jimContext));
    }


}
```

Appendix H

Program Templates used as Search Queries

```

import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.table.TableColumnModel;
import java.util.*;

/**
 * view a table's information and attributes
 *
 */

public class TableInfoPanel extends JPanel
{

  public TableInfoPanel(MySQLTreeNode Node, MySQLDatabase Conn)
  {
    setLayout(new BorderLayout());
    setBorder(BorderFactory.createEtchedBorder());
    connection = Conn;
    node = Node;

    add("North", getInfoPanel());
    add("Center", new JSeparator());
    add("South", getUpdatePanel());

  }

  ////////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////////

  /**
   *
   */
  public Insets getInsets()
  {
    return new Insets(8, 8, 8, 8);
  }

  /**
   * alters the table
   */
  public void UpdateTable ()
  {
    String query;

    query =
    "alter table \""+ owner+"\".\""+tableName+"\" MAX_TRANS "+maxTrans.getText()+
    " PCT_FREE "+pctFree.getText()+" PCT_USED "+pctUsed.getText()+" STORAGE("+
    "NEXT "+next.getText()+" K PCTINCREASE "+pctIncrease.getText()+" MAXEXTENTS "+
    maxExt.getText()+")";

  }


  ////////////////////////////////////////////////////////////////////////////
  // private methods
  ////////////////////////////////////////////////////////////////////////////


  /**
   * create a new panel containing the table's information
   */

  private JPanel getInfoPanel()
  {
    tablespace = new JTextField(20);
    tablespace.setEditable(false);
    pctIncrease = new NumericTextField(20);
    pctIncrease.setEditable(false);
    pctFree = new NumericTextField(5);
    pctFree.setEditable(false);
    pctUsed = new NumericTextField(5);
    pctUsed.setEditable(false);
```

```
      initial = new NumericTextField(15);
      initial.setEditable(false);
      next = new NumericTextField(10);
      next.setEditable(false);
      minExt = new NumericTextField(10);
      minExt.setEditable(false);
      maxExt = new NumericTextField(10);
      maxExt.setEditable(false);
      iniTrans = new NumericTextField(10);
      iniTrans.setEditable(false);
      maxTrans = new NumericTextField(10);
      maxTrans.setEditable(false);

      JPanel infoPanel = new JPanel();

      infoPanel.setLayout(new GridLayout(0,2,3,3));

      infoPanel.add(new JLabel("Tablespace Name:     "));
      infoPanel.add(tablespace);
      infoPanel.add(new JLabel("Percent Increased:   "));
      infoPanel.add(pctIncrease);
      infoPanel.add(new JLabel("Percent Free:        "));
      infoPanel.add(pctFree);
      infoPanel.add(new JLabel("Percent Used:        "));
      infoPanel.add(pctUsed);
      infoPanel.add(new JLabel("Initial Extent (KB):"));
      infoPanel.add(initial);
      infoPanel.add(new JLabel("Next Extent (KB):    "));
      infoPanel.add(next);
      infoPanel.add(new JLabel("Min Extents:         "));
      infoPanel.add(minExt);
      infoPanel.add(new JLabel("Max Extents:         "));
      infoPanel.add(maxExt);
      infoPanel.add(new JLabel("Intial Transactions:"));
      infoPanel.add(iniTrans);
      infoPanel.add(new JLabel("Max Transactions:    "));
      infoPanel.add(maxTrans);

      tableName = node.getParent().toString();
      owner = node.getOwner();
      String query = null;
      query = "select OWNER, TABLE_NAME, TABLESPACE_NAME, PCT_FREE, " +
      "PCT_USED,INI_TRANS,MAX_TRANS,INITIAL_EXTENT,NEXT_EXTENT," +
      "MIN_EXTENTS, MAX_EXTENTS,PCT_INCREASE,FREELISTS," +
      "FREELIST_GROUPS,LOGGING,BACKED_UP,NUM_ROWS," +
      "BLOCKS,EMPTY_BLOCKS,AVG_SPACE,CHAIN_CNT,AVG_ROW_LEN," +
      "AVG_SPACE_FREELIST_BLOCKS,NUM_FREELIST_BLOCKS,DEGREE," +
      "INSTANCES,CACHE,TABLE_LOCK,SAMPLE_SIZE,LAST_ANALYZED," +
      "PARTITIONED,IOT_TYPE,NESTED from ALL_TABLES " +
      "where TABLE_NAME= '" + tableName + "' and OWNER ='" +
      owner + "'";

