UTILIZATION OF JAVA REFLECTION IN DETECTING OBJECT CONCEPT SIMILARITIES

A master project submitted to the Graduate School of in partial
fulfillment of the requirement for the degree of
Master of Science (Information Technology) (MSc.IT)
Universiti Utara Malaysia.

by

TAN CHOO JUN

Sekolah Siswazah
(Graduate School)
Universiti Utara Malaysia

PERAKUAN KERJA KERTAS PROJEK
(Certification of Project Paper)

Saya, yang bertandatangan, memperakukan bahawa
(I, the undersigned, certify that)

TAN CHOO JUN

calon untuk Ijazah
(candidate for the degree of) Master of Science (Information Technology)

telah mengemukakan kertas projek yang bertajuk
(has presented his/her project paper of the following title)

UTILIZATION OF JAVA REFLECTION
IN DETECTING OBJECT CONCEPT SIMILARITIES

seperti yang tercatat di muka surat tajuk dan kulit kertas projek
(as it appears on the title page and front cover of project paper)

bahawa kertas projek tersebut boleh diterima dari segi bentuk serta kandungan, dan meliputi bidang ilmu dengan memuaskan.
(that the project paper acceptable in form and content, and that a satisfactory knowledge of the field is covered by the project paper).

Nama Penyelia
(Name of Supervisor): Mohd. Zamberi Saad

Tandatangan
(Signature) : [illegible]

Tarikh
(Date) : 8th March 2000

ABSTRAK

Projek ini membangunkan aplikasi "Java Reflection". Ia menggunakan package system java.lang.reflect yang dibangunkan oleh Sun Microsystem dalam JDK versi 1.2 dan ke atas. "Reflection", juga dikenali sebagai "Introspection", mempunyai keupayaan "melihat kandungan" sesuatu kelas atau objek. Ia digunakan untuk mengekplotasi kandungan fail kelas. Dengan bantuan enjin penganalisa, aplikasi yang dibangunkan berkeupayaan menghasilkan maklumat kesamaan antara objek tanpa merujuk kepada kod asal. Metodologi Object-Oriented, khususnya Teknik Permodelan Objek, digunakan untuk membangunkan aplikasi ini. Terdapat empat tahap untuk dirujuk dalam methodologi ini, iaitu analisis, rekabentuk sistem, rekabentuk objek, dan implementasi. Inputnya adalah fail objek Java, dan output aplikasi mengandungi maklumat persamaan pada fail-fail input. Maklumat objek dibahagikan kepada lima kategori, iaitu Modifier, Interface, Field, Method, dan Constructor. Sistem akan menghasilkan maklumat kesamaan di antara dua fail objek untuk setiap kategori, termasuk kategori yang digunakan dan dikemukakan. Kekerapan maklumat kesamaan juga merupakan sub komponen output terperinci sistem. Kesimpulannya, aplikasi ini adalah alat alternatif untuk membandingkan sekumpulan fail objek dengan pantas dan menghasilkan output yang mudah difahami. Contoh kegunaan aplikasi ini adalah sebagai satu alat membantu pensyarah menilai tugasan pelajar dengan satu model ideal dan kriteria penilaian yang malar. Selain itu, ia juga sesuai digunakan untuk membandingkan tahap kesamaan antara dua jawapan pelajar bagi mengesan plagiarisme.

ABSTRACT

This project is about developing a Java reflection application. It utilizes the reflection features in the system package java.lang.reflect of Sun Microsystems' JDK version 1.2 and above. Reflection, also named as Introspection, has the ability to "look inside" a class or an object (Lemay, 1996). It uses to explore the content of the class tiles. With the help of the analyzer engine, the developed application is capable to produce similarity object's information between the inputs without referring to the source code. The Object-Oriented Methodology, specifically the Object Modeling Technique, is used to develop this Reflection Application. There are four stages involving analysis, system design, object design, and implementation that are followed in this methodology. The input is the Java object files, and the output contains of similarity information of those object files. The object's information is divided into five categories including Modifier, Interface, Field, Method, and Constructor. The system address the similar information for each category between two object files to the user, which including the similar used and declared category. The similar items' frequency will also be an element of the system's detail output. As a conclusion, this application is an alternative tool to compare a group of object files in fast mode with readable result in application's output. The example of the application usage is as a contributing tool to help lecturers to evaluate student assignments with an ideal model answer with constant evaluating criteria requirements. It is also suitable to be used in determining student plagiarism.

ACKNOWLEDGEMENTS

Many people contributed to the successful completion of my project at Universiti Utara Malaysia. First, I would like to express deep gratitude to my supervisor Mohd. Zamberi Saad for his valuable guidance and advice, which contributed substantially towards the completion of this study. My gratitude is also extended to my course coordinator, Associate Professor Dr. Wan Rozaini Sheik Osman, for all the help extended and meticulously undergoing the project, and giving numerous suggestions in overall presentation.

Special thanks give to Associate Professor Haji Abdul Razak Ismail, deputy Dean of Language and Scientific Thinking School, who guide me in completing this academic writing in Information Technology's field. Also, not forgetting the former Dean of Information Technology School, Associate Professor Shahrum Hashim, Dr. Mubarak Rahamathulla Ali, and William Koh Siew Yan who inspired me to explore various fields in research, and finally undertake this programming area of the project. I am grateful to Mr. Wong Chee Onn, lecturer of Information Technology School, for his kindliness to correct this report's grammars and spelling mistakes.

Finally, I express a deep sense of gratitude to my family and my dearest Joyce Ong for their valuable untiring and moral support.

CONTENT PAGE

Chapter One

Introduction

This project uses Java technology, a platform-independent[1] application, developed by Sun Microsystems. The Java programming offers "reflection" features, which is capable to examine or interpret Java compiled codes, and turns them into actual uses of classes, method and so on. The main technology applied in this project is the Java's reflection feature that uses it to reflect the similarities of object concepts.

[1] Refer to Appendix E: Glossary, page 79.

Chapter Two

The purpose and problem statement

The purpose of the study is to explore the capability of Java Reflection in reflecting the content of object file, and in producing meaningful information in Java application's output. In Whale's research (1990), the work stated the plagiarism detection system is critically depends on the choice of program representation to identify similarity of program in large populations. There were two categories of conventional software metric used to represent program as listed by Whale (1990), and they were not efficient in addressing program information when applied to complex program. From the previous study in detecting plagiarism system showed that for the system to cope with sophisticated plagiarism techniques, it needs to involve complex programming (Whale (2), 1990).

Therefore, this project is carried out to discover the difficulties in gathering information for plagiarism analysis. The produced system is used to compare a number of different object files for detecting object concept similarities. It is capable to address destination's object information through the object file. The object concept is divided into five categories including Modifier, Interface, Field, Method, and Constructor. The developed system will address the similar information for each category between two object files to the user. For example the category Method, the similar information addresses by the system including the similar used Method and declared Method occur in particular set of both object files. The similar items' frequency will also be an element of the system's detail output. For partial system's output, a percentage will be computed by the system separately subject to particular set of two object files upon all the category similarities' value. This exploration might be the solution to address complex program information in Whale's (1990) work.

As mentioned before, this application is using the reflection features in Java technology to collect the hidden class information in Object-Oriented environment from the object file. Unfortunately, the exploration of this application is only valid in Java environment. This means that the source input or destination object file to be compared must be originated from Java's object file.

2.1 The background of the solution

This application is a console program, and it is different as compared to Java Applet[2]. It will only read input from the keyboard, and display text output on the screen. Although it is a console program, the nature of the program is not limited to user interaction (Jaworski, 1998) comparing to Java Applet, which is reading the keystrokes and mouse clicks, and display graphical shapes in a windows. Console program's strength is minimizing the program's complexities,

[2] Ibid.

especially in term of excluding the Graphical User Interfaces' components coding. The emphasize of this project is more to the engine's logic to examine orderly the hidden content information in object file compared to the presentation of graphical interface to the user.

The most basic Input-Output (I/O) performed by this console application is reading data entered at the user's keyboard and writing data to the user's console. In Java, the system class of the java.lang package is used to perform keyboard input and console output (Jaworski, 1998). The java.lang package is one of the packages of the Core Java API, and it provides a number of classes and interfaces that are fundamental to Java programming. The Package class[3] is new to JDK 1.2. It is used to provide version information about a package. The Object class[4] is at the top of the Java class hierarchy. All classes are subclasses of Object and therefore inherit its methods. The Class class[5] is used to provide class descriptors for all objects created during Java program execution.

The java.lang.reflect package is part of the system class of the java.lang package (Arnold, 1997). It contains the classes and interfaces, which enable classes, interfaces, and objects to be examined, and their public fields, constructors, and methods to be discovered and used at runtime. These capabilities are used by Java Beans, object inspection tools, Java runtime tools such as the debugger, and other Java applications and applets.

[3] Ibid.
[4] Ibid.
[5] Ibid.

The java.lang.reflect package consists of the Member interface[6] and seven classes: AccessibleObject[7], Array[8], Constructor[9], Field[10], Method[11], Modifier[12], and ReflectPermission[13] (Jaworski, 1998).

2.2 The reflection

The package java.lang.reflect contains the Java reflection package, the classes that can be used to examine a type in detail. Reflection, also called as introspection, has the ability to "look inside" a class or an object (Lemay, 1996). It enables the application to get information about destination object's variables and methods as well as actually set, and get the values of those variables and to call methods. In other case's example, the developer can write a complete type browser using classes, or write an application that interprets code that a user writes, turning that code into actual uses of classes, method, etc.

[6] Ibid.
[7] Ibid.
[8] Ibid.
[9] Ibid.
[10] Ibid.
[11] Ibid.
[12] Ibid.
[13] Ibid.

2.2.1 The use of reflection

Object reflection is useful for tools such as class browsers or debuggers, where getting the information of an object. It allows the user to explore what that object can do, or for component-based programs, such as Java Beans, where the ability for one object to query another object about what it can do (and then ask it to do something) is useful to build larger applications.

2.2.2 The support version of JDK for reflection

The classes that support reflection of Java classes and objects are part of the core Java 1.1 API and above. It means this reflection feature was introduced since Java Development Kit (JDK) 1.1. They are not available in the earlier version, like JDK version 1.0.2.

2.2.3 The content of java.lang.reflect

A package, java.lang.reflect, contain classes to support reflection, which includes the following as mentioned above:

a) Field, for managing and finding out information about class and instance variables.

b) Method, for managing class and instance methods.

c) Constructor, for managing the special methods for creating new instances of classes.

d) Array, for managing arrays.

e) Modifier, for decoding modifier information about classes, variables and methods.

In newest JDK version 1.2 or until version 1.2.2, it provides the capability to identify the same features as above with additional part in Java language access controls (Jaworski, 1998). This permits reflection to be better used with the more flexible JDK version 1.2 security model.

Chapter Three

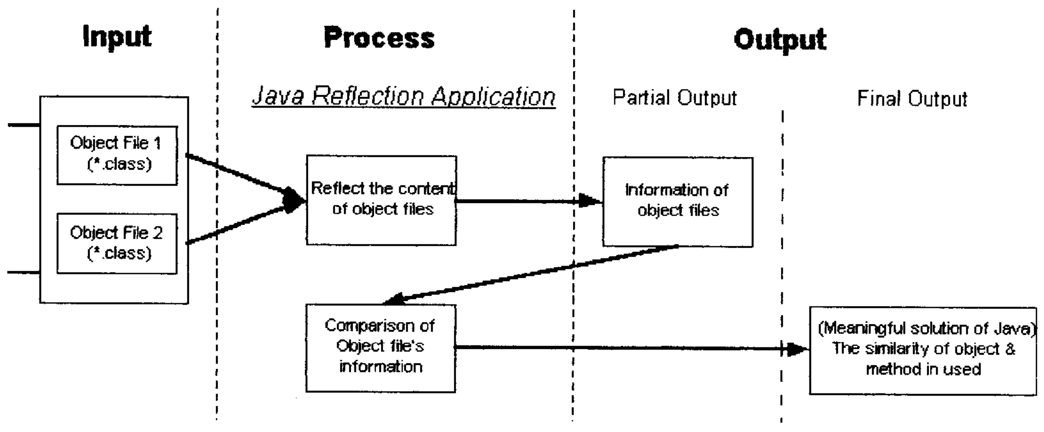
The project definition

The purpose of the study is to explore the capability of Java Reflection in reflecting the content of object file, and producing meaningful information in Java application's output. This project develops a Java console application by utilizing the Reflection features in the system package java.lang.reflect of Sun Microsystems's Java Development Kit[14] version 1.2.

3.1 The nature of the application

In executing this application, at least two Java object file is required to be located in the destination folder. With the Java technology, the application is capable to detect the similarity of object information between these two object files. To detect the object file's information, the application will use the system package java.lang.reflect, which have the ability to look inside a class file of Java. The information reflected in this application includes the Modifier, Interface, Field, Method, and Constructor, including the used and declared, and the frequency they invocated.

[14] Ibid.



Input
Process
Output
Java Reflection Application
Partial Output
Final Output
Object File 1 (*.class)
Object File 2 (*.class)
Reflect the content of object files
Information of object files
Comparison of Object file's information
(Meaningful solution of Java) The similarity of object & method in used

To examine the previous mentioned application's partial output, Java classes were built to compare the object files information. The result of examination includes the similarity of object and method in two object files.

By using the similarity of object's information, one can compare a number of different systems for detecting potential plagiarisms in program's source code. In Verco & Wise paper (1996), Parker and Hamblen define software plagiarisms as a program that has been produced from another program with a small number of routine changes. There are many systems built based on attribute counting and structure-metric system in literature review, which were discovered in Verco & Wise paper regarding the detecting suspected plagiarism. These automatic plagiarism detection systems used to overcome the persistent problem in generating program source code, for example in carrying out programming assignments of university courses. This is due to the relative ease to own the same capability program without owning the skill by altering another people's program.

This application can also be used to examine the differential of two-object file directly without referring to their source code. From the information obtained by the application, the program version can be easily retrieved. In developing large application, one cannot avoid confusion with the contents of sheet source code. Sjoberg, Welland & Atkinson (1997) emphasized that the software constraint of construction for large application system is to prevent applications from becoming unnecessarily large, complex and confusing. With this application, programmer not only can retrieve the program's version, but also the solution for above confusion. The application's output will be the meaningful information of two checked object files, which is also the summary information of sheets source code. With the similarities object information, the programmer can trace back the developing phases, and only concentrate on the differential in program. Indirectly, this application will contribute towards faster proceeding in further system enhancement process.

With the capability of this application to reflect class information, and with the help of system package java.lang.reflect, of course, the user still can obtain benefits of type-safety to linguistic reflection yields, which is stated by Kirby & Morison (1998), in proceeding of generating safely new programs.

3.2 The project assumption

This application will be preferable for those who do not own the source code but intend to examine object code. With the help of this application, the user can get the information of the object file. To execute this application, the user has to fulfill the following requirements.

a) The application is utilizing the Java technology, so the input of application must originate from a Java object file. This requirement is the key to ensure the validation of operation of the application.

b) The application is platform-independent application, and the user can execute the application in any platform with pre-installed and setup the JDK, at least with the tested version 1.2.

c) To suit the user's specification in object concept analysis, user is required to examine the information in file **user.ini**, and alter them to obtain the expected result.

d) This application is a tool developed for those who are familiar with Java application development. One must be well known in using the JDK development tools, or at least the steps to execute Java application and compile the source code.

3.3 The scope and objective

The coverage of the project only consists of the development of reflection application with Java programming language. The project put emphasis on the engine of reflecting, and the final output in presenting the similarities result in percentage form. The reflection application exploring the object information base on the content of object files, which should be located in the same folder where the reflection application resides. The final output percentage is gathering the similarity percentage of Modifiers, Interfaces, Fields, Methods, and Constructors. This result is presented on the screen after executing the application. However, the user is recommended using the Java Integrated Development Environment (IDE) tools, such as Tek-Tools KAWA IDE, to execute the application to obtain the scrollable output on screen.

