Execution-based retrieval of object-oriented components for reuse.

Hongjian. Niu
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000 ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
EXECUTION-BASED RETRIEVAL OF

OBJECT-ORIENTED COMPONENTS FOR REUSE

by

Hongjian Niu

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

Windsor, Ontario, Canada
1999
© 1999 Hongjian Niu
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-52621-6
Abstract

Software reuse is recognized as one of the promising techniques to improve software development. A central problem associated with object-oriented software reuse is how to effectively search and retrieve the desired components from a class library. This thesis proposes a methodology of execution-based retrieval. By alternatively executing a test program on each class in a class library, the proposed retrieval system tries to locate those classes whose behaviours are similar to the desired behavior.

The system organizes the user-input test data into a test program. Then, it dynamically executes each message in the test program on each class in the class library to find the match information for the class which includes the numbers of matched methods and non-matched methods. The very basic execution is trying to find a match between a message and a class method in the argument numbers and orders, the arguments’ types and values, and the return types and values. If a match is not achieved for a message, the system will search the superclasses of the executed class to find a match. Finally, two strategies are implemented to retrieve the candidate classes by examining the match information of the classes.

A retrieval engine is implemented in Java with client/server module and multithreading feature. Several GUIs are provided for input and output, and also for validating the user input or selection. By searching classes in a class library, the system is trying to find the
candidate classes with the desired behavior if they exist, or find the best approximation relative to the desired behavior if no such class exists. Also, candidate classes can be narrowed down by repeating execution on the previous results. The retrieval system also provides the functionality of class browsing, insertion and deletion. It is the first time that a methodology of execution-based retrieval of object-oriented components and its full-integrated implementation are provided.
Dedication

This paper is dedicated to my wife Crystal and my daughter Helena, for their love and support.
Acknowledgements

I wish to thank Dr. Young Park, whose excellent instruction and advice on both the courses and the thesis greatly stimulated my interesting in the information technology, and will benefit my future endeavor. He also significantly contributed to the quality and consistency of this thesis by thorough reviews and by providing excellent feedback.

Special thanks to Dr. Indra Tjandra for providing innumerable suggestions and comments for improvement.

I also would like to give special thanks to Dr. Steven Atkinson in University of Queensland at Australia, who discussed with me the key aspects of the methodologies.

Finally I acknowledge the support of the Java Team at Sun Microsoft, whose prompt and accurate feedback for the Java language greatly helped the code implementation.
# TABLE OF CONTENTS

Abstract iii

Dedication v

Acknowledgements vi

Chapter One  Introduction 1

1.1 Software Reuse 1

1.2 Retrieval of Reusable Software Components 3

1.3 WISER System 4

1.4 Retrieval from Object-Oriented Components 5

1.5 Overview of the Thesis -- Motivation for Execution-Based Retrieval of Classes 6

1.6 Organization of the Thesis 9

Chapter Two  Methodology of Execution-Based Retrieval 11

2.1 General Concepts of Retrieval by Execution 11

2.2 Approach to Execution-Based Retrieval 14

2.3 Message and Test Program 15

2.4 Class and Class Behavior 18

2.5 Match -- A Way to Compare Class Method and Test Message 22

2.6 Match Information of a Class -- Degree of Closest to a Test Program 23

2.7 Retrieval Strategies 25
2.7.1 Max, min and union functions 26
2.7.2 Strategy 1 – maximize, then minimize 27
2.7.3 Strategy 2 – minimize, then maximize 27
2.7.4 Union for retrieving classes 28

2.8 A Simple Example 29

Chapter Three A Prototype System 35

3.1 System Design 35
3.2 Client/Server Application 36
3.3 Construction of Class Library 39
3.4 Organization of Test Program 41
3.5 Execution 44

3.5.1 Input to the retrieval process 44
3.5.2 The definition of a type match 45
3.5.3 The definition of a method match 46
3.5.4 Executing a test program on a class 48

3.6 Finding Candidate Classes 50

3.6.1 Three assistant functions 50
3.6.2 Strategy One 51
3.6.3 Strategy Two 52
3.6.4 Union two sets of candidate classes to get candidate classes 53
3.6.5 Filtering the candidate classes 53

3.7 Class Browser - A Way to Specification-Based Retrieval 55
3.8 Class Library Maintenance 56
3.9 Error Handling 59
3.10 A Scenario 61

Chapter Four Evaluation and Conclusion 65
  4.1 Evaluation of the Methodology and Implementation 65
    4.1.1 Generosity 66
    4.1.2 Large-scale reuse 67
    4.1.3 Precision and recall 67
    4.1.4 Efficiency 68
    4.1.5 Reusable class selection 69
  4.2 Comparison to Other Retrieval Methods 69
    4.2.1 Similarity 70
    4.2.2 Differences 71
    4.2.3 Suitability 72
  4.3 Conclusion 73
  4.4 Future Work 76

