2001

Execution-based retrieval of object-oriented classes: An improved method.

Shaochun. Xu
University of Windsor

Execution-Based Retrieval of Object-Oriented Classes:
An Improved Method

By

Shaochun Xu

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2001

Abstract

Software reuse has become a topic of much interest in the software community due to its potential benefits, which include increased product quality and productivity and decreased product cost and time. The software reusability can be enhanced by the Object-Oriented approach. The potential problem in software reuse is to find an effective and efficient way to retrieve the candidate components from the library.

An improved methodology of retrieval by execution on object-oriented (OO) classes is proposed in this thesis. The system allows users to enter the data on the constructor, observer and modifier in order. The system then organizes them into a test program and executes the classes from the selected library. Finally, the system returns to the user a list of candidates according to the matching number of methods and constructors. The user does not need to take care of the argument order, and the system handles each case. This proposed method is the first retrieval by execution that works on OO classes and discloses the complete class behavior. Characteristics of OO components such as information hiding, inheritance, overloading and overriding are fully considered. Compared with the previous execution-based retrieval method, this method greatly improves the retrieval precision, recall and efficiency.

A prototype system, called EBCRS, is developed using HTML, JavaScript, Applet and Servlets. This system could be used to retrieve, browse and save the Java classes from the class library. It also allows the administrator to manage the class library such as adding to and deleting from the class library. This system is Internet and Intranet ready.

Dedication

This thesis is dedicated to my wife, Lan Li, and my daughter, Ying Xu, for their love, understanding and support.

Acknowledgements

First of all, I would like to thank Dr. Y. Park for dedicated instruction and advice on both the courses and the thesis. Many valuable discussions with him and his excellent suggestions have not only significantly contributed to the quality of this thesis, but have also benefited my future endeavors. Dr. Park's excellent teaching and supervising ability on research impressed me very much. Without his supervision, this thesis would not be so valuable.

Special thanks to Dr. L. Li, my internal reader, for providing many suggestions and comments to improve the quality of this thesis.

I also would like to thank Dr. S. Suh, my external reader, who showed immense interest in my work, offered many comments on my work, which contributed to many improvements.

Finally, I acknowledge the prompt responses from Sun Microsystems Inc. with my questions about the Java language.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Concept of Software Reuse

Software reuse, broadly defined, is the practice of using existing software components in more than one system. In other words, it is the process of creating software systems from predefined software components (Standish, 1984). Software reuse has two sides: (1) the systematic development of reusable components and (2) the systematic reuse of these components as building blocks to create new systems (Prieto-Diaz, 1993). Software specifications, designs, test cases, data, prototypes, plans, documentation, frameworks, code and templates are all candidates for reuse (Prieto-Diaz, 1993; Zaremski and Wing, 1995; Lee et al., 1999).

Reuse makes sense because the similarity found across software systems is enormous and undeniable. When we compare software systems, we usually find 60-70% commonality from one application to another (Jones, 1984; Mili et al., 1995; McClure, 1995). This includes code, design, functional and architectural similarities. At all levels of development from requirement specifications to code, there are components that by the nature of implementing tasks and representing information on a computer must appear over and over again in software applications. New technologies such as object-orientation, software automation and client-server application do not change this; however, they do make it easier to take advantage of software similarities (Krueger, 1992; Cheng, 1993; Lee et al., 1999). Some software similarities can be predefined and built into software tools (such as reusable code patterns in generators); others can be created as reusable components that are stored in software reuse libraries. The potential for reuse is enormous since the majority of each new application could be assembled from reusable components if the appropriate components could be predetermined and built prior to system development.

The main advantages of software reuse are listed as follows (Mili et al., 1995; Aranow, 1997):

- Increase software productivity and shorten software development time: Because software reuse uses existing, tested and documented software components to develop the new system, programmers need less work and the productivity is highly improved. It is no question that the time of software development is also shortened.
- Improve software system interoperability: The tested components are high quality; thus the new system built with those components should have better interoperability.
- Reduce software development and maintenance costs: As we reuse the existing components, no coding (or less coding) is needed. A few testing cases are required. Therefore, the cost will be largely reduced.
- Produce more standardized, better quality and reliability software: Because the reusable components have been tested and proved in the real-world system, the accumulated defect fixes result in higher quality components. These components are more reliable than new components (Lim, 1994). Each time the reusable components are reused, they are debugged and tested.

1.2 Object-Oriented Programming and Software Reuse

In the procedural programming environment, the common modules of software components are procedures and functions. They are difficult to be reused, because they are less likely matched with the requirements of a new application (Cheng, 1993).

Object-Oriented Programming (OOP) has often been promoted on the basis of its reusability merits (Prieto-Diaz, 1993; Liao and Wang, 1993; Chou et al., 1996). Meyer (1988) used "Open-closed principle" to explain the OOP and reuse. According to this principle, a module should be available for further extension; once a module is tested and accepted into a library, it should be able to be used without being "open up". In the procedural programming environment, there are no components, which are both open and closed. However, classes in OOP are such modules being open and closed. Encapsulation makes classes closed and inheritance makes classes open.

Cheng (1993) discussed in detail about the relationship between OOP and reuse. In OOP, encapsulation and inheritance play an important role in software reuse. Encapsulation or information hiding defines a data structure and a group of methods to access (Booch, 1994). It makes a class as black box, which can easily be shipped to other applications. Inheritance is a mechanism for representing similarity among classes. It has been widely recognized as an important mechanism for constructing new reusable software components from existing components. In real world, a software component is rarely exactly suitable for a new application. So, it is necessary to adapt the component in other way. With inheritance, this adaptation can be done easily by extension/specialization. Otherwise, the adaptation can be done only by modification.

Composition is also one of the characteristics of OOP. The most common way that two different entities can be statically structured together is by composition, where one object is used as a component part of the other. This facility thus makes further use of the reusability of objects and classes (Biddle and Tempero, 1995).

Overloading and polymorphism can allow us to create a generic code and use in different cases with different types. It enables users to design and implement components or system that are easily re-implemented and extended (Nelson and Poulis, 1995).

There are many different ways for OOP reuse: A helper class for a class library was built to let the user specify keywords to search the library for a match (McManis, 1996).

Damiani et al. (1997) described a descriptor-based approach to OO code reuse. In their approach, an application engineer maintains the system for an audience of users or application developers. COOR provides tools for the classification and search for reuse candidates. The experience with COOR suggests that descriptors can provide several advantages over class interfaces. COOR descriptors support different levels of component classification granularity (like application frameworks rather than single class).

Etzkorn and Davis (1997) introduced the Patricia (Program Analysis Tool for Reuse) system, developed at the University of Alabama, which is used to automatically identify reusable components in OO code. To understand a program, Patricia uses a unique heuristic approach deriving information from the linguistic aspects of comments and identifiers and from other nonlinguistic aspects of OO code such as class hierarchy.

Although OOP was considered with its great potentiality in software reuse (Cheng, 1993), Honiden thought that OO analysis is not fully mature because its specification process has never been described in detail (Honiden, 1993). By the way, OOP also has some side effects in software reuse. The use of polymorphism and inheritance introduces a large number of dependencies between components. Dynamic binding increases the number of implementations to be examined, and the dispersion of functionality into different components makes global understanding difficult (Gonzalez and Fernandez, 1997).

1.3 Component Retrieval Approaches

The growth of reuse and the advent of software repository have led to the design of mechanisms to retrieve reusable components. The purpose of component retrieval is concerned with finding component behaviors that solve a problem described by a query.

There are many ways to retrieve components from the reuse library such as browsers, semantics-based, keyword-based, type-based, specification based, composition-based, and execution-based retrieval (Steigerwald, 1992; Mili et al., 1995). Many classification schemes also exist in the literature. As retrieval methods largely depend on the library organization and also depend on the matching a candidate component against a user query, so it is often and reasonable to classify the retrieval methods according to the query representation/ and or the library structure.

An elegant classification on the existing library organization/retrieval methods has been proposed recently (Mili et al., 1998). Mili et al. (1998) classified all the retrieval methods into 6 categories:

- Information retrieval method
- Descriptive method
- Operational semantics method
- Denotational semantics method
- Topological method
- Structural method

Information retrieval method

The executable software components can be seen as documents containing information as the same way as the books or hypertext documents. The information retrieval methods act as the natural library and the stored information focuses on textual information represented in natural language (Maarek et al., 1991). The difficulty of this method is how to formulate the information of each component in order to get most of the relevant documents for each query.

One of the examples is the GURU project conducted by Maarek et al. (1991) and Helm and Maarek (1991). GURU automatically assembles large components by using information retrieval techniques. The construction of their library consists of two steps: first, attributes are automatically extracted from natural language documentation by using an indexing scheme based on the notions of lexical affinities and quantity of information; then a hierarchy for browsing is automatically generated using a clustering technique which draws only on the information provided by the attributes. Three phases, query specification, linear retrieval and browsing make the retrieval process. First, the user formulates a query based on the authorized vocabulary, then the system identifies all the components that match the query, and finally, the user can browse the library to search for the relevant components.

Descriptive method

Descriptive methods are characterized by matching a keyword-based query against assets that are represented by structured lists of descriptive keywords. A component is selected whenever the keywords that form its representation match all (or most) of the keywords that form the query. Because it is easy to understand, it has been widely used at present. The library for this method is usually organized by manual indexing. The administrator defines the abstraction of the component that adequately and concisely describes the component and provides the descriptors. A conceptual separation between the library components and its representation is present. Due to the difficulty of maintaining consistency and accuracy, pure keyword based approaches are scarcely used and only limited to a small size of library (Mili et al., 1995).

The faceted approach could see an improvement over keyword method, which is widely discussed in the literature. Prieto-Diaz first introduced this approach by defining a multi-dimensional search space with each dimension referred to as facet made up of a set of predefined keywords (Prieto-Diaz, 1991, 1992). For example, function, object type and system-type can be treated as facets. For each facet, a set of keywords is given for function such as add, append and close.

Operational semantic method

Operational semantic methods use the executability of software components to select the candidate components from the library. The principle of behavioral retrieval was first suggested by Podgurski and Pierce (1992): a retrieval technique that generalizes the simple idea of executing each component and test input, reporting those that compute correct output. The execution-based retrieval method can generally offer accurate components and is easy to implement. It also guarantees that the retrieval components behave the same or close to the user's expectation. Their method was improved and extended by Hall (1993).

The idea of using a lattice structure as the basis of behavioral retrieval has been suggested by Mili et al. (1994). The lattice idea of Mili et al. (1994) and the behavior-sampling ideal of Podgurski and Pierce were combined by Atkinson and Duke (1995), who provided a methodology of behavior retrieval on OO classes, but they did not provide an implementation. According to Atkinson and Duke (1995), the behavior of class is represented by an input/output pair where the input is a sequence of incoming messages and the output is the corresponding sequence of responses.

Niu and Park (1999) extended the methodology of Atkinson and Duke (1995) and proposed a relatively effective and accurate method for class retrieval. They also implemented a prototype.

Denotational semantics method

The components in the library of these methods can be represented at various levels of abstraction including functional signatures, functional abstractions and requirements frameworks. Specification-based retrieval and signature-based retrieval are included in this category. Signature-based retrieval can be seen as a specific case of specification-based retrieval.

Signature matching is the process of determining which library component matches a query signature. Signature matching takes advantage of information about program modules that is essentially free. The signature of a function is simply its type.

One of the examples is a method proposed by Rittri (1990) for software component storage and retrieval based on signature matching in a function language. The matching criterion is using polymorphic type systems and provides independence of the order of components in a type.

Zaremski and Wing (1995) described various applications of signature matching as a tool for using software library such as in Standard ML. They gave definitions for a variety of

matches at both the function and the module levels. A signature is used as a key to find a set of reusable components in their retrieval methods. A hierarchy of matching criteria is defined and discussed. The basic matching condition provides for equality between the two signatures, module variable renaming and parameter ordering.

Specification matching goes beyond signature matching. It is not only based on the function type, but also on the behavior. In other words, signature describes a component's type information and specification describes the component's dynamic behavior. For example, Jeng and Cheng (1992) discussed an organization of a software library that is based on formal specification of components and queries. They organized the library into a two-layered hierarchy by means of a clustering algorithm, where the top layer stores related software components. They applied two steps retrieval processes: first performing a coarse-grained search to identify a cluster and then performing a fine-grained search on the selected cluster. They extended this method by defining matching criteria between components and between methods (Jeng and Cheng, 1992, 1995). The matching criteria make provisions for sub-typing, variable renaming and parameter permutation.

The advantages of using formal specification for component retrieval are that they are free from ambiguity and they are subject to stronger forms of transformation than other specification methods (Wing, 1990). Based on the performance, query by formal specification can offer increased precision over keyword and multi-attribute approaches. However, it is difficult to implement and processing times for the search algorithms may be excessive depending on the approach taken (Steigerwald, 1992).

It is worthy to point out that any of above classifications could not draw clear boundary between two methods. Some retrieval methods could be put in either of categories. Moreover, a combination of two methods is now being used.

Helm and Maarek (1991) used formal specification as a medium to describe the components in class and used information retrieval method to browse the reusable components in their work.

Isakowitz and Kauffman (1996) proposed an approach to automatically classify components using faceted and but retrieve by hypertext. ORCA utilizes a faceted classification approach that can be implemented using hypertext. AMHYRST can automatically create hypertext networks that represents and link objects in term of a number of distinguishing features.

Signature matching could be used for first stage of retrieval, which returns a subset of components from the library against a particular query, i .e. it is a filter for another tool. Thus, signature matching can be seen as a complementary approach to other retrieval techniques.

Fischer and Struckmann (1995) combined the signature matching and specification method together to implement a VCR VDM–based component retrieval tool. A preprocessing phase utilizes signature matching to filter promising candidates out of component library. A specification matching phase builds proof obligations from the specifications of keys and candidates and feeds them into a theorem prover. Validated obligations denote matching components.

1.4 Overview of the Thesis

1.4.1 Motivation

As discussed above, Niu and Park (1999) proposed a new method of behavior retrieval, which is an extension of Atkinson and Duke's (1995) methodology. Although the behavior retrieval method has been proposed since 1992 (Podgurski and Pierce, 1992), this was the first time that this methodology has been applied to the Object-Oriented Environment with a full implementation. Their method generally improves the precision of retrieval in terms of the behavior of classes. One of the most important improvements is considering the hierarchy of classes. This method also allows repeated retrieval and provides two matching algorithms. However, some shortcomings still exist.

```
public class ObjectEquals {
        public boolean equals (Object arg1, Object arg2)
        {
                return arg1.equals (arg2);
        }
        public boolean notEquals (Object arg1, Object arg2)
        {
                return !arg1.equals(arg2);
        }
        public boolean identical (Object arg1, Object arg2)
        {
                return arg1==arg2;
        }
        public boolean notIdentical (Object arg1, Object arg2)
        {
                return !(arg1==arg2);
        }
}
```

Figure 1.1: A simple class in Niu and Park's library

1. **Real "class"**: Although their methodology is used on the Java class library, in fact, their class component library does not differ from the function component library. All the classes in the library only perform mathematical calculations, string manipulations or object comparisons (Figure 1.1). They are different from "real-world" classes that should perform some operations on one or more attributes rather than only perform calculation on the passed arguments.

2. **Uncaptureable behavior**: An OO class consists of attributes, constructors, modifier methods and observer methods within its scope. Their retrieval method only executes on those methods whose functionality is the same as the functions in a procedural language (those methods with return value); it will not perform actions on those methods whose functionality is similar to that of procedures (i. e. constructor and observer methods in OO languages). Thus, their retrieval method does not capture the complete behavior of the classes. To test a class, all the member functions including constructors, observer methods and modifier methods declared for a "class" should be tested.

3. **Default constructor:** Their method exclusively uses the default constructor of a class to dynamically create the instance of each class and then perform the execution based on the test program. However, classes usually have user-defined constructors and the compiler may not be allowed to provide the default. In this case, even if a class matches the query, its methods may not be called because no instance exists. Thus, this class will be skipped and the recall will be reduced.

4. **The order of arguments**: Their methodology requires the user to supply the correct order of arguments when performing the retrieval. Even if there is a class whose behavior is similar to or the same as the user's expected one; it may not be retrieved because the order of arguments of one or several methods is different from the user's input. Thus, the retrieval recall could be low.

5. **Time complexity and cost**: Their retrieval method loads each class from the library every time even when the class has been tested (loaded) for the previous message. Thus, the system takes a lot of time to unnecessarily reload classes from the library each time, thus the cost is high. If a method of a class does not match the input data type, this method or even this class may not need to be reloaded and tested.

1.4.2 Objective of the Thesis

The objective of this thesis is to develop an improved execution-based retrieval methodology. The most critical improvements are summarized as follows:

1. The proposed methodology fully considers the characteristics of OO classes. An entity of a class encapsulates all the data and method implementation and provides all the necessary interfaces. All the methods are used to update or inspect the attributes in the object (the states of the object). The components of our retrieval methodology are "real-world" Java classes.
2. The interface of a class includes constructors, modifier methods and observer methods. The complete behavior of a class should include all the responses from all

the class methods when a test program is sent to the class and executed on these class methods. In our method, the user is allowed to input a test program including the messages on constructors, observers and modifiers. The user may input several test programs that will improve the retrieval precision.

3. The user does not need to consider the order of arguments for each test message. The system will evaluate each case of different orders and then perform testing and executing on the corresponding classes. This technique will greatly increase the retrieval recall.
4. The signature (return type and parameter list) of each method is very important during execution. If this signature does not match, then no further loading and testing on the class is necessary. When the class is loaded into system from the class library for the first time, the return type and parameter list of each method is stored into a dynamic data structure. Before executing the testing messages, the type match checking is performed by comparing those of the test messages and those of methods in the class stored in the system. If the types match, then the system will load the class and execute them. Otherwise, this method or the class will be skipped. This improvement will reduce the time complexity, and thus increase the retrieval efficiency.
5. The user will have the choices to define his/her own expected constructors to create the object using these constructors and to send the messages using this object. The user does not solely depend on the default constructor to create object and then perform retrieval. If the class in the library does not have a default constructor, it can still be executed and compared. This also greatly improves the retrieval recall.
6. An Internet-ready and HTML (JavaScript-Applet)-Servlets client/server web application is implemented. HTML-JavaScript-Applet can provide a user-friendly interface and servlets are taking the place of CGI in the server side. Such application will increase the communication speed between the client and the server and also the security of transformation. It is easy to put the whole system on the Internet. Servlets can establish network connections without the sandbox problems. Servlets can write to files by accessing the client machine. Since servlets can handle multiple requests

concurrently, the requests can be synchronized with each other to support multi-user application.

1.4.3 Organization of the Thesis

This thesis is organized into four chapters. The brief description of each chapter is as follows:

Chapter 1 gives a brief introduction of software reuse, advantages, the objects of software reuse, the component retrieval techniques and the overview of this thesis.

Chapter 2 discusses the general idea of execution-based retrieval. An improved methodology is proposed and discussed in detail.

Chapter 3 describes a prototype of execution-based retrieval based on the improved methodology and implemented using HTML/JavaScript/Applet and Java Servlets. The distributed servlets client/server application is discussed. UML is used to describe the functionality of the system. All the components of a prototype system are systematically described in detail. Screen shots of important GUIs of both client and server sides are provided. The system security is also discussed.

Chapter 4 compares the proposed methodology with the previous methods. It also covers the remaining problems and possible future work in this area.

Chapter 2 An Improved Execution-Based Retrieval

2.1 Concept of Execution-Based Retrieval

Execution-based retrieval executes each component on the test data and compares the results with desired output. The other terms such as Operational Method (Mili et al., 1998); Behavior Retrieval (Podgurski and Pierce, 1992, 1993) and General-Behavior-Based Retrieval (Hall, 1993) are used. Obviously, these methods use the executability of software components as a basis for the selection of candidate components from a software library.

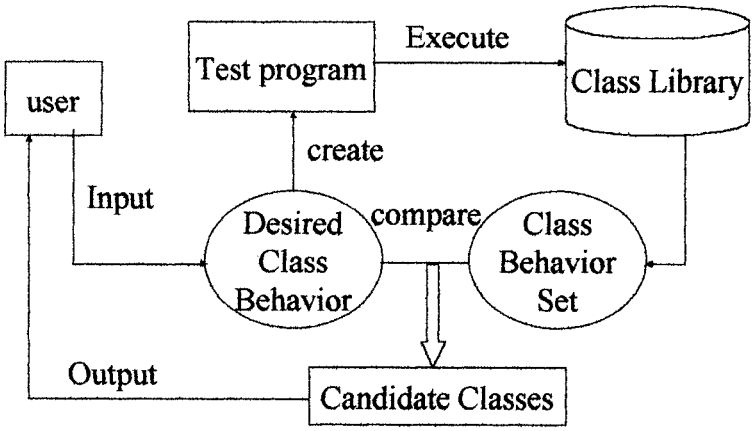


Figure 2.1: Execution-based retrieval

The general algorithm of class retrieval by execution is shown in Figure 2.1 (Atkinson and Duke, 1996; Niu and Park, 1999). Programs are executed using components and the responses are recorded. Retrieval is achieved by selecting those components whose responses are closest to a predetermined set of desired responses. In detail, the user first inputs some data about the desired components. The system organizes the test data into a test program and then executes each message of the test program on each class in the library to find the match information. The system returns all the related components. It then compares the retrieval components behavior with the user's desired one and chooses the closest components and then returns the result to the user.

Execution-based retrieval has the following characteristics (Mili et al., 1998):

- Component: These methods only apply to segment of code such as functions in a procedural language and classes in the OO language, because they need the executability of the software component to determine if it matches the query.
- Scope of Library and Storage Structure: All the proposed libraries for execution-based retrieval are on relatively small scale because of limitation of execution time. The previous methods usually do not pay attention to the library organization. The components are stored in the flat file without any structure.
- Query Representation: The query is composed of a sample of input data.
- Retrieval Goal and Matching Criterion: Some proposals aim to retrieve all the components exactly matching the requirements (Podgurski and Pierce, 1992; Hall, 1993). New versions also try to retrieve approximate matching (Atkinson and Duke, 1995; Niu and Park, 1999). All the methods compare the behavior of candidate components with the expected behavior as described by the user.

2.2 Review of the Previous Execution-Based Retrieval Methodologies

Behavior retrieval was first proposed by Podgurski and Pierce (1992) for function components. They observed that a software component could be uniquely identified within a large software library on the basis of its behavior on a few selected sample inputs. Basic behavior sampling identifies relevant components by executing candidates on a user-supplied sample of operational inputs and by comparing their output to the output provided by the searcher. Extensions to basic behavior sampling were also proposed to improve its recall and to make it applicable to the retrieval of abstract data type and object classes. But their behavioral sampling methodology may not collect all the possible execution responses but rather than randomly select the responses over a number of executions and exercised the most commonly used operations based on a probability distribution. Their method was improved and extended by Hall (1993). Hall overwhelmed the shortcomings of behavior sampling by making the provision in the

matching condition, letting the user to select the sample input and controlling the executing time and the termination of the component invoked. Hall's methodology also works only on functional components.