The objective of this project is to produce to those who interested to discover the similarity of Java object file information without referring to the source code. It is an alternative tool to comparing a group of object files in fast mode with readable result in application's output.

Chapter Four

The literature review

Linguistic reflection defined by Kirby & Morrison (1998) as a program's ability to generate new program fragments, and to integrate them into its own execution. By incorporating these new program fragments into the ongoing computation make the persistent applications environment, including supporting safe evolution of long-lived programs and data, and specifying highly generic programs that may be reused in many contexts. Kirby & Morrison showed this style of reflection in the compiled, strongly typed language Java, and used as a paradigm for program generation.

In strongly typed systems, the linguistic reflection process includes checking of the generated program fragments to ensure type safety. Two varieties of type-safe linguistic reflection[15] can be identified. These vary as to the time at which generator execution takes place. Kirby & Morrison (1998) only concentrated on the run time linguistic reflection, and emphasized on the type-safe generation and binding of new code. They also described the nature of linguistic reflection, and the way it can be integrated with a compiled strongly typed language, such as Java, by the facility of JDK version 1.1 to provide the basic facilities of linguistic reflection. The finding shows that the strongly typed run time linguistic reflection allows programs to

[15] Ibid.

generate new programs safely. The strongly typed run time linguistic reflection consists more information to reflective computation, in the form of systematically required types.

In an earlier paper, Stemple (1993) and group proved that the integration of linguistic reflection, strong typing and static checking can provide a uniform mechanism for the production and evolution of data and programs that exceeds the capabilities of present database programming languages.

From above review, they showed that linguistic reflection application is supporting evolution in strongly typed persistent systems. The inevitable changes to meta-data in long-lived systems give rise to the problem of consistently changing all the affected programs and data. Given some mechanism for locating the relevant programs and data, linguistic reflection can be used to introduce transformed versions in a controlled manner. Unfortunately, the results had shown only understood by those who know what they are doing in the application or the content of particular class.

Another issue raised up in Whale's research (1990) was plagiarism detection system is critically dependent on the choice of program representation to identify similarity of program in large populations. There were two categories of conventional software metric used to represent program were listed by Whale (1990), and they were not efficient in addressing program information when they are applied to complex program. In another work of Whale, he stated that for detecting plagiarism system to cope with sophisticated plagiarism techniques, the system have to involve complex programming (Whale (2), 1990). This complex programming involvement occurs as to cope with the sophisticated plagiarism techniques in source code; example stated by Whale is demonstrating similarities to non-specialists in quantifying the degree of similarity between large nominated program pairs. It was another difficulty to gather information for plagiarism analysis in previous works.

Kirby (1992) made a comment that the reflective languages are hard to use as they contain a mixture of several kinds of code to respect to their role in reflection. The languages used in early stage, which stated by Kirby (1998), in reflecting program are Lisp, Scheme, TRPL, CRML, PS-algol, Napier88, and POP-2. In the same research, Kirby suggested further work could be carried on Java-specific aspects of linguistic reflection, and providing better support for writing generators. Unfortunately, all of them were based on nominated source code input to obtain specific result. Example, in the Kirby (1998) suggestion, the same package of JDK was used as this project, but it was used to make changes in the core classes so that the developed system by Kirby can concurrently implementing the main memory based compiler. The nominated target input is source code. This meant the existing application could only capable to analyze or "look into" the source code, and not the object code that had been compiled by the compiler, for specific purposes according to the researcher's research objective.

In this project, the same Java package, which had suggested by Kirby (1998), is used to develop the reflection application, but the required nominated target input for developed application is object file. This main project's idea is to discover the capability of Reflection feature in Java programming language, and adopt it to detecting the object concept similarity also suitable for plagiarism analysis. The output of the application is easy to understand by non-expert Java programming user, whereby the comparison similarity result, which compiled in a separate set of two object files, is presented in percentage of similarity. The detail of similarity items and frequency occurrences of particular items can be traced back through the detail information output. The detail information is listed in set of two object files, and with spacing to separate each set of comparison. It is easy to view and identify similar programs in large populations, with the help of value after each object file's name in detail information list. The developed application is ready to quantify the degree of similarity for large nominated program pairs without altering the logic or source code of the application.

There are two kind of Java program that can be created, which is console application, and graphics applet (Horstmann, 1998). This project is developed using the Console Application style, and it is lack of Graphic User Interface (GUI) components. The project purpose is to explore the capability of Java Reflection in reflecting the content of the object file, and producing meaningful information in the output to whom that interested with the similarity of Java object file's information without referring to the original source code. More emphasis is pressed on the engine of reflecting a number of object files, and producing a set of similarity result among the nominated target files in percentage form.

However, with console programs, this application is not limited to user interaction (Jaworski, 1998) compared to Java Applet, which is reading the keystrokes and mouse clicks, and display graphical shapes in a window. Console programs tend to minimize the program's complexities with excluding the GUI components' coding. The Sojberg's work in discovering the software constraints for large application system (1997) stated that to make the application system live longer, the application is needed to prevent from becoming unnecessarily large, complex and confusing. With the uses of console application environment, the application can continue to be developed and enhanced to serve the purposes of future plagiarism analysis.

Chapter Five

The methodology

An application is developed to reflect the two or more object files. The Java programming language, an Object Oriented (OO) programming language, is used to develop the application.

5.1 The conceptual development

The validity of the project is based on the features offered by JDK version 1.2 and above, especially the system package of java.lang.reflect. The application development is carried in Microsoft Windows 98 environment.

The JDK is used to compile the source code that is created from the text editor. To execute the application, the Java interpreter in JDK is used to examine the program coding result. The debugging of application will be done in text editor from time to time in the process of application development.

Above is the environment to develop, and tools used in the process. The development process follows the Object-Oriented Methodology. It consists of building a model of an application domain, and then adding implementation details to it during the design of a system. According to Rambaugh (1991), this approach can be called as Object Modeling Technique. The methodology has the following stages:

a) Analysis

The first stage of the methodology concerns with devising a precise, concise, understandable, and correct model of the real world. In others words, the task is to model the real world system so that it can be understood. The success key of this stage is "what" the desired system must do, not "how" it will be done. There are five steps to be followed in this stage.

i) The first step is to state the requirements. Through the finding from literature review, this step stated downs "what" to be done. In the previous research, Whale's research (1990), the work stated the plagiarism detection system is critically dependent on the choice of program representation to identify similarity of program in large populations (Whale, 1990). Furthermore, the existing software metrics conventional, which used to represent program were listed by Whale (1990), were not efficient addressing program information when they adapted to complex program. From the previous study, it showed the weakness in detecting plagiarism system always coping sophisticated plagiarism techniques, and another finding stated that it needed in involving complex programming (Whale (2), 1990). Therefore, this project is to discover the difficulties in gathering information for source code analysis, especially addressing the program information for complex program, specifically for Java application.

ii) The second step is to construct an Object Model, which shows the static data structure of the real world system, and organizes it into workable pieces. The object classes are identified from the application domain, which is listed as below.

- The object concepts (Modifier, Interface, Field, Method, Constructor);
- Their weightage influencing the final output;
- The tasks of analyzing the collected vocabs (reflecting the concepts, multiply value, adding value, converting data type, creating percentage, validation, gathering vocabs, filtering vocabs, matching, counting frequency);
- The tasks presenting output to the user;
- The tasks request input from the user; and
- The tasks manage the vocabs and vectors.

The objects are grouped to form the classes[16] as below.

- The class **ReflectApp**, the main class of the application, which has the main role to perform the reflection tasks, such as the tasks request input from the user, the tasks manage the vocabs and vectors, reflecting the concepts.

[16] Ibid.

- The class **AnalyzeVocab**, the class consists of the objects, which have the common behavior of performing the analyzing tasks. The object members including are converting data type, creating percentage, validation, gathering vocabs, matching, and counting frequency.
- The class **ObjPropWeightage**, the group of objects have the tighten relationship in adding the weightage before the final output, such as object concept's weightage influencing the final output, and filtering vocabs.
- The class **Util**, the groups of objects have the most reusability role in supporting above classes. They are multiply value, adding value, and the tasks presenting output to the user.

The above grouping tasks are done according to the operations, roles, and the level of relevant to others objects in the group. The class Util is created to reduce the redundant objects appear in different classes, and to increase the performance of application under the reusability concept in Object-Oriented Programming.

iii) The data dictionary is created since this stage, which has the role to explain and interprets the isolated words during the application development. This data dictionary is compiled at the end of this report in the appendix E: Glossary.

iv) Any dependency between two or more classes is identified. In the methodology, this task is called identifying associations[17]. The multiplicity and missing associations' issues is solved in this stage. The role name is added appropriate to describe the role that a class in the association plays from the point of view of the other class.

v) The next task is to identify the attributes[18] of the objects. The attribute is gathered through the association classes. They can used to show the relationship between two objects, such as the similarity item, and the frequency occurrence within the objects (Appendix C Figure 1: Object Model Diagram). From the diagram, the binary relationships occur are one-to-many, and many-to-many between the objects.

vi) Then the classes are reorganized by using concept inheritance to share the common structure. There are two directions to inherit the classes, which are listed as below.

- Generalizing common aspects of existing classes into a superclass (bottom up), and

[17] Ibid.

[18] Ibid.

- Refine existing classes into specialized subclasses (top down).

In this application, the first approach is used to inherit the classes, which is searching classes with similar attributes, associations, and operations. After the inheritance, the access path is tested, and write off the repeatedly works before proceed to next step in creating the Dynamic Modeling.

vii) The next model, in third step, that shows the time dependent behavior of the system, and the object in it is the Dynamic Model. This model is looking for events that externally visible stimuli and responses. It is consists of the summarize permissible event sequences for each object with a state diagram. This dynamic model is important for interactive systems (Rumbaugh, 1991). The first step of creating this Dynamic Model is to list out the scenarios of typical interaction sequences, and the scenarios are as below.

- After user execute the application, the user is required identifying the target file to be compared with.
- The application accepts one number file from user.
- The application performs the reflecting and analyzing tasks, and print out the summary result of similarity percentage subject to target file.

- Then the user is requested to choose the option to view the detail of the similarity results.
- The application accepts the user input, and searching the target comparison set's result.
- The application print out the detail of similarity of each comparison set.

Above scenarios are the major interaction, and because of the nature of the application is console based, there is no interface format require in the process of development. There are only two printing message interact with user for proceeding the use of application.

viii) The next task is to identifying all the external events, such as the input from the user or the response to the system's request (Appendix C Figure 2: Event trace for Reflection Application). In this development, there are only two responses from the user, and they are listed as below.

- The target file number to be compare with, and
- The response on printing the detail information of each comparison set of previous analyzing results.

A State Diagram is built for the dynamic behavior of application, and it shows the events the object receives and sends (Appendix

C Figure 4: State diagram for class ReflectApp). From the diagram, the behavior of the system responding to the user feed back can be viewed. If the error or invalid input occurs from behalf of user, the error message will appear consequently by the system itself. The diagram viewed up overall process will be occurred in the system from beginning towards the stage of system termination.

ix) Above State Diagram will merge with previous scenarios to form up the Event Flow Diagram of the application. At the same time, checking the consistency and completeness of events is done while drawing out the Event Flow Diagram (Appendix C Figure 3: Event flow diagram). It is a summary flow of diagram when the system deals with system user. From the diagram, the user's inputs or outputs can be referred easily to achieve the event consistency and completeness in the system.

x) Next, the fourth step is creating the Functional Model, which is used to show how the values are computed or the background computational operation, without regard for sequencing, decisions, or object structure. Inside this model, it shows which value depends on which other values and the functions that relates them. The mentioned functions expressed in various ways, but in this application the function is use the mathematical equations as the base. To form up this model, the input and output values have to identify first (Appendix C Figure 5: Input and output values for ReflectApp system). In other words, these values are the parameters of events between the system and the outside world.

xi) From the above findings, the Data Flow diagram is constructed to show how each value is computed or used from input values. This diagram is usually constructed in layers (Appendix C Figure 6-9: Data flow diagram).

In level 0 Data Flow diagram, it is a process-like model that actually illustrates the system's interfaces to outside world. It has a bit similar with the Context diagram in methodology SSADM. It shows the input and output, or how the system communicates with outside world.

In level 1 Data Flow diagram, the process in level 0 is explained in detail. For example, in process 1.0, the process of detecting object files, the detail operation of detecting tasks is shown in the diagram. The output from this process 1.0 will be found at earlier level of Data flow diagram, which is the Object files' list. The same things will happen in process 2.0, which is regarding the analyzing object files' tasks. These tasks will carry by various classes in the application by creating the vocabs first until the step producing the similarity result subject to user's target file. Whereas the last process 3.0 is regarding the task of searching from exist vocabs of the similarity result's detail information.

xii) Based on above Data Flow diagram, the next step is to identify the constraints between the objects. Constraints are functional dependencies between objects that are not related by an input-output dependency.

- If the user locates less than two files, the system will terminate.
- If the user input the target file out of range, the system will terminate.
- If the user responds on viewing detail information as "n", the system will terminate.

xiii) For the conclusion from above works, the analysis places less emphasis on defining operations compare to traditional programming-based object-oriented methodologies. The reason point out by Rumbaugh (1991) was the operations are open-ended, and it was difficult to know when to stop modifying them in this non-traditional programming based object-oriented methodologies. However, with this methodology still can correspond with object-oriented programming, such as queries the attribute or associations in Object Model, queries the events in

Dynamic Model, and functions the Functional Model. The next step is performing the operation from above accumulated analysis stage's work by verifying, iterating, and refining the above three model. As mentioned before, for Object Model, operation can be done including reading and writing attribute values and association links. While for events' operations, each event sent to an object corresponds to an operation on the object. In Functional Model, each function in the Data Flow Diagram corresponds to an operation on an object or objects.

xiv) Finally, the last step for this stage is to iterating the analysis' works. The refining overall analysis model is required to overcome the inconsistencies and imbalances within and across models, and ready as the basic for design in next stage.

b) System design

This stage is consists of designing overall architecture of the application. Due to the size of this application, there is no sub system organized. The issues concerns in this stage are what performance characteristics to optimize, strategy of attacking the problem, and make tentative resource allocations. Generally, there are eight steps to follow in this stage.

i) The first step is omitted due to the size of application, which is small. So there is no sub system can be broken up.

ii) Next step is to identify the concurrency in objects' implementation. This application is not a critical system, and in practice, still many objects, which are listed before, can be implemented on a single processor. Therefore, there is no need to implement inherent concurrent in separate hardware unit, and this step is omitted to examine in the design stage.

iii) The estimation of the hardware resources requirement's step also is omitted as this project is developing an independent, and has specify small scope coverage application. In addition, this application can run on a single computer with platform independent feature. No other physical units must be arranging among themselves.

iv) In this application, there is no exist any internal or external data store, so the step in methodology used to manage the data store is omitted too.

v) Only one global resource, which is the logical name for position of next input in vocabs, needed to be taken care to avoid conflicting access in a shared environment of future implementation. This resource is partitioned logically by declared as "private static", so that it did not shared with other classes, and it is an independent control approach.

vi) The next step is choosing software to control the implementation. There are two-control flows in a software system, which is external control, and internal control. In this application, the external control exists in the flow of externally visible events among the objects in the system. For example, the procedure-driven system control, the control resides within the program code. The system control work as it requests input from user and then wait for it; when the input arrives, control resumes within the procedure that made the call.

For the internal control system, as events passed between objects, and the same implementation mechanisms above is used. However, the main different between these two control system is the External control system's interactions inherently involving waiting for events because difference objects are independent and cannot force other objects to respond; whereas the Internal control system's operations are generated by objects as part of the implementation algorithm, so their patterns are predictable.

vii) In this step, the tasks involve the boundary conditions. For example the initialization, the application brought from a quiescent initial state to a sustainable steady state condition.