      try
      {
        ResultSet rs = connection.executeQuery(query);
        if (rs.next())
        {
      tablespace.setText(rs.getString("TABLESPACE_NAME"));
      pctIncrease.setText(Integer.toString(rs.getInt("PCT_INCREASE")));
      pctFree.setText(Integer.toString(rs.getInt("PCT_FREE")));
      pctUsed.setText(Integer.toString(rs.getInt("PCT_USED")));
      initial.setText(Integer.toString(rs.getInt("INITIAL_EXTENT") / 1024));
      next.setText(Integer.toString(rs.getInt("NEXT_EXTENT") / 1024));
      minExt.setText(Integer.toString(rs.getInt("MIN_EXTENTS")));
      maxExt.setText(Integer.toString(rs.getInt("MAX_EXTENTS")));
      iniTrans.setText(Integer.toString(rs.getInt("INI_TRANS")));
      maxTrans.setText(Integer.toString(rs.getInt("MAX_TRANS")));
      rs.close();
        }
      }
      catch (Exception argh)
      {
        MessageBox.showDebug(argh.getMessage());
      }

      return infoPanel;
    }


    /**
     * create`a new update panel for the table
     */

    private JPanel getUpdatePanel()
    {
      return new UpdatePnl();
    }



    ////////////////////////////////////////////////////////////////////////////////
    // inner classes
    ////////////////////////////////////////////////////////////////////////////////


    class UpdatePnl extends JPanel
    {
      JCheckBox editable;
      JPanel pnl = new JPanel();
      JSeparator sep = new JSeparator();
      JButton update = new JButton();

      public UpdatePnl()
      {
        update.setEnabled(false);
        pnl.setLayout(new FlowLayout(FlowLayout.CENTER));
        super.setLayout(new BorderLayout(0,5));
        update.setToolTipText ("Update Table Settings");
        update.setText ("Update Table");
        update.addActionListener (new ActionListener ()
        {
          public void actionPerformed (ActionEvent evt)
          {
            UpdateTable();
          }
      });


        editable = new JCheckBox("Editable?");
        editable.addItemListener(new ItemListener ()
        {
          public void itemStateChanged(ItemEvent e)
          {
            if (e.getStateChange() == ItemEvent.DESELECTED)
            {
              pctIncrease.setEditable(false);
              pctFree.setEditable(false);
              pctUsed.setEditable(false);
              next.setEditable(false);
              minExt.setEditable(false);
              maxExt.setEditable(false);
              maxTrans.setEditable(false);
              update.setEnabled(false);
            }
            else
            {
              pctIncrease.setEditable(true);
              pctFree.setEditable(true);
              pctUsed.setEditable(true);
              next.setEditable(true);
              minExt.setEditable(true);
              maxExt.setEditable(true);
              maxTrans.setEditable(true);
              update.setEnabled(true);
            }
          }
      });

      super.add(sep, "North");
      super.add(pnl, "Center");
      pnl.add(editable);
      pnl.add(update);
      }
    }


    protected String           tableName;
    protected String           owner;
    protected JTextField       tablespace;
    protected NumericTextField pctFree;
    protected NumericTextField pctUsed;
    protected NumericTextField pctIncrease;
    protected NumericTextField initial;
    protected NumericTextField next;
    protected NumericTextField minExt;
    protected NumericTextField maxExt;
    protected NumericTextField iniTrans;
    protected NumericTextField maxTrans;
    protected MySQLDatabase   connection;
    protected MySQLTreeNode   node;
}

```