The idea of using a lattice structure as the basis of behavioral retrieval for the function components has been suggested by Mili et al. (1994). In their retrieval system the nodes of the lattice were relations serving as surrogates for specifications. The lattice idea of Mili et al. (1994) and the behavior-sampling ideal of Podgurski and Pierce were combined by Atkinson and Duke (1995). They provided a methodology of behavior retrieval on OO classes, but did not provide an implementation. According to Atkinson and Duke (1995), the behavior of a class is represented by an input/output pair, where the input is a sequence of incoming messages and the output is the corresponding sequence of responses. From this formal method they derived a partial ordering on class behaviors and discussed the lattice properties that join only under compatibility condition.

Niu and Park (1999) extended the methodology of Atkinson and Duke (1995) and proposed a relatively effective, practical and accurate method for class retrieval. Their main methodology is as follows: the user first inputs some data about the desired components. The system organizes the test data into a test program and then executes each message of the test program on each class in the library to find the match information. It then compares the retrieved components behavior with the user's desired one and chooses the closest components and then returns the result to the user. The most important improvement over Atkinson and Duke's methodology is that they considered the hierarchy of class. Their method supports repeatable retrieval and provides two different matching algorithms. They also implemented a prototype to prove their methodology that is open to Internet and allows user to update the class library. However, their methodology has some problems as described in the chapter 1.

2.3 Organizing the Input Program

2.3.1 Class, Object and Class behavior

We assume that the Object-Oriented classes (Java classes) are the components in our library. We also assume the classical definition of classes and objects from the OO literature.

Class: A class is a blueprint that defines the variables and the methods common to all objects of a certain kind.

A class definition specifies (a) the name of the class; (b) the kind of the class; (c) inheritance for the class; (d) fields that each object of the class will contain; (e) methods that will provide services for objects in the class and (f) constructors for objects of the class.

A class is either a concrete class or an abstract class. Abstract class cannot be instantialized and it may have some abstract methods that are not implemented. As its behavior cannot be fully defined, abstract class will not be considered as a candidate class although it could be stored in the library for its subclass's extension.

Fields, methods and constructors can appear in any order within a class definition.

Field: A field in a class defines a data item that exists in each instance of the class. In Java, there are two types of attributes: instance variable and class variable. Class variable is defined by keyword static and shared by all the instances. Class variable acts as global variable. In our method, we will not consider the static variables, as it is difficult for the user to control the state of static variables.

Constructor: A constructor defines actions that are performed when an object is created. A constructor returns a new object of the class when it is called.

A class definition can contain zero or more constructors. A constructor can have zero or more parameters. A predefined constructor called default constructor is offered by the compiler when no constructor is implemented by the programmer.

Method: A method is a procedure or a function that provides a service (or operation) for the objects of a class. A method is activated for a specific object of the class.

Methods are declared within the definition of a class and are marked with access modifiers such as public, protected, default and private.

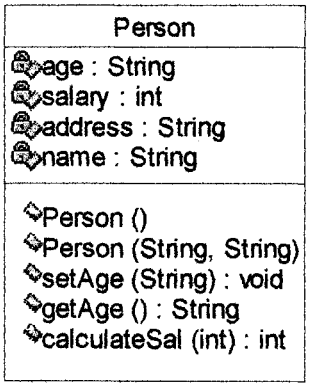


Figure 2.2: Class diagram of the class Person

A method may have input parameters and return type. A typical signature of a Java method might look like Figure 2.2.

Among methods, there are two types: observer method and modifier method. A modifier method changes the state of the object and modifies the existing class instance (i. e. the value of one or more of the data attributes in the object). An observer method inspects the object and returns a value characterizing one or more of its state attributes. It does not accept any parameter. The main difference between observer method and modifier method is that observer method does not cause side effects on the object states.

Among the Person class, Person (), Person (String) are two constructors, setAge (String) and calculateSal (int) are modifier methods and getAge () is an observer method.

In general, class definition contains two parts: an interface including a list of operations (methods) that can be performed by the instances of the class and a body consisting of the implementation of the operations and the data attributes of the class for an instance. In other words, a class is composed of a set of variables (instance or class), one or more constructors (including default one), zero or more observer methods and zero or more modifier methods. In general, a class can be expressed as follows:
Class N = ({F}, {C}, {O}, {M}) where F: fields in the class N; C: constructors in the class N; O: observers in the class N and M: modifiers in the class N.

```
public class ObjectEquals {
        public boolean equals (Object arg1, Object arg2)
        {
                return arg1.equals (arg2);
        }
        public boolean notEquals (Object arg1, Object arg2)
        {
                return !arg1.equals(arg2);
        }
        public boolean identical (Object arg1, Object arg2)
        {
                return arg1==arg2;
        }
        public boolean notIdentical (Object arg1, Object arg2)
        {
                return !(arg1==arg2);
        }
}
```

Figure 2.3: The class ObjectEquals from Niu and Park's class library

Although Niu and Park's methodology is used on the class library, in fact, their class component library is not different from a function component library. All their classes in the library only perform mathematical calculation, string manipulation or object comparison (Figure 2.3). No attributes, constructors and observer methods are defined in the classes. Therefore, they are different from "real-world" classes that should perform

some operations on one or more attributes. The retrieval methods on OO classes should also perform actions on those "real world" classes.

Types	Return		Parameters	
	Type	value	Type	value
Constructors	none	none	any	any
Observers	Any (except void)	Any (except none)	none	none
Modifiers	Any (include none)	Any (include none)	any	any

Table 2.1: The comparison among constructor, observer and modifier

The differences among constructors, observer methods and modifier methods can be listed as follows (Table 2.1):

- Constructors have no return type (even void type), but may have 0 or more parameters.
- Observer methods have return type (void excluded), but without parameters.
- Modifier methods may have return type (void included), and usually have 1 or more parameters.

Object: An object is a combination of instance variables for the object's state and methods that implement the object's behavior. In other words, an object is a software bundle of variables and related methods.

Software objects are modeled after real world objects in that they also have states and behavior. A software object maintains its state in one or more variables and implements its behavior with methods. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. An object has a public interface that other objects can use to communicate with it.

Class behavior: The behavior of a class is represented by an input/output pair. The input is a sequence of incoming messages and the output is the corresponding sequence of responses.

If a retrieval technique cannot access whole the public interface of a class, that method will not handle a complete coverage of the class's behavior. For example, in the class Person shown in Figure 2.2, there are two constructors and three methods. If we only execute the test program on the method calculateSal (), the complete behavior of this class will not be disclosed.

Niu and Park's retrieval method only executes on those methods whose functionality is the same as the functions in a procedural language (those modifier methods with return value), it will not perform actions on those methods whose functionality is similar to that of procedures (i. e. constructor and observer methods in OO languages). Thus, their retrieval method does not capture the complete behavior of a class. To test a class, all the member functions including constructors, observer methods and modifier methods declared for the class should be tested.

In order to disclose the class behavior, at least one object of class needs to be created and then the message can be sent through this object. The previous method exclusively depends on the default constructor of the class to create the instance of each class and then performs execution based on the test program. However, classes usually have user-defined constructors and the compiler may not be allowed to provide the default one. In this case, even this class matches the query, its methods may not be called because no instance exists. Thus this class will be skipped and the recall will be reduced. In an OO class retrieval method, the user should be allowed to construct his/her own expected instance of the class and then use this instance to pass the messages for execution.

2.3.2 Messages and Test Program

Message: In OOP, the message refers to communication between objects. Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B.

We employ the message definition of Niu (1999) for our retrieval methodology. Message is the name of the operation and a set of possible input (including type and value), the output (type and value) associated with this operation and the user expected output (value and type).

The message always includes two parts. The first part may consist of one to many pairs of input type and value and the second part, which is the last pair in the set, is the user desired return type and value.

For example, consider the following messages:

- Message 1 {(int, 5), (none, none)}: is a message on the constructor which consists of one argument (type =int, value=5) and without return type and return value.
- Message 2 {(int, 2), (boolean, true), (void, none)}: is a message on the void-returned modifier method. It is composed of two pairs of arguments (type=int, value =2; type=boolean, value=true).
- Message 3 {(none, none), (int, 2)}: is a message on the observer method that only consists of the user-desired return type (int) and return value (2).
- Message 4 {(String, "Jerry"), (int, 10), (char, 'p'), (String, "done")}: is a message on the non-void returned modifier method. It is composed of three pairs of arguments (type=String, value="Jerry"; type=int, value=10; type=char, value ='p') and the user-desired return type (String) and value ("done").

It is obvious that the type of methods (constructor, observer and modifier) can be recognized by the messages input by the user. Based on these, the system can select corresponding constructors and methods in the class to execute.

Test Program: Test program is a set of messages to be sent to the system. A class is composed of constructors, observer methods and modifier methods. Therefore, in order to have a complete behavior of the class, a wide range of test messages are needed to compose a test program.

In our retrieval method, we can let users to input various messages on all the interfaces of a class including constructors, modifier methods and observer methods. The system then executes them according to the order of the input and employs the corresponding instance to modify or inspect the states of objects.

2.3.3 State Inspection

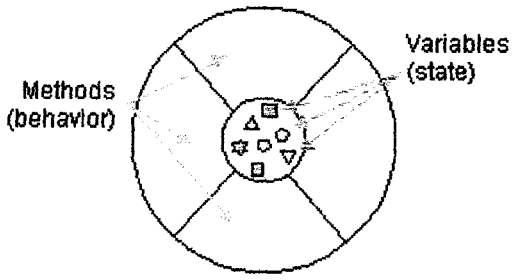


Figure 2.4: The states and methods of an object

An instance or object invokes an operation and sends a message to the object. Every object has a state that can be characterized either by its history of method invocation or the current values of its attributes (Figure 2.4). In other words, the instance variables for an object directly represent the state of the object and an object of class changes states when one or more of methods are called on one or more attributes. However, an instance must be created before it can be updated or deleted.

The previous retrieval methods only perform an incomplete execution on the class: only methods whose functionality is similar to that of functions (i. e., modifier methods with return type in OOE) are called, and those methods whose functionality is similar to that of procedures (observer methods and constructors in OOE) are not executed.

In order to disclose the complete behavior of a class, we need to create an instance and then call the methods to update the states of the instance and inspect each state to find whether it is in the state as expected. Therefore, the following order should be followed:

1. Execute one or more constructors to produce some instances with initial states; if no, a default constructor method is executed.
2. Execute zero or more modifier methods to modify the states of the instances.
3. Apply observer methods to the instance. Each observer should return a result characterizing some state attributes.
4. The results returned by the observer methods are compared with user-expected result.

If the modifier method has return type rather than void type, then a modified order is described below:

1. Execute one or more constructors to produce some instances with initial states; if no, a default constructor method is executed.
2. Execute 1 or more modifier methods to modify the states of the instance.
3. The results returned by the modifier method whose return type is not void are compared with user-expected result.

Let C be a constructor, M be a modifier method and O be an observer method. Then, the retrieval algorithm should be formulated by the following regular expression: CM*O+ or CM+

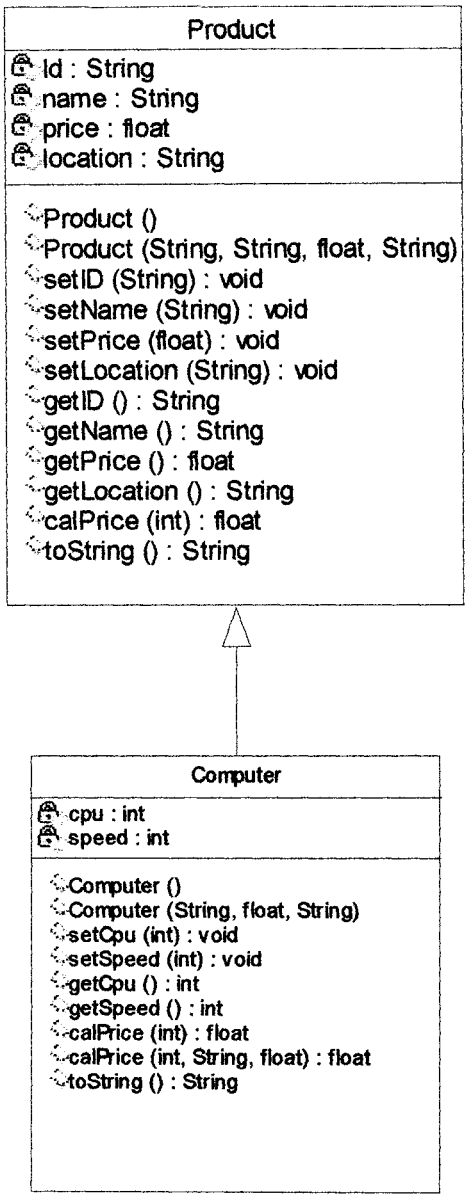


Figure 2.5: Class diagrams of the class Product and the class Computer

For example, the class Product has four attributes (Figure 2.5), two constructors and 10 methods, including 5 observer methods (getID(), getName(), getPrice(), getLocation() and toString()) and 5 modifier methods (setID(String), setName(String), setPrice(float), setLocation(String) and calPrice(int)).

If we want to retrieve a class whose behavior is the same as or close to the class Product, then an ideal test program should be at least as follows:

- Message 1 {(void, null), (none, none)} - create an instance of class Product using non-argument constructors
- Message 2 {(String, "Table"), (void, null)} - modify the state of the instance using modifiers
- Message 3 {(void, null), (String, "Table")} - observe the state of the instance using observers
- Message 4 {(String, "123"), (String, "Chair"), (float, 13.5), (String, "Windsor"), (none, none)}- use a new constructor to create another instance of the class Product
- Message 5 {(int, 1.5), (float, 20.25)}- get a value during changing the state of instance

After execution of the above test program, the class behavior can be obtained as Behavior ={(none, none), (void, null), (String, "Table"), (none, none), (float, 20.25)}.

It should be noted that the user might create different objects (instances) of one class and pass different messages to these objects. In this case, it is important for the system to maintain the order of messages, execute them in that order and inspect various states of the objects. Otherwise, the result will not be presented as the user's expectation.

For example, users may enter the following messages in order:

- Message 1 {(void, null), (none, none)} - create an instance of class Product using non-argument constructors
- Message 2 {(String, "Table"), (void, null)} - modify the state of the instance using modifiers
- Message 3 {(void, null), (String, "Table")}- observe the state of the instance using observers

- Message 4 {(String, "Chair"), (String, "125"), (double, "Chair"), (String, "Windsor") (none, none)} - create a new instance of the class Product using argument constructors
- Message 5 {(void, null), (String, "Chair")}- observe the state of the instance using observers

The expected behavior is {none, null, "Table", none, "Chair"} rather than {none, null, "Table", none, "Table"}. Of course, such methodology may allow many test programs to be entered in the form of the following regular expression: {CM*O+|CM+}*

2.4 Executing the Test Program

After the system organizes the input messages into a test program, it begins to execute the test program against all the classes from a class library.

2.4.1 Type Checking and Run-Time Storage

The previous execution-based retrieval method loads the class from the library every time even when the class has been loaded and tested for the previous messages. Thus, it wastes a lot of time to reload classes and the cost is high. If a method of a class does not match the input data type, then this method or even this class may not need to be reloaded.

The signature of each method is very important during execution. If this signature (return type and parameter list) does not match the user input, then no further loading and testing is necessary. When the class is loaded into system from a class library for the first time, this information for each method can be stored into a dynamic data structure such as vector or array. Before executing test messages, the type match checking is performed against that information of methods in class stored in the system. If types match, then the system will load the class and execute them. Otherwise, this method or this class will be skipped. This improvement will reduce the execution time and increase the system efficiency.

Suppose f(t) =({constructors_test_data (arglist)}, {methods_test_data (arglist, rt, rv)}) where

- f(t) : a test program composed of a list of test messages (constructors and methods)
- constructors_test_data (arglist): an order of test messages on constructors (arglist: list of arguments for each constructor)
- methods_test_data (arglist, rt, rv): a list of test messages on methods (arglist: argument list, rt: return type, rv: return value, if the return type is void, then the return value is empty "").

Given a class in the library, the interfaces could be represented as follows:

F(c) = ({constructors_in_class (paramlist)}, {methods_in_class (paramlist, ret)}) where

- F(c): The interfaces of the class from the library
- constructors_in_class (paramlist): constructor list in the class (paramlist: list of parameters for each constructor)
- methods_in_class (paramlist, rt): method list in the class (paramlist: parameter list, ret: return type).

The type-matching algorithm is as follows:

For each constructor_test_data (arglist) in the f(t)
If (this is the first constructors_test_data)
Load constructors_in_class (paramlist) for each class from the library, store the paramlist in a vector
If (arglist==paramlist in the vector)
Execution on the constructor
Else skip

For each methods_test_data (arglist, rt, rv) in the f(t)
If (this is the first methods_test_data)
Load methods_in_class (paramlist, ret) for each class from the library, stored the paramlist and ret in an array

If (arglist==paramlist in the vector && rt==ret)

If (there is no instance of the class)

Call the default constructor

Execution on the method

Store the value

Else skip

If the method information is not stored, the class needs to be reloaded from the library during executing each message. Thus the system will be wasted for reloading the classes. On the other hand, the type checking is necessary for execution-based retrieval. Otherwise, a run-time exception will be thrown.

2.4.2 Class Retrieval Regardless of Argument Order

Some retrieval methodologies require the user to supply the correct order of arguments during performing class retrieval. Even if there is a class whose behavior is similar to or the same as the user's expected one, this class may not be retrieved as a candidate because of the order of arguments of one or more methods different from the user's input. Thus, the retrieval recall could be low. In fact, the user usually does not know the argument order of a method ahead or he does not even care about it. The most important criterion for the component retrieval is the whole behavior of a component (class).

Once the user inputs each test message, the system stores it into a test program. Each case of different orders for every test message will be evaluated. The system then performs testing and executing on each method of the corresponding class.

The algorithm of the execution part is as follows:

For each composition of arglist

If (arglist==paramlist in the vector)

execution on the constructor or method

Else skip

This technique will obviously increase the retrieval recall.

For the class Computer in Figure 2.3, if the test program contains one message:

Message 1 {(String, "Pantium"), (int, 1), (float, 1.5), (float, 1358.7)}, the system can determine that this is a message for the modifier method by checking all the pairs. The system also counts out 3 arguments and stores them in a vector. The system then finds that there are 6 compositions of the argument orders:

(String, int, float)

(String, float, int)

(int, String, float)

(int, float, String)

(float, String, int)

(float, int, String)

The system will evaluate all the cases recursively.

2.4.3 Matching Process and Candidate Classes

A matching between a message and a class method is established when executing the message on the class method, the same output or similar output compared with the desired output of the message is obtained.

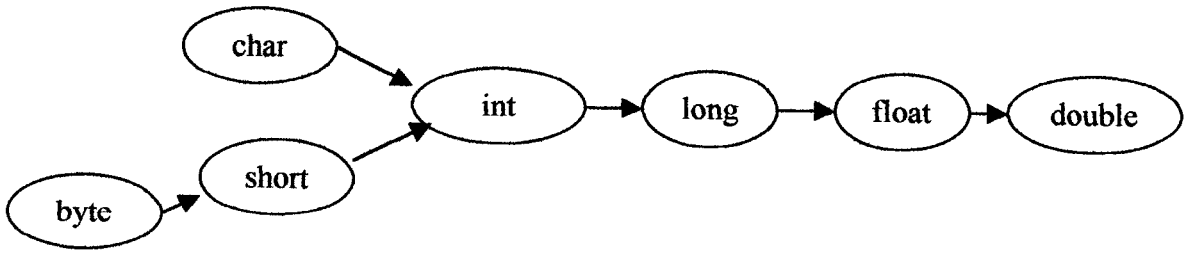


Figure 2.6: The Java widening conversion rules

The primitive type conversion rules of the Java language is as follows (Figure 2.6):

- A boolean may not be converted to any other type.
- A non-boolean type may be converted to another non-boolean type if it is widening conversion.
- A non-boolean may not be converted to another non-boolean type if it is narrowing conversion.

After executing a message, a match is found if the following conditions are met:

- The test message and the method are the same type (constructor, observer or modifier).
- The test message and the method have the same number of arguments and the same or convertible types.
- The test message and the method have the same or convertible return types and the same return value.

The matching algorithm is as follows:

For each test message

If (the test message is a constructor)

If (check each case of argument lists with the parameter list of each constructor)

Construct the instance

Return true

Else return false

Else

If (check the return type with that of method)

If (check each case of argument lists with the parameter list of each method)

If (there is no instance)

Call default constructor

Execute on the method using the instance

If (the result is the same as the expected)

Return true

Else

Return false

Else

Return false

Return false

A candidate class is the class that has the maximum matches (at least one match) to a test program after the test program is executed on this class.

The system will store the matching information on a data structure associated with the class name after finishing execution on all the test messages on all the classes from the library. The system then returns the first 10 class names (if there are more than 10) to the user and each of them must have at least one match. If there is no method or constructor that matches those in all the classes, then no class name will be returned.

The algorithm for this matching process is as follows:

1. *Let f(c) be the list of the classes returned by the retrieval process, each of which has at least one match.*
2. *Sort the list according to the matching number associated with each candidate class.*
3. *Display the first 10 candidates from the list according to the sorted list.*

2.4.4 Inheritance, Overloading and Overriding

Object-Oriented Programming (OOP) has often been promoted on the basis of its reusability merits as discussed before. The retrieval method dealing with OO classes should consider these OO characteristics.

Inheritance: Inheritance is a means of defining a class that is an extension or refinement of another class (super-class). A class definition that contains a class-inheritance-list specifies a class that inherits from each of the class names specified in the class inheritance list. As an implicit part of its definition, a descendant class (subclass) contains the definitions of all fields, methods and constructors of each of its ancestor classes (super-classes). In Java, a class cannot have itself as an ancestor, inherit two or more times from the same class or inherit from two classes at the same time.

A class that is a descendant of another class can add definitions for new fields, methods and constructors to those it inherits. It can also override inherited methods and constructors and thereby associates new bodies with these entities. However, it cannot remove the definition of an entity defined in an ancestor class, nor can it modify the signature of any method, constructor or the type of any field.

The retrieval methodology on OO components should consider their inherited methods as a part of its methods and count the matching info as well on those methods defined as public or protected. Private methods are not inherited, and therefore they should not be counted. In Java, the default access modifier is package, which means only the class in the same package could access the methods in its super-class. Because we could modify the accessor (changing from package to public or protected) during adaptation stage after selecting the candidate class, such methods may be considered as inherited.

For example, in Figure 2.3, the class Computer is a subclass of the class Product. The class Computer inherits all the methods defined in the class Product such as setId(), getID() etc. When executing the test program on the class Computer, the inherited methods should also be tested and the matches should be counted for the class Computer except those overridden methods that is discussed later.

Overloading: Overloading occurs when two (or more) methods have the same name but different parameter lists. All the methods are called overloaded methods. For an overloaded method, selection of the actual method to be invoked is based on matching of

the types of the actual arguments against the types of the formal parameters. Therefore, overloading will not need to be treated specially during execution-based retrieval. The system will execute all the test messages against each method whether the method is overloaded or not. In the class Computer, the method calPrice () is overloaded (calPrice (int) and calPrice (int, String, float)). During execution, these two methods are checked and tested separately.