Things to be initialized in this system include parameters, global variable of **nextVocab**, and tasks.

For failure issues, the application is planned termination in the system. Failure can arise from user errors in this application, such as the invalid input in choosing the target file's number. In the case of user picked the number, which is out of the detected files' number, the system will terminate with an error message on screen. Application was designed with a plan for orderly failure, and most of the approaches used were the "If-else" statements in the development. Other example, if the user responds to the system of viewing the detail information is not character "y" or "n"; the system will terminate with an invalid's error message on screen.

viii) The next step is to determine the trade off priorities in the system. In the process development, the detection of quantity object files by the system is located as high priorities, before proceed to other steps. Next is the selected target file by the user to be comparing with, so that the system can narrow down the scope of printing and discovery of analysis. Then the next priority is pass to the option of printing the detail information of the similarity-matched items. The priority statement is just the design philosophy to guide the design process.

c) Object design

It is a design built based on the analysis model but contains implementation details. It also added details to the design model in accordance with strategy established during the system design above. The focus of this stage is on the data structures and algorithms needed to implement in each class. There are eight steps to be followed in this stage as below.

i) The first step is to combine the three models. For example, the application's Object Model did not showed operations. It only shows the objects and their relationships. In this step, the tasks are convert the actions and activities of the Dynamic Model, such as the scenarios for the application, and the processes of the Functional Model, such as the data flow of the application, into operations attached to classes in the Object Model.

ii) Each operation specified in the Functional Model needed to transform as an algorithm or "how" it is done. This algorithm is subdivided into calls on simpler operations until the lowest level operations are simple enough to implement directly without further refinement, for example the procedural definitions, and the mathematical definitions for computing the function. Then, the task is choosing the data structure the algorithm work on. It is regarding the tasks of organizing the operation in a form convenient for the algorithms that use it. Examples of data structures have used in this application are as arrays, two dimensional arrays, vectors, associations, etc. Array is used to store the known size of data, such as the object file name. Whereas, the vector is used to store the unknown ones, such as the items reflected by the Java package. The two dimensional array is used to store, for example, the unrepeated vocabs reflected by the Java package.

In the expansion of algorithms, new classes of objects are needed to hold intermediate results. It involves the defining internal classes, and these operations must be done in Object Design, as they are not visible externally. Example in this project is the validation of multiplier's parameters, the low-level operation is performed to validate the condition of parameter before proceeding the multiply operation in class **Util**.

iii) The optimization work is done during this stage. The inefficient but semantically correct analysis model can be optimized to make the implementation more efficient in Object design's stage. To optimize the model, in this application development, is rearranging the computation for greater efficiency, and save the derived attributes to avoid re-computation of complicated expressions.

iv) The next step is to refine the strategy for implementing the state-event models present in the Dynamic Model (Appendix C Figure 10: ReflectApp control). Firstly, using the location within the program to hold state, which is involving the previous discussed procedure-driven system, then direct the implementation of a state machine mechanism, which is used the event-driven system.

v) With above model, the next task is to adjust the inheritance for classes and operations. Example inheritance class in this application is the class **ExtensionFilter**, which is used to filter the file extension, specifically in searching or detecting the user's object files in destination folder. These tasks are performed by rearranging, adjusting the class and operations to increase the inheritance.

vi) Then the next step involves analysis of the associations' design. During the Object Design stage, a strategy is formulated for implementing the associations in the Object Model. Here, this step is to analyze and optimize the design for flexibility to change the decision with minimal effort. Example in this application is the separated frequent used methods in the class **Util**, such as the print2DArray, multiplyString, addString. The key in developing application in this stage is hiding the implementation using the access operations to traverse and update the association.

vii) After redesign the associations, the next step is to determine the object representation. It is regarding the issue of choosing objects when to use primitive types in representing objects, and when to combine groups of related objects.

viii) Then the final step is to complete the physical packing. In this application, this task only involves the compilation and distribution of the source code and object files to be used by the user.

d) Implementation

In this stage, the object classes and relationships developed during Object Design are translated into Java programming language for implementation purposes. Theoretically, the programming is relatively minor and mechanical part of the development cycle, but practically, it involves most of the developing period because the hard decisions made during design phases are keep revising for improvement. The fact is the target language, Java, have some extent to influence the design decisions, but the design decisions did not have fully dependent on the fine details of the Java programming language.

Java is an Object-Oriented programming language, and Object Modeling techniques is producing the Object-Oriented design, so it is relatively easy to implement the design since the language constructs are similar to design constructs (Rumbaugh, 1991). There are several steps to be followed to implement the Object-Oriented design in an Object-Oriented language.

i) The first step to implement the design is to declare the object classes. Each attribute and operation in an object diagram must be declared as part of its corresponding class. Java is supporting the declaration of "public" and "private", which meant that the public features could be accessed by any method, while private features are only accessible by methods of the same class.

ii) Next step is to create the objects for the application. Compared to C++, Java is easy to use because for the new created object, the Java does not need to allocate storage for its attribute values and assign a unique object ID in storage block. Objects used in the application are **AnalyzeVocab** and **objPropWeightage**.

iii) During the invention of the object class, the operations have been called consequently. In Java, each operation has one or more implicit argument, the target object, indicated with a special syntax. Java is permitted the developer to passing arguments as ready-only values and as references to values that can be updated by a procedure. For example, the vocabs, which pass in and out for each reflecting categories of object concepts is referred during the in operation, and has been updated as it out from each reflecting engine.

iv) The application also uses the inheritance as provided by the Java language. In the developed application, the inheritance, especially for the class ExtensionFiltering, is implemented in static condition and it is bound at runtime.

5.2 The logical development / program flow

In this section, the program flow explains the detail of how the application works in logical. From this section, the reader will gain the knowledge of how the application performs the tasks, where the source information from and forwarded for next task. They will descript according to the source code line by line.

Generally, the application is developed in three objects or classes, which is listed as below.

a) The **ReflectApp** class, consist of the main engine to perform the direction of reflection analysis.
b) The **analyzedVocab** class, consist of the engine to analyze the collected reflection result.
c) The **objPropWeightage** class consists of the engine evaluating the selected target file with user predetermines information.

At beginning of the source code, the application import four package, which has a set of types and sub-packages as members, for different purposes as remark at the end of the lines.

```
import java.lang.reflect.*; // For reflecting purposes
import java.util.*;         // For the use of Vector
                            //                 & retrieving file
import java.io.*;           // For requesting input from user
import java.math.*;         // For round up the percentage
```

In the **ReflectApp** class, integer **nextVocab**, is declared as private static, and it is used internally only. It is variable for position of array **vocab** in method **addToVocab**.

In the **main** of the ReflectApp class, the application performs the task to search all the object files, which locate in the destination folder or same level as the application's object files located. This task is performed by the class **ExtensionFilter** in searching the entire file with the extension of "class". The engine locates the found object files in array **fileNames** (Figure program flow 1), except five object files used by this application, which is listed as below.

a) ReflectApp.class

b) ReflectApp$ExtensionFilter.class

c) AnalyzeVocab.class

d) ObjPropWeightage.class

e) Util.class

Immediately after the searching, a two-dimension array, **vocabs**, is declared to collect all object files' information.

An **If-else** statement is used to examine the length of array **fileNames**. If the length is bigger than zero, which means there is one or more object files locate in the destination folder, the application will print the collected object file's or files' name, and will continue analysis tasks later. Else, an error message will appear to warn user that there is no object file exit for comparison tasks, and the system will terminate.

If the array length is bigger than zero, and the length is equal to one, an error message will appear to warn the user that there is not enough object files to be compare and carry on the analysis tasks. The application will terminate immediately after printing the error message.

If the array length is bigger than zero, and the length is equal to more than one, the application will print message to request user key in the number to be the subject of later comparison purposes. Each early detected object files will be represented by listed number on screen. User request to key in the number that represents specify object file, and press Enter key for continuing the process. If user input the correct number within the range which is shown on screen, the application would convert the number to actual file name to be used in future analysis process. For the example, this converted target file name will be used in method **compileAllItem**, and method **filterResult**.

Then the application will reflect each object files. To obtain the object concept the application perform the task with the method **reflectClassOrInterface**. Then, the application will load weightage information from the file **user.ini**, which locate in the same folder as this application's object files. Overall, that is ten item's information need to be load from the file **user.ini**. The default value in the file **user.ini** is listed as below.

```
a) ModifiersWeightage=2/10
b) InterfacesWeightage=2/10
c) FieldsWeightage=2/10
d) MethodsWeightage=2/10
e) ConstructorsWeightage=2/10

f) itemMatchingWeight=5/10
g) freqExactMatchWeight=2/10
h) freqRangeMatchWeight=3/10

i) StatusNotExitIsSimilar=yes

j) freqRange=20
```

The items a) to e), are the weightage for five category directly contribute to final similarity percentage, which have been consider for each object file. Their total should sum up for one, and each category said above can be omitted by assigning 0/10. For example, if user just want to discover the similarity in category **method**, so the item d) should assign the value of 1/1, and the others (a), b), c), e)) should assign the value of 0/1.

Whereas, for the items f) to h) is the weightage specifically for category **Field**, **Method**, and **Constructor** for each object file. The reason this weightage is created because three categories mentioned before encountering the frequency of each similarly item or items. The user can relocate the weight for these sub categories to obtain accurate expected results. These three items should contribute to total of one, and they can be omitted by assigning value 0/1. Item j) is specifically created to determine the frequency range to be match for inside the category h). The item j) assign value is based on the percentage value, such as original frequency plus or minus 20% as mention above.

The item i) is an option to let the user determine specifically non exit item in analyzed categories (Modifier, Interface, Field, Method, Constructor) to be consider as a similarity item or not. The default value assigned the non-exit item is considered as similar, and contribute towards to the final similarity percentage.

Then the application will analyze five categories, which is Modifier, Interface, Field, Method, and Constructor, for each object files separately. These tasks are performed separately with various methods. Example for Modifier and Interface, the application will invocate the class **AnalyzeVocab**, specifically the method getCompareModifiers (for Modifier) or getCompareInterfaces (for Interface), to comparing the similar item appear between the object files. Then the method **collectResult** from the class **AnalyzedVocab** is used to collect the object files' name and the similarity item's value between two object files.

For the Field, Method, and Constructor, additional method is invocated, which is method **freqMatching**, to compare the frequency between two object files. To compile all the result for items matching, exact frequency matching, and range frequency matching, method **getCompileResult** is invocated. During this invocation, the items' information from **user.ini** (f), g), h)) is retrieved to determine the value will be produced.

Before present the output to user, all the result from Modifier, Interface, Field, Method, and Constructor need to compile together. To perform this tasks, the class **ObjPropWeightage** is invocated. At the same time, the weightage for mentioned categories from **user.ini** is retrieved while compiling the results. The status of non-exit item will submit together to produce the final similarity percentage by method **compileAllItem**.

The final output will print by invocate the method **print2DArray** in class **Util**. The output consists of the comparing set's name, which subject to only two files for each set, and the similarity percentage among these two files or set. The target file or the file name choose by user at the beginning of the application is executed will be the subject comparing to all the rest of object files. Thus, the number comparing set will the total object files detected by application minus one.

Immediately the printing of above output, a request input's message from user will appear. User can have the detail of similarities for above output by typing "y" to request the application print out all the matching items, exact matching frequency, and range matching frequency. If the user feed back is "n", the system will terminate, and wish the user. To print out the detail of similarity matching, the application uses the method **filterResult** to filtering the complete source, which produced by method **getCompareMethods**, method **getCompareInterfaces**, method **getCompareFields**, method **getCompareMethods**, and method **getCompareConstructors**. The target file name is submitted together to filtering out the irrelevant subject files, and details.

In class **ReflectApp**, there are 12 methods were created to perform the analysis tasks, and listed as below.

a) **filterResult**, used for filtering purposes before printing the detail information of the similarity matching of the application.
b) **addTo2DVocabs**, used to add new found two-dimensional array to existing accumulated two-dimensional array.
c) **reflectClassOrInterface**, the main engine to reflect the object concepts, which contain inside the object file.
d) **reflectNameAndModifiers**,
e) **reflectInterfaces**,
f) **reflectFields**,
g) **reflectMethods**, and
h) **reflectConstructors**, all five methods above (d), e), f), g), h)) are sub engine to reflect the object concepts of object file. The first task of these methods is to record down the comparison object file's name. For d) and e), the methods reflect the used Modifiers, and Interfaces. Whereas, for f), g), and h), the methods reflect the used plus declared Fields, Methods, and Constructors.

i) **addToVocab**, used to create a new and unrepeated accumulated vector. Inside this method, there is a checking engine to perform the examination tasks to make sure there is no redundancy item occurred.

j) **filteringString**, this method is used to filter out the object file's name from the matching items. If found the object file's name exist in matched item, this method will replace it with *FileName*. The purpose implement this engine is achieve the full analyzing tasks on the matching section, especially in items matching, without considering the differential of object file's name.

k) **checkEntryExist**, this method is used to support the method **addToVocab**, especially to create the unrepeated accumulated vector.

l) **countFrequency**, is used to count the items frequency occur in the object file. This method will compare the full-unrepeated accumulated vocabs to the found items, including the used and declared items, in the object files.

In class **AnalyzedVocab** has created one constructor, **tempVocabs**, at very beginning of the class. This constructor is reused internally in methods of class **AnalyzedVocab**. Overall, there are 16 methods is created in this class, and is listed as below.

a) **getModifiers**,

b) **getInterfaces**,

c) **getFields**,

d) **getMethods**, and

e) **getConstructors**, above five methods is used to direct the collection of specify information from the complete vocabs, which located in two dimensional array **tempVocabs**.

f) **getCreate**, is an engine to collect information directly from **tempVocabs**. Other words, this method is the base engine to perform the methods a), b), c), d), and e).

g) **convertObjectToStringArray**, is used to convert the object array to string array. This method is useful while converting the vector to array, because after converting from vector to array, the output is in object array form. With this method, the mentioned converted output can be easily turned into string array.

h) **getNumberOfSet**, is used to calculate the number of comparison set, which has direct relation to number of detected object files by the application.

i) **getCompareModifiers**,

j) **getCompareInterfaces**,

k) **getCompareFields**,

l) **getCompareMethods**, and

m) **getCompareConstructors**, are used to compare the specific categories according to methods. The source of these methods is came from a), b), c), d), and e). The comparison is done between the source and destination object file, which locate from the above sources. These methods will generate a two dimensional array as output.

n) **freqMatching**, is perform the frequency-matching task for categories Field, Method, and Constructor only.

o) **collectResult** is used to collect the matching result from the unrepeated accumulated vocabs, specifically for object files' name, and the similarity item's value between two object files.

p) **getCompileResult**, is used to compile all the result for items matching, exact frequency matching, and range frequency matching. In this method, the weightage information, which has been loaded from **user.ini**, is used to produce the similarity value between objects. The source for this method is came from the method **collectResult**.

In class **ObjPropWeightage**, there are five constructors is created, and is listed as below.

a) **ModifiersWeightage**,

b) **InterfacesWeightage**,

c) **FieldsWeightage**,

d) **MethodsWeightage**, and

e) **ConstructorsWeightage**, are the constructors in class **ObjPropWeightage**. Their value is loaded from file **user.ini**, and reused internally in method **CompileAllItem**.

There are four methods in this class, and are listed as below.

a) **compileAllItem**, is used to compile all collected information from discovery categories, which includes Modifier, Interface, Field, Method, and Constructor. The weightage information gather before will be multiplied, and add-ins to final value. Each category's result, which has been multiplied, will be added to compute the final two-dimensional array together with the names of two comparison object files.

b) **convertToPercentage**, is used to convert the computed value in method **compileAllItem** to a percentage form before presenting to user.

c) **filterResult**, is used to filter the output of method **compileAllItem** before presenting to user on screen.

d) **checkValidDouble**, is used to check the validation of double condition in method **compileAllItem**, especially before performing the multiply task. This method is considered as non-exist item's status. If the non-exist item is considered as similarity, then the value of 0/0 will change to 1/1, else the value will be 0/1.