```
import java.io.*;
import java.util.Map;
import java.util.Map.Entry;
import java.awt.Image;

public class JdaiPhotoFile implements JdaiPhoto {

    private JdaiSection section;
    private String id;
    private String fileName;
    private EXIFInfo exif;
    private JdaiProgressListener progress;

    private static LRUCache cache = new LRUCache(150);

    private static class LRUCache extends java.util.LinkedHashMap {
        private int maxsize;
        protected boolean removeEldestEntry(Entry eldest) {
            return size() >= maxsize;
        }
        public LRUCache(int maxsize) {
            super(maxsize * 4 / 3 + 1, 0.75f, true);
            this.maxsize = maxsize;
        }
    }

    public JdaiPhotoFile(JdaiSection section, String id, String fileName) {
        this.section = section;
        this.id = id;
        this.fileName = fileName;
    }

    public JdaiSection getSection() {
        return section;
    }

    public String getId() {
        return id;
    }

    /**
     * Get a thumbnail of the photo as a BufferedImage for displaying.
     */
    public Image getThumbnail() throws JdaiReadException {
        Image image = null;
        if (cache.containsKey(fileName)) {
            image = (Image) cache.get(fileName);
        } else {
            setupExif();
            if (exif.hasThumbnail()) {
                try {
                    image = exif.getThumbnail();
                } catch (IOException e) {
                    throw new JdaiReadException(e.getMessage());
                }
            } else {
                File thumbFile = new File(fileName + ".thm");
                if (thumbFile.exists()) {
                    image = JdaiImageHelpers.readJpegFile(thumbFile);
                } else {
                    image = getImage(160, 160);
                    try {
                        JdaiImageHelpers.writeJpegFile(image, thumbFile);
                    } catch (JdaiWriteException e) {
                    }
                }
            }
            int rotation = getSection().getInfoStore().getRotation(this);
            image = JdaiImageHelpers.rotate(image, rotation);
            cache.put(fileName, image);
        }
        return image;
    }

    /**
     * Refresh the thumbnail of this photo. Is a thumbnail has not been loaded
     * this method does nothing - otherwise it tells the photo to reload the
     * thumnail next time it's needed.
     */
    public void refreshThumbnail() {
        if (cache.containsKey(fileName))
            cache.remove(fileName);
    }

    /**
     * Get the photo itself as an Image for displaying.
     * @return The image.
     * @exception JdaiReadException Thrown when image could not be read.
     */
    public Image getImage() throws JdaiReadException {
        return getImage(0, 0);
    }

    /**
     * Get the photo itself as an Image for displaying. This
     * method supports resizing the image in a bounding box (the image
     * is never enlarged).
     * @param width Maximum width of the image
     * @param height Maximum height of the image
     * @return The image.
     * @exception JdaiReadException Thrown when image could not be read.
     */
    public Image getImage(int width, int height) throws JdaiReadException {
        Image result = JdaiImageHelpers.readJpegFile(new File(fileName), width,
height, progress);

        int rotation = getSection().getInfoStore().getRotation(this);

        result = JdaiImageHelpers.rotate(result, rotation);
        return result;
    }

    /**
     * Compare to another photo (sort).
     * @param o The other photo.
     * @return The compare value (see Comparable interface)
     */
    public int compareTo(Object o) {
        int result;
        if ((result = section.compareTo(((JdaiPhoto) o).getSection())) == 0)
            result = id.compareTo(((JdaiPhoto) o).getId());
        return result;
    }

    public boolean equals(Object o) {
        if (o == null)
            return false;
        return compareTo(o) == 0;
    }

    /**
     * Get a readable string representation of the photo.
     * @return The string representation.
     */
    public String toString() {
        return getId();
    }