Overriding: Overriding is the redefinition of the implementation of a visible inherited method. Overriding method is the method in the subclass. Overridden method is the method in the super-class. The signature of an overriding method must be the same as the signature of the overridden method. Overriding method does not cause any side-affect for the execution-based retrieval, but the overridden method should be taken care of. Therefore, when performing execution on the inherited methods from the super-class, the overridden method should not be executed whether its type is matching or not, because the overridden method is hidden from the subclass.

The vector used to store the signature (return type and parameter list) of the methods of each class also saves the method names during the first loading stage. When executing on its inherited methods, a comparison between the names, parameter lists and return types is performed to ensure that the overridden methods are not executed and counted in the matches.

For example, the method toString () of the class Computer is overriding the method toString () in the class Product. When executing on the class Computer, if there is no match, and then the inherited methods begin to be executed. However, because the method toString () in the class Product is hidden from the object of the class Computer, this method should not be reloaded and tested.

2.5 Browsing the Candidate Classes

In most cases of software reuse, the problem is how a user can know whether there is a required component or not and how to use this component. That is why a searching mechanism is definitely required. This is also the purpose of this thesis looking for a more effective and efficient retrieval method.

Of course, a good library or retrieval mechanism should not only provide an effective way to retrieve the components that perfectly or partially match the query, but also allow a user to browse the components. The difference between browsing and retrieving is that while the latter aims to identify and extract components that satisfy a predefined matching criterion, the former consists of inspecting components for possible extraction without predefined matching criterion (Mili et al., 1998). On the other hand, the browsing functionality allows users to familiarize themselves with the classes and to allow administrators to maintain the class library.

Because the execution-based retrieval typically works on a small-scale library, it could allow users to directly browse the implementation of each class in the library. Even after retrieval, the system could return a list of candidates that satisfy exactly or closely the requirements of the user. All the candidates should be arranged according to the number of matches among which the candidate components with most matches appear first.

Because of the limitation of keyword browsing such as unfamiliar and ambiguous keywords, it is not regarded as a preferred method for browsing. The system should rather offer the ability for users to see the implementation in order to choose what they want. Users could select the first several candidate classes to browse their codes in order to see if the classes satisfies their requirement and design structure to determine the usability. Because of the ordered arrangement of the candidates, this approach is practical and useful.

2.6 An Example Class Library

A simple class library is used to demonstrate the hierarchical structure of the library and the retrieval methodology achieved above. All the classes are standard Java classes and they can be compiled and executed by Java compilers. They are described in UML notations.

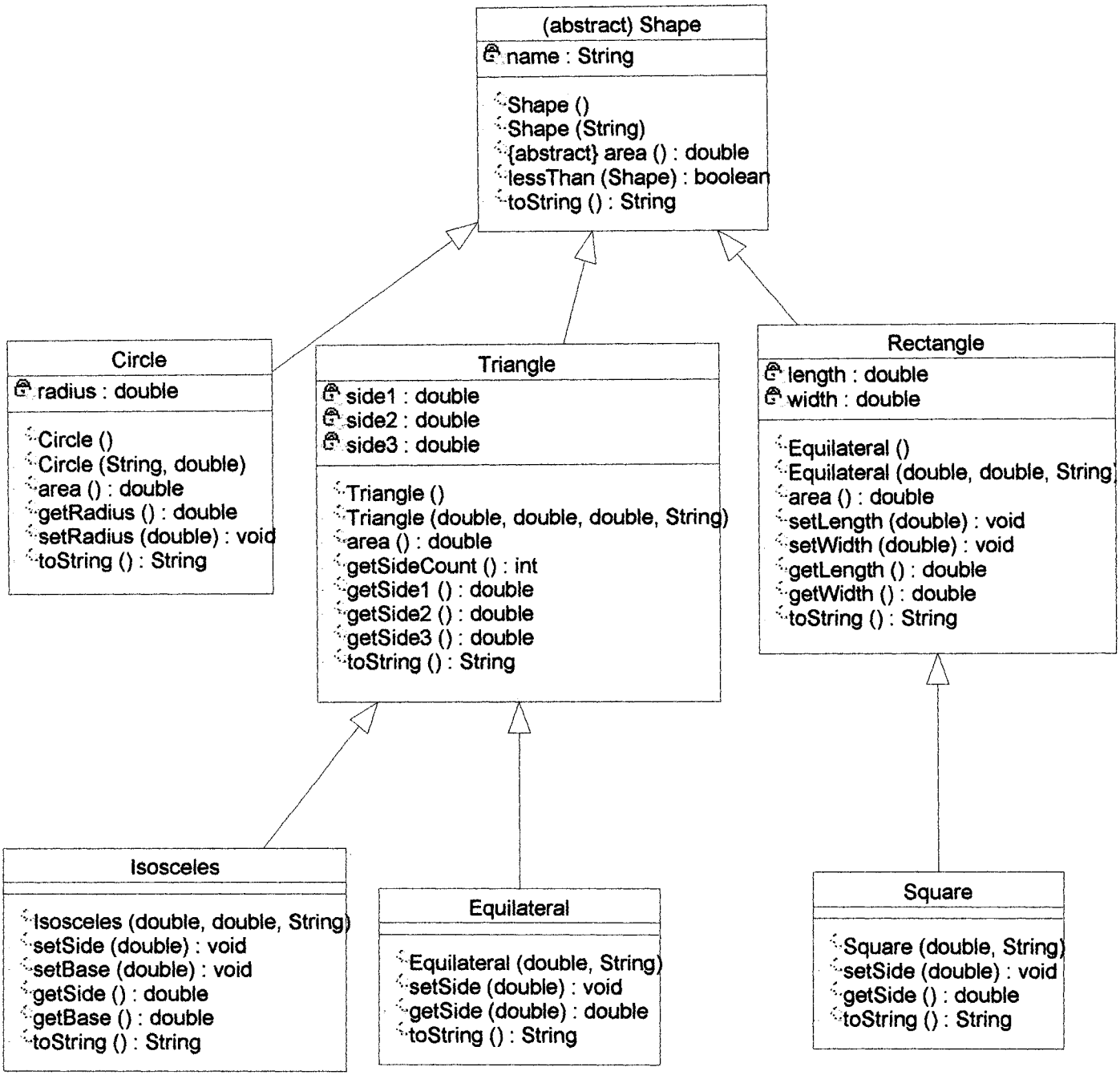


Figure 2.7: Class diagram of the class library 1

Consider a small library called Geometry Library that is composed of 7 classes related to geometry diagrams such as Shape, Circle, Triangle, Rectangle, Equilateral, Isosceles and Square. Class Shape is an abstract class. Other classes are concrete subclasses of the class Shape. The detailed inheritance relationship and all the members (including fields, constructors and methods) are shown in Figure 2.7.

Note that all the classes inherit the method lessThan () from the class Shape and override the method toString(). Classes Isosceles and Equilateral extend the class Triangle and then have the method area(). The constructors of the subclass normally call super () in the first line if there is no distinct call in the first line of the constructor.

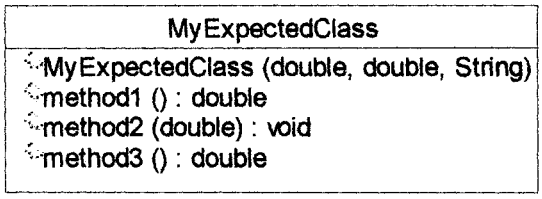


Figure 2.8: Class diagram of the class MyExpectedClass

Suppose the user is looking for a class with the following interfaces (Figure 2.8): MyExpectedClass (double, double, String) is a constructor with an argument list. Method1 performs area calculation. Method2 modifies one of the attributes by passing a value. Method3 observes one of the attributes in the object.

The user can enter the test data as follows:

{(double, 5), (double, 8), (String, "name"), (none, none)}
{(none, none), (double, 40)}
{(double, 7), (void, none)}
{(none, none), (double, 7)}

The system organizes the above input into a test program:

{[(double, 5), (double, 8), (String, "name")], [(none, none)], [(double, 7)], [(none, none)]} with the expected behavior {none, 40, none, 7}.

Class Name	Number of matching methods
Isosceles	4
Rectangle	3
Triangle	1

Table 2.2: Number of matching methods of each class in the class library 1

After execution on each class in the library, a match list is shown in Table 2.2. The system returns the list of candidates in the order of matching method number: Isosceles, Rectangle and Triangle. The user then can browse the three classes to see if they satisfy the requirements. Finally, the system should allow the user to save the retrieved candidates into his/her hard-drive if the user would like to.

Chapter 3 A Prototype System

3.1 Introduction

Based on the improved execution-based retrieval methodology, we have developed a prototype system named EBCRS that stands for Execution-Based Class Retrieval System. EBCRS is designed by UML and implemented using Java Servlets, HTML, JavaScript and Applet. Four class libraries of Java classes are constructed to test the effectiveness of the system. EBCRS is a completely distributed system and is Internet ready. A part of source codes of the EBCRS system is given in the appendix.

Java applets and servlets can be used together in the design of today's multi-tiered web applications. Applets combined with HTML and JavaScript provide a convenient mechanism for building powerful and dynamic interfaces to applications, while servlets give us a highly efficient means to handle requests on a web or application server.

Servlets are protocol- and platform-independent server side components written in Java that dynamically extend web servers running on Java-enabled hosts. Servlets are analogous in many ways to CGI programs. They handle web requests, returning data or HTML programmatically rather than from a static file. They can access databases, perform calculations and communicate with other components. Unlike CGI programs, however, servlets are persistent – they are instantiated once and continually handle requests (usually many simultaneous requests) for the life of the web server. They are therefore operating at a much higher level of efficiency than CGI programs. Servlets run within a servlet engine usually on a web or application server. Servlets can be used to create an easily accessible interface between clients such as applets and web browsers and the core enterprise applications behind them. Servlets may be loaded from remote http://directories, and thus provides true distribution of services in a network.

3.2 Analysis and Design

3.2.1 General Requirements

Overview Statement:

The purpose of this work is to design and to implement a prototype system called EBCRS for retrieving Java classes from the library by execution.

Goals:

- Having a convenient and user friendly interface for the user and administrator
- Retrieving candidates from the library efficiently and effectively
- Having low time and space complexity

System Functions:

- Accept primary data type as arguments
- Accept any Java classes in the library (except abstract class and interface)
- Support execution-based retrieval
- Support multithread
- Internet and Intranet ready

Ref. No.	Functions	Category
R1.1	Input data by the user	Evident
R1.2	Retrieve the class candidates	Evident
R1.3	Open the class file	Evident
R1.4	Browse the class structure	Evident
R1.5	Browse the class implementation	Evident
R1.6	Save the Java file into the local drive	Evident
R1.7	Display the class list	Evident
R1.8	Display error messages	Evident
R1.8	Add classes into the class library	Evident
R1.9	Delete classes from the class library	Evident
R1.10	Accept constructor, modifier and observer input	Evident
R1.11	Allow user to exit the program	Evident
R1.12	Perform recursive on arguments	Hidden
R1.13	Perform recursive retrieval	Hidden
R1.14	Login by the user and administrator	Evident

Table 3.1: System functions of the prototype system

Attributes	Details and boundary Constraints
Interface metaphor	GUI with input, execute, browse and exit
Response time	(boundary constraint) when inputting a test program, the result should appear within 30 seconds
Fault tolerance	If the power is off, the storage should not be affected
OS platform	Window 95/98/NT, Unix

Table 3.2: System attributes (non-functional) of the prototype system

The detailed system requirements are listed in Table 3.1 and Table 3.2.

User Case Analysis

- *User authentication*: Authorized users would first log into the Web site to access the system. Only authorized users should be able to browse class components and perform retrieval from the class library.
- *Library Browsing*: The users should be able to browse the complete class list on the Web and view detailed description and implementation of each class.
- *Class Retrieval*: The users can input the test program, execute it on the class library and retrieve the candidates. They should then be able to browse the retrieved candidates.
- *Library Management*: The administrator can add and delete classes from/to the library.
- *Administrator authentication*: Only the administrator can modify the class library
- *Exit system*: The users log out from the system.

3.2.2 The Distributed Client/Server Application Architecture

The above system requirements can be mapped into a matched application model.

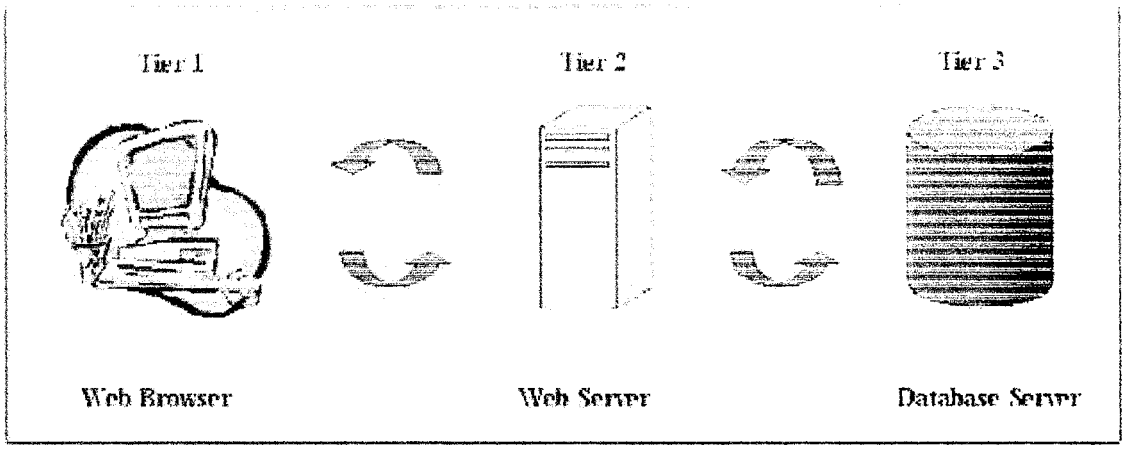


Figure 3.1: The three-tier architecture of the prototype system

The application can be partitioned into three tiers: the user interface layer, the business rule layer and the data store layer that is a 3-tier HTML-Servlets-Database application that used HTML, Applet, Java servlets, MS Access and the Java Database Connection (JDBC) (Figure 3.1).

The first tier is a web browser that serves as our remote client. In the first phase of the application, an HTML (JavaScript-Applet) front-end is used for user-input and displaying the database query results. The HTML (JavaScript and Applet) approach is taken because it offers a user-friendly interface.

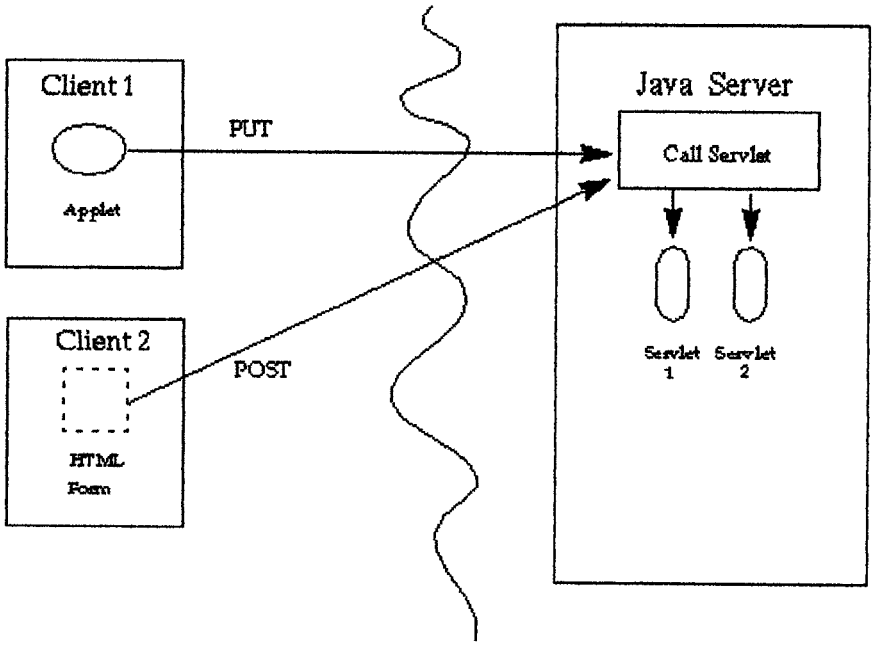


Figure 3.2: The communication between the client and the server

The second tier of the application is implemented with a Web server being capable of executing Java servlets. The Java servlet harnesses the power of JDBC to access the database and to access files in the server and then to store/retrieve information as needed. The server sends the results to the client through the HTTP protocol or socket protocol (Figure 3.2). In some cases, a dynamic HTML page is generated by the servlet based on the database results.

The third tier is composed of our back-end database server. The database server stores the information that is used by the application including MS access files, text files and Java class files. Through the JDBC API, servlets can access database in a portable fashion by using the SQL call-level interface.

We employ three methods for the Applet-Servlets communication (Figure 3.2). The simplest way for an applet to exchange information with a servlet is through an HTTP

text stream, which is used in the User Login Use Case, the Administrator Login Use Case, the Exit System Use Case and the Save File Use Case. Java's URL and URLConnection classes make it easy to read data from a URL without having to worry about sockets and other normally complex issues of network programming. All we need is a server-side component that can deliver information via a URL. We use the URLConnection in the Browse Use Case. However, only one communication path is available for this method. One big benefit of using raw sockets is that the connection is persistent and bi-directional. With a raw socket approach, we can establish a connection with the server and continually receive updates as they occur which is needed in the Retrieve Use Case and the Manage Library Use Case.

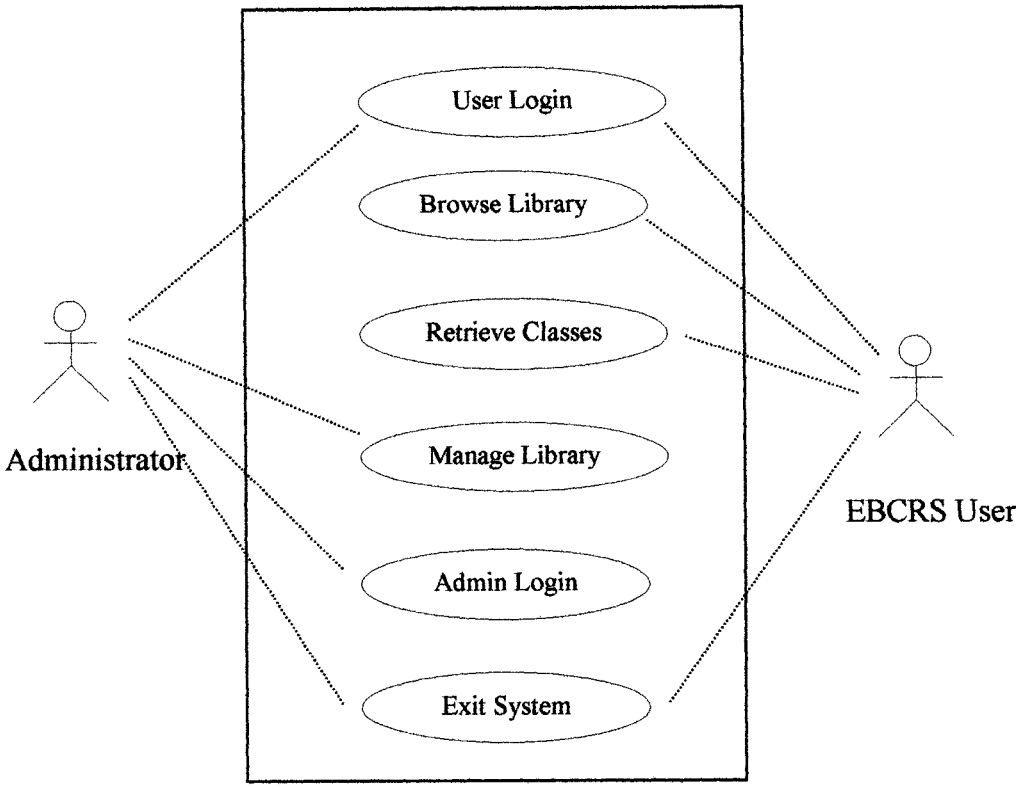
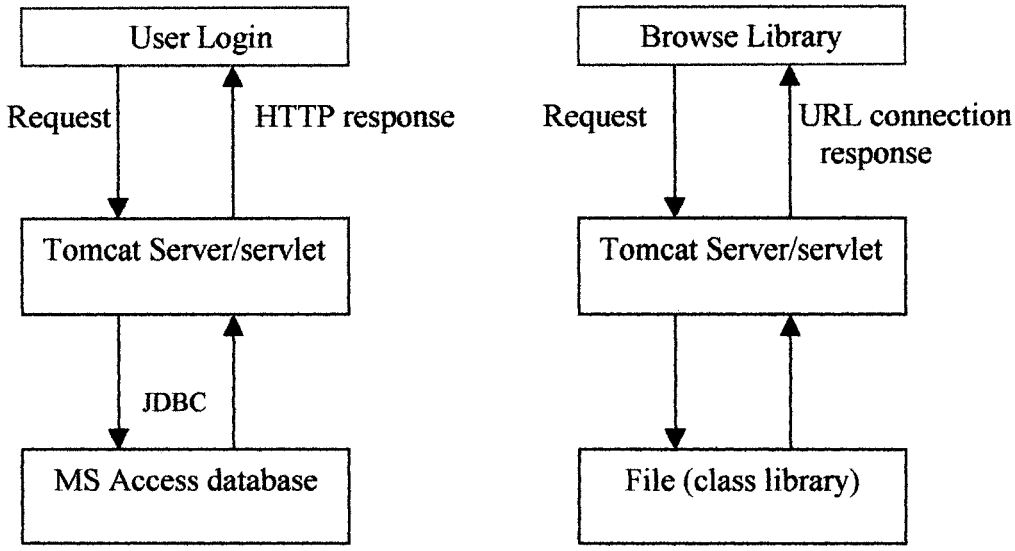


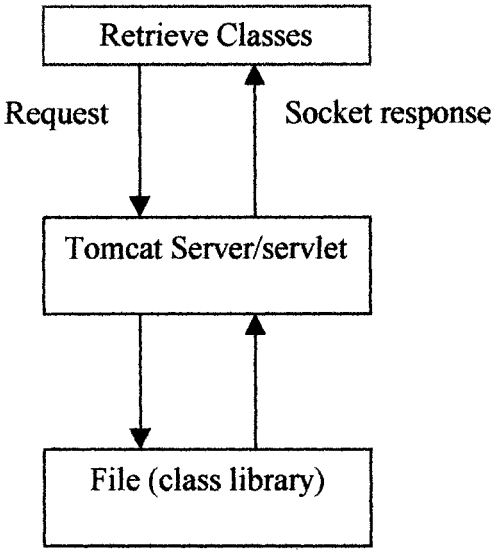
Figure 3.3: Use case diagram of the prototype system

The general use case diagram is shown in Figure 3.3.