In class **Util**, there are four methods created to serve the above classes. Three of the four methods, except the method **getDivider**, can be called outside of the class **Util** to perform the tasks. The methods are listed as below.

a) **multiplyString**, is used to multiply integer in string form.

b) **addString**, is used to add similarity value in string form.

c) **getDivider**, is used to find the divider of value, so that it can simplify the value before performing the multiplying or adding tasks in the above methods.

d) **print2DArray**, is used to print out the two dimensional array. Most of the outputs from the application's methods are in two-dimensional array. In order to present them to the user, this method is needed.

5.3 The expected input and output

The user needs to locate at least two or more Java object file in the folder, which is same level as the Reflection Application object files. This application will searching and detecting the quantity of object file can be analyzed.

Next, the user is required to choose the file number that represents the certain file, which has been listed on screen. It is a target file to be considered in comparing with other object files. After key-ins the selected number, the application will perform the analyzing tasks, and lists the similarity percentage, which is subject to the user's target file. The expected output of the application will be the result of comparative two-object file, and percentage of similarities between two selected object files.

The application will offer an option, after above listing, in listing all the details of similarities of above result. The user is require to key-ins "y" for yes or "n" for not to list the detail information of the analyzing results.

Chapter Six

Conclusion

The expectation of final product in project will be a list of comparison result of two-object file with percentage. The detail information of similarities can be listed through the order of user to enhance the percentage's information. As overall, this expected output is helping those who are innocent in the content application understanding the original class developer's idea by reflecting the detail object concept. In addition, it is a tool to help user to discover the similarity information between two-object files, in the fast mode, through the percentage between two-object files.

6.1 The contribution of project

The purpose of the project is to discover the capability of Java Reflection, and it is uses in producing similar object concept between two object files. After compilation, there is no way for developer to read the hidden source code from the compiled object file. The only way to get back the full and accurate object concept is to obtain the object file's source code, and analyzed it line by line. It is hard and cost more effort to complete this task. In this project, the application is developed to reflect the hidden information in object file. Through the sample output (Appendix A), it successfully reflects the similar hidden information that locates in two-compiled object file. It helps the developer to discover the similar object concept without referring to the complicated source code, and obtain the similar object concept's information in fast mode.

The expected output of the application come with similar percentage between two-object files is useful in detecting the plagiarism cases in academic institution. The students fault work can be discovered through the similar object concept that contain inside the object files. The analyzing tasks are look into the similarity section of Modifier, Interface, Field, Method, and Constructor. Each section will consist of exact item matching, exact frequency item occurrence, and predetermine range frequency item occurrence. These analyzing tasks' result is contributing towards the final similarity percentage. This application result can help the educator judge the plagiarism cases in student work since it producing the informative object concept from the student object files.

Besides, the capability of programming can be used to detect the plagiarism cases among the student's work, which wrote by Verco & Wise (1996), from this project, this programming application also contributes as a tool to help the lecturer to evaluate student's assignment. To perform this task an ideal answer needs to be ready before proceeding checking with the student's assignment. It also performs the cross checking among the students' work, whereby, each student's work will be compared among the submitted object files in the folder.

Each student will compare with the lecturer's ideal model answer. However, the expected only showing the target object file, which had been selected by the user. From the expected result, the lecturer can discover that the content of student work consist of how many percentage the lecturer's expected output as in the ideal answer. For logical analysis, the application will match the quality of object and method in used, which needs to be present in student's work. The higher matching of student work represents the knowledge own by particular student. Evaluation marks on student can be given through this degree of matching guideline. It is useful for those lecturers who need to mark a big group of programming course's student. It will help in increasing the lecturer productivity of the daily workload, especially in evaluating the student assignments, with constant requirements of evaluating programming content.

6.2 The contribution for future study

The project is reflecting the object concept information, and this task can be done through the analysis of source code. For future study, the work should compare the performance of existing analyzing approach with the source code analyzer in achieving the preferable result of detecting the similarities object concept information.

For better presentation of the application, this application should be presented in improved interactive graphical user interface. This proposed work is to increase the user friendliness in using the exiting application.

6.3 The limitation

The limitation of the application is listed as below.

a) This project uses Object-Oriented methodology, specifically the Object Modeling Technique, however the development process did not fully used the guideline that ruled by Rambaugh (1991). The main reason is the unsuitability issues occur, such as size of the project is small, and impractical fully following the stated steps, especially the steps in system design.

b) This application only tested in windows environment, specifically in Microsoft Windows 95/98. This was due to lack of facility in university environment. It should work on other platforms as the Sun Microsystems promised the Java technology as platform independent. Further testing should be carried out in different platform for this application to achieve independent-platform criteria to ensure the portability of application.

c) The design of the classes is not fully achieved the object-oriented purposes. This may cause the application's performance decrease, and does not achieve the reuse purposes, which offered by Java, a true Object-Oriented Programming Language.

d) The modifications of the weightages' information in the file of **user.ini** have to be done in external text editor. The application did not design internally for alternate the **user.ini** content. The enhancement needs to be done in exist application for user alternate the mentioned file before performing the analysis tasks.

Bibliography

Arnold, Ken & Gosling, James. (1997). The Java™ Programming language (2nd Edition). England: Addison-Wesley Longman Inc.

Awad, Elias M. (1996). Building Expert System: Principles, Procedures, and Applications. New York: West Publishing Company.

Horstmann, Cay. (1998). Computing Concepts with Java Essentials. New York: John Willy & Sons, Inc.

Jaworski, Jamie. (1996). JAVA Developer's Guide. Indianapolis: Sams.net Publishing.

Jaworski, Jamie. (1998). Java 1.2 Unleashed. Indianapolis: Macmillan Computer Publishing

Kirby, G. N. C. (1992). Persistent Programming with Strongly Typed Linguistic Reflection. http://www-ppg.dcs.st-and.ac.uk/Publications/1992.html#persistent.programming (Accessed on 18/08/1998).

Kirby, Graham & Morrison, Ron. (1998). Linguistic Reflection in Java. http://www-ppg.dcs.st-and.ac.uk/Publications/1998.html#java.reflection (Accessed on 18/08/1998).

Lemay, Laura & Perkins, Charles L. & Morrison, Michael. (1996). Teach Yourself Java in 21 Days (Professional Reference Edition). Indianapolis: Sams.net Publishing.

Rambaugh, James. & Blaha, Michael. & Premerlani, William. & Eddy, Frederick. & Lorensen, William. (1991). Object-Oriented Modeling and Design. New York: Prentice-Hall International, Inc.

Sjoberg, Dag I.K. & Welland, Ray & Atkinson, Malcolm. (1997). Software Constraints for Large Application Systems. The Computer Journal. Vol.40 No.10 1997. Pg 598-616.

Stemple, D. & Morrison, R. & Kirby, G.N.C. & Connor, R.C.H. (1993). Integrating Reflection, Strong Typing and Static Checking. http://www-ppg.dcs.st-and.ac.uk/Publications/1993.html#integrating.reflection (Accessed on 18/08/1998).

Verco, Kristina L. & Wise, Michael J. (1996). Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting System. Presented paper of First Australian Conference on Computer Science Education, Sydney, Australia, July 3-5 1996, John Rosenberg (Ed), ACM.

Whale, Geoff. (1990). Software Metrics and Plagiarism Detection. Journal of System Software. Vol.13 1990. Pg 131-138.

Whale, G. (2). (1990). Identification of Program Similarity in Large Populations. The computer Journal. Vol.33 No.2 1990. Pg 141-146.

Appendix A:

Sample output with description

Sample output with description

This sample output is generated from windows platform, specifically from Microsoft Windows 98, with the software called Tek-Tools KAWA IDE Version 3.5. The output is bold and used the Courier New font to differential from the report texts.

```
D:\jdk1.2.2\bin\java.exe    ReflectApp
Working Directory - E:\collection\Java\3p\
Class Path - D:\jdk1.2.2\jswdk-
~1.1\lib\jspengine.jar;.;D:\jdk1.2.2\jswdk-
~1.1\lib\servlet.jar;D:\jdk1.2.2\jswdk-
~1.1\lib\xml.jar;D:\jdk1.2.2\lib\msql-jdbc-2-
0b5.jar;D:\jdk1.2.2\lib\tools.jar;D:\Kawa3.5\kawaclasses.zip;d:\jdk1
.2.2\jre\lib\rt.jar
```

The Reflection Application is executed by the KAWA IDE. The application also can be executed in MS-Dos Prompt environment by typing the command of **java ReflectApp**.

```
The collected Java Object's files (*.class) list:

      File 1 : ReflectApp7

      File 2 : ComboArray

      File 3 : TestScore

      File 4 : ReflectApp7a

      File 5 : ComboAxxx


5 files found!

Please select file name to be consider?

(Choose 1-5 and Press Enter) > 1
```

Above is the application finding. The application detected five object files, which is contain the file extension of class (*.class), in the folder of **E:\collection\java\3p\** to be analyzed. Each of the files is represent by one number, and it is easy to let the user choose the subject of comparison. The application is waiting for the user to key-ins the target file's number. In above sample output, the user choose "1", and press the Enter key!

```
********** For selectedPlagiarismResult **********

Comparing subject to  : ReflectApp7, ComboArray,
Similarity percentage : 54.0%,
Comparing subject to  : ReflectApp7, TestScore,
Similarity percentage : 54.0%,

Comparing subject to  : ReflectApp7, ReflectApp7a,
Similarity percentage : 100.0%,

Comparing subject to  : ReflectApp7, ComboAxxx,
Similarity percentage : 54.0%,

********** End selectedPlagiarismResult **********
Do you want to see the detail of above comparison?
(Choose "y" or "n") >  y
```

After the user pressed the Enter key, the application performed the analysis tasks, and the output is listed as above. Each of the comparison is subject to the target file, as the user chooses at previous step. In this sample output, the user choose number "1", which meant the target file is pointing to the file's name called ReflectApp7. Then, each set of result in comparison will consider the target file, which is ReflectApp7, as the base to compare with.

The output is the selected plagiarism's result, which meant each set, between two files, there is a percentage to show the level of similarity between them. The higher the percentage, the higher the plagiarism occurrence between this two files. In above sample output, the set, which contains of ReflectApp7 and ReflectApp7a, is 100% similar. It is the highest score that can be achieved by this application, which means the plagiarism occurs between these two files.

A request input will appear immediately after the selected plagiarism result. The application offer two option to the user to view the detail of similarity occur between the files. The option "y" is to view the detail, and the option "n" is to deny the further printing detail. In above sample output, the user typed the character "y" to view the detail of similarity comparison.

Below is the listed detail similarity information for each set of comparison. The detail information is divided into five categories for each set, which reflected from both object files, including the Modifier, Interfaces, Field, Method, and Constructor.

```
****************************************************
Below is the detail of comparison subject to ReflectApp7
****************************************************

-----------------------------------
********** For Modifiers **********
-----------------------------------

Comparing subject to  : ReflectApp7, 0/0, ComboArray, 0/0,
Similarity item/s     : notExit,

Comparing subject to  : ReflectApp7, 0/0, TestScore, 0/0,
Similarity item/s     : notExit,

Comparing subject to  : ReflectApp7, 0/0, ReflectApp7a, 0/0,
Similarity item/s       : notExit,

Comparing subject to    : ReflectApp7, 0/0, ComboAxxx, 0/0,
Similarity item/s       : notExit,
```

For example, the Modifier as above, the section **Similarity item/s** filled with result "notExit", which meant the Modifier does not exist in particular set of object files. User can refer to the value locate behind of the object file's name, if the value is 0/0, it is an evidence to show that there is no item exist in particular object file.

```
-----------------------------------
********** For Interfaces *********
-----------------------------------
Comparing subject to    : ReflectApp7, 0/0, ComboArray, 0/0,
Similarity item/s       : notExit,

Comparing subject to    : ReflectApp7, 0/0, TestScore, 0/0,
Similarity item/s       : notExit,

Comparing subject to    : ReflectApp7, 0/0, ReflectApp7a, 0/0,
Similarity item/s       : notExit,

Comparing subject to    : ReflectApp7, 0/0, ComboAxxx, 0/0,
Similarity item/s       : notExit,

-----------------------------------
********** For Fields *************
-----------------------------------

Comparing subject to    : ReflectApp7, 0/1, ComboArray, 0/2,
Similarity item/s       : notFoundSimilarity,
Frequency(1st subject): notFoundSimilarity,
Frequency(2nd subject): notFoundSimilarity,

Comparing subject to    : ReflectApp7, 0/1, TestScore, 0/9,
Similarity item/s       : notFoundSimilarity,
Frequency(1st subject): notFoundSimilarity,
Frequency(2nd subject): notFoundSimilarity,

Comparing subject to    : ReflectApp7, 1/1, ReflectApp7a, 1/1,
Similarity item/s       : static int *FileName*.nextVocab,
Frequency(1st subject): 1,
Frequency(2nd subject): 1,

Comparing subject to    : ReflectApp7, 0/1, ComboAxxx, 0/2,
Similarity item/s       : notFoundSimilarity,
Frequency(1st subject): notFoundSimilarity,
Frequency(2nd subject): notFoundSimilarity,
```

For example the last set of comparison, the Field as above, the items exist in both of the object files, the evidence can be getting from the value locates at behind the object file's name, such as 0/1, and 0/2. Unfortunately, there is no similarity among the items exist.

```
----------------------------------
********** For Methods ************
----------------------------------

Comparing subject to  : ReflectApp7, 9/21, ComboArray, 9/9,

Similarity item/s     : public boolean
java.lang.Object.equals(java.lang.Object), public final native
java.lang.Class java.lang.Object.getClass(), public int
java.lang.Object.hashCode(), public final native void
java.lang.Object.notify(), public final native void
java.lang.Object.notifyAll(), public java.lang.String
java.lang.Object.toString(), public final void
java.lang.Object.wait() throws java.lang.InterruptedException,
public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException, public final void
java.lang.Object.wait(long,int) throws
java.lang.InterruptedException,

Frequency(1st subject): 1, 1, 1, 1, 1, 1, 1, 1, 1,


Frequency(2nd subject): 1, 1, 1, 1, 1, 1, 1, 1, 1,



Comparing subject to  : ReflectApp7, 9/21, TestScore, 9/17,

Similarity item/s     : public boolean
java.lang.Object.equals(java.lang.Object), public final native
java.lang.Class java.lang.Object.getClass(), public int
java.lang.Object.hashCode(), public final native void
java.lang.Object.notify(), public final native void
java.lang.Object.notifyAll(), public java.lang.String
java.lang.Object.toString(), public final void
java.lang.Object.wait() throws java.lang.InterruptedException,
public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException, public final void
java.lang.Object.wait(long,int) throws
java.lang.InterruptedException,

Frequency(1st subject): 1, 1, 1, 1, 1, 1, 1, 1, 1,

Frequency(2nd subject): 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

For example, the above comparison set, for Method, the items matched were nine among 21 from the first object file and 17 from the second object file. Whereas for the below comparison set, the matching is full matched between these two object files.

```
Comparing subject to  : ReflectApp7, 21/21, ReflectApp7a, 21/21,

Similarity item/s     : static java.util.Vector
*FileName*.addToVocab(java.lang.Object[],java.lang.Object[]), static
boolean
```

User manual

This Reflection Application contains seven files, which is listed as below.

a) ReflectApp.java (source code)
b) ReflectApp.class
c) ReflectApp8$ExtensionFilter.class
d) AnalyzeVocab.class
e) ObjPropWeightage.class
f) Util.class
g) User.ini

Step 1:

These files should locate in the same folder. It is recommend for non-expert user to create a new folder, and copy these files into the created new folder. Let us name the folder as "RefApp"

Step 2:

Then, copying all the object files into the "RefApp" folder.
Note: In order to make the application function correctly, user must copy at least two or more object file (*.class),.

Step 3:

User can customize the application setting by editing the file **user.ini**. To edit this file, do use external text editor, such as Windows Notepad or Windows Word Pad, to edit the file. User can refer the example inside the file **user.ini** to customize the analysis setting.