    /**
     * Copy to another file-based photo.
     * @param other The photo to copy to.
     */
    public void copyTo(JdaiPhoto other) throws JdaiReadException,
JdaiWriteException {
        if (other instanceof JdaiPhotoFile) {
            JdaiPhotoFile o = (JdaiPhotoFile) other;
            try {
                FileInputStream in = new FileInputStream(new File(fileName));
                FileOutputStream out = new FileOutputStream(new
File(o.fileName));
                byte[] buf = new byte[1024];
                int c;
                while ((c = in.read(buf)) != -1)
                    out.write(buf, 0, c);
                in.close();
```

```java
            } catch (FileNotFoundException e) {
                throw (new JdaiReadException(e.getMessage()));
            } catch (IOException e) {
                throw (new JdaiWriteException(e.getMessage()));
            }
            JdaiPhotoInfoStore is1 = getSection().getInfoStore();
            JdaiPhotoInfoStore is2 = o.getSection().getInfoStore();
            int r;
            if ((r = is1.getRotation(this)) != JdaiPhotoInfoStore.NORTH)
                is2.setRotation(other, r);
            String s;
            if (!(s = is1.getCaption(this)).equals(""))
                is2.setCaption(other, s);
            if (!(s = is1.getKeywords(this)).equals(""))
                is2.setKeywords(other, s);
        }
    }

    public void delete() throws JdaiReadException, JdaiWriteException {
        File photoFile = new File(fileName);
        if (!photoFile.delete()) throw new JdaiWriteException("Unable to delete file: " + fileName);
        getSection().getInfoStore().deleteInfo(this);
    }

    /**
     * Get meta information from the photo (e.g. EXIF from digital camera photos).
     * @return A Map of String, String pairs of metadata.
     */
    public Map getMetaInfo() {
        Map infoMap;
        setupExif();
        infoMap = exif.getEXIFMetaData();
        infoMap.put("Id", getId());
        return infoMap;
    }

    private void setupExif() {
        if (exif == null) {
            exif = new EXIFInfo(new File(fileName));
        }
    }

    /**
     * Get meta information from the photo (e.g. EXIF from digital camera photos).
     * @return An HTML String with pretty-printed metadata.
     */
    public String getMetaInfoHtml() {
        Map infoMap = getMetaInfo();
        String[] fieldList = EXIFInfo.getFieldList();
        StringBuffer infoStrBuf = new StringBuffer();
        infoStrBuf.append("<table cellspacing=0 cellpadding=0>");

        for (int i = 0; i < fieldList.length; i++) {
            String key = fieldList[i];
            if (infoMap.containsKey(key)) {
                infoStrBuf.append("<tr><td><font face=\"Helvetica,Arial,sans-serif\" size=\"-1\"><b>");
                infoStrBuf.append(JdaiGuiHelpers.escapeHtml(EXIFInfo.getFieldName(key)) + ":&nbsp;");
                infoStrBuf.append("</b></font></td><td><font face=\"Helvetica,Arial,sans-serif\" size=\"-1\">");
                infoStrBuf.append(JdaiGuiHelpers.escapeHtml((String) infoMap.get(key)));
                infoStrBuf.append("</font></td></tr>");
            }
        }
        infoStrBuf.append("</table>");
        return infoStrBuf.toString();
    }


    /**
     * Sets which listener should receive info about progress of reads
     *
```

```
            this.progress = progress;
        }
    }

```

```
public class QueryData extends EmbeddedData
   implements javax.servlet.jsp.tagext.TryCatchFinally {
   private static Log logCat = LogFactory.getLog(QueryData.class.getName());

   private String query;

   public void setQuery(String query) {
      this.query = query;
   }


   public String getQuery() {
      return query;
   }


   public void doCatch(Throwable t) throws Throwable {
      throw t;
   }


   public void doFinally() {
      query = null;
      super.doFinally();
   }


   protected List fetchData(Connection con) throws SQLException {
      logCat.info("about to execute user defined query:" + query);

      ResultSetVector   rsv = null;
      PreparedStatement ps = con.prepareStatement(query);

      try {
         rsv = new ResultSetVector();