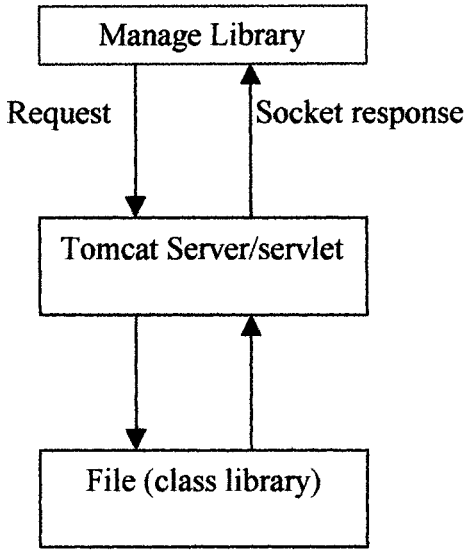


A) User Login Use Case B) Browse Library Use Case

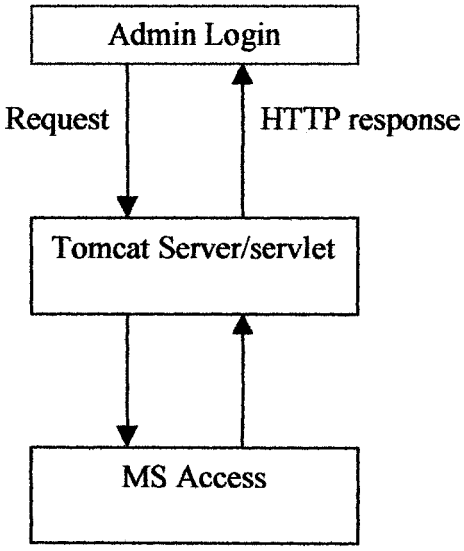
Figure 3.4: Detailed architecture of each use case



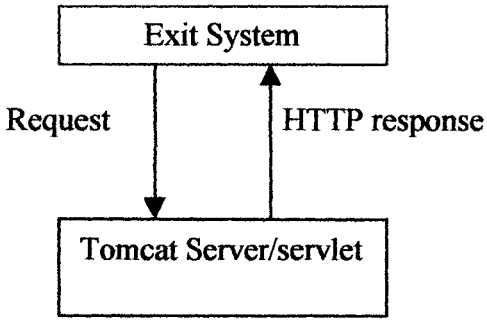
C) Retrieve Classes Use Case



D) Manage Library Use Case



E) Administrator Login Use Case



F) Exit System Use Case

Figure 3.4: Detailed architecture of each use case (cont.)

The architecture of different use cases is shown in Figure 3.4.

3.2.3 General GUI of the EBCRS system

This system is a full functional multi-tiered client/server application. The client side is a Java applet-based application with user interfaces implemented with Java AWT. The applet is embedded in an HTML web page with JavaScript so that the applet can be executed anytime and anywhere through a web browser. The web server is the Tomcat from the Jakarta project. We have extended the functionality of the web server with several Java servlets. Each servlet serves a specific request from the client.

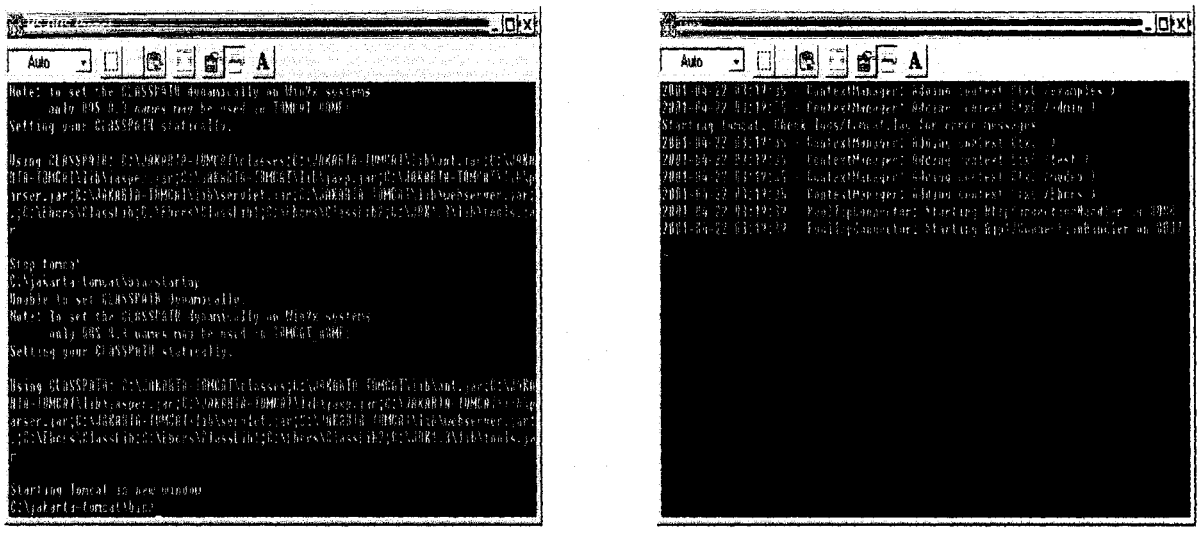


Figure 3.5: The screen capture of the Tomcat server

The web server uses JDBC to make connection to the database or directly read from files and execute the query and update requests from the client. Each request will be sent by a browser to a web server and processed by a Java servlet (Figure 3.5).

This system supports two types of users: administrator and authorized users. The administrator has the privilege to add or delete classes and maintain the class library. The users can browse the class library, check the implementation of class and execute the program. However, the users cannot make any change to the class library in the database.

The server side

The server side has many components working together to offer the services requested by the client side. These components are stored in Ebcrs\Web-Inf\classes. The functionality of each component is listed below.

DaemonHttpServlet and Daemon: Daemon is a class that extends Thread. DaemonHttpServlet is an abstract class extending HttpServlet and performing low-level socket management. The init() method creates and starts a new Daemon thread which is in charge of listening for incoming connections. The destroy() method stops the thread. The Daemon thread begins by establishing a ServletSocket to listen on some specific socket port. The socket port is determined by a call to the servlet's getSocketPort() method. After establishing the ServerSocket, the Daemon thread waits for incoming requests with a call to serverSocket.accept().

MainServlet: This is the most important servlet, which extends DaemonHttpServlet and implements a handleClient(Socket) method that spawns a new MainConnection thread. It accepts the client message and lets the handleClient () method to perform action based on the request given. This servlet will be loaded at startup time of the server so that it can be accessed directly.

MainConnection: MainConnection extends Thread performing all the action according to request type sent by the client when MainServlet orders. When it is created, it saves a reference to the MainServlet and a reference to the Socket so that it can communicate with the client. When the MainConnection instance is created, its run() method is called. In this method, MainConnection uses some protocol to communicate with the client.

Retrieve: This class handles all the retrieval requests. It executes the test program according to the order of messages and type of messages (constructor, observer and modifier). It stores the method's signature in a vector during the first time loading from the server. Each time it compares the argument list and return type between the test message and loaded classes before execution. It then compares the returned value and the expected result.

ClassLibInfo: It is a class that stores the class library information. It handles adding and deleting classes from/to the class library. It also provides a method to access the class lib path.

InputProg: InputProg is class to organize the input data from the user. It stores each message into a vector and provides access methods.

AccessServlet: The servlet handles the user login. It stores the userid and password entered by the user, accesses MS Access database through JDBC, compares the userid and password with those stored in the database and dynamically creates a HTML page or redirect to the MainInterface.

SaveFileServlet: This servlet handles the save file action. Once a client sends a message to save a selected file from the selected folder and then the servlet will send this file to the client folder.

ExitServlet: The servlet handles the user logout. It allows user to enter the system again by redirect to the login page or dynamically creates an HTML page.

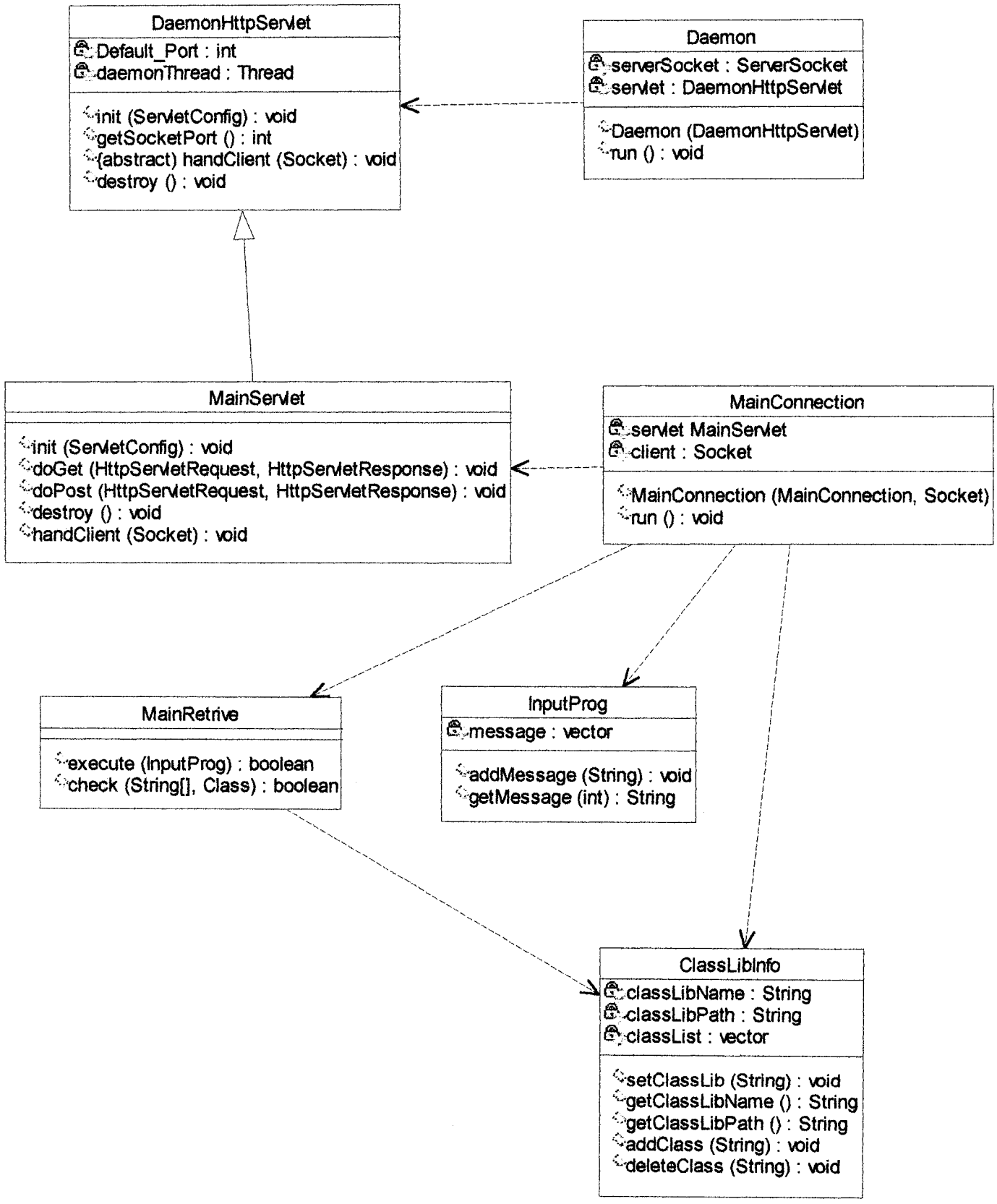


Figure 3.6: Class diagram of the server-side main classes

The main class diagrams in the server side are as Figure 3.6. Some servlets such as AccessServlet, ExitServlet and SaveFileServlet are omitted here because of similarity with general servlet structure.

The client side

The client side includes HTML, JavaScript and Applet. The applets allow users to access the system friendly by providing good-looking interfaces such as browsing the library, executing the classes, managing the library and exiting the system.

LoginInterface: The user needs to enter his/her user name and password. A servlet (AcessServlet) on the web server host will search the user name and password in the MS Access database on the host. If the user has been found, the user will be redirected to the MainInterface. If the user id or password is invalid, the user will be prompted to login again dynamically.

MainInterface: It controls whole the system functionality by connecting other interfaces for different operations. It displays a list of class libraries for the user to select and has four buttons: "Browse Library"- go to the browse library interface, "Retrieve Classes"- allows users to go to the execution interface to input his/her test program and perform retrieval on the library. "Manage Library"- allows the administrator to go to the administrator interface in order to add or delete classes from the selected library. "Exit System"- let the user to go to the exit interface and logout from the system.

BrowseInterface: It will allow the user to check all the classes in the libraries by browsing directly the implementation or the class structure such as methods, constructors, fields and super-classes.

ExecutionInterface: It provides the user with different input interfaces such as constructor, observer method and modifier method in order to enter the test program. The user can check the list of arguments and input the value of arguments.

CandidateInterface: It lists all the retrieved candidates and allows the user to browse the implementation of classes, save the class file into the local disk or return to the main interface.

ManageInterface: It allows the administrator to login for managing the class library. "Add Class" – go to the add class interface, "Delete Class" – go to the delete class interface, "Back to Main" – let the user go back to the main interface.

AddInterface: It allows the administrator to select a Java class file and add into the selected library. The user can also return to the main interface.

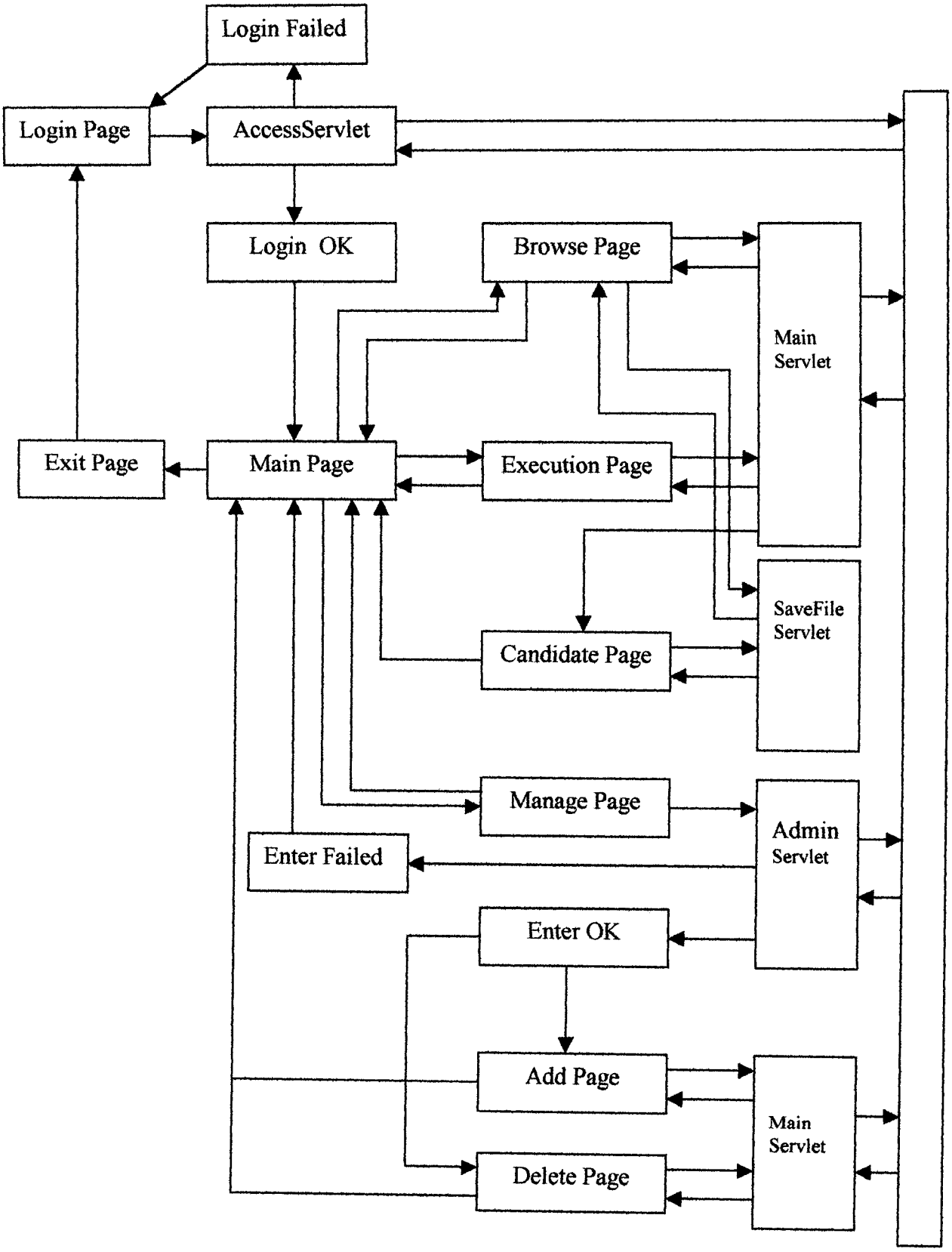


Figure 3.7: System function flow diagram in the client side

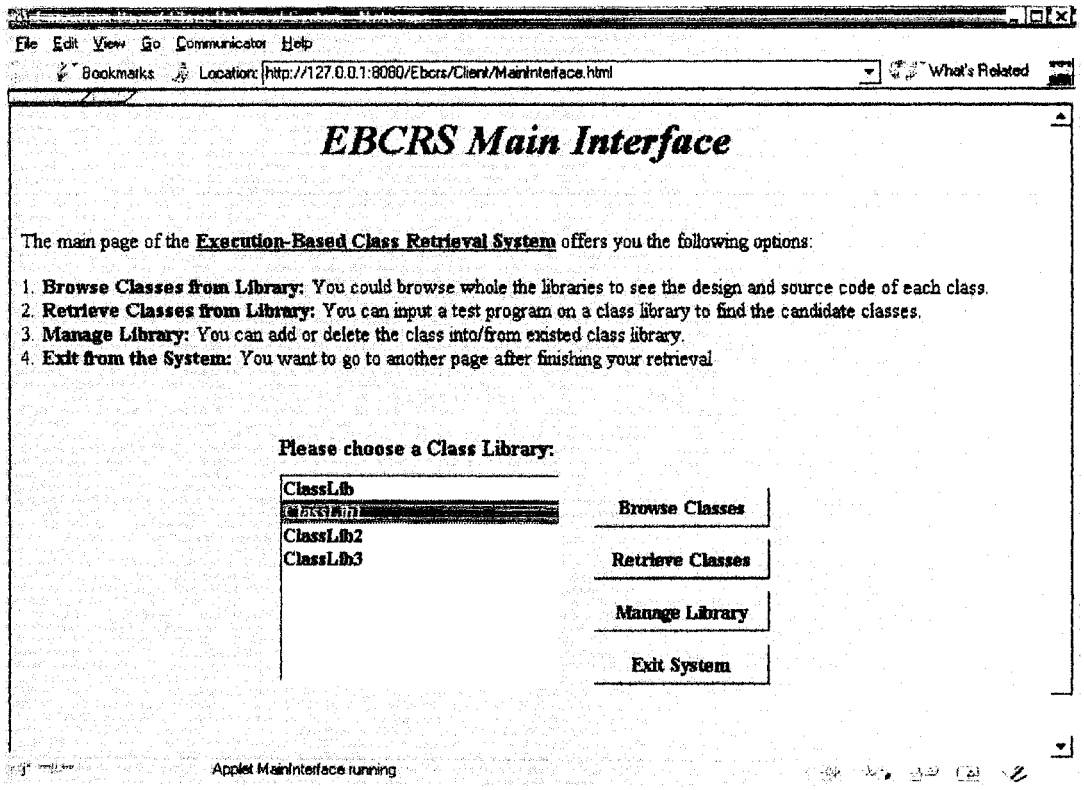


Figure 3.8: The main interface of the EBCRS system

DeleteInterface: It allows the administrator to delete a selected Java class file from the selected library. The user can also return to the Main Interface.

ExitInterface: It allows the user to exit the EBCRS system or return to the system.

The system function flow diagram for the client side is shown in Figure 3.7. The main interface is shown in Figure 3.8.

3.3 Organizing a Class Library

Because the class library is constructed for testing this proposed methodology, no much time has been spent on the organization of the class library. However, classes are stored

into different sub-libraries according to their functionality. For example, the Geometry diagrams related classes are in one sub-library ClassLib1; String related operation classes are stored into another sub-library ClassLib2; Person related classes, such as Employee, Student, Customer, Person, Company etc are classified into one sub-library ClassLib3 and AWT event related classes are put into the sub-library ClassLib (Figure 3.8).

- All the classes are standard Java classes that are allowed to inherit from the class in the same folder or from the Java API classes.
- All the classes are allowed to have fields, constructors and methods with various access modifiers.
- Although the input variable types are limited to primitive and String, the methods in classes could accept different user-defined data types.
- Classes are permitted to have static methods or fields.
- Although some abstract classes are stored in the library, they are not listed as entry for retrieval because abstract class or interface cannot be instantialized. They are put there because other classes may extend them.

Each sub-library is a folder where all the class files are stored. A class list file that lists all the class names in the sub-library is stored as a standard text file in each folder.