```
ModifiersWeightage=2/10
InterfacesWeightage=2/10
FieldsWeightage=2/10
MethodsWeightage=2/10
ConstructorsWeightage=2/10
// The total of 5 should equal to 1
// example: (2/10) + (2/10) + (2/10) + (2/10) + (2/10) = 1

itemMatchingWeight=5/10
freqExactMatchWeight=2/10
freqRangeMatchWeight=3/10
// The total of 3 should equal to 1
// example: (5/10) + (2/10) + (3/10) = 1

StatusNotExitIsSimilar=yes
// yes or no

freqRange=20
// example: 20    -> +- 20%
// example: 28.59 -> +- 28.59%
```

```
*FileName*.checkEntryExist(java.lang.String,java.lang.Object[]),
static java.lang.String[]
*FileName*.convertObjectToStringArray(java.lang.Object[]), static
void
*FileName*.countFrequency(java.lang.Object[],java.lang.Object[],java
.lang.Object[]), static void
*FileName*.describeClassOrInterface(java.lang.Class,java.lang.String
), static void
*FileName*.displayConstructors(java.lang.reflect.Constructor[]),
static void *FileName*.displayFields(java.lang.reflect.Field[]),
static void *FileName*.displayInterfaces(java.lang.Class[]), static
void *FileName*.displayMethods(java.lang.reflect.Method[]), static
void *FileName*.displayModifiers(int), public static void
*FileName*.main(java.lang.String[]), static void
*FileName*.printArray(java.lang.String[]), public boolean
java.lang.Object.equals(java.lang.Object), public final native
java.lang.Class java.lang.Object.getClass(), public int
java.lang.Object.hashCode(), public final native void
java.lang.Object.notify(), public final native void
java.lang.Object.notifyAll(), public java.lang.String
java.lang.Object.toString(), public final void
java.lang.Object.wait() throws java.lang.InterruptedException,
public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException, public final void
java.lang.Object.wait(long,int) throws
java.lang.InterruptedException,

Frequency(1st subject): 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,

Frequency(2nd subject): 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,



Comparing subject to  : ReflectApp7, 9/21, ComboAxxx, 9/9,

Similarity item/s     : public boolean
java.lang.Object.equals(java.lang.Object), public final native
java.lang.Class java.lang.Object.getClass(), public int
java.lang.Object.hashCode(), public final native void
java.lang.Object.notify(), public final native void
java.lang.Object.notifyAll(), public java.lang.String
java.lang.Object.toString(), public final void
java.lang.Object.wait() throws java.lang.InterruptedException,
public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException, public final void
java.lang.Object.wait(long,int) throws
java.lang.InterruptedException,

Frequency(1st subject): 1, 1, 1, 1, 1, 1, 1, 1, 1,

Frequency(2nd subject): 1, 1, 1, 1, 1, 1, 1, 1, 1,


---------------------------------
********** For Constructors *******
---------------------------------
```

```
Comparing subject to   : ReflectApp7, 0/1, ComboArray, 0/1,
Similarity item/s      : notFoundSimilarity,
Frequency(1st subject): notFoundSimilarity,
Frequency(2nd subject): notFoundSimilarity,

Comparing subject to   : ReflectApp7, 0/1, TestScore, 0/1,
Similarity item/s      : notFoundSimilarity,
Frequency(1st subject): notFoundSimilarity,
Frequency(2nd subject): notFoundSimilarity,

Comparing subject to   : ReflectApp7, 1/1, ReflectApp7a, 1/1,
Similarity item/s      : public *FileName*(),
Frequency(1st subject): 2,
Frequency(2nd subject): 2,

Comparing subject to   : ReflectApp7, 0/1, ComboAxxx, 0/1,
Similarity item/s      : notFoundSimilarity,
Frequency(1st subject): notFoundSimilarity,
Frequency(2nd subject): notFoundSimilarity,


Process Exit...
```

For above sample output, it depends on the information in the file **user.ini** as below. According to below information, the non-exit items are considered as similar for the analysis. There is a guideline in this **user.ini** to guide user to modify the content on file. For example, the range of frequency matching is sated in percentage form. There are some example are listed for user.

```
ModifiersWeightage=2/10
InterfacesWeightage=2/10
FieldsWeightage=2/10
MethodsWeightage=2/10
ConstructorsWeightage=2/10
// The total of 5 should equal to 1
// example: (2/10) + (2/10) + (2/10) + (2/10) + (2/10) = 1

itemMatchingWeight=5/10
freqExactMatchWeight=2/10
freqRangeMatchWeight=3/10
// The total of 3 should equal to 1
// example: (5/10) + (2/10) + (3/10) = 1

StatusNotExitIsSimilar=yes
// yes or no

freqRange=20
// example: 20    -> +- 20%
// example: 28.59 -> +- 28.59%
// example: 200   -> +- 200%
```

Appendix B:

User manual