         HttpServletRequest     request = (HttpServletRequest) pageContext
            .getRequest();
         DbEventInterceptorData data = new DbEventInterceptorData(request,
               getConfig(), con, null);
         data.setAttribute(DbEventInterceptorData.PAGECONTEXT,
                pageContext);
         rsv.addResultSet(data, ps.executeQuery());
      } finally {
         ps.close(); // #JP Jun 27, 2001
      }

      return formatEmbeddedResultRows(rsv);
   }
}
```

```

import javax.swing.*;
import java.awt.*;
import java.net.URL;
import java.io.*;
import java.awt.event.*;


public class PictureApplication {
    /* General Application variables */
    private PictureApplet myApplet;
    private Container myContentPane;
    private PicturePanel myPanel;
    private PictureControls myControls;
    private int paramWidth, paramHeight;
    private int actualWidth, actualHeight;
    private URL baseURL;
    private ActionListener myTimerListener;
    private Timer myTimer;

    /* Pictures variables */
    private BufferedReader pictureListReader;
    private String pictureListFile;
    private Picture pictures[];
    private Image currentImage, offscreenImage;
    private int nbPictures, currentPosition;
    private final int MAX_IMAGES = 2048;
    private PictureWorker pictureWorker;

    public void init(PictureApplet applet) {
        int i = 0;
        String fileList[] = new String[MAX_IMAGES];
        myApplet = applet;

        paramWidth = Integer.parseInt(myApplet.getParameter("width"));
        paramHeight = Integer.parseInt(myApplet.getParameter("height"));
        pictureListFile = myApplet.getParameter("pictureList");
        myContentPane = myApplet.getContentPane();

        baseURL = myApplet.getCodeBase();
        try {
            pictureListReader = new BufferedReader(new
InputStreamReader(getURL(baseURL, pictureListFile).openStream()));
            while((fileList[i++] = pictureListReader.readLine()) != null) {}
        }
        catch (Exception e) {}

        nbPictures = i - 1;
        currentPosition = 0;

        pictures = new Picture[MAX_IMAGES];
        for(i = 0; i < nbPictures; i++) {
            pictures[i] = new Picture(getURL(baseURL, fileList[i]));
        }

        myContentPane.add("North", myControls = new PictureControls(this));
        myContentPane.add(myPanel = new PicturePanel(this));

        myTimerListener = new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                pictureWorker = new PictureWorker();
                pictureWorker.start();
            }
        };
        myTimer = new Timer(1000, myTimerListener);
        myTimer.start();

        myControls.hidePreviousButton();
        myControls.hideNextButton();
    }

    /*
     * Scales the original image to fit in the applet window and writes it
     * on an offscreen buffer
    */
    private void createOffscreenImage() {
        float ratio;
        float iw, ih;
        Graphics big;
        Graphics2D big2D;

        big = offscreenImage.getGraphics();
        big2D = (Graphics2D) big;
        big2D.clearRect(0,0,actualWidth,actualHeight);

        currentImage = pictures[currentPosition].getImage();

        iw = (float)currentImage.getWidth(myPanel);
        ih = (float)currentImage.getHeight(myPanel);
        ratio = iw / ih;

        if (actualHeight * ratio > actualWidth) {
            big.drawImage(currentImage, 0, 0, actualWidth,
(int)(actualWidth/ratio), myPanel);
        }
        else {
            iw = (int)(actualHeight*ratio);
            big.drawImage(currentImage, (int)((actualWidth - iw)/2), 0,(int)iw,
actualHeight, myPanel);
        }
    }

    public Image getOffscreenImage() {
        return offscreenImage;
    }

    public PicturePanel getPanel() {
        return myPanel;
    }

    public void previousPicture() {
        if (currentPosition == 0) {
            currentPosition = nbPictures - 1;
        }
        else currentPosition--;
        setButtons();
        createOffscreenImage();
        myPanel.repaint();
    }

    public void nextPicture() {
        if (currentPosition == nbPictures - 1) {
            currentPosition = 0;
        }
        else currentPosition++;
        setButtons();
        createOffscreenImage();
        myPanel.repaint();
    }