Reference 78
Vita Auctoris 82
1.1 Software Reuse

Software reuse is the process of creating software systems from existing software assets, rather than building them from scratch (Krueger, 1992). Demands for lower software production and maintenance costs along with increased quality can only be met by widespread and systematic reuse (Somerville, 1997). There are some advantages in software development with reuse:

- In the aspect of software quality, reuse provides an opportunity to use the similar systems by capturing the functionality, and thus it can increase the system reliability, usability, efficiency, maintainability, and portability;
- Reuse can greatly reduce the software development costs, shorten the lead time, and thus increase the productivity;
- Overall process risk is reduced;
- Organizational standards can be embodied in reusable components.

The process of software reuse can be divided into three phases (Bai, 1995): classification/storage, location/retrieval, and adaptation/integration. Now, more and more software development companies adopt software reuse in their development
process, or even adopt the reuse-driven software development. In the reuse-driven process (Sommerville, 1997), the system requirements can be modified according to the reusable components available (Fig. 1-1), and the search for reusable components plays a very important role in the whole process.

Fig. 1-1 A reuse-driven development process (modified after Sommerville, 1997).

There are several different levels of software reuse (Somerville, 1997):

- *Application system reuse*: The whole of an application system may be reused;
- *Sub-system reuse*: Major sub-systems of an application may be reused;
- *Module or object reuse*: Components of a system representing a collection of functions may be reused;
- *Function reuse*: Software components that implement a single function may be reused.

A fundamental concern in all levels of software reuse is how to locate potential components for reuse.
1.2 Retrieval of Reusable Software Components

It has been anticipated for some time that comprehensive libraries of reusable software components will transform and mature the practice of software engineering. Furthermore, object orientation has been seen as the technology most likely to achieve this through the construction of class libraries, because it has these merits: information hiding, modularity, abstraction, inheritance, and so on. Thus reusability seems one of the most important promises of object-oriented approach (Booch, 1994). In practice, however, the problems associated with software reuse are formidable despite considerable recent effort (Atkinson and Duke, 1995).

The central problem associated with libraries of reusable software components is that of retrieval. A library is of little use unless it can be easily searched and suitable components retrieved. Until recently, most effort in solving the retrieval problem has gone into software classification, with emphasis upon domain-specific aspects such as the signature of operations, and the determination of static metrics that can be used when searching and retrieving. These strategies have somewhat simplified the retrieval problem by narrowing the scope of the universe of discourse.

The retrieval schemes proposed in the literature can be classified into three categories, based upon the technique used to index components during the search process:
• **Classification-based schemes**: Keyword, faceted, knowledge-based and feature classification schemes all seek relevant components using controlled vocabularies, properties and ontology external to the component;

• **Structural schemes**: Structural schemes, such as signature and specification matching, seek relevant components using their structural characteristics;

• **Behavioral schemes**: Behavioral retrieval schemes seek to take advantage of a distinguishing property of software -- its executability. Behavior-based schemes seek relevant components by comparing input and output spaces of components.

In these three retrieval schemes, the behavioural schemes is of great importance, as it is the dynamic behaviour of objects that is of the most concern when building applications (Atkinson and Duke, 1995). At heart, the problem of retrieval is to find those components that behave in some specified way when sent messages by their environment.

### 1.3 WISER System

WISER is a reusable software base system developed by the group in School of Computer Science at University of Windsor in a functional programming environment. It contains an evolving structured software repository of primitive Miranda functions. Components are classified on the number of arguments, and stored in a record structure. The software library was designed to provide retrieval based on function types, function semantics, and function executions. It belongs to the above-described function reuse.
1.4 Retrieval from Object-Oriented Components

A reusable component is any component that is specifically developed to be used, and is actually used, in more than one context (Karlsson, 1995). An important aspect of a reusable component is its generality, which means the component can be specified, parameterized or configured by different users. In object-oriented regime, a class is a general-purpose template which consists of the common behaviors of a set of related or similar objects. So it is more likely that a class rather than a function will be reused in the software development process.

The great advantage of object-oriented development is to use the existing object-oriented components (named classes, modules, or templates in different languages) in building the new system, but it is a great adventure to search the huge and complicated class libraries to find the desired components. Until recently, most effort in solving the retrieval problem of class library has gone into software classification or description-based retrieval, with emphasis upon domain-specific aspects (Liao and Wang, 1993; Nelson and Poulis, 1995).