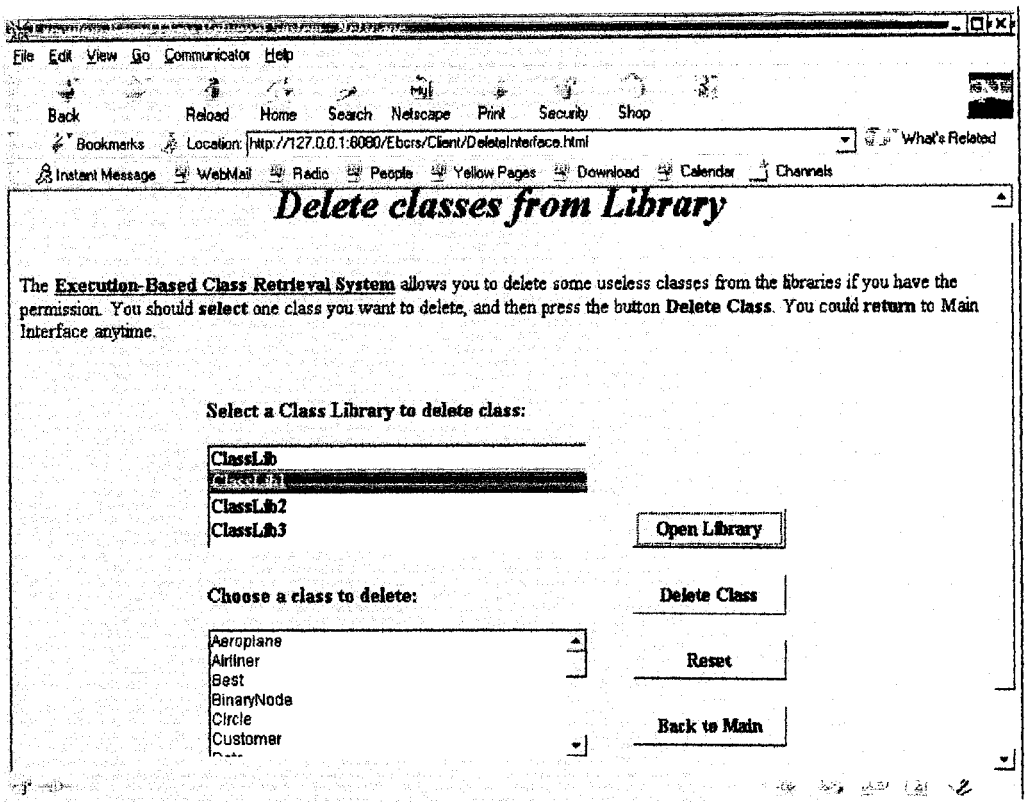


Figure 3.9: The delete interface of the EBCRS system

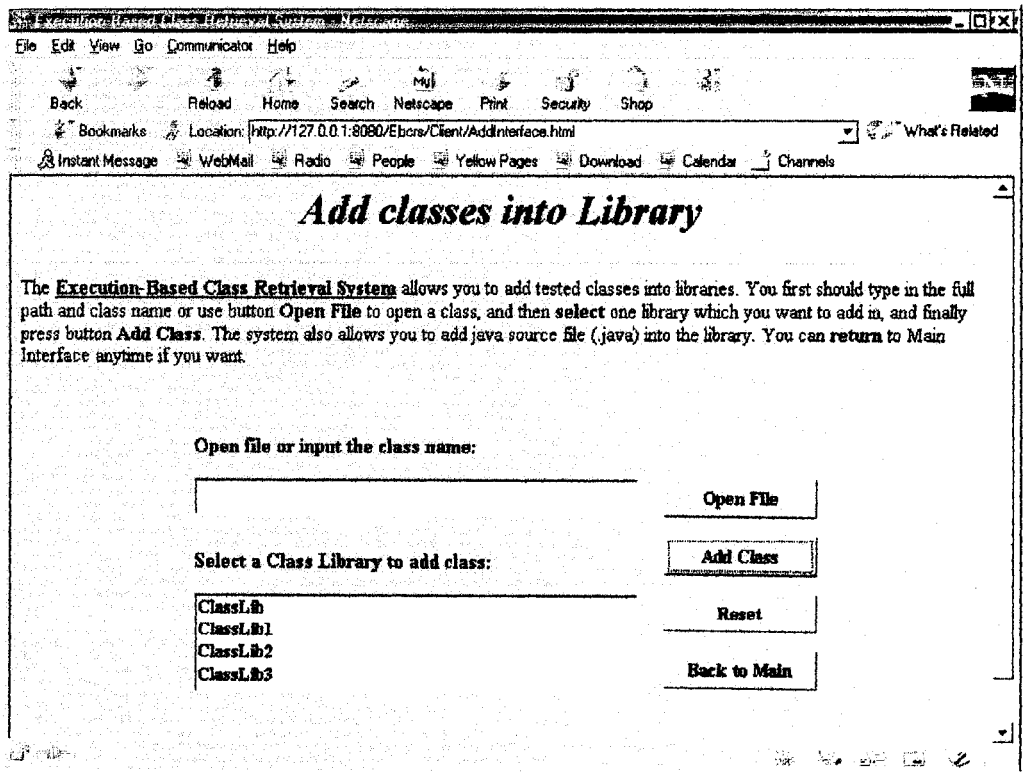


Figure 3.10: The add interface of the EBCRS system

The administrator can login to reorganize the library by adding and deleting classes to/from the class library. He/she should first select one library and then choose a class to perform action. The two interfaces (Add Interface and Delete Interface) will help administrator to maintain the class library (Figure 3.9 and Figure 3.10).

3.4 The Retrieval Process

After the class library is organized and all the classes are assembled, we could perform the retrieval process to find the candidate classes exactly or appropriately match the user's requirements.

3.4.1 Inputting Data

Because the class behavior is composed of all the responses from the interfaces (including constructor, observer method and modifier method) and all the three types can be easily distinguished by their return type and parameter list, there are three input interfaces to accept individual type of input. Another reason to have three input interfaces is that the input order decides which instance of the class is used to send a message. Each time when the user input a message, he or she should push the button "Done One". When all the input data are accepted by the system, the user then pushes button "Execute" to retrieve. The system will execute each message in order of inputting.

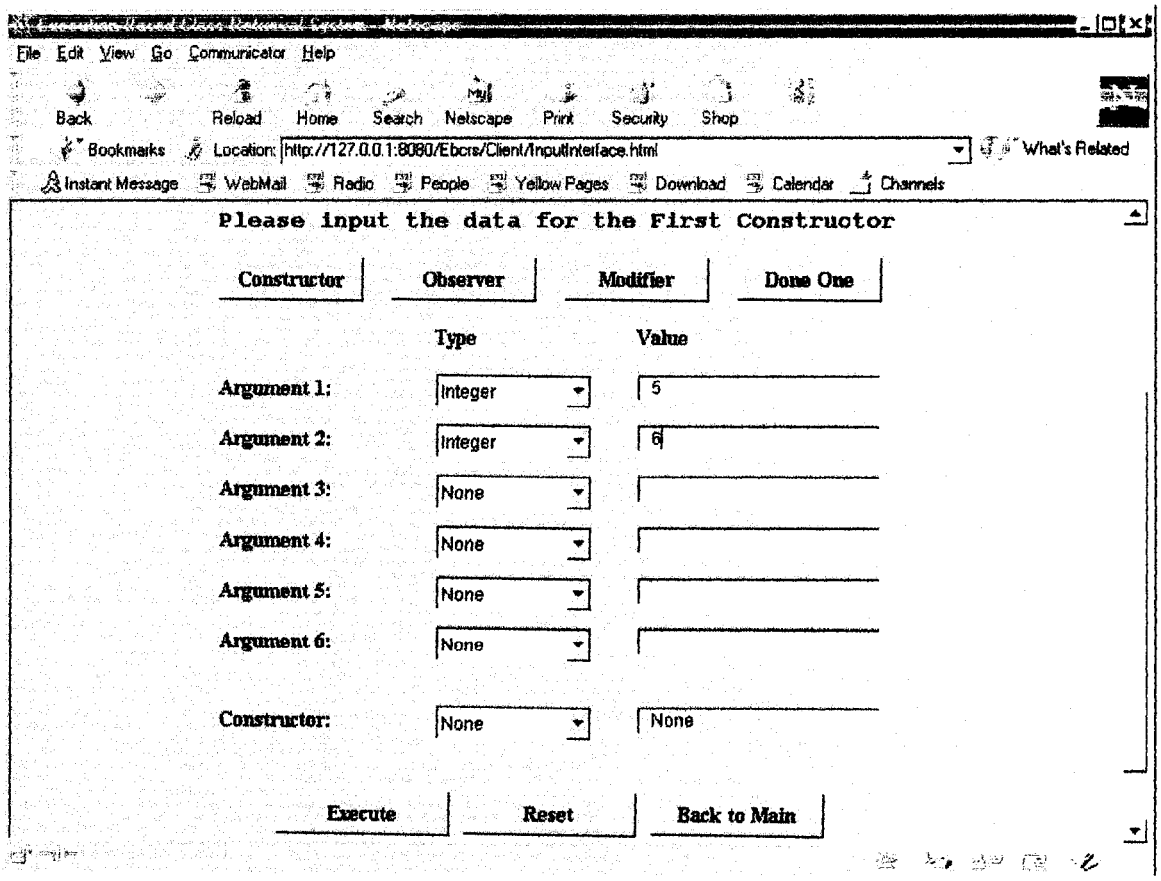


Figure 3.11: The input interface for constructor

The constructor-input interface only allows the user to select argument type and input the value. No return type and value are needed to enter (Figure 3.11).

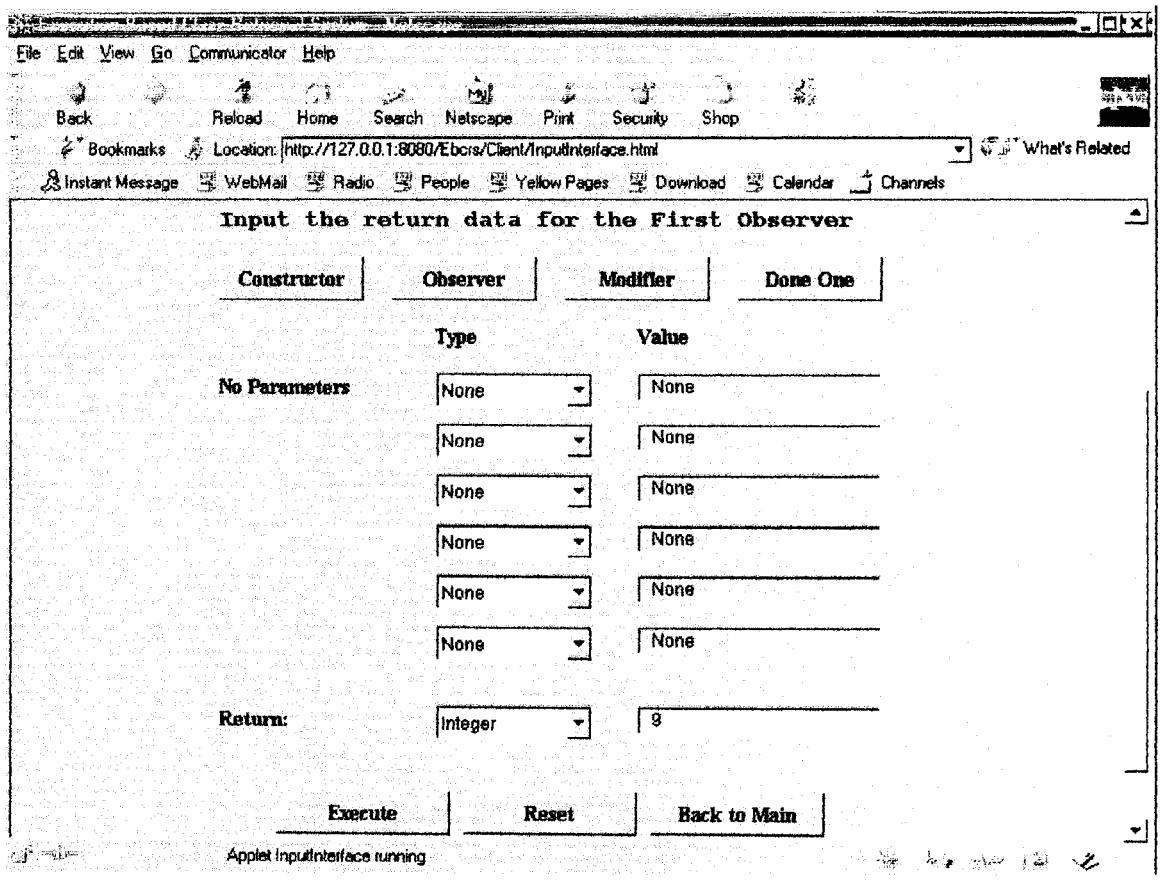


Figure 3.12: The input interface for observer

The observer-input interface does not allow the user to input any argument (Figure 3.12). The user only needs to select the return type and value because the observer does not cause side effect of an object. The message for the observer methods will use the existing instance of the class to execute. If no instance exists, then a default constructor is called to create an instance.

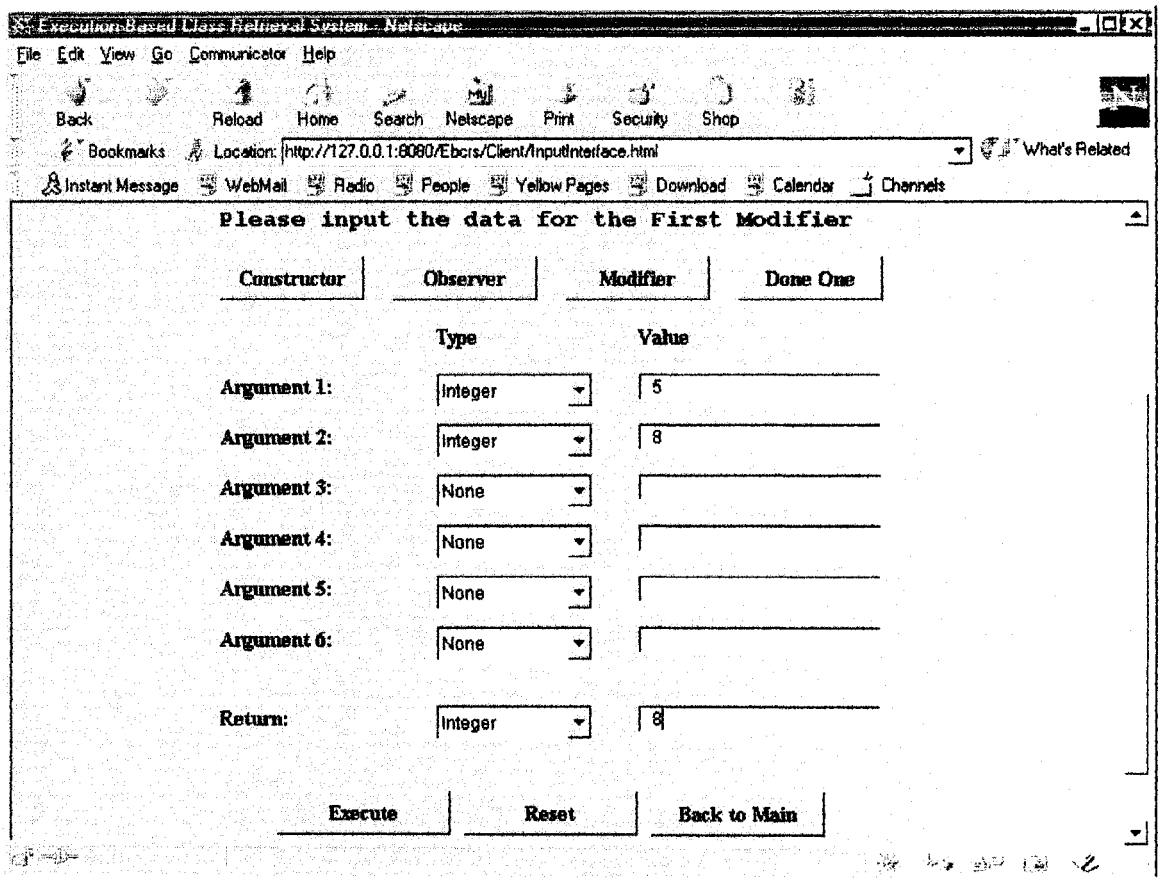


Figure 3.13: The input interface for modifier

The modifier-input interface asks the user to input all the information about a test message such as argument list and return type and value (Figure 3.13). The return type may include void with empty value ("").

In this system, only primitive types and String are allowed such as char, short, int, long, float, double and boolean. After the user's input, the system will check the matching problem between the type selected by the user and the value inputted. If the type selected by the user and the value entered are not matched, then an error message is displayed. The Java widening rule is adopted in the system as discussed in chapter 2. After the input is done, the user pushes the "Execute" button and sends all test messages to the server.

3.4.2 Executing the Program

The system will organize the test messages into a test program according to the input order after receiving the test messages. This is very important. The user may have created various instances and use them to perform different actions on various objects. If the order is not kept, then the result may be out of control.

During execution of the first message of each type (constructor, observer and modifier), the system loads each class and store the parameter list and the number of the parameters in a vector and also the method name and return type if it is an observer or modifier. And then comparison between the argument list of the test message and the parameter list of the method is performed and is followed by the comparison of return types. If any mismatching happens, the argument order is switched and a new test message is created until all the orders have been tested.

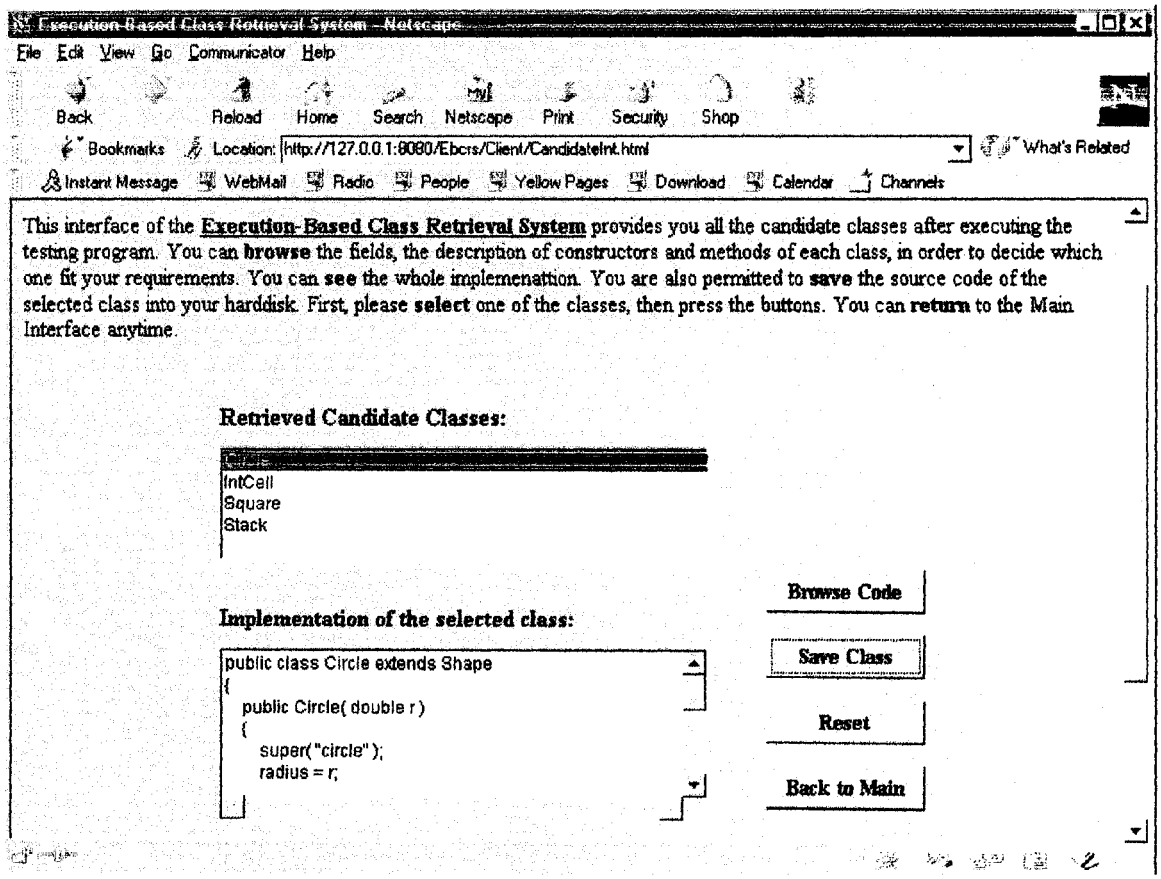


Figure 3.14: The candidate interface of the EBCRS system

If all match, then the system loads the class from the database and performs execution with the test message. The return value is stored and compared with the expected result. If they are the same, then a match is returned and recorded. Finally, after all the match information has been obtained, the candidate class names are sent back to the client according to the order of match's number (Figure 3.14).

3.4.3 Browsing the Library

After retrieval, a list of maximum ten class names is sent by the server and listed in the browser. The user can browse the class implementation (code) to determine if they match his/her requirement. Because the number of candidates is not very large, browsing the implementation of each class is useful and practical.

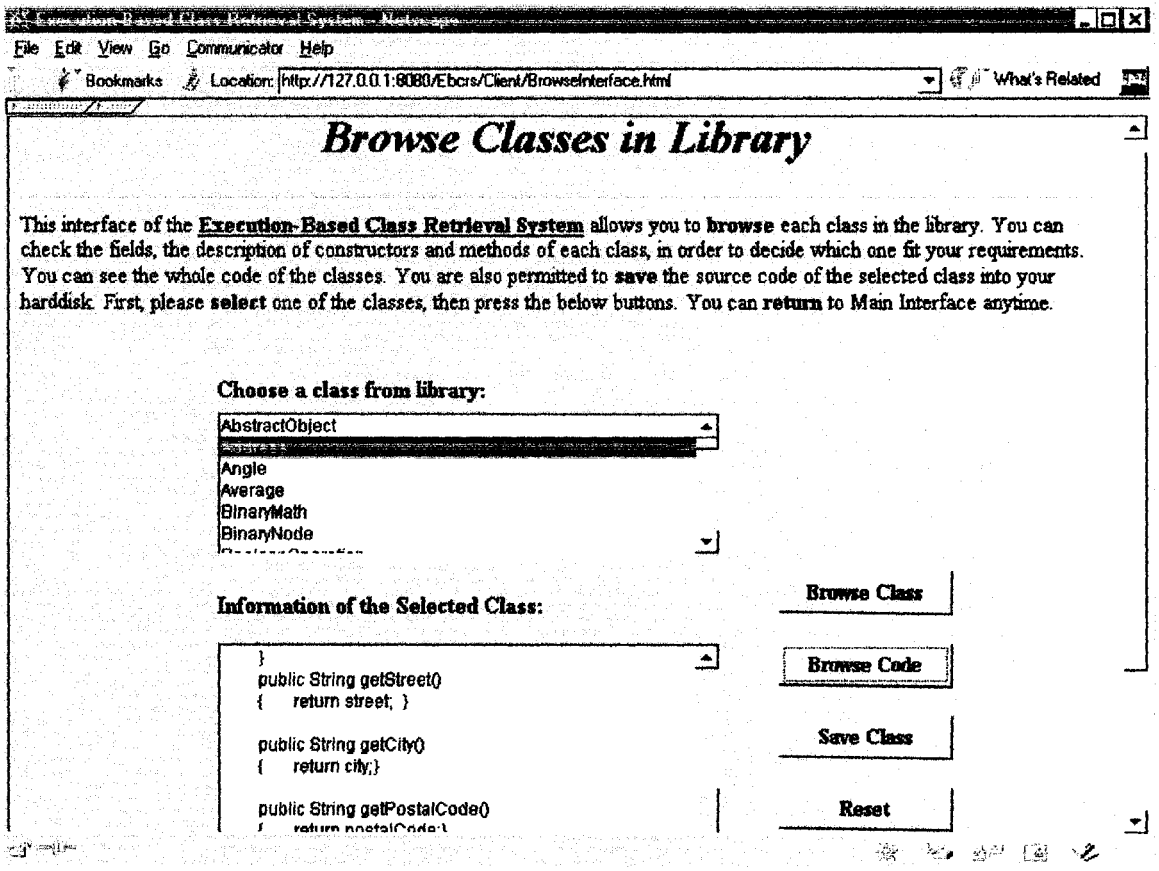


Figure 3.15: The browse interface of the EBCRS system

After entering the system, the user can also directly enter the Browse Interface to check all the information about each class such as the implementation code (Figure 3.15).

3.4.4 Saving the Candidates

Applet code is served from a host web server and executed in the client's browser on the end user's machine. To prevent the proliferation of viral applets that could wreak havoc on unsuspecting surfers, applets are bound by security constraints that allow them to communicate only with their host server and prevent them from interacting with the end user's machine. They cannot read or write from its file systems, execute programs or examine certain sensitive environmental properties.