```
// example: 200   -> +- 200%
```

The default setting for **user.ini** is showed as above. There are eight weightages, one exist status, and one frequency's range retrieved by the application. The first five weightages are the weightage for Modifier, Interface, Field, Method, and Constructor. These weightages will bring direct effect to final application's output.

The next three weightages have the direct effect on the categories Field, Method, and Constructor only. They are divided into item matching weightage, exact frequency weightage, and the range frequency weightage. For above the first five weightages or the next three weightages should sum up as one.

Next, it is the status of not exit similarity for five categories, which are Modifier, Interface, Field, Method, and Constructor, have been considered in this application. The value for this status can be "yes" or "no" only.

The last item is the range of frequency matching, specifically for Field, Method, and Constructor. This item should filling with percentage value as in the sample in file **user.ini**.

<u>Step 4:</u>

Execute the application.

It is recommended that the user use the tested environment to execute the application, which is in Tek-Tools KAWA IDE (can be download form http://www.tek-tools.com, and install it with pre-install JDK version 1.2 or above) environment. In KAWA IDE, user just need to open up the ReflectApp.java file, compile the file (press F7), and run the file (press F4).

Of course, the user can also run the application in MS-Dos Prompt environment, by typing this command line in as below.
C:\RefApp\ java ReflectApp press **Enter**

Note: To execute the application in any platform, the platform must pre-installed the JDK version 1.2 and above (can be download from http://java.sun.com) before executing the application.

Appendix C:

Figure / diagram

Figure 1: Object model diagram

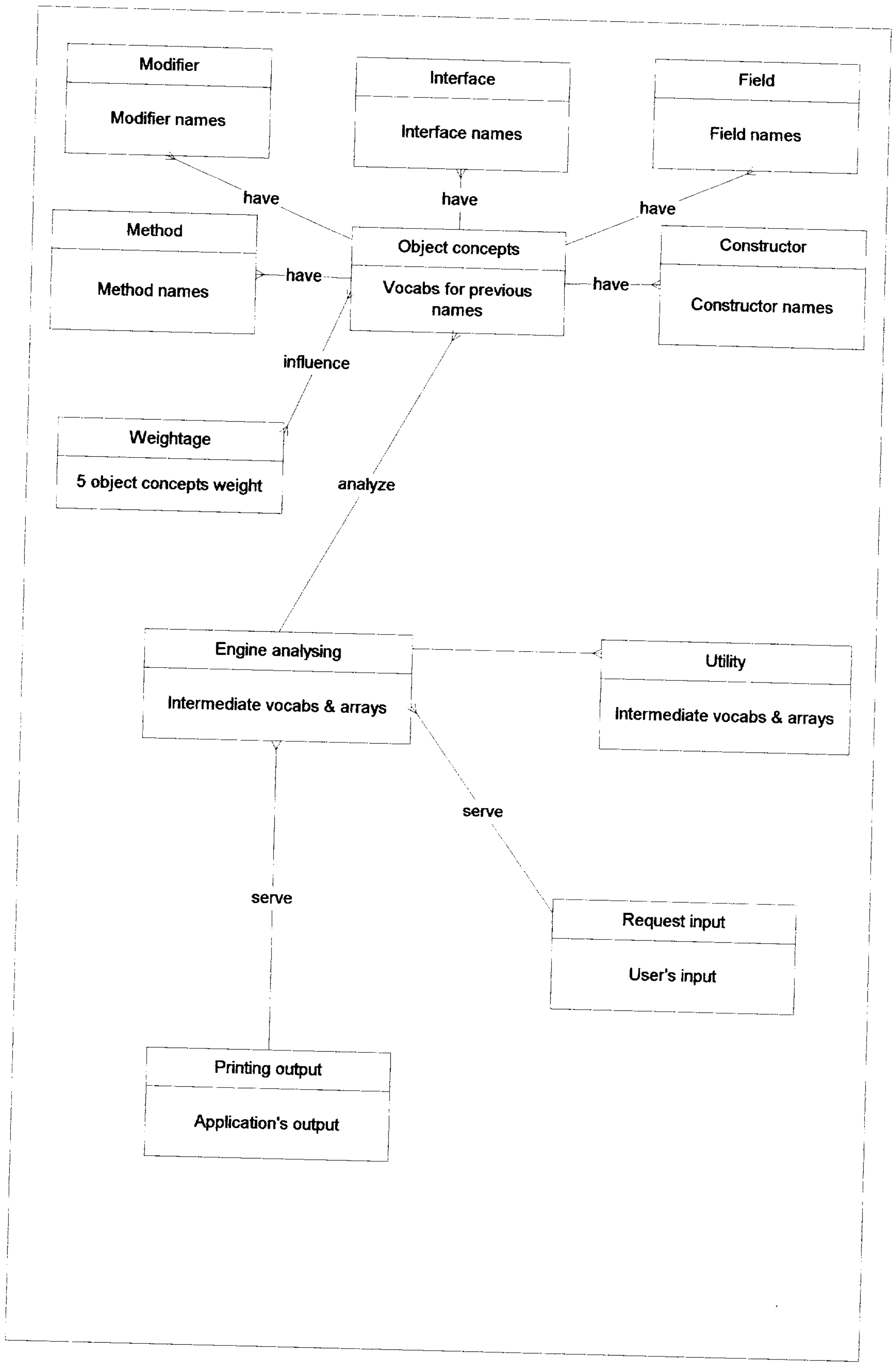


Figure 2: Event trace for Reflection Application

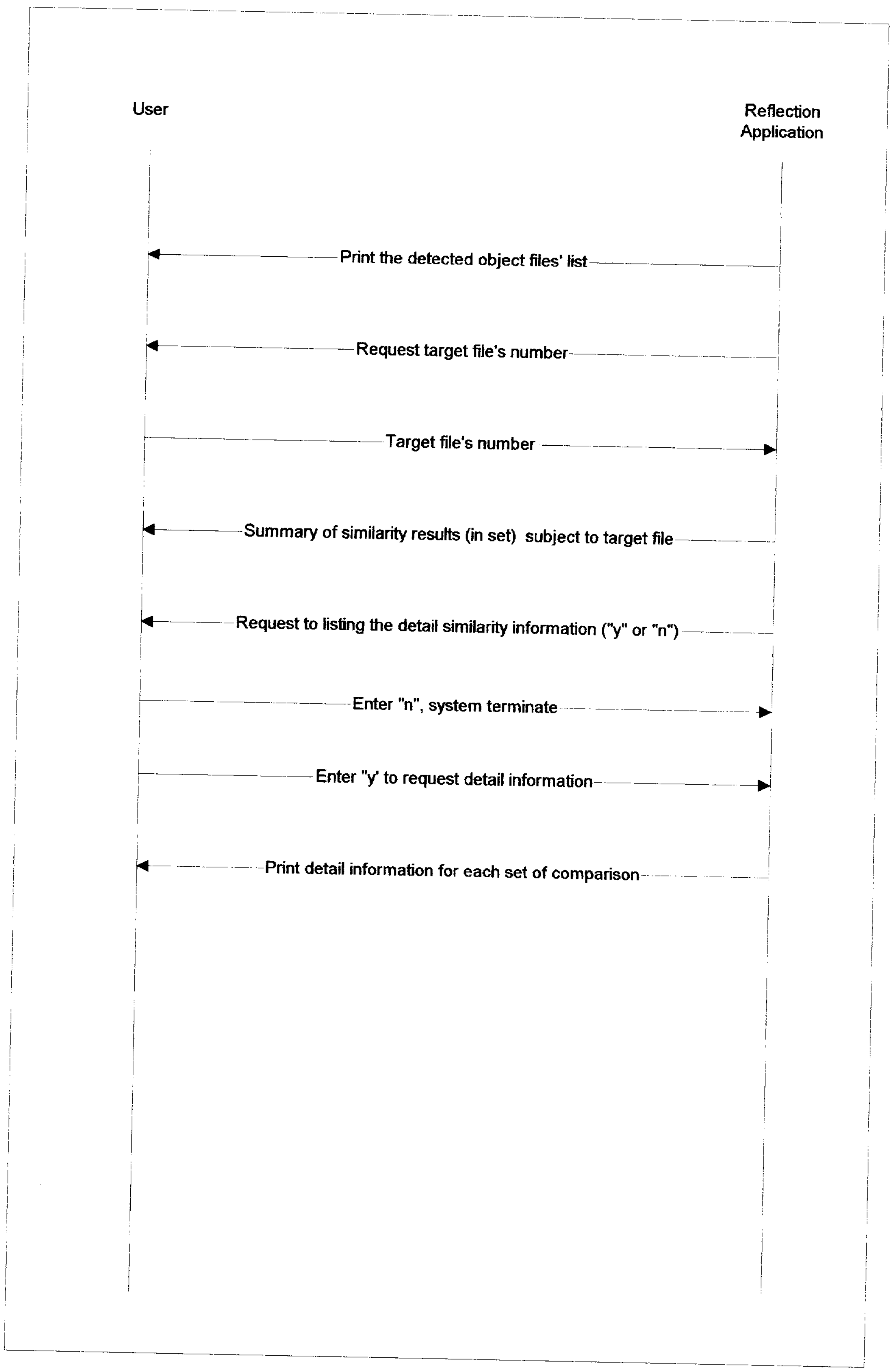

User
Reflection Application
Print the detected object files' list
Request target file's number
Target file's number
Summary of similarity results (in set) subject to target file
Request to listing the detail similarity information ("y" or "n")
Enter "n", system terminate
Enter "y' to request detail information
Print detail information for each set of comparison

Figure 3: Event flow diagram

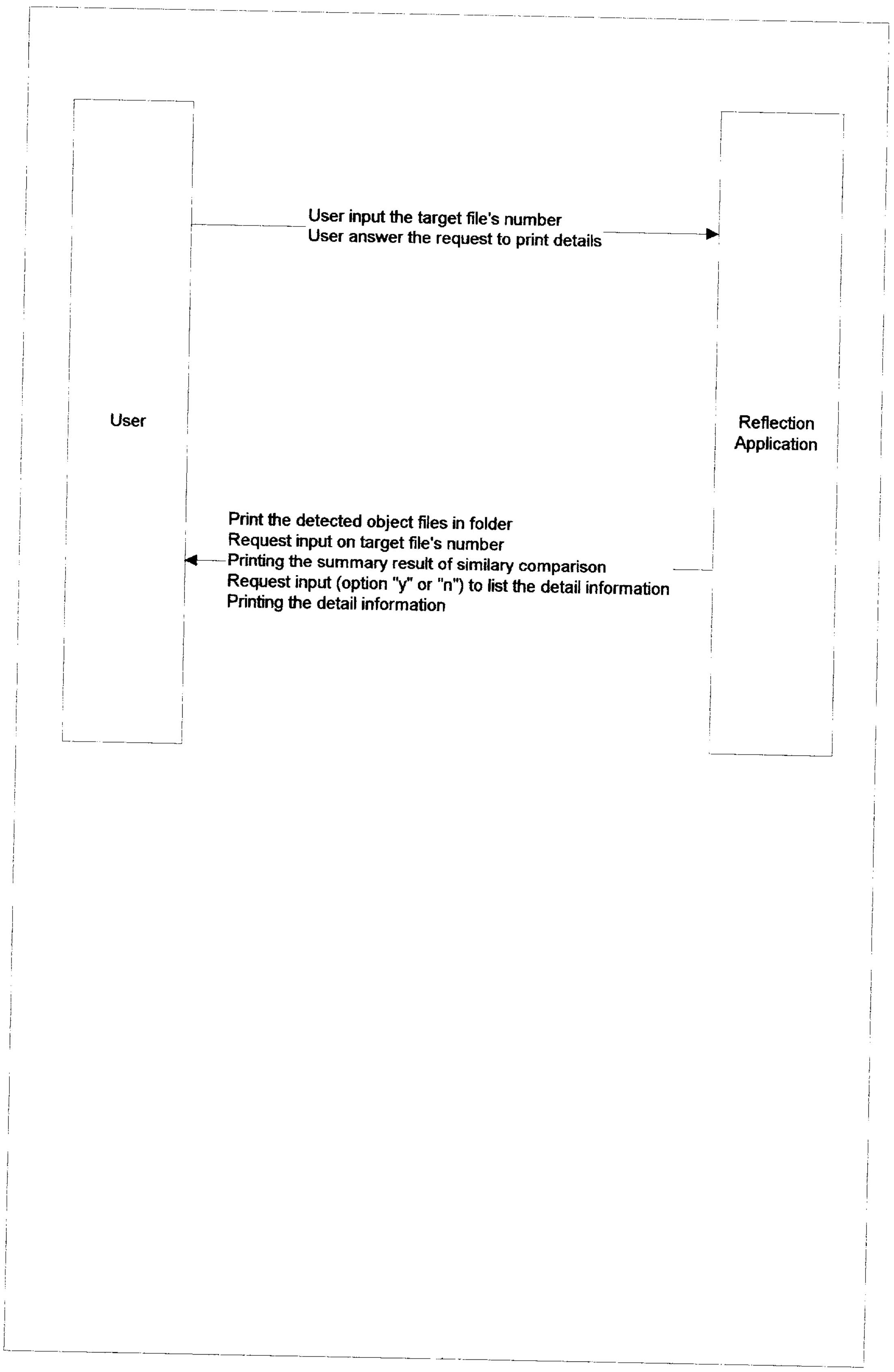

User
User input the target file's number
User answer the request to print details
Reflection Application
Print the detected object files in folder
Request input on target file's number
Printing the summary result of similary comparison
Request input (option "y" or "n") to list the detail information
Printing the detail information

Figure 4: State diagram for class ReflectApp

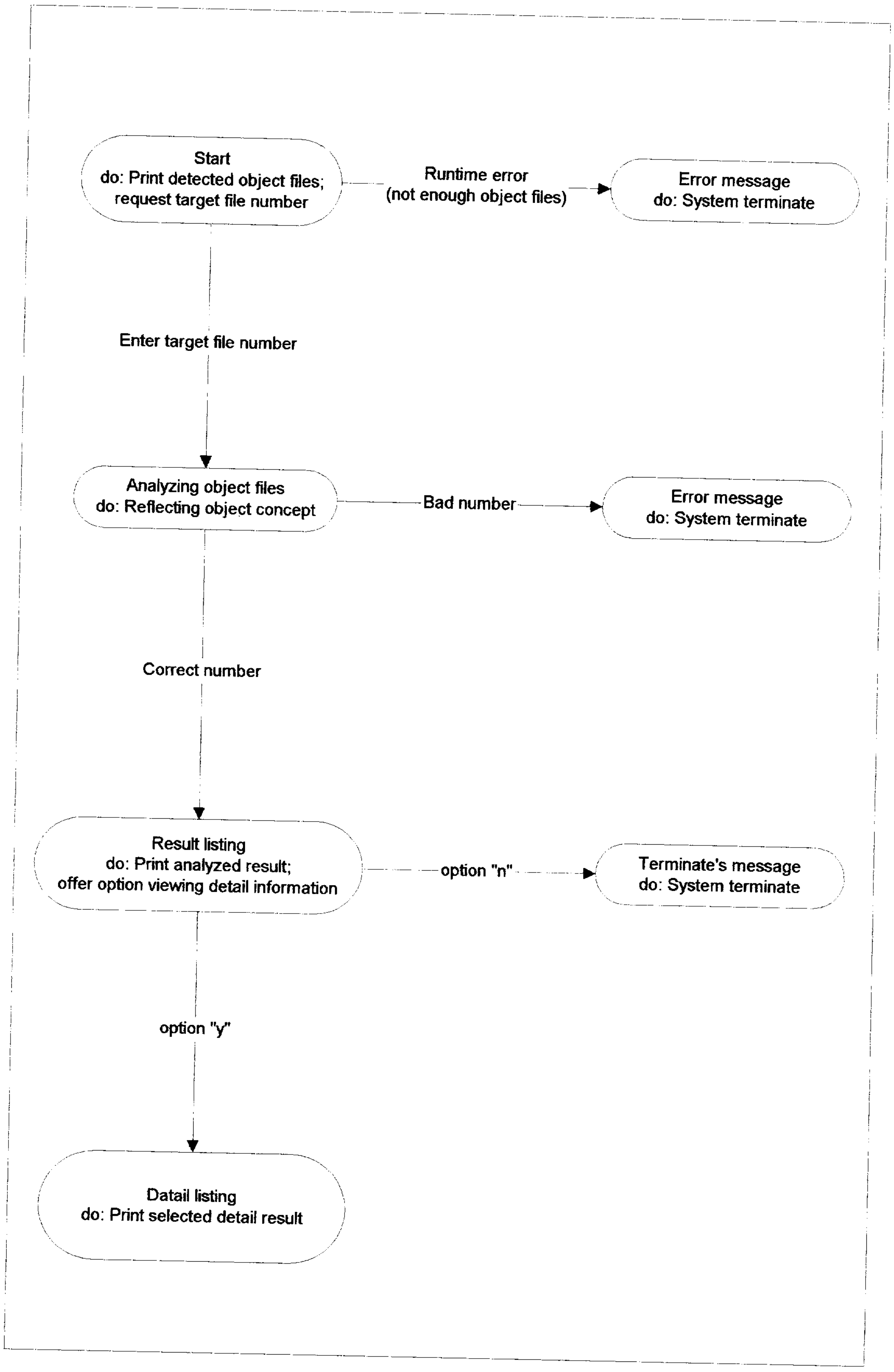


Figure 5: Input and output values for ReflectApp system

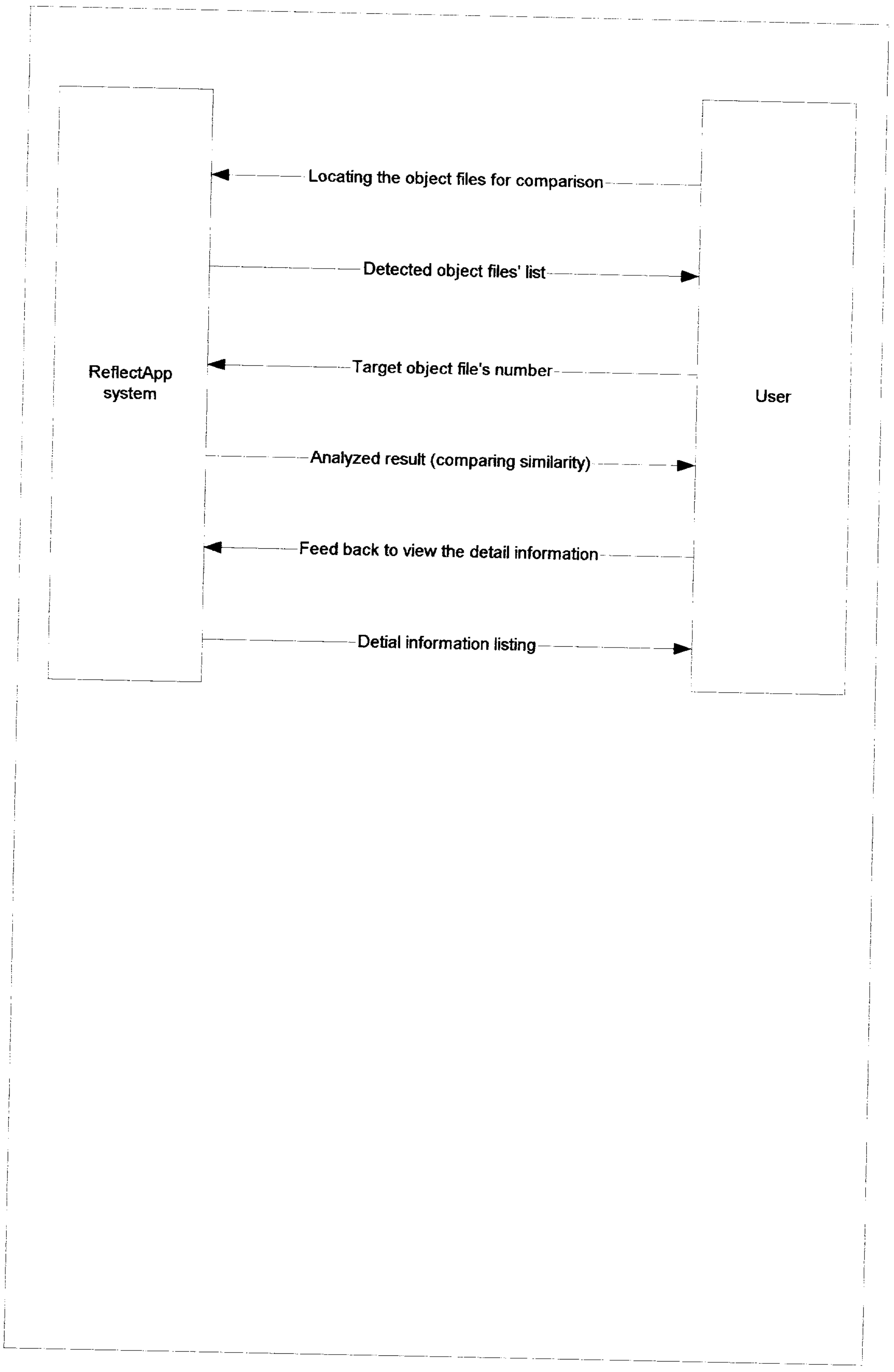

Locating the object files for comparison
Detected object files' list
ReflectApp system
Target object file's number
User
Analyzed result (comparing similarity)
Feed back to view the detail information
Detial information listing

Figure 6: Data flow diagram (Level 0)
(Context diagram)

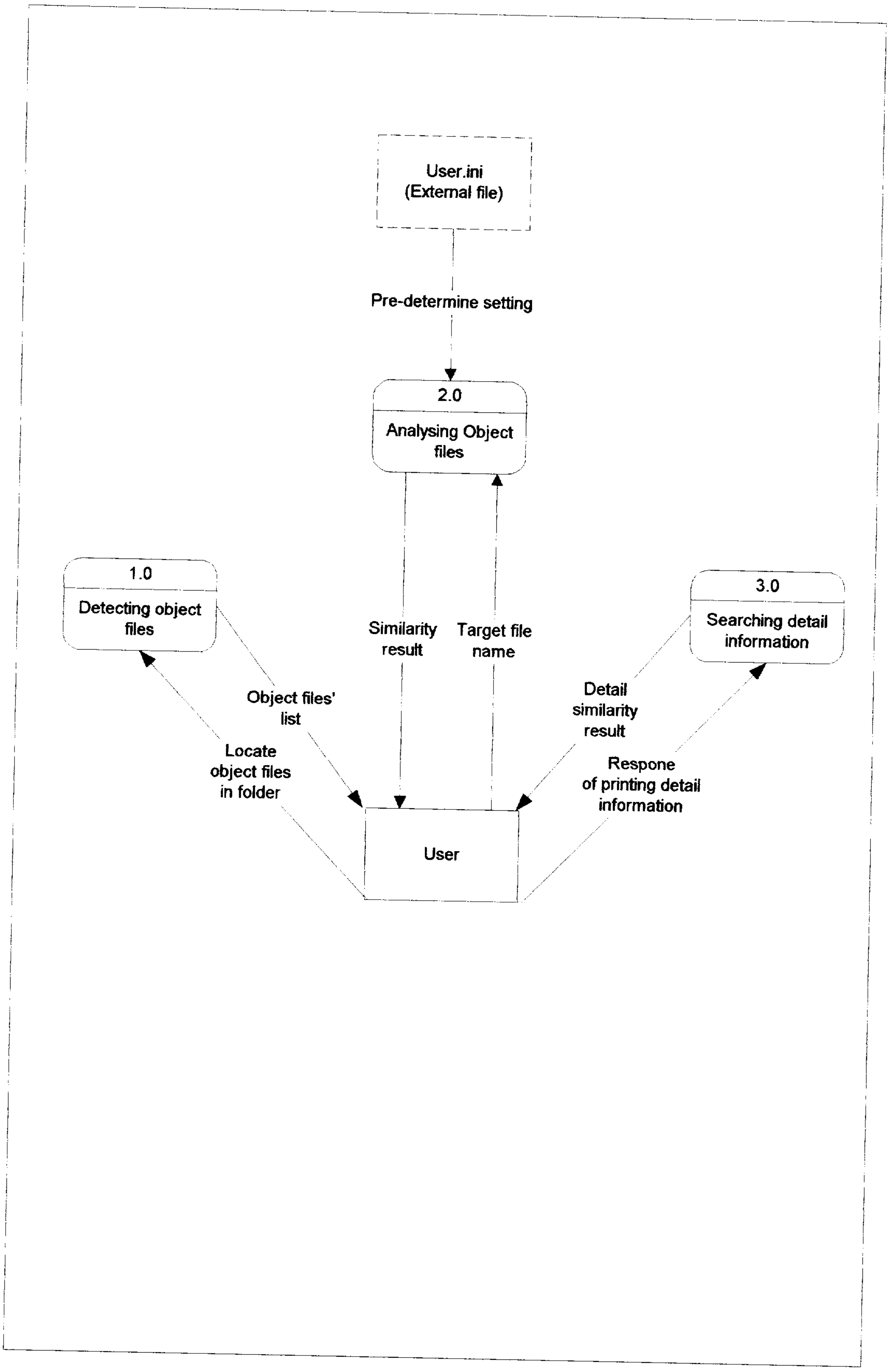


Figure 7: Data flow diagram (Level 1)
Process 1.0 (Detecting object files)

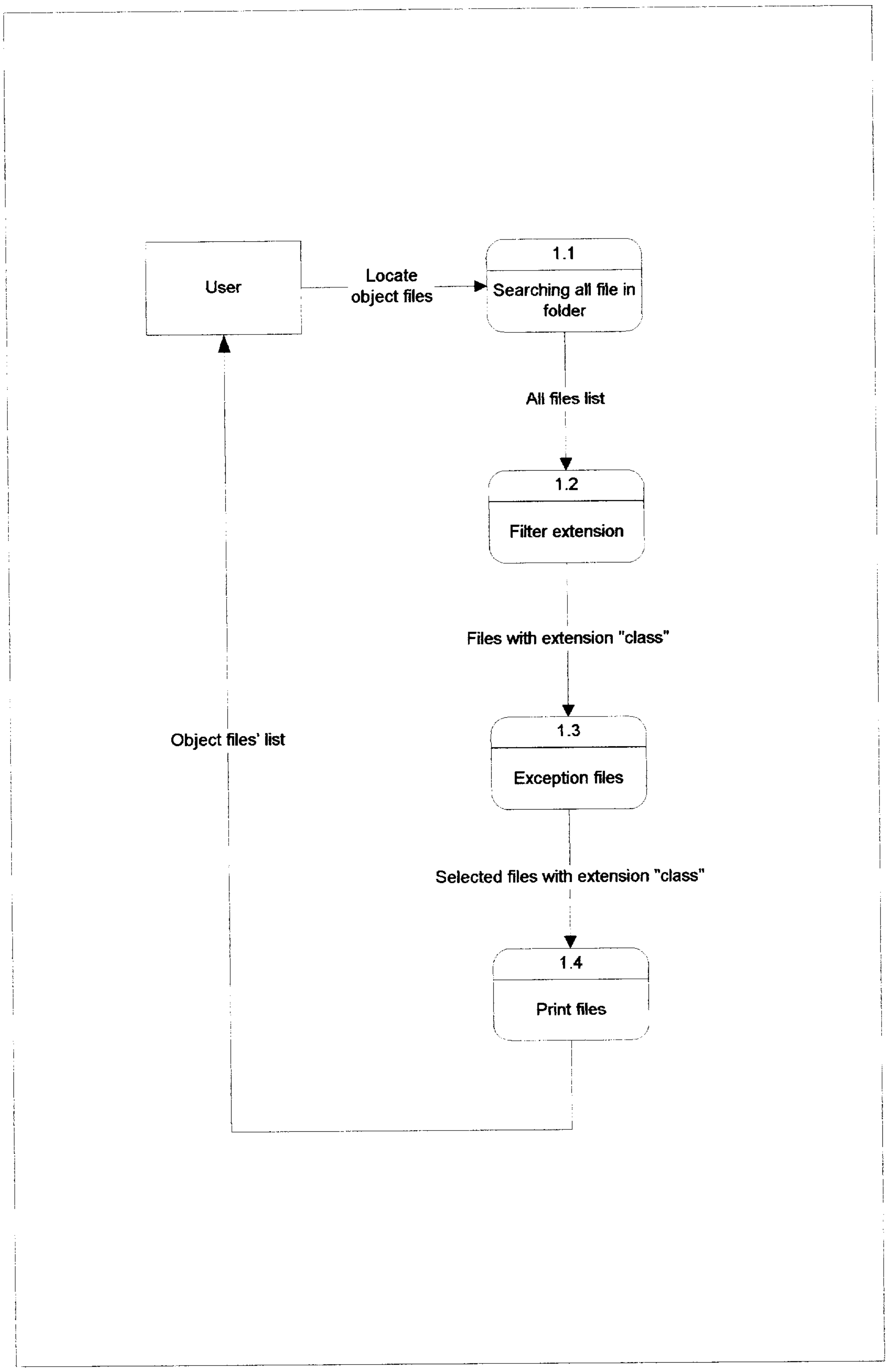


Figure 8: Data flow diagram (Level 1)
Process 2.0 (Analysing object files)

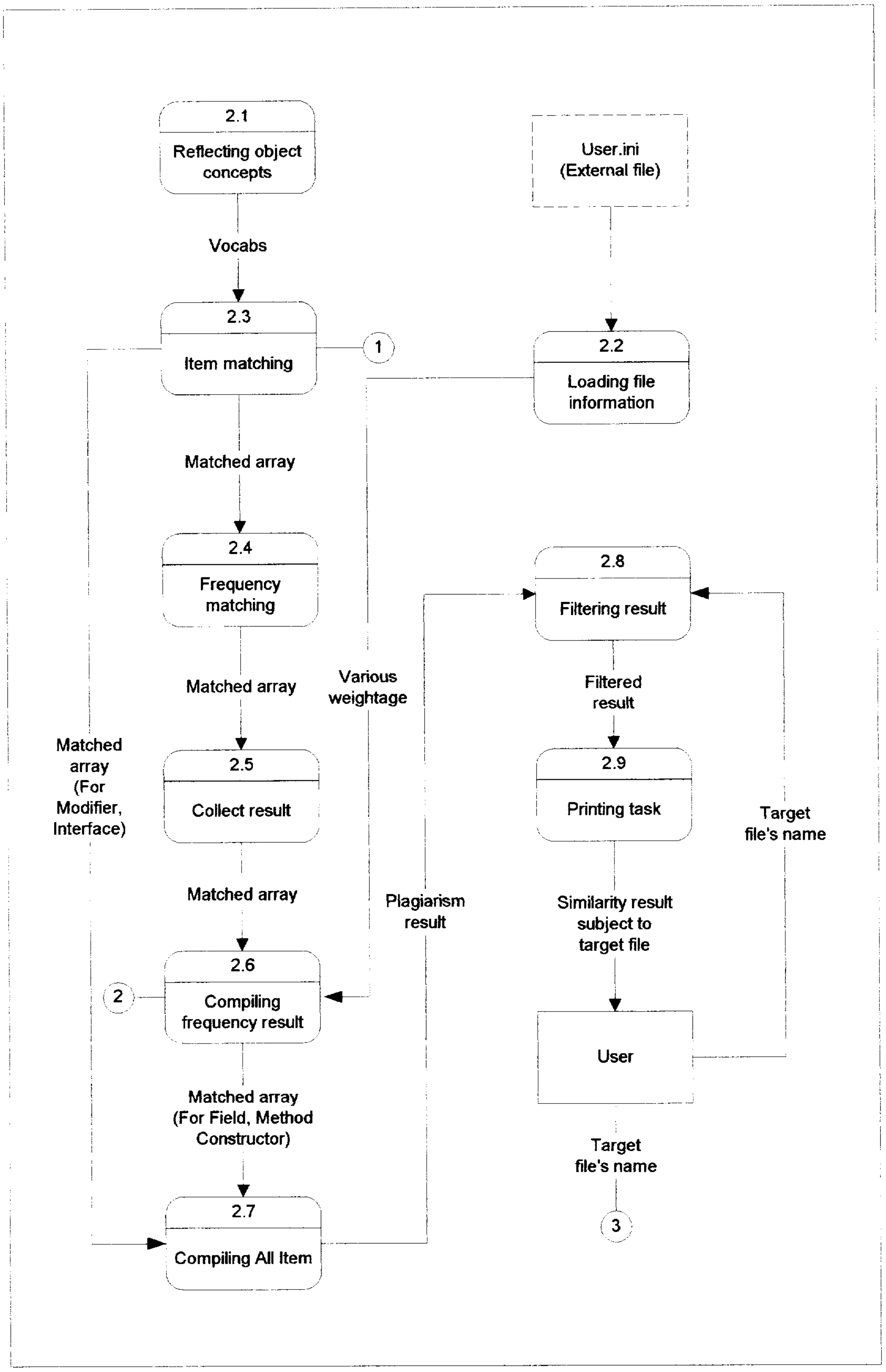
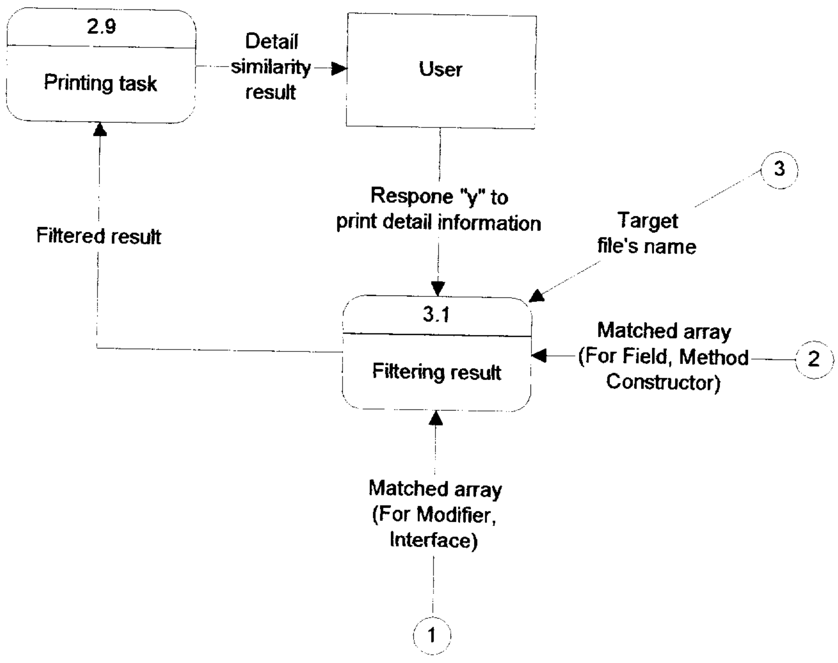
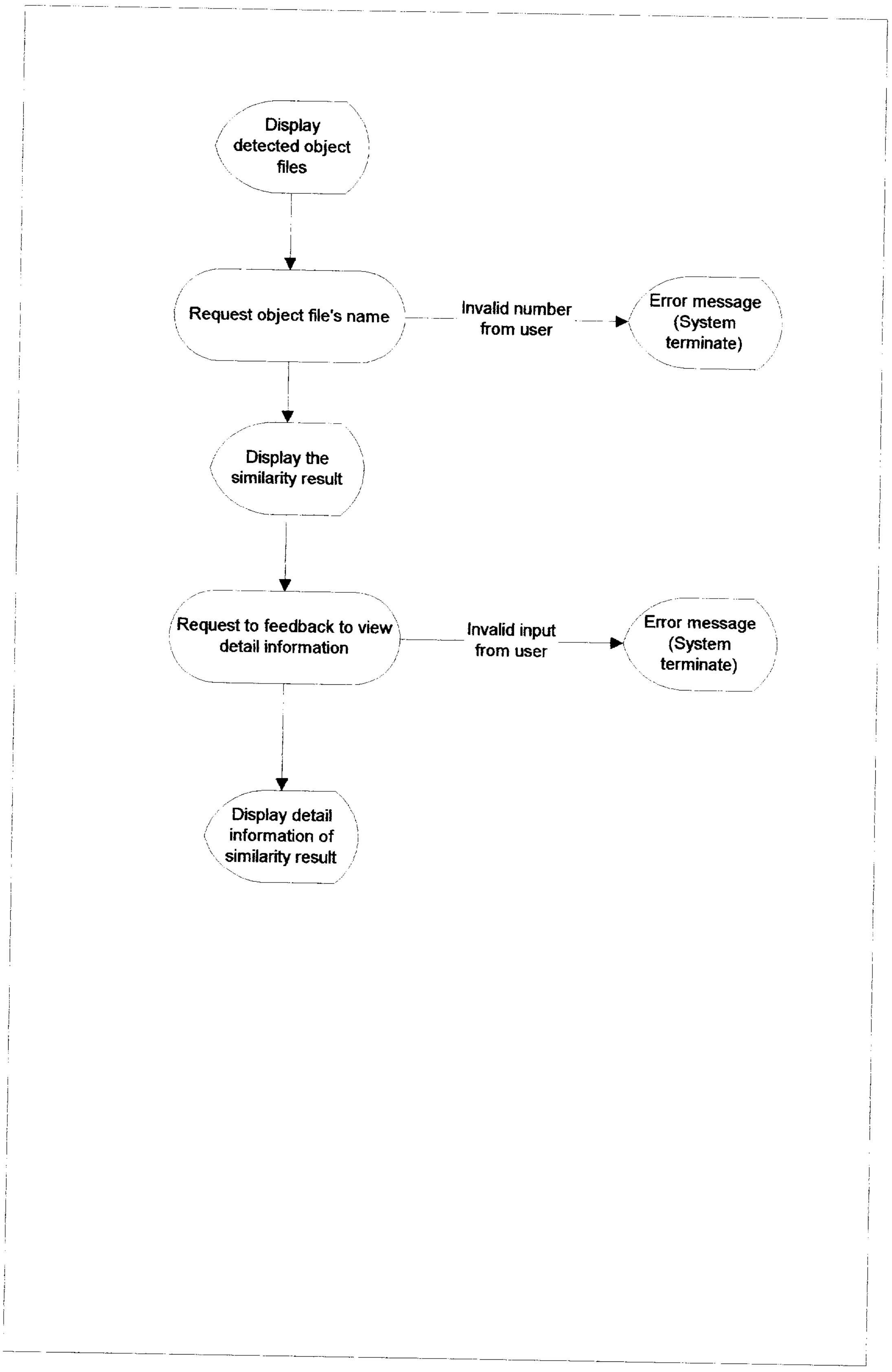


Figure 9: Data flow diagram (Level 1)
Process 3.0 (Searching detail information)



2.9
Printing task
Detail similarity result
User
Respone "y" to print detail information
Filtered result
Target file's name
3
3.1
Filtering result
Matched array (For Field, Method Constructor)
2
Matched array (For Modifier, Interface)
1

Figure 10: ReflectApp control


Display detected object files
Request object file's name
Invalid number from user
Error message (System terminate)
Display the similarity result
Request to feedback to view detail information
Invalid input from user
Error message (System terminate)
Display detail information of similarity result

Appendix D:

Gantt chart

Gantt chart

Gantt chart for development of Java Reflection Application (14 weeks)

	Month			**November**				**December**				**January 2000**					**February**			
	Date	**Start**	**Finish**	6	14	20	28	4	11	18	25	1	8	15	22	29	5	12	19	26
	Week			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	**Planning**																			
	Review project proposal	6-Nov	18-Dec	*	*	*	*	*	*	*										
	Exploring Java reflection	6-Nov	8-Jan	*	*	*	*	*	*	*	*	*	*							
	Exploring Java syntax and data structures	6-Nov	19-Feb	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
2	**Analysis stage**																			
	Comparing exist & proposed techniques	14-Nov	4-Dec		*	*	*	*												
	Identifying problem	14-Nov	28-Nov		*	*	*													
	Discovering user requirements	20-Nov	29-Jan			*	*	*	*	*	*	*	*	*	*	*				
	Object modeling	28-Nov	22-Jan				*	*	*	*	*	*	*	*	*					
	Dynamic modeling	28-Nov	29-Jan				*	*	*	*	*	*	*	*	*	*				
	Functional modeling	4-Dec	5-Feb					*	*	*	*	*	*	*	*	*	*			
3	**System design stage**																			
	Logical design	4-Dec	22-Jan					*	*	*	*	*	*	*	*					
4	**Object design stage**																			
	Object design	18-Dec	12-Jan							*	*	*	*	*	*	*	*	*		
	Verifying object	15-Jan	19-Jan											*	*	*	*	*	*	
4	**Implementation stage**																			
	Coding	11-Dec	19-Feb						*	*	*	*	*	*	*	*	*	*	*	
	System testing	15-Jan	19-Jan											*	*	*	*	*	*	
5	**Documentation**	22-Jan	26-Feb												*	*	*	*	*	*

Appendix E:

Glossary

Glossary

1. **Platform independence** means that a program can run on any computer system. Java programs can run on any system for which a Java Virtual Machine has been installed (Lemay, 1996).

2. According to Horstmann (1998), there are two kind of Java program that can be created, which is **console** application, and graphics **applet**. Java Applet uses to write application similar like window applications, which have Graphic User Interface (GUI) components. For handling the events associated with those components, the console program take into place, where there are no windows or GUI components, and in simple programs, the only event that user usually need to handle is the entering of keyboard data by the user. The primary difference between **console** programs, and window applications or **applets** is that console programs lack a graphical user interface. Console programs have the same entry point source code as window applications--a main() method with a String[] argument.

3. Java software development is based upon the use and reuse of packages. Both Java 1.0 and Java 1.1 used packages. However, the **Package class** is new to JDK 1.2. It provides methods for obtaining package version information stored in the manifest of .jar files. The Package class provides fourteen methods that can be used to retrieve information about packages. The static **getPackage()** and **getAllPackages()** methods provide Package objects that are known to the current class loader. The **getName()**, **getSpecificationTitle()**, **getImplementationTitle()**, **getSpecificationVersion()**, **getImplementationVersion()**, **getSpecificationVendor()**, and **getImplementationVendor()** methods return name, title, version, and vendor information about the specification and implementation of packages. The **getSealBase()** method returns the base URL of a signed package. The **isSealed()** method is used to determine if a package is sealed. The **isCompatibleWith()** method is used to determine whether a package is comparable with a particular version. The **hashCode() and toString()** methods override those inherited from the Object class.