    public void start() {
        actualWidth = paramWidth;
        actualHeight = paramHeight - myControls.getHeight();

        offscreenImage = myPanel.createImage(actualWidth, actualHeight);
        createOffscreenImage();
        myPanel.repaint();
    }

    private URL getURL(URL codeBase, String file) {
        URL url = null;
            try {
            url = new URL(codeBase, file);
        }
        catch (java.net.MalformedURLException e) {
                System.out.println("Couldn't create image: "
                + "badly specified URL");
            return null;
        }
        return url;
    }

    public URL getURL(String file) {
        return getURL(baseURL, file);
    }

    public void setButtons() {
        if (currentPosition != nbPictures - 1) {
            if (pictures[currentPosition + 1].isFinishedLoading()) {
                myControls.showNextButton();
            }
            else {
                myControls.hideNextButton();
            }
        }
        else {
            if (pictures[0].isFinishedLoading()) {
                myControls.showNextButton();
            }
            else {
                pictures[0].getImage();
                myControls.hideNextButton();
            }
        }

        if (currentPosition == 0) {
            if (pictures[nbPictures - 1].isFinishedLoading()) {
                myControls.showPreviousButton();
            }
            else {
                pictures[nbPictures - 1].getImage();
                myControls.hidePreviousButton();
            }
        }
        else {
            if (pictures[currentPosition - 1].isFinishedLoading()) {
                myControls.showPreviousButton();
            }
            else {
                myControls.hidePreviousButton();
            }
        }
    }

    /*
     * A thread to always keep the previous image
     * in memory and preload next images so the wait time is not
     * too dramatic :-)
    */
    private class PictureWorker extends SwingWorker {
        private final int NB_PICTURES = 5; // to look ahead
        public Object construct() {
            int i;
            int start=0, stop=nbPictures;
            if (currentPosition == nbPictures - 1) start = 1;
            if (currentPosition == 0) stop = nbPictures - 1;

            if (currentPosition > 0) pictures[currentPosition - 1].getImage();

            for(i = currentPosition; i < currentPosition + NB_PICTURES; i++) {
                if (i < stop) {
                    pictures[i].getImage();
                }
            }
            for(i = currentPosition + NB_PICTURES; i < stop; i++) {
                pictures[i].delete();
            }
            for(i = start; i < (currentPosition - 1); i++) {
                pictures[i].delete();
            }
            setButtons();

            return null;
        }

        public void finished() {
        }
    };
};
```

APPENDIX I

POST-EXPERIMENTS QUESTIONS AND EVALUATION FORM

<u>Questions Asked in the Post-Experiment</u>

Name(*optional*): .... Year of Study: ....

1. How many years have you been writing source code program? ______________

2. What is your current major programming language? ________________

3. When did you start learning Java? How often do you program with it?____

4. What did you think your programming level/skill in Java is, on a scale from 1(beginner) to 5(master)? _________

5. During most of your time developing application, how often do you also act as the system analysis, responsible for problem solving? Rate yourself using rating scale of 1(never) to 5(all the time). _________

6. Do you have any knowledge on designing problem solving, specifically using design patterns? If so, given a rating scale 1(no knowledge at all) to 5(master), how do you rate yourself? _________

7. Based on the scale of 1(no knowledge at all) to 5(master), how do you rate yourself on the knowledge of software metrics (eg: couplings between objects, weighted methods per class? __________

8. Based on the given task and using rating scale of 1(not satisfied at all) to 5(very satisfied), how satisfied are you with the delivered programs? __________

TASK: Pick Photo Panel

- Panel that allows the user to configure where the photos are currently stored.
- Implements **one-to-many** dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.

PROGRAMS (Combination Approach)	**RELEVANCY** (tick √ where appropriate)	**Rank/Order of Relevant Programs (Precision)**
RenameAndDescribePanel		
FullScreen		
SummaryPanel		
Worker		
PickOutputPanel		
UnformattedTextHandler		
SlideShow		
JasperReportServlet		
LineReportServletAbstract		
AppAnisS		