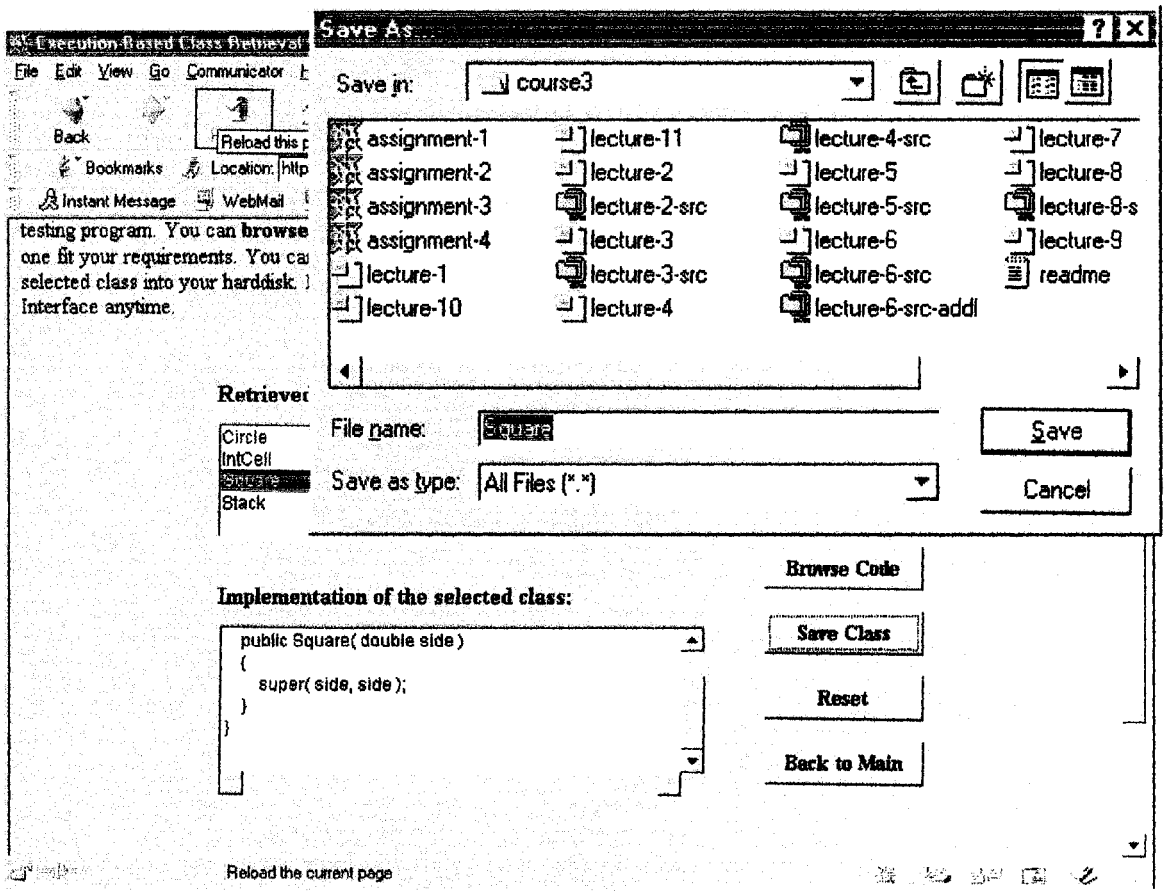


Figure 3.16: The save file interface of the EBCRS system

Because of these restrictions, we must employ special strategies to communicate information to the client through the applet. We take the advantage of servlet: it can use

HTTP OutputStream send all the data to the client without restriction (Figure 3.16). The code of SaveFileServlet is as below.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException
    {
      String path = "c:\\Ebcrs\\"+request.getParameter("folder")
            +"\\"+request.getParameter("file");

       File file = new File(path);
       response.setContentType("application/octet-stream");
       response.setContentLength((int) file.length());
       response.setHeader("Content-Disposition",
            "attachment; filename=\"" + file.getName() + "\"");
       InputStream in = new BufferedInputStream(new FileInputStream(file));
       OutputStream out = response.getOutputStream();
       byte[] buffer = new byte[4096];
       while (true)
            {
           int bytesRead = in.read(buffer, 0, buffer.length);
           if (bytesRead < 0)
                    break;
           out.write(buffer, 0, bytesRead);
        }
        out.flush();
        out.close();
        in.close();
    }
```

3.5 System Security

An ideal system should maintain different levels of privileges depending on various users. Two levels are employed in the EBCRS system. The user login interface allows the user to input the userid and password and then send them to the server for checking. The AccessServlet loaded in the server handles this matter by retrieving the registered user name and password from the MS Access database. If the user enters a wrong message, then the system will dynamically allow the user to try again.

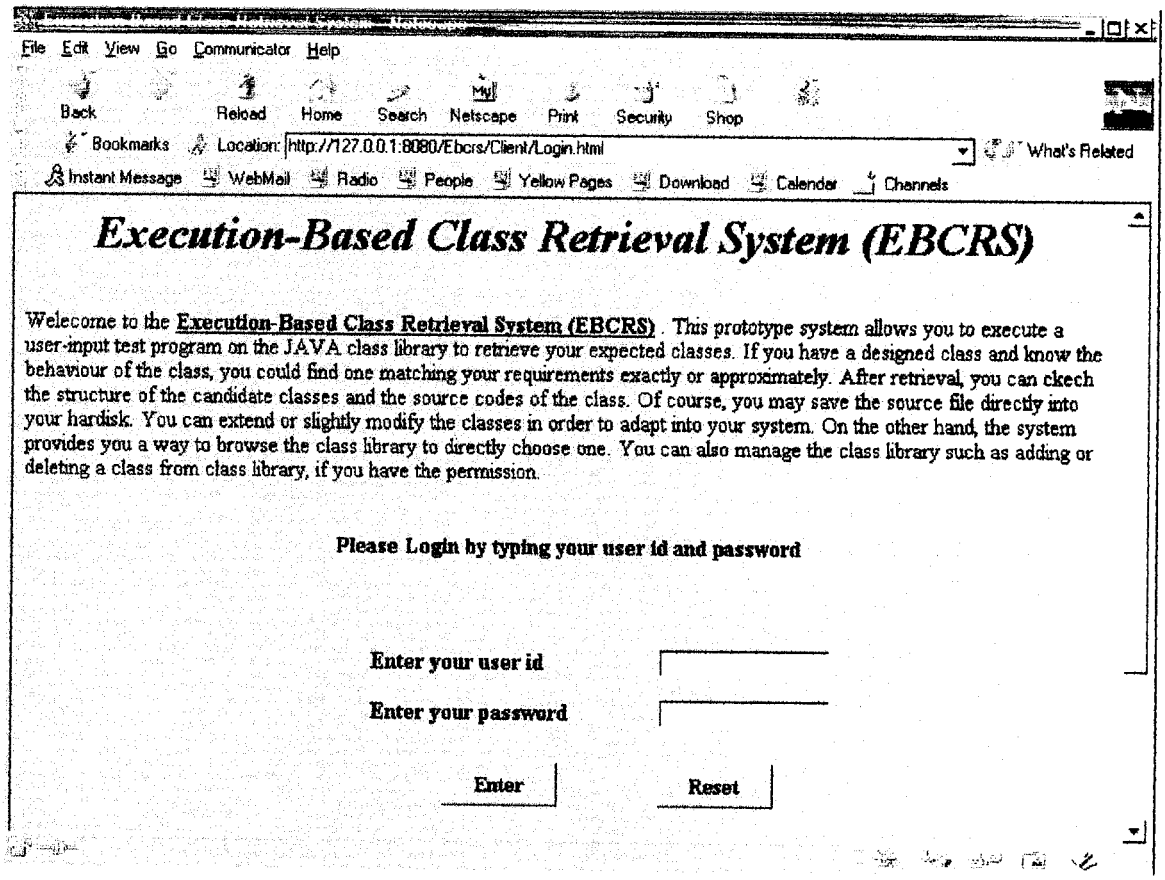


Figure 3.17: The user login interface of the EBCRS system

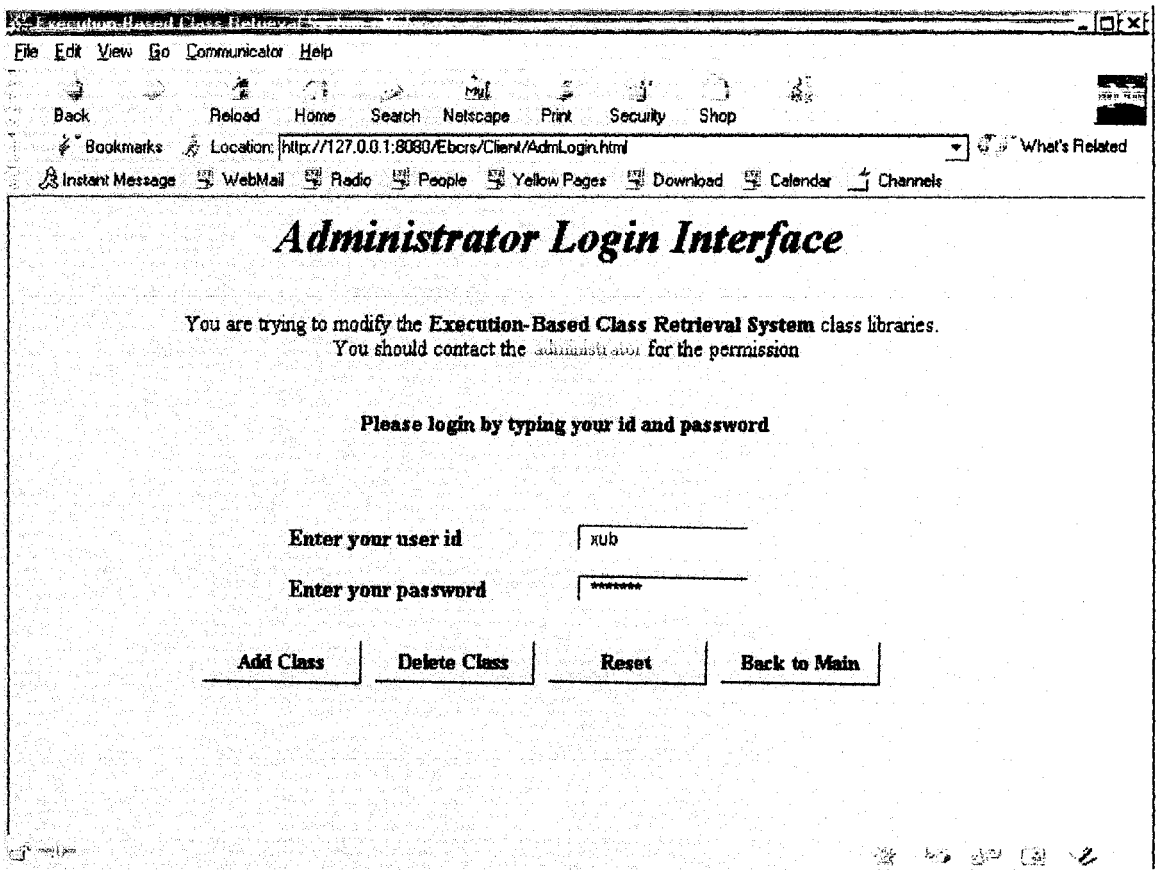


Figure 3.18: The administrator login interface of the EBCRS system

The user login interface is shown in Figure 3.17. The same is for the administrator login page (Figure 3.18). If the user has no authorization to enter the management page, then the system will direct him/her to the main interface. Once authorization is finalized, the administration page will direct to the adding class page or the deleting class page respectively. This is handled by the AdminServlet in the server, which connects the MS Access database through JDBC-ODBC bridge.

The system should also handle long running time or non-terminating problems. A certain time quota (30 seconds) is assigned to the retrieval process. Once the time is out the system will terminate the process and return to the main page.

3.6 A Scenario

In this section, we try to use the class library described in the chapter 2 to show in detail how the EBCRS system works and how the client gets the retrieval result.

Suppose that the user wants to build up a Java class with 2 constructors and 3 methods that have the following functionality:

- Constructor 1 accepts one double and one String arguments.
- Constructor 2 accepts two double and one String arguments.
- Method 1 accepts a double as an argument, to update one of the attributes.
- Method 2 accepts no argument, but get a circle area with the radius input with method 1.
- Method 3 accepts no argument, but get a stereo volume of a cylinder.

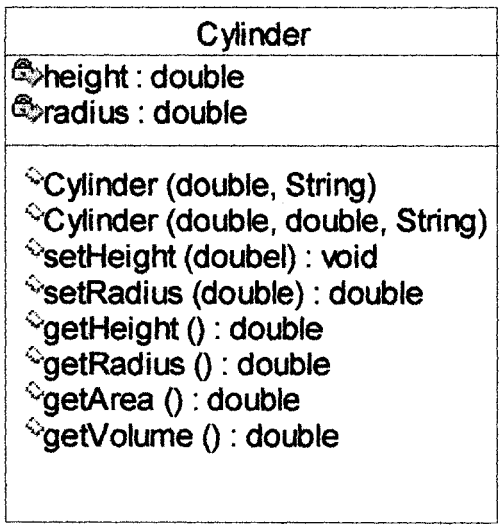


Figure 3.19: Class diagram of the expected class Cylinder

Obviously the user wants a class that is similar to the class Cylinder in Figure 3.19. However, the user does not want to implement it by himself/herself and then comes to the

EBCRS system to login into the system. The user pushes the "Retrieve Class" button to enter the Execution Interface.

The user can enter the test data as follows:

- Message 1 {(double, 10), (String, "cylinder"), (none, none)}
- Message 2 {(double, 10), (double, 20), (String, "cylinder"), (none, none)}
- Message 3 {(double, 5), (void, none)} - set the radius
- Message 4 {(none, none), (double, 78.54)} - expect an area with radius 5
- Message 5 {(none, none), (double, 785.38)} - expect a volume with radius 5, height 20

The system organizes the above input into a test program 2: {[(double, 10), (String, "cylinder")], [(double, 10), (double, 20), (String, "cylinder")], [(double, 5)], [(none, none)], [(none, none)]} with the expected behavior {none, none, none, 78.54, 785.38}.

Class Name	Number of matching methods
Circle	4
Isosceles	3
Rectangle	3
Equilateral	3
Square	3

Table 3.3: Number of matching methods of each class
in the class library 1 on the test program 2

After execution on each class in the library, a match list is shown in Table 3.3. The system returns a list of candidates in the order of matching method number: Circle, Isosceles, Rectangle, Equilateral, and Square. The user then browses all the five classes to see their implementation and finds that other classes have no close relation with the

expected one and only class Circle satisfies his/her requirements. The class Circle can be simply extended to include several more methods.

Chapter 4 Evaluation, Conclusion and Future Work

4.1 Evaluation

The complete criteria for evaluation of the retrieval methodology have been defined and summarized by Mili et al. (1998). They include technical, managerial and human criteria.

1. Technique Criteria:
    - Precision and Recall: The retrieval method should have high precision and high recall, otherwise, the user may need to spend time to understand those not reusable components, or some components may be available for reuse but are not recognized.
    - Coverage ratio: It refers to the average number of components that are visited over total size of the library. This ratio should be ensured so that there is no excluded portion of library containing relevant components.
    - Efficiency (time complexity and logical complexity): The retrieval method should take a short time, i .e., high efficiency, otherwise the user should wait for a long time to see the result.
    - Automation potential: The retrieval method should be possible for automation. This will save a lot of human input errors.
2. Managerial criteria:
    - Investment and operating cost: The cost to setup this software library should be low, otherwise the library may not be useful and it is better to build the component from scratch.
    - Pervasiveness: The retrieval method should be available for many platforms and many situations.
3. Human criteria:
    - Difficulty of use: The retrieval method should be easily used even for untrained users.
    - Transparency: Users prefer to understand how the method works.

We evaluate our methodology according to above criteria.

Precision: Precision is defined as the proportion of retrieved material that is relevant; it measures how well the system retrieves only the relevant components (Mili et al., 1995). Analytical evidence (Podgurski and Pierce, 1992) and experimental evidence (Podgurski and Pierce, 1992; Hall, 1993) show that the behavior retrieval provides a high precision because it actually compares the behavior of class with the expected behavior. Although the behavior of some classes may not exactly match the expected, recursively retrieval allows improve the precision. The present method allows the user to input data on constructor, modifier and observer, and thus improves the precision. The precision could become 100% theoretically.

Recall: Recall is defined as the proportion of relevant material and it measures how well the system retrieves all the relevant components. As the matching criterion of execution-based retrieval is a necessary condition to the relevance criterion, it causes no loss of recall under certain circumstance.

The previous execution-based retrieval method does not offer the facility to let users define their constructors, but only allow the system to call the default constructor of each class in order to create object for invoking the methods. For those classes that do not have the default constructor, it will not be retrieved although they may satisfy the query. The previous method requires users to give the correct order of arguments, and thus some relevant candidate classes may be excluded from the retrieved list. The proposed method does allow the system to use different constructors, and does not require the specific order of arguments, and thus largely improves the retrieval recall.

Coverage ratio: The execution-based retrieval method in fact invokes an execution on whole class library or sub-library, thus has a coverage ratio 100%. Further retrieval may select those with most matching and thus reduces the coverage ratio.

Efficiency: Although this method executes the test program on each class of the library, we try to classify classes into several sub-libraries in order to increase the retrieval efficiency. Moreover, before actually performing execution, the system performs type checking against the input data and thus reduces the time-complexity. When the class is loaded into the system from a class library for the first time, the signature of each method is stored into a dynamic data structure. Before executing the test messages, the type match checking is performed against those stored in the system. If types match, then the system will load the class and execute on it. Otherwise, this method or this class will be skipped. This improvement will reduce the time complexity and execution time and thus increase the system efficiency.

Automation potential: The execution-based retrieval generally involves an automated organization of the test program and execution. In our system, users only need to select the argument types and to input the values and the system will organize those input data into a test program. Therefore, it is simple and eminently automatable.

Investment and operating cost: This method does not require the classification of the class library and no complex mechanism is required. This method offers Internet facility and thus reduces the cost of installation.

Pervasiveness: The EBCRS system is designed and implemented using a distributed HTML-Servlet Client/Server architecture. Java makes an application portable, platform independent and local/remote transparent. This implementation is Internet ready. Although this implementation is tested on the Java class library it could be easily modified to accept other OO class libraries such as C++ library using the Java JNI package and considering the multi-inheritance.

User friendliness: In order to retrieve the candidate classes from a library, the user only needs to input the corresponding argument type and value and the return type and value. It is very easy to use. Moreover, the interface is implemented using HTML and Applet and thus even a new user can still know how to use this system in a short time.

Transparency: The retrieval mechanism in this method is totally performed by the system once the user inputs his/her expected test program and the return data. The user does not need to understand the inner structure of the system.

4.2 Comparison

In chapter 1, we have discussed different retrieval approaches. Obviously, the proposed method is an improvement over the Niu and Park's execution-based retrieval, which belongs to the operational semantics category. The comparison with Niu and Park's methodology is as follows:

Similarities

- Both methods execute the test program on all the components in the library and compare the returned result and the expected result.
- Both component libraries store Java classes.
- Both methods support the class hierarchy.
- Both methods allow exact matching and appropriate matching.

Differences

- Our class libraries are "real world classes" rather than those only providing function such as calculation. Our classes can have user-built-in attributes and parameters.
- Our method can capture the complete behavior of the class by providing the constructor, modifier and observer with different input interfaces, and by organizing the input messages into a test program according to the input order. The precision is higher.
- Our method allows users to call different constructors to create various instances and test different messages rather than use the default constructor as Niu and Park's method. The recall is improved.
- Our method does not need users to consider the argument order during the input. This is handled by the system and the recall is higher.

- Our method stores the information of the class from the library during the first time loading and compares the test message with the stored information before execution. It saves loading and executing time and increases the efficiency.

4.3 Conclusion

In this thesis, an improved execution-based retrieval methodology is proposed for retrieving reusable class components from the Java class library. This method based on the user's expectation selects some of the sub-libraries, and executes the user's input data on the classes to retrieve the class candidates whose behaviors closely match the query. The mechanics of this method are described and the comparison with previous methods is made. Based on this method, a prototype called the EBCRS system is developed using HTML-JavaScript-Applet and Java Servlets. This system could be used to retrieve, browse and save the Java classes from the class library. It also allows the administrator to manage the class library such as adding and deleting classes into/from the class library.

Although some previous retrieval methods are designed for object-oriented classes, they do not actually refer to the OO classes or provide no implementation. The proposed methodology is the first one to retrieve the complete behavior of a class from the class library. It also largely improves the retrieval efficiency and effectiveness. The follows are the main achievements:

- A true OO component retrieval methodology is provided on the "real world" class library and a useful tool is implemented.
- Not only the default constructor but also the user-selected constructors are allowed to create instances of a class with which the test messages are sent for execution. The retrieval precision and recall have been greatly improved.
- Any order of arguments and an organization of the class library into several sub-libraries could largely improve the retrieval recall.
- Runtime storage of the method return type and parameter list could reduce the time complexity during retrieval execution.

- Characteristics of OO classes such as inheritance, overloading and overriding are fully considered.
- An HTML (Applet)-Java servlet implementation allows the system to be Internet ready.

4.4 Future Work

A number of ways are possible for further extension of the proposed methodology and implementation. The class library could be organized using the type (functionality) based method, which could largely improve the system efficiency. Although the non-primitive type variables are allowed in the class definition as attributes or as parameters, no such types are allowed during input. A simple extension of the class library including those objects in the library could overcome this shortcoming. An access modifier matching may be added to before execution in order to achieve more accurate retrieval. An auto-generated test program could be added to for some users who may not want to input the data by themselves.

References

1. Aranow, E., 1997. Software reuse: because the waters are rising. http://www.reuse.com/waterris.html.
2. Atkinson, S. and Duke, R., 1995. Behavioral retrieval from class libraries. Australian Computer Science Communication, 17 (1): 13-20.
3. Biddle, R. and Tempero, E., 1995. Understanding OOP language support for reusability. In Proceedings of the 7th Annual Workshops on Institutionalizing Software Reuse, St. Charles, IL, USA, pp. 1-7.
4. Booch, G., 1994. Object-oriented analysis and design with application. Menlo Park, CA: Benjamin/Cummings, 2nd ed.
5. Cheng, J., 1993. Improving the software reusability in object-oriented programming. ACM SIGSOFT Software Engineering Notes, 18(4): 70-74.
6. Chou, S. H., Chen, J. Y. and Chung, C. G., 1996. A behavior-based classification and retrieval technique for object-oriented specification reuse. Software - Practice and Experience, 26(7): 815-832.
7. Damiani, E., Fugini, M. G. and Fusaschi, E., 1997. A descriptor-based approach to OO code reuse. Computer, 30(10): 73-80.
8. Etzkorn, L. H. and Davis, C. G., 1997. Automatically identifying reusable OO. Computer, 30(10): 66-71.
9. Fischer, M. K. and Struckmann, W., 1995. VCR: A VDM-based software component retrieval tool. http://www.cs.tu-bs.de/softech/papers/icse17-fmws.html, pp. 30-38.
10. Gonzalez, P. and Fernandez, C., 1997. A knowledge-based approach to support software reuse in object-oriented libraries. In Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97 Knowledge System Institute, Skokie IL, pp. 520-527.
11. Hall, R. J., 1993. Generalized behavior-based retrieval. In Proceedings of the 15th International Conference on Software Engineering, Baltimore, MD, USA, pp. 371-380.

12. Helm, R. and Maarek, Y. S., 1991. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In Proceedings of OOPSLA'91, pp. 47-61.
13. Honiden, S., 1993. Formal specification modeling in OOA. IEEE Software, 10(1): 54-66.
14. Isakowitz, T. and Kauffman, R. J., 1996. Supporting search for reusable software object. IEEE Transaction on Software Engineering, 22(6): 407-423.
15. Jeng, J. J. and Cheng, B. H. C., 1992. Using automated reasoning techniques to determine software reuse. International Journal of Software Engineering and Knowledge Engineering, 2(4): 523-546.
16. Jeng, J. J. and Cheng, B. H. C., 1995. Specification matching for software reuse: A foundation. In Proceedings of the ACM SIGSOFT Symposium on Software Reuse, Seattle, Washington, USA, pp. 97-105.
17. Jones, T. C., 1984. Reusability in programming: A survey of the state of the art. IEEE Transactions on Software Engineering, 10(5): 488-494
18. Krueger, C. W., 1992. Software reuse. ACM Computing Surveys, 24(2): 131-183.
19. Lee, S., Choi, H., Yang, Y. and Lee, S., 1999. Storage and management of object-oriented frameworks. IEEE Transactions on Software Engineering, 25(4): 762-767.
20. Liao, H. and Wang, F., 1993. Software reuse based on a large object-oriented library. Software Engineering Notes, 18(1): 74-80.
21. Lim, W. C., 1994. Effects of reuse on quality, productivity, and economics. IEEE Software, 11(5): 23-30.
22. Maarek, Y. S., Berry, D. M. and Kaiser, G. E., 1991. An information retrieval approach for automatic constructing software libraries. IEEE Transactions on Software Engineering, 17(8): 800- 813.
23. McClure, C., 1995. Model-driven software reuse: Practicing reuse information engineering style. http://www.reusability.com/papers2.html.
24. McManis, C., 1996. Code reuse and object-oriented systems: Using a helper class to enforce dynamic behavior. http://www.javaworld.com/jw-12-1996/jw-12-indepth.html

25. Meyer, B., 1988. Reusability: The case for object-oriented design. IEEE Software, 5(1): 50-64.
26. Mili, A., Mili, R. and Mittermeir, R., 1994. Storing and retrieving software components: A refinement-based system. In Proceedings of the 16th International Conference on Software Engineering, Sarrento, Italy, pp. 91-100.
27. Mili, H., Mili, F. and Mili, A., 1995. Reusing software: Issues and research directions. IEEE Transactions on Software Engineering, 21(6): 528-562.
28. Mili, A., Mili, R. and Mittermeir, R., 1998. A survey of software reuse libraries. Annals of Software Engineering, 5: 349-414.
29. Nelson, M. and Poulis, T., 1995. The class storage and retrieval system: Enhancing reusability in object-oriented systems. OOPS Messenger, 6(2): 28-36.
30. Niu, H., 1999. Execution-based retrieval of object-oriented components for reuse, M. Sc. Thesis, School of Computer Science, University of Windsor.
31. Niu, H. and Park, Y., 1999. An execution-based retrieval of object-oriented component. In Proceedings of the ACM Southeast Conference (ACMSC), pp. 160-167.
32. Podgurski, A. and Pierce, L., 1992. Behavior sampling: A technique for automatic retrieval of reusable components. In Proceedings of the 14th International Conference on Software Engineering, ACM Press, New York, USA, pp. 300-304.
33. Podgurski, A. and Pierce, L., 1993. Retrieval reusable software by sampling behavior. IEEE, ACM Transaction on Software Engineering and Methodology, 2(3): 286-303.
34. Prieto-Diaz, R., 1991. Implementing faceted classification for software reuse. Communication of ACM, 34(5): 88-97.
35. Prietro-Diaz, R., 1992. A domain analysis process model. SPC-92032, Software Productivity Consortium, Herndon, VA.
36. Prieto-Diaz, R., 1993. Status report: Software reusability. IEEE Software, 10(3): 61-66.
37. Rittri, M., 1990. Retrieving library identifications via equation matching of types. In Proceedings of the 10th International Conference on Automated Deduction, Kaiserslautern, Germany, pp. 603-617.

38. Standish, T. A., 1984. An essay on software reuse. IEEE Transactions on Software Engineering, 10(5): 494-497.
39. Steigerwald, R. A., 1992. Reusable component retrieval with formal specification. In Proceedings of the 5th Annual Workshop on Software Reuse, Victoria, British Columbia, Canada, pp. 21-27.
40. Wing, J. M., 1990. A specifier's introduction to formal methods. Computer, 23(9): 8-24.
41. Zaremski, A. M. and Wing, J. M., 1995. Signature matching: A tool for using software libraries. ACM Transactions on Software Engineering and Methodology (TOSEM), 20(4): 6-17.

Vita Auctoris

Name: Shaochun Xu

Place of Birth: Hubei, P. R. China

Date of Birth: February 5, 1965

Education:

1999-2001 M. Sc., School of Computer Science, University of Windsor, Windsor, Ontario, Canada

1997-1999 Post-doctorate Research Fellow, Department of Geological Sciences, University of Manitoba, Winnipeg, Manitoba, Canada

1991-1996 Ph. D., Department of Geological Sciences, University of Liege, Liege, Belgium

1984-1987 M. Sc., Chinese Academy of Geological Sciences, Beijing, China

1980-1984 B. Sc., Department of Geology, Peking University, Beijing, China

Appendix - A Part of Java Source Codes