4. The **Object class** does not have any variables and has only one constructor. However, it provides 11 methods that are inherited by all Java classes and support general operations used by all objects. For example, the **equals()** and **hashCode()** methods are used to construct hash tables of Java objects. Hash tables are like arrays, but are indexed by key values and dynamically grow in size. They make use of hash functions to quickly access the data that they contain. The hashCode() method creates a hash code for an object. Hash codes are used to quickly determine whether two objects are different.

   The **clone()** method creates an identical copy of an object. The object must implement the Cloneable interface. This interface is defined within the java.lang package. It contains no methods and is used only to differentiate clonable classes from non-clonable classes.

The **getClass()** method identifies the class of an object by returning an object of Class.

The **toString()** method creates a String representation of the value of an object. This method is handy for quickly displaying the contents of an object. When an object is displayed, using **print()** or **println()**, the **toString()** method of its class is automatically called to convert the object into a string before printing. Classes that override the toString() method can easily provide a custom display for their objects.

The **finalize()** method of an object is executed when an object is garbage-collected. The method performs no action, by default, and needs to be overridden by any class that requires specialized finalization processing.

The Object class provides three **wait()** and two **notify()** methods that support thread control. These methods are implemented by the Object class so that they can be made available to threads that are not created from subclasses of class Thread. The wait() methods cause a thread to wait until it is notified or until a specified amount of time has elapsed. The notify() methods are used to notify waiting threads that their wait is over.

5. The **Class class** provides over 30 methods that support the runtime processing of an object's class and interface information. This class does not have a constructor. Objects of this class, referred to as class descriptors, are automatically created and associated with the objects to which they refer. Despite their name, class descriptors are used for interfaces as well as classes.

   The **getName()** and **toString()** methods return the String containing the name of a class or interface. The toString() method differs in that it prepends the string class or interface, depending on whether the class descriptor is a class or an interface. The static **forName()** method loads the class specified by a String object and returns a class descriptor for that class.

   The **getSuperclass()** method returns the class descriptor of a class's superclass. The **isInterface()** method identifies whether a class descriptor applies to a class or an interface. The **getInterfaces()** method returns an array of Class objects that specify the interfaces of a class, if any.

   The **newInstance()** method creates an object that is a new instance of the specified class. It can be used in lieu of a class's constructor, although it is generally safer and clearer to use a constructor rather than newInstance().

   The **getClassLoader()** method returns the class loader of a class, if one exists. Classes are not usually loaded by a class loader. However, if a class is loaded from outside the CLASSPATH, such as over a network, a class loader (Classes that are loaded from outside the CLASSPATH require a class loader to convert the class byte stream into a class descriptor. ClassLoader is an abstract class that is used to define class loaders) is used to convert the class byte stream into a class descriptor. The Class class contains a number of other methods that begin with get and is. These methods are as follows:

a) **getClasses()** -- Returns an array of all classes and interfaces that are members of the class.
b) **getComponentType()** -- Returns the component type of an array.
c) **getConstructor()** and **getConstructors()** -- Return Constructor objects for the class.
d) **getDeclaredClasses()**, **getDeclaredConstructor()**, **getDeclaredConstructors()**, **getDeclaredField()**, **getDeclaredFields()**, **getDeclaredMethod()**, and **getDeclaredMethods()** -- Return the classes, constructors, fields, and methods that are declared for a class or interface.
e) **getDeclaringClass()** -- Returns the class in which the referenced class is declared (if any).
f) **getField()** and **getFields()** -- Returns a specific Field object or all Field objects of a class or interface.
g) **getMethod()** and **getMethods()** -- Returns a specific Method object or all Method objects of a class or interface.
h) **getModifiers()** -- Returns the class or interface modifiers as a coded integer.
i) **getResource()** and **getResourceAsStream()** -- Locates system resources. System resources are objects that are used by the runtime system or local Java implementation.
j) **getSigners()** -- Returns the signers of a class.
k) **isArray()** -- Returns true if the Class object represents an array.
l) **isAssignableFrom()** -- Used to determine whether an object of one class can be assigned to an object of another class.
m) **isInstance()** -- Equivalent to the isInstanceOf operator.
n) **isPrimitive()** -- Returns true if the object represents a primitive type.

6. The **Member interface** is used to provide information about a Field, Constructor, or Method. It defines two constant variables and three methods. The **DECLARED** constant identifies the class members (fields, constructors, and methods) that are declared for a class. The **PUBLIC** constant identifies all members of a class or interface, including those that are inherited. The **getName()** method returns the name of the referenced Member. The **getModifiers()** method returns the modifiers of the referenced Member encoded as an integer. The Modifier class is used to decode this integer. The **getDeclaringClass()** method returns the class in which the Member is declared.

7. The **AccessibleObject class** is introduced with JDK 1.2. It is the superclass of the Constructor, Field, and Method classes. It was added to the class hierarchy to provide the capability to specify whether an object suppresses reflection-access control checks. The **isAccessible()** method identifies whether the object

suppresses access control checks. The **setAccessible()** method is used to set the accessibility of an object or array of objects.

8. The **Array class** is used to obtain information about, create, and manipulate arrays. It consists of 21 static methods. The **getLength()** method is used to access the length of an array. The **get()** method is used to access an indexed element of an array. The **getBoolean()**, **getByte()**, **getChar()**, **getDouble()**, **getFloat()**, **getInt()**, **getLong()**, and **getShort()** methods are used to access an indexed element of an array as a particular primitive type. The **set()** method is used to set an indexed element of an array. The **setBoolean()**, **setByte()**, **setChar()**, **setDouble()**, **setFloat()**, **setInt()**, **setLong()**, and **setShort()** methods are used to set an indexed element of an array to a value of a particular primitive type. The **newInstance()** method is used to create new arrays of a specified size.