```
/*************************************************************************/
/* Class AccessServlet handles the user login. It stores the userid and password entered */
/* by the user, accesses MS access database through JDBC, compares the userid and*/
/*password with those stored in database and dynamically creates a HTML page or      */
/* redirect to MainInterface.                                                        */
/*************************************************************************/

import java.io.*;
import java.util.*;
import java.net.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class AccessServlet extends HttpServlet
{

   // modify this constants
   final static String DB_NAME     = "Records";
   final static String DB_USER     = "";
   final static String DB_PASSWORD = "";
   // end modify

   String dbUrl   = "jdbc:odbc:" + DB_NAME;
   Connection con  = null;

   public void doPost (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
  {
         Statement stmt = null;
                  // get the servlet output stream
         ServletOutputStream out = res.getOutputStream();

         // set the content type
         res.setContentType("text/html");

         // get the query parameter from the html form or from the direct call
         String userid = req.getParameter("userid");
         String password=req.getParameter("password");

         try
         {
                  // Load the sun jdbc-odbc bridge driver
                  Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");

                  // connect to the jdbc-odbc bridge driver
```

```
            con = DriverManager.getConnection(dbUrl, DB_USER, DB_PASSWORD);

            // Create a Statement so we can submit statements to the driver
            stmt = con.createStatement();

            // Submit the query, creating a ResultSet object

            ResultSet rs = stmt.executeQuery ("select * from myTable where name
                    = '"+userid+"' and pass = '"+ password+"'");

            if (rs.next())
            res.sendRedirect("http://127.0.0.1:8080/Ebcrs/Client/MainInterface.html");
            else
            {
                    display (out, userid);
            }
            rs.close();
            // Close the statement
            stmt.close();
            // Close the connection
            con.close();
        }

        catch (SQLException ex)
        {
                System.out.println("SQL exception");
        }
        catch (Exception ex)
        {
                System.out.println("Exception: ");
        }
        out.close();

}

public void doGet (HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
{
        doPost (req, res);
}


private void display (ServletOutputStream out, String userid)
   throws SQLException, IOException
{

        //Display first the column headings
        out.println("<HTML><HEAD>");
        url=\"http://127.0.0.1:8080/Ebcrs/UserLogin.html\"");
        out.println("<TITLE><Acess Denied></TITLE></HEAD>");
```

```
        out.println("<BODY><center><B><h2>" + userid + ", You login and password are
        invalid.</h2></B><BR><BR>");
        out.println("<b><i><h1>Please <a
        href=\"http://127.0.0.1:8080/Ebcrs/Client/UserLogin.html\">try again
        </h1></i></b></a>");
        out.println("</center></BODY></HTML>");
}

} // end of servlet



/*****************************************************/
/* Class Daemon accepts the requests from client side,          */
/* and performs different operations according the the requests  */
/*****************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;


class Daemon extends Thread {

   private ServerSocket  servletSocket;
   private DaemonHttpServlet servlet;

   public Daemon (DaemonHttpServlet servlet) {
        this.servlet = servlet;
}

   public void run () {

        try{
                //create a server socket to accept connection
                servletSocket=new ServerSocket (servlet.getSocketPort());
        }
        catch (Exception e) {
                servlet.getServletContext().log(e, "Problem establishing server socket");
                return;
        }

        try{
                while (true)
                {
                        try{
                                servlet.handleClient(servletSocket.accept());
                        }
                        catch (IOException e)
```

```
                {
                        servlet.getServletContext().log(e, "Problem establishing server
                        socket");
                }
        }

    } catch (ThreadDeath e) {
        //when thread is killed, close the server socket
        try{
                servletSocket.close();
         }
        catch(IOException ioe)
        {
                servlet.getServletContext().log(e, "Problem closing server socket");
        }
    }
  }
}

/**************************************************************************/
/* Class DaemonHttpServlet is an abstract class extending HttpServlet, performing low-
level socket management. The init() method creates and starts a new Daemon thread
which is in charge of listening for incoming connections. The destroy() method stops the
thread. The Daemon thread begins by establishing a ServletSocket to listen on some
specific socket port. The socket port is determined by a call to the servlet's
getSocketPort() method. After establishing the ServerSocket, the Daemon thread waits
for incoming requests with a call to serverSocket.accept() */
/**************************************************************************/


import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;


public abstract class DaemonHttpServlet extends HttpServlet {

  protected int DEFAULT_PORT =1313;  // not static or final
  private Thread daemonThread;

  public void init (ServletConfig config) throws ServletException
  {
        super.init(config);
        //start a daemon thread
        try {
                daemonThread = new Daemon (this);
                daemonThread.start ();
```

```
        }
        catch (Exception e) {
                System.out.println("Problem starting socket server daemon thread");
         }
 }

 protected int getSocketPort() {

        try {

                return Integer.parseInt(getInitParameter("socketPort"));
        }
        catch (NumberFormatException e) {
                return DEFAULT_PORT;
        }
  }

  abstract public void handleClient(Socket client);

  public void destroy () {

        try {
                daemonThread=null;
        }
        catch (Exception e) {
                System.out.println("Problem stopping socket server daemon thread");
                }
  }

}
```

/**/
/* Class MainServlet class is the most important servlet which extends DaemonHttpServlet and implements a handleClient(Socket) method that spawns a new MainConnection thread. It accepts the client message and lets the handleClient method to perform action based on the request given. This servlet will be loaded at startup time of the server so that it can be accessed directly. */
/**/

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class MainServlet extends DaemonHttpServlet {
```

```
  public void init (ServletConfig config) throws ServletException
  {
        super.init(config);
  }


  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
        ServletException, IOException
 {

 }


  public void doPost (HttpServletRequest req, HttpServletResponse res) throws
        ServletException, IOException
  {
                doGet(req, res);
  }

  public void destroy () {
                super.destroy();
  }

  //handle a client's socket connection by spawning
  public void handleClient(Socket client) {
                new MainConnection (this, client).start();
  }

}

class MainConnection extends Thread {

  MainServlet servlet;
  Socket client;
  BufferedReader in = null;
  PrintWriter out   = null;

  MainConnection (MainServlet servlet, Socket client) {
        this.servlet=servlet;
        this.client=client;
        setPriority (NORM_PRIORITY-1);

        try {
                in = new BufferedReader(new InputStreamReader(client.getInputStream()));
                out = new PrintWriter(new BufferedWriter(new OutputStreamWriter
                        (client.getOutputStream())), true);
        } catch (IOException e) {
                System.out.println("IOException: " + e);
        }
}
```

```
public void run () {
	String flag = new String();

	/* Read the interface flag from client */
	 try {
		flag = in.readLine();
	} catch (IOException e) {
		System.out.println("IOException: " + e);
	}


	if (flag.equals("MAIN INTERFACE")) {
		System.out.println("Begin Processing Request from Main Interface...");

		String ClassLibString = null;

		/* The class list to be executed comes from class library */
		ClassLib.setFromClassLib(true);
		MatchInfo.initialize();

		 /* Send the class lib name list to the client */
		try {
			RandomAccessFile ClassLibListFile = new
			RandomAccessFile("C:/Ebcrs/Server/ClassLibList.txt", "r");
			while ((ClassLibString = ClassLibListFile.readLine()) != null) {
				out.println(ClassLibString);
			}
			ClassLibListFile.close();
		} catch (Exception e) {
			System.out.println("Error opening and reading class lib list: " + e);
		}
		out.println("END CLASS LIB LIST");

		/* Send the current class lib name to the client */
		out.println(ClassLib.getFullClassLibName());

		/* Read the current class lib name from the client */
		try {
			ClassLibString = in.readLine();
			in.readLine();  /* String "END MAIN INTERFACE" */
		} catch (Exception e) {
			System.out.println("Error getting info from client: " + e);
		}

		 /* Set up the current class library */
		if (ClassLibString == null || ClassLibString.compareTo("") == 0)
		ClassLibString = new String("C:/Ebcrs/ClassLib/ClassLib.txt");
		ClassLib.setClassLib(ClassLibString);
		System.out.println("End Processing Request from Main Interface...");
	}
```

```
if (flag.equals("BROWSE INTERFACE")) {
        System.out.println("Begin Processing Request from Browse Interface...");

        Vector classList = ClassLib.getClassList();
        String className = new String();

        /* Send the current class lib path to the client */
        out.println(ClassLib.getClassLibPath());

        /* Send the class list in the class library to client */
        for (int i = 0; i < classList.size(); i++)
                out.println(((String)classList.elementAt(i)).trim());
        out.println("END CLASS LIST");

        /* Send class info of a specific class to client */
        try {
                while ((className = in.readLine()).compareTo("END BROWSE
                INTERFACE")!= 0) {
                        Class myClass = Class.forName(className);

                        System.out.println("This class name ====" +className);

                        ClassInfo classInfo = new ClassInfo(myClass, out);
                        classInfo.generalInfo();
                        classInfo.fieldInfo();
                        classInfo.constructorInfo();
                        classInfo.methodInfo();
                        out.println("END CLASS INFO");
                }
        } catch (Exception e) {
                System.out.println("Exception: " + e);
        }
        System.out.println("End Processing Request from Browse Interface...");
}


else if (flag.equals("INPUT INTERFACE")) {
        System.out.println("Begin Processing Request from Input Interface...");

        TestPrg testPrg     = new TestPrg();
        String  message     = new String(); /* One line from client */
        Vector  wholeMessage  = new Vector(); /* Hold a whole test program */
        Vector  methodMessage = new Vector(); /* Hold a whole test message */
        Vector  classList   = new Vector();

        try {
                 while ((message = in.readLine()).compareTo(
                "END INPUT INTERFACE") != 0)
                {
```

```
        if (message.compareTo("END TEST PROGRAM") != 0)
        {
                wholeMessage.addElement(message);
        }
        else
        {
                for (int i = 0; i < wholeMessage.size(); i++) {
                        if (((String)wholeMessage.elementAt(i)).
                        compareTo("END MESSAGE INFO")== 0) {
                                /* Build a test message info */
                        /* Add the test message into test program */
                                testPrg.addMessageInfo(new
                                MessageInfo(methodMessage));

                                methodMessage.removeAllElements();
                                /* For next message */
                        }
                        else
                        {
                                methodMessage.addElement(
                                (String)wholeMessage.elementAt(i));
                        }
                }
        }
}
/* If the classes to be executed are from class library */
if (ClassLib.getFromClassLib())
        classList = ClassLib.getClassList();

/* If the classes to be executed are from candidate classes */
else
{
        classList = MatchInfo.getCandidateClasses();

        int i=0;
        while (i<classList.size())
        {
                System.out.println("Class=="+classList.elementAt(i));
                i++;
        }

}
/* Begin the execution-based retrieval */
MatchInfo.initialize();
Retrieve retrieve = new Retrieve(testPrg, classList);

/* The execution is Successful */
if (retrieve.execute()) {
/* Call the MatchInfo to get the candidate classes */
        MatchInfo.execute();
           out.println("END EXECUTION-BASED RETRIEVAL –
```

```
                        SUCCESS");
                }

                /* The execution is fail */
                else
                        out.println("END EXECUTION-BASED
                                RETRIEVAL -- FAIL");

                        wholeMessage.removeAllElements(); /* for next test program */

                } catch (Exception e) {
                System.out.println("Exception: " + e);
        }
        System.out.println("End Processing Request from Input Interface...");
}


else if (flag.equals("ADD INTERFACE")) {
        System.out.println("Begin Processing Request from Add Interface...");

        String className = new String();

        try {
                while ((className = in.readLine()).compareTo
                        ("END ADD INTERFACE")!= 0) {
                        ClassLib.addClass(className);
                        out.println("END ADD CLASS -- SUCCESS");
                }
         } catch (Exception e) {
                System.out.println("Exception: " + e);
                out.println("END ADD CLASS -- FAIL");
        }
        System.out.println("End Processing Request from Browse Interface...");
}


else if (flag.equals("DELETE INTERFACE")) {
        System.out.println("Begin Processing Request from Delete Interface...");

        String className = new String();
        String classLibName=new String();
        String line=new String();


        try {
                System.out.println("before read line");
                line=in.readLine();  // get the first library name
                System.out.println("the class lib 1=="+line);

                classLibName= "c:/Ebcrs/"+line+"/"+line+".txt";
                ClassLib.setClassLib (classLibName);
```

```
                while ((className = in.readLine()).compareTo
                                ("END DELETE INTERFACE")!= 0) {
                        ClassLib.deleteClass(className);
                        out.println("END DELETE CLASS -- SUCCESS");

                        line=(in.readLine()).trim();
                        System.out.println("the class lib 2=="+line);
                        classLibName= "c:/Ebcrs/"+line+"/"+line+".txt";
                        ClassLib.setClassLib (classLibName);

                }
        } catch (Exception e) {
                System.out.println("Exception: " + e);
                out.println("END DELETE CLASS -- FAIL");
        }
        System.out.println("End Processing Request from Delete Interface...");
}


else if (flag.equals("CANDIDATE INTERFACE")) {
        System.out.println ("Begin Processing Request from Candidate Interface...");

        /* Send the current class lib path to the client */
        out.println(ClassLib.getClassLibPath());

        /* Send the candidate classes to client */
        Vector CandidateClasses = MatchInfo.getCandidateClasses();
        for (int i = 0; i < CandidateClasses.size(); i++)
                out.println((String)CandidateClasses.elementAt(i));
        out.println("END CANDIDATE CLASSES");

        try {
                String message = null;

                /* If client requests class info */
                while ((message=in.readLine()).compareTo(
                "END CANDIDATE INTERFACE") != 0 &&
                        message.compareTo("FILTER CANDIDATE CLASSES
                        BUTTON") != 0 ) {
                        Class myClass = Class.forName(message);
                        ClassInfo classInfo = new ClassInfo(myClass, out);
                        classInfo.generalInfo();
                        classInfo.fieldInfo();
                        classInfo.constructorInfo();
                        classInfo.methodInfo();
                        out.println("END CLASS INFO");
                }

                /* If the client wants to filter the candidate classes */
```

```
                    if (message.compareTo("FILTER CANDIDATE CLASSES BUTTON")
                    == 0) {
                        ClassLib.setFromClassLib(false);
                        message = in.readLine();
                        /* "END CANDIDATE INTERFACE" */
                    }
                } catch (Exception e) {
                 System.out.println("Exception: " + e);
                }
                System.out.println
                    ("End Processing Request from Candidate Interface...");
                }

        try {
                in.close();
                out.close();
                client.close();
        } catch (IOException e) {
                System.out.println("IOException: " + e);
        }
    }

}

/*****************************************************************************/
/* Class Retrieve handles all the retrieval requests. It executes the test program according
to the order of messages and type of messages (constructor, observer and modifier). It
stores the method's signature in a vector during the first time loading from the server.
Each time it compares the argument list and return type between the test message and
loaded classes before execution. It then compares the returned value and the expected
result */
/*****************************************************************************/

import java.io.*;
import java.lang.reflect.*;
import java.util.*;

public class Retrieve {

  TestPrg testPrg  = new TestPrg();
  Vector  classList = new Vector();
  int totalMethods;  /* total methods in a class and/or its superclass(es) */
  int totalConstructors;
  final int RUNTIME = 60;   /* the whole execution time allowed in second */



/* Constructor -- accepts a test prgram and a class list */
```

```
public Retrieve (TestPrg testPrg, Vector classList) {
        this.testPrg   = testPrg;
        totalMethods   = 0;
        totalConstructors=0;
}

/* Execute the test program on the class list */

public boolean execute () {
        Class myClass  = null;

        /* Use a timer to handle endless execution */
        GregorianCalendar totalTime = new GregorianCalendar();
        totalTime.add(Calendar.SECOND, RUNTIME);

        for (int i = 0; i < classList.size(); i++) {
         /* If time expired, stop the process */
                GregorianCalendar currentTime = new GregorianCalendar();
                if (currentTime.after(totalTime)) return false;

                String className = (String)classList.elementAt(i);

                try {
                        myClass = Class.forName(className);
                } catch (Exception e) {
                System.out.println(className + " Exception: " + e);
                }

                try{
                        executeTestPrg(myClass);
                }
                catch (Exception e) {
                        System.out.println(" One Exception: " + e);
                }
         }
        return true;
}

/* Method executeTestPrg executes the test program on one class by
executing the test program on each method in the class (and/or its
superclasses), and stores the match information into MatchInfo class */

private void executeTestPrg (Class myClass) {
  boolean match         = false;
  int    matchMethods   = 0;
  int    nonMatchMethods = 0;
  int    maxTotalMethods = 0;
  String  matchedMethodDesc = new String();
  Object instanceOfClass   = null;  //
  String storedClass =new String (myClass.getName()) ;
        // because myClass may change to superclass
```

```
  /* Matched methods in a class and/or its superclass(es) */
  Vector matchedMethodList = new Vector();

  for (int i = 0; i < testPrg.getNumOfMessages(); i++) {
    /* Execute a test message on a class */

    matchedMethodDesc = executeMessageOnClass(
        testPrg.getMessageInfo(i), myClass, matchedMethodList, instanceOfClass);

    if (matchedMethodDesc.compareTo("NO MATCH FOUND") != 0) {
      matchedMethodList.addElement(matchedMethodDesc);
      matchMethods++;
    }
  }

  maxTotalMethods=totalMethods+totalConstructors;


  /* save match info, exclude the classes without match methods */
  if (matchMethods > 0) {
        MatchInfo.addMatchInfo(storedClass,
                matchMethods);
        System.out.println("Send to MatchInfo: " + storedClass + " " +
                matchMethods);
  }
}


/* Method executeMessageOnClass executes a test message on a class to
find a match. If a match is not found, the program will search to its
superclass(es) to find a match. The matched method name is returned. */

public String executeMessageOnClass (MessageInfo message, Class myClass,
                        Vector matchedMethodList, Object instanceOf) {

  Class   thisClass = null;
  try{
        thisClass=Class.forName(myClass.getName());
   }
   catch (Exception e) {
        System.out.println("exceptttt");
  }

  boolean match    = false;
  Vector  methodsInSubclass = new Vector();
  Object  instanceOfClass   = instanceOf;

  totalMethods = 0; // total methods in this class and/or its superclass(es)
  totalConstructors =0;
```

```
//execute the message on constructor  if it is constructor
 if (message.getReturnType().equals("None"))  {

        Constructor constructors[]=thisClass.getDeclaredConstructors();
        totalConstructors +=constructors.length;

        for (int i = 0; i < constructors.length; i++) {

                match = executeMessageOnConstructor(message, constructors[i],
                thisClass, instanceOfClass);
                if (match) return constructors[i].toString();
        }
 }
 else {   //execute on methods
        while ((thisClass.getName()).compareTo("java.lang.Object") != 0) {
                Method methods[] = thisClass.getDeclaredMethods();
                totalMethods += methods.length;

                if (instanceOfClass==null)   //if no constructor is called
                {
                        try {
                                instanceOfClass = thisClass.newInstance();
                        } catch (Exception exception) {
                                System.out.println(thisClass.getName() + " Exception: " +
                                        exception);
                                return "NO MATCH FOUND";
                        }
                }

                for (int i = 0; i < methods.length; i++) {
                 /* If a method is overridden in the subclass, skip it */
                if (methodsInSubclass.contains(methods[i].getName())) continue;

                /* If a method is already matched before, skip it*/
                /* use toString() rather getName() to catch the exact method if there
                 are several methods in a class with same name */
                if (matchedMethodList.contains(methods[i].toString())) continue;

                        match = executeMessageOnMethod(message, methods[i], thisClass,
                                instanceOfClass);

                        if (match) return methods[i].toString();
                }

 /* Otherwise, executes its superclass to find a match */
 thisClass = thisClass.getSuperclass();

 //because instanceOfClass now is for superclass
 instanceOfClass=null;

 methodsInSubclass = addMethods(methodsInSubclass, methods);  //this may not right
```

```
        } //end of while
    } //end of else
    return "NO MATCH FOUND";
}


/* Method executeMessageOnConstructor executes a test message on a class constructor to find a
match */

public boolean executeMessageOnConstructor (MessageInfo message, Constructor constructor,
                        Class thisClass, Object instanceOfClass) {

    boolean match = false;

    int     numOfMessageParameters = message.getNumOfParameters();
    String[] messageParameterTypes  = new String[numOfMessageParameters];
    Object[] messageParameterValues = new Object[numOfMessageParameters];

    String[] messageParameterTypes1  = new String[numOfMessageParameters];  //for sorting use

    message.getParameterTypes().copyInto(messageParameterTypes);
    message.getParameterTypes().copyInto(messageParameterTypes1);  //inorder to sort

    message.getParameterValues().copyInto(messageParameterValues);

    /* Compare numbers of parameters */
    Class parameterTypes[] = constructor.getParameterTypes();
    System.out.println("constructor=="+constructor.toString());

    // if this is no-arg constructor

    System.out.println("no aragume= "+ numOfMessageParameters+" no parameter= "+
        parameterTypes.length);


    if (numOfMessageParameters==0&&parameterTypes.length==0)
    {
        try {
                instanceOfClass = thisClass.newInstance();
                // instanceOfClass = constructor.newInstance();
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
        return true;
    }

    if (numOfMessageParameters != parameterTypes.length) return false;

    /* Compare each parameter types in order*/
```

```
int parameterLength =parameterTypes.length;
int j;
for (j = 0; j < parameterTypes.length; j++) {
        match = matchType(messageParameterTypes[j].toString(),
                parameterTypes[j].toString());
        if (! match) break;
}
if (j==parameterLength)
{
        try {
                instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
    return true;   //if it matches
}

/* Compare each parameter types in reverse order*/
for (j = 0;  j < parameterLength; j++) {

  match = matchType(messageParameterTypes[j].toString(),
                parameterTypes[parameterLength-j-1].toString());
  if (! match) break;
}

if (j==parameterLength)
{
        messageParameterValues=reverse (messageParameterValues);
                //change the order of value
        try {
                instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
        return true;   //if it matches
}


/* if there is less than 3 parameters then no match  */
if(parameterLength<3)
                return false;

/* if there are more than three parameters, we increase more chances
order=1, second-first-.....last
order=2  third--second-first....
order =3 first-third-second....
order=4  third--first-second-...
order=5  second-third--first--fourth-...
order=6  fourth-second-third--first---...
```

```
please note each change will infect later change */

/* order=1 second-first-.....last */
messageParameterTypes1=changeTypeOrder (messageParameterTypes1, 1);
for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes1[j].toString(),
                parameterTypes[j].toString());
        if (! match) break;
}
if (j==parameterLength)
{
        messageParameterValues=changeValueOrder (messageParameterValues, 1);
        try {
                instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
        return true;   //if it matches
}

//order =2 third-second-first..., the different messageParameterTypes
messageParameterTypes1=changeTypeOrder (messageParameterTypes1, 2);
for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes1[j].toString(),
                parameterTypes[j].toString());
        if (! match) break;
}
if (j==parameterLength)
{
        messageParameterValues=changeValueOrder (messageParameterValues, 2
        try {
                instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
        return true;   //if it matches
}

//order =3, first-third-second-..., the different messageParameterTypes
messageParameterTypes1=changeTypeOrder (messageParameterTypes1, 3);
for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes1[j].toString(),
                parameterTypes[j].toString());
        if (! match) break;
}
if (j==parameterLength)
{
        messageParameterValues=changeValueOrder (messageParameterValues, 3);
        try {
```

```
                instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
        return true;   //if it matches
}

//order =4, third--first--second-..., the different messageParameterTypes
 messageParameterTypes1=changeTypeOrder (messageParameterTypes1, 4);
 for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes1[j].toString(),
              parameterTypes[j].toString());
        if (! match) break;
}
 if (j==parameterLength)
 {
        messageParameterValues=changeValueOrder (messageParameterValues, 4);
        try {
                instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
        return true;   //if it matches
}

//order =5, second--third--first---..., the different messageParameterTypes
 messageParameterTypes1=changeTypeOrder (messageParameterTypes1, 5);
 for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes1[j].toString(),
              parameterTypes[j].toString());
        if (! match) break;
}
 if (j==parameterLength)
 {
        messageParameterValues=changeValueOrder (messageParameterValues, 5
        try {
                instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
                System.out.println(thisClass.getName() + " Exception: " + exception);
                return false;
        }
        return true;   //if it matches
}

if(parameterLength<4)
        return false;

/*fourth-second--third--first--... */
messageParameterTypes1=changeTypeOrder (messageParameterTypes1, 6);
```

```
  for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes1[j].toString(),
              parameterTypes[j].toString());
        if (! match) break;
  }
  if (j==parameterLength)
  {
        messageParameterValues=changeValueOrder (messageParameterValues, 6);
        try {
              instanceOfClass = constructor.newInstance(messageParameterValues);
        } catch (Exception exception) {
              System.out.println(thisClass.getName() + " Exception: " + exception);
              return false;
        }
        return true;  //if it matches
  }
  return false;
}

/* Method executeMessageOnMethod executes a test message on a class method to find a match
*/

public boolean executeMessageOnMethod (MessageInfo message, Method method,
                        Class thisClass, Object instanceOfClass) {

  boolean match  = false;
  Class  methodReturnType  = null;
  Object  methodReturnValue  = new Object();

  String  methodReturnString = new String();

  /* Test message info */
  int     numOfMessageParameters = message.getNumOfParameters();
  String[] messageParameterTypes  = new String[numOfMessageParameters];
  Object[] messageParameterValues = new Object[numOfMessageParameters];
  message.getParameterTypes().copyInto(messageParameterTypes);
  message.getParameterValues().copyInto(messageParameterValues);
  String messageReturnType  = message.getReturnType();
  String messageReturnValue = message.getReturnValue();

  /* Compare numbers of parameters */
  Class parameterTypes[] = method.getParameterTypes();
  if (numOfMessageParameters != parameterTypes.length) return false;

  /* Compare the return types */
  methodReturnType = method.getReturnType();
  match = matchType(messageReturnType, methodReturnType.toString());
  if (! match) return false;

 /* Compare each parameter types in order*/
  int parameterLength =parameterTypes.length;
```

```
int j;
for (j = 0; j < parameterTypes.length; j++) {
  match = matchType(messageParameterTypes[j].toString(),
            parameterTypes[j].toString());
  if (! match) break;
}
if (j==parameterLength)
{
        try {
              methodReturnValue = method.invoke(instanceOfClass,
                    messageParameterValues);
        }
        catch (Exception exception) {
              System.out.println("From invoke Method " + method.toString()
               + " in Class " + thisClass.toString() + ": " + exception);
      }
      /* Compare the return values */
      if (methodReturnValue == null) methodReturnString = new String("NULL");
      else methodReturnString = methodReturnValue.toString();

      match = matchValue(messageReturnValue, methodReturnString);
      if (match) return true; /* One match is found */
}


/* Compare each parameter types in reverse order*/
for (j = 0;  j < parameterLength; j++) {

  match = matchType(messageParameterTypes[j].toString(),
            parameterTypes[parameterLength-j-1].toString());
  if (! match) break;
}
if (j==parameterLength)
{
      messageParameterValues=reverse (messageParameterValues);

      try {
              methodReturnValue = method.invoke(instanceOfClass,
                    messageParameterValues);
      }
      catch (Exception exception) {
              System.out.println("From invoke Method " + method.toString()
               + " in Class " + thisClass.toString() + ": " + exception);
      }

      /* Compare the return values */
      if (methodReturnValue == null) methodReturnString = new String("NULL");
      else methodReturnString = methodReturnValue.toString();

      match = matchValue(messageReturnValue, methodReturnString);
      if (match) return true; /* One match is found */
```

```
}

 /* if there is less than 3 parameters, then no match  */
if(parameterLength<3)
        return false;

/* if there are more than three parameters, we increase more chances
order=1, second-first-.....last
order=2  third--second-first....
order =3 first-third-second....
order =4 third-first-second...
order=5  second-third--first--fourth-...
order=6  fourth-second-third--first---...
please note each change will infect later change */

 /* order=1 second-first-.....last */
 messageParameterTypes=changeTypeOrder (messageParameterTypes, 1);
 for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes[j].toString(),
              parameterTypes[j].toString());
        if (! match) break;
 }
 if (j==parameterLength)
 {
    messageParameterValues=changeValueOrder (messageParameterValues, 1);
    try {
                methodReturnValue = method.invoke(instanceOfClass,
                      messageParameterValues);
        }
        catch (Exception exception) {
                System.out.println("From invoke Method " + method.toString()
                 + " in Class " + thisClass.toString() + ": " + exception);
        }

        /* Compare the return values */
        if (methodReturnValue == null) methodReturnString = new String("NULL");
        else methodReturnString = methodReturnValue.toString();

        match = matchValue(messageReturnValue, methodReturnString);
        if (match) return true; /* One match is found */

 }

 //order =2 third-second-first..., the different messageParameterTypes
 messageParameterTypes=changeTypeOrder (messageParameterTypes, 2);
 for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes[j].toString(),
              parameterTypes[j].toString());
        if (! match) break;
 }
```

```
if (j==parameterLength)
{
    messageParameterValues=changeValueOrder (messageParameterValues, 2);
        //change the order of value
        try {
                methodReturnValue = method.invoke(instanceOfClass,
                        messageParameterValues);
        }
        catch (Exception exception) {
                System.out.println("From invoke Method " + method.toString()
                 + " in Class " + thisClass.toString() + ": " + exception);
        }

        /* Compare the return values */
        if (methodReturnValue == null) methodReturnString = new String("NULL");
        else methodReturnString = methodReturnValue.toString();

        match = matchValue(messageReturnValue, methodReturnString);
        if (match) return true; /* One match is found */

}

//order =3, first-third-second-..., the different messageParameterTypes
messageParameterTypes=changeTypeOrder (messageParameterTypes, 3);
for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes[j].toString(),
              parameterTypes[j].toString());
        if (! match) break;
}
if (j==parameterLength)
{
    messageParameterValues=changeValueOrder (messageParameterValues, 3);
        //change the order of value
        try {
                methodReturnValue = method.invoke(instanceOfClass,
                        messageParameterValues);
        }
        catch (Exception exception) {
                System.out.println("From invoke Method " + method.toString()
                 + " in Class " + thisClass.toString() + ": " + exception);
        }

        /* Compare the return values */
        if (methodReturnValue == null) methodReturnString = new String("NULL");
        else methodReturnString = methodReturnValue.toString();

        match = matchValue(messageReturnValue, methodReturnString);
        if (match) return true; /* One match is found */
}

//order =4, third first-second-..., the different messageParameterTypes
```

```
messageParameterTypes=changeTypeOrder (messageParameterTypes, 4);
for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes[j].toString(),
                parameterTypes[j].toString());
        if (! match) break;
}
if (j==parameterLength)
{
    messageParameterValues=changeValueOrder (messageParameterValues, 4);
        //change the order of value
        try {
                methodReturnValue = method.invoke(instanceOfClass,
                        messageParameterValues);
        }
        catch (Exception exception) {
                System.out.println("From invoke Method " + method.toString()
                 + " in Class " + thisClass.toString() + ": " + exception);
        }

        /* Compare the return values */
        if (methodReturnValue == null) methodReturnString = new String("NULL");
        else methodReturnString = methodReturnValue.toString();

        match = matchValue(messageReturnValue, methodReturnString);
        if (match) return true; /* One match is found */
}

//order =5, second--third--first--..., the different messageParameterTypes
messageParameterTypes=changeTypeOrder (messageParameterTypes, 5);
for (j = 0;  j < parameterLength; j++) {
        match = matchType(messageParameterTypes[j].toString(),
                parameterTypes[j].toString());
        if (! match) break;
}
if (j==parameterLength)
{
        messageParameterValues=changeValueOrder (messageParameterValues, 5);
        try {
                methodReturnValue = method.invoke(instanceOfClass,
                        messageParameterValues);
        }
        catch (Exception exception) {
                System.out.println("From invoke Method " + method.toString()
                 + " in Class " + thisClass.toString() + ": " + exception);
        }

        /* Compare the return values */
        if (methodReturnValue == null) methodReturnString = new String("NULL");
        else methodReturnString = methodReturnValue.toString();

        match = matchValue(messageReturnValue, methodReturnString);
```

```
        if (match) return true; /* One match is found */
    }

    if(parameterLength<4)
        return false;

    /*fourth-third--first--second... */
    messageParameterTypes=changeTypeOrder (messageParameterTypes, 6);
    for (j = 0; j < parameterLength; j++) {
        match = matchType(messageParameterTypes[j].toString(),
            parameterTypes[j].toString());
        if (! match) break;
    }
    if (j==parameterLength)
    {
        messageParameterValues=changeValueOrder (messageParameterValues, 6);
        try {
            methodReturnValue = method.invoke(instanceOfClass,
                messageParameterValues);
        }
        catch (Exception exception) {
            System.out.println("From invoke Method " + method.toString()
             + " in Class " + thisClass.toString() + ": " + exception);
            return false;
        }

        /* Compare the return values */
        if (methodReturnValue == null) methodReturnString = new String("NULL");
        else methodReturnString = methodReturnValue.toString();

        match = matchValue(messageReturnValue, methodReturnString);
        if (match) return true; /* One match is found */
    }

    return false; /* no match is found */
}


/* Function matchType tests if two types are same or can be converted */

private boolean matchType (String type1, String type2) {

    /* Check if type2 is a primitive type */
    if (type2.equals("boolean")) type2 = new String("Boolean");
    if (type2.equals("byte"))    type2 = new String("Byte");
    if (type2.equals("short"))   type2 = new String("Short");
    if (type2.equals("int"))     type2 = new String("Integer");
    if (type2.equals("long"))    type2 = new String("Long");
    if (type2.equals("float"))   type2 = new String("Float");
    if (type2.equals("double"))  type2 = new String("Double");
    if (type2.equals("char"))    type2 = new String("Character");
```

```
if (type2.equals("void"))    type2 = new String("Void");

/* Get the type2 name without prefix description */
int location = type2.lastIndexOf(".");
if (location != -1)
  type2 = type2.substring(location + 1, type2.length());

/* Compare the two types */

if (type2.equals("Object")) return true;

if (type1.equals("Boolean")) {
  if (type2.equals("Boolean")) return true;
}
else if (type1.equals("Byte")) {
  if (type2.equals("Byte")    || type2.equals("Short") ||
     type2.equals("Integer") || type2.equals("Long")  ||
     type2.equals("Float")   || type2.equals("Double"))
    return true;
}
else if (type1.equals("Short")) {
  if (type2.equals("Short") || type2.equals("Integer") ||
     type2.equals("Long")  || type2.equals("Float")   ||
     type2.equals("Double"))
    return true;
}
else if (type1.equals("Integer")) {
  if (type2.equals("Integer") || type2.equals("Long") ||
     type2.equals("Float")   || type2.equals("Double"))
    return true;
}
else if (type1.equals("Long")) {
  if (type2.equals("Long") || type2.equals("Float") ||
     type2.equals("Double"))
    return true;
}
else if (type1.equals("Float")) {
  if (type2.equals("Float") || type2.equals("Double")) return true;
}
else if (type1.equals("Double")) {
  if (type2.equals("Double")) return true;
}
else if (type1.equals("Character")) {
  if (type2.equals("Character")) return true;
}
else if (type1.equals("String")) {
  if (type2.equals("String")) return true;
}
else if (type1.equals("Void")) {
  if (type2.equals("Void")) return true;
}
```

```
    return false;
  }


  /* Function matchValue tests if two values are same or within the tolerance */

  private boolean matchValue (String value1, String value2) {
    Double doubleValue1 = null;
    Double doubleValue2 = null;

    /* If the two vales are same */
    if (value1.equals(value2)) return true;

    if (value2 == null) {
      if (value1.compareTo("NULL") == 0||value1.equals("")) return true;
      else return false;
    }

    try {
      doubleValue1 = new Double(value1);
      doubleValue2 = new Double(value2);
    } catch (Exception e) {
      return false;
    }

    if (doubleValue1 != null && doubleValue2 != null) {
      if (Math.abs(doubleValue1.doubleValue() -
          doubleValue2.doubleValue()) <= 10E-5) return true;
    }

    return false;
  }



  /* Method addMethods adds the methods into a vector */

  public Vector addMethods (Vector methodInSubclass, Method[] methods) {
    for (int i = 0; i < methods.length; i++) {
      methodInSubclass.addElement(methods[i].getName());
    }
    return methodInSubclass;
  }

  //change the order of value for constructor use
  private Object[] reverse (Object [] messageParameterValues)  {
          int length=messageParameterValues.length;
          Object temp[]=new Object[length];

        for (int i=0; i<length; i++)
          {
```

```
                temp[i]=messageParameterValues[length-i-1];
        }
            return temp;
}

/*change the parameter type order in different way
 order=1, second-first-.....last
  order=2  third--second-first....
  order =3 first-third-second....
  order=4  third-first--second-...
  order=5  second-third--first--fourth-...
  order=6  fourth-second-third--first---...
*/
private String[] changeTypeOrder (String [] messageParameterTypes, int order) {
        int length = messageParameterTypes.length;
        String [] temp = new String [length];
      int i=0;

        switch (order) {
                case 1:
                        temp[0]=messageParameterTypes[1];
                        temp[1]=messageParameterTypes[0];
                        for (i=2; i<length; i++)
                                temp[i]=messageParameterTypes[i];

                        break;
                case 2: case 3: case 5:
                        temp[0]=messageParameterTypes[2];
                        temp[1]=messageParameterTypes[0];
                        temp[2]=messageParameterTypes[1];
                        for (i=3; i<length; i++)
                                temp[i]=messageParameterTypes[i];

                        break;

                case 4:
                        temp[0]=messageParameterTypes[1];
                        temp[1]=messageParameterTypes[0];

                        for (i=2; i<length; i++)
                                temp[i]=messageParameterTypes[i];

                        break;

                case 6:
                        temp[0]=messageParameterTypes[3];
                        temp[1]=messageParameterTypes[0];
                        temp[2]=messageParameterTypes[1];
                        temp[3]=messageParameterTypes[2];
                        for (i=4; i<length; i++)
                                temp[i]=messageParameterTypes[i];
```

```
                        break;
        }

        return temp;
}

/*change the parameter value order in different way
 order=1, second-first-.....last
  order=2  third--second-first....
  order =3 first-third-second....
  order=4  third-first-second...
  order=5   second--third-first-fourth--...
  order=6  fourth--second--third-first---...
*/
private Object[] changeValueOrder (Object [] messageParameterValues, int order) {
        int length = messageParameterValues.length;
        Object [] temp = new Object [length];
        int i=0;

        switch (order) {
                case 1:
                        temp[0]=messageParameterValues[1];
                        temp[1]=messageParameterValues[0];
                        for (i=2; i<length; i++)
                                temp[i]=messageParameterValues[i];

                        break;
                case 2:
                        temp[0]=messageParameterValues[2];
                        temp[1]=messageParameterValues[1];
                        temp[2]=messageParameterValues[0];
                        for (i=3; i<length; i++)
                                temp[i]=messageParameterValues[i];

                        break;

                case 3:
                        temp[0]=messageParameterValues[0];
                        temp[1]=messageParameterValues[2];
                        temp[2]=messageParameterValues[1];
                        for (i=3; i<length; i++)
                                temp[i]=messageParameterValues[i];

                        break;
                case 4:
                        temp[0]=messageParameterValues[2];
                        temp[1]=messageParameterValues[0];
                        temp[2]=messageParameterValues[1];

                        for (i=3; i<length; i++)
```

```
                    temp[i]=messageParameterValues[i];

                break;

            case 5:
                temp[0]=messageParameterValues[1];
                temp[1]=messageParameterValues[2];
                temp[2]=messageParameterValues[0];

                for (i=3; i<length; i++)
                    temp[i]=messageParameterValues[i];

                break;

            case 6:
                temp[0]=messageParameterValues[3];
                temp[1]=messageParameterValues[1];
                temp[2]=messageParameterValues[2];
                temp[3]=messageParameterValues[0];

                for (i=4; i<length; i++)
                    temp[i]=messageParameterValues[i];

                break;

        }

        return temp;
    }
}
```

```
/*****************************************************************************/
/* Class SaveFileServlet handles the save file action. Once client sends a message to save a
selected file from the selected folder and then the servlet will send the file to client folder.
*/
/*****************************************************************************/

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;


public class SaveFileServlet extends HttpServlet {

        public void doPost(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException
        {
                String path = "c:\\Ebcrs\\"+request.getParameter
                ("folder")+"\\"+request.getParameter("file");
```

```
        // ServletContext.getRealPath(request.getParameter("file"));

        File file = new File(path);
        response.setContentType("application/octet-stream");
        response.setContentLength((int) file.length());
        response.setHeader("Content-Disposition",
                "attachment; filename=\"" + file.getName() + "\"");
        InputStream in = new BufferedInputStream(new FileInputStream(file));
        OutputStream out = response.getOutputStream();
        byte[] buffer = new byte[4096];
        while (true)
        {
                int bytesRead = in.read(buffer, 0, buffer.length);
                if (bytesRead < 0)
                        break;
                out.write(buffer, 0, bytesRead);
         }
         out.flush();
         out.close();
         in.close();
    }

    public void doGet (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
    {
        doPost (req, res);
    }
}
```