9. The **Constructor class** is used to obtain information about and access the constructors of a class. It consists of nine methods. The **getName()** method returns the name of the constructor. The **getDeclaringClass()** method identifies the class to which the constructor applies. The **newInstance()** method is used to create a new instance of the class to which the constructor applies. The **getParameterTypes()** method provides access to the parameters used by the constructor. The **getModifiers()** method encodes the constructor's modifiers as an integer that can be decoded by the Modifier class. The **getExceptionTypes()** method identifies the exceptions that are thrown by the constructor. The **equals()**, **hashCode()**, and **toString()** methods override those of the Object class.

10. The **Field class** is used to obtain information about and access the field variables of a class. It consists of 25 methods. The **getName()** method returns the name of the variable. The **getDeclaringClass()** method identifies the class in which the variable is declared. The **getType()** method provides access to the data type of the variable. The **getModifiers()** method encodes the variable's modifiers as an integer that can be decoded by the Modifier class. The **get()** method is used to access the value of the variable. The **getBoolean()**, **getByte()**, **getChar()**, **getDouble()**, **getFloat()**, **getInt()**, **getLong()**, and **getShort()** methods are used to access the value as a particular primitive type. The **set()** method is used to set the value of the variable. The **setBoolean()**, **setByte()**, **setChar()**, **setDouble()**, **setFloat()**, **setInt()**, **setLong()**, and **setShort()** methods are used to set the value to a particular primitive type. The **equals()**, **hashCode()**, and **toString()** methods override those of the Object class.

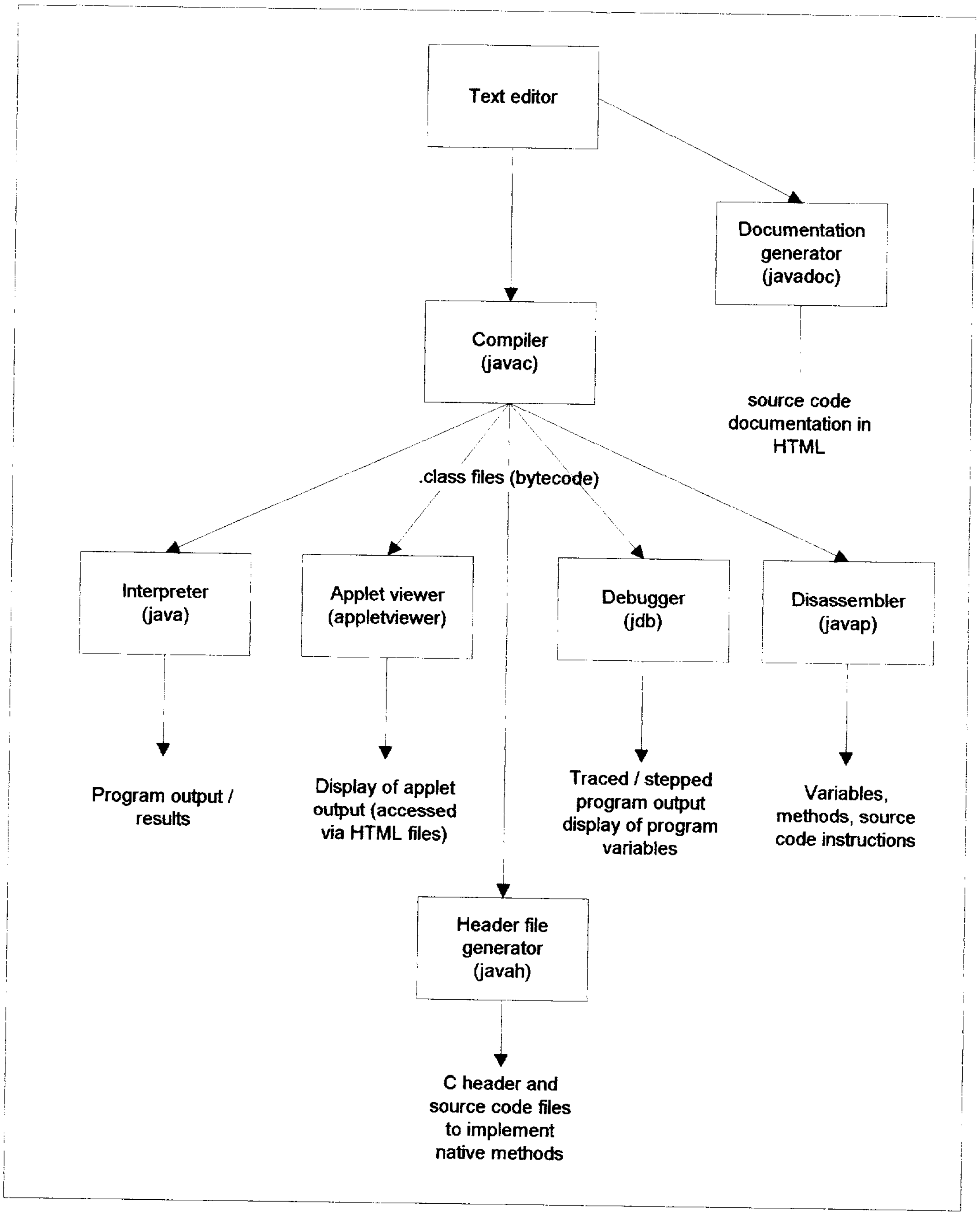
11. The **Method class** is used to obtain information about and access the methods of a class. It consists of 10 methods. The **getName()** method returns the name of the method. The **getDeclaringClass()** method identifies the class in which the method is declared. The **invoke()** method is used to invoke the method for a particular object and list of parameters. The **getParameterTypes()** method provides access to the parameters used by the method. The **getModifiers()** method encodes the method's modifiers as an integer that can be decoded by the Modifier class. The **getExceptionTypes()** method identifies the exceptions that

are thrown by the method. The **getReturnType()** method identifies the type of object returned by the method. The **equals()**, **hashCode()**, and **toString()** methods override those of the Object class.

12. The **Modifier class** is used to decode integers that represent the modifiers of classes, interfaces, field variables, constructors, and methods. It consists of 11 constants, a single parameterless constructor, and 12 static access methods. The 11 constants are used to represent all possible modifiers. They are **ABSTRACT**, **FINAL**, **INTERFACE**, **NATIVE**, **PRIVATE**, **PROTECTED**, **PUBLIC**, **STATIC**, **SYNCHRONIZED**, **TRANSIENT**, and **VOLATILE**. The **toString()** method returns a string containing the modifiers encoded in an integer. The **isAbstract()**, **isFinal()**, **isInterface()**, **isNative()**, **isPrivate()**, **isProtected()**, **isPublic()**, **isStatic()**, **isSynchronized()**, **isTransient()**, and **isVolatile()** methods return a boolean value indicating whether the respective modifier is encoded in an integer.

13. The **ReflectPermission class** is a permission class introduced with Java Development Kit version 1.2. It is used to specify whether the default language access should be suppressed for reflected objects.

14. **Java Development Kit (JDK)** provides a complete set of tools for the development, testing, documentation, and execution of Java programs and applets. Generlly JDK consists of seven programs, which are javac (compiler), java (interpreter), jdb (debugger), javap (disassembler), appletviewer (Applet viewer), javadoc (documentation generator), javah (C language header file generator). The latest JDK can be downloaded form the Sun Microsystems™ web page, which located at download page of **http://java.sun.com/** .



Text editor
Documentation generator (javadoc)
Compiler (javac)
source code documentation in HTML
.class files (bytecode)
Interpreter (java)
Applet viewer (appletviewer)
Debugger (jdb)
Disassembler (javap)
Program output / results
Display of applet output (accessed via HTML files)
Traced / stepped program output display of program variables
Variables, methods, source code instructions
Header file generator (javah)
C header and source code files to implement native methods

Usually (Jaworski, 1996), the Java program is writing with a text editor to develop Java source files. These files consist of source code packages that declare Java classes and interfaces. Source files use the **.java** extension. The Java compiler, **javac**, is uses to convert the source files into files that can execute with Java interpreter. These files are referred to as byte code files and end with the **.class** extension. The Java interpreter, **java**, executes classes from the byte code (.class) files. It verifies the integrity, correct operation, and security of each class as it is loaded and executed, and interacts with the host operating system, windowing environment and communication facilities to produce the desired program behavior. The debugger, **jdb**, is like the interpreter in that it executes Java classes that have been compiled into byte code files, but it also provides special capabilities to stop program execution at selected breakpoints and to display the values of class variables. These capabilities are very useful in finding programming errors. The disassembler (**javap**) takes the byte code files and displays the classes, fields (variables), and methods that have been compiled into the byte codes. It also identifies the byte code instructions used to implement each method. The disassembler is a handy tool for recovering the source code design of those compiled Java classes for which no source code is available-for example, those that you would retrieve from the Web. The applet viewer, **appletviewer**, displays Java applets contained within Web pages, located on your local file system, or at accessible websites. It is used to test applets that you develop. The automated documentation tool, **javadoc**, is used to convert portions of Java source files into Hypertext Markup Language (HTML) files. HTML is the language used to write Web pages. The HTML files generated by javadoc document the classes, variables, methods, interfaces, and exceptions contained in Java source files based on special comments inserted into these files. The C programming language's header file tool, **javah**, is used to generate C-language header and source files from a Java byte code file. The files generated by javah are used to develop native methods-Java classes that are written in languages other than Java.

15. Two varieties of type-safe linguistic reflection: With **compile time linguistic reflection**, the generators are evaluated during the course of program compilation and the new code produced is incorporated into the program being compiled. This technique could be viewed as a sophisticated form of macro expansion, where the language used to evaluate the macro is the same as the programming language itself. With **run time linguistic reflection**, the generators are evaluated during program execution and the new code produced is compiled and executed in the same context.

16. **Class** is a description of a group of objects with similar properties, common behavior, common relationship, and common semantics.

17. **Association** is a reference from one class to another.

18. **Attributes** are the properties of individual objects, and not the objects themselves.