For example, McManis (1996) built a helper class for a class library to let the user specify the key words to search the library for a match. Damiani et al. (1997) described COOR which is a descriptor-based approach. Liao and Wang (1993) improved the code
searching by reorganizing the object-oriented library with facet classification scheme and thesaurus. Nelson and Poulis (1995) developed a CSRS system which uses the object-oriented database management system to query the specific classes. These strategies have somewhat simplified the retrieval problem by narrowing the scope of searching, but they are neither accurate nor practical to implement. Atkinson and Duke (1995) proposed an abstract methodology for behavioral retrieval of class library, but did not provide the implementation.

1.5 Overview of the Thesis -- Motivation for Execution-Based Retrieval of Classes

The principle of behavioural retrieval from software libraries was suggested by Podgurski and Pierce (1992). The basic idea of execution-based (behaviour) retrieval is that the system executes each component on the test data provided either by the user or generated by the system and then determines the components to be retrieved by comparing the results with the known outputs. Two ideas were proposed: Sampling behaviour (Podgurski and Pierce, 1992) and Generalized behaviour-based retrieval (Hall, 1993). The two ideas were both based on the retrieval of function components.

However, few efforts have been spent on the behavioral retrieval of object-oriented components, which are mainly classes. Atkinson and Duke (1995) proposed an abstract methodology for behavioural retrieval of class library. Firstly, in their proposal, an
abstract model of classes and class behaviour is defined, laying the foundation for a
general theory of behavioural retrieval, independent of any particular language
formalism. In particular, a notion of approximate behaviour is made precise. Secondly, a
notion of failure is included in the behavioural model, and the idea that failures can be
removed is incorporated into the underlying partial order between behaviours. Thirdly, a
prescription for behavioural retrieval involving the join and meet lattice operations is
given.

Atkinson’s methodology provided a way to theoretically search and locate the potential
classes from a class library by executing the user-input test data on the classes, and
comparing the result class behaviors to the desired behavior. It is the first time that a
mechanism of retrieval by class execution was proposed. However, there are still
something left in their methodology:

- The methodology treats each class as an individual component, so it retrieves the
  potential classes based on the behavior of the individual classes. However, in object-
  oriented schema, every class is in a position of class hierarchy, which means a class is
closely related to other classes. When we consider the behaviour of one class, we
must also consider its superclasses or subclasses.

- Generally, a class method is deterministic, which means it responses the same
  message with the same way. It is different to Atkinson’s methodology.

- No implementation was provided, as it is very difficult for the methodology. So the
  methodology is somewhat impractical to be used in real system development.
In this thesis, a new methodology is proposed which extends Atkinson’s methodology in the following aspects:

- The proposed methodology is based on the match information between the user-desired behaviour and the class behaviors after executing a user input test program on the classes in the class library. It is reasonable, easy to implemented, practical and efficient.

- Considering the class inheritance. Besides executing all the methods in a class, the retrieval system also searches its superclass to find a match. This makes the retrieval engine object-oriented.

- The methodology provides a way to narrow down the retrieved classes for more specific candidate classes. This gives the user a way to recursively process until the result components are satisfied.

- Constructing a practical implementation using Java language. A client–server engine lets the user input the tests messages, and retrieve the candidate classes. The engine is a dynamic one as it provides the functionality of adding components to and deleting components from the class library.

- A class browser enables the user to peep the description and detailed information of each class in the class library at any stages of retrieval. This is very important for the user who is unfamiliar to the class library or who has no concrete knowledge about what he or she really wants to retrieve.
• The retrieval engine checks the type (both the input types and output types) match before retrieval, which facilitates the retrieval process because the execution on each class is very time-consuming. Counting the description of the classes, the retrieval system is actually a combination of the specification-based, type-based and execution-based retrievals.
• Java programming language and client/server module make it easy to put the whole system on the internet, which enables the user to perform the retrieval from remote sources.

1.6 Organization of the Thesis

This thesis is organized into four chapters.

Chapter 1 gives a brief introduction of the software reuse, retrieval of reusable components, retrieval of object-oriented components, and the overview of the thesis.

Chapter 2 discusses the general concept of retrieval by execution. A methodology based on the class behaviors and match information of classes is provided with the definitions and detailed description.

Chapter 3 gives a prototype of implementation in Java language. A client/server module is outlined. All the modules used in the retrieval system are systematically described with
several samples. Also, the maintenance of class library, class browser, and error handling are discussed.

Chapter 4 compares the proposed methodology with the former researches and other retrieval tools. It also concludes problems left and the future works in this area.
Chapter Two

Methodology of Execution-Based Retrieval

Retrieval of reusable object-oriented components that satisfy given requirements is a critical step in software reuse after the class library is built up. An important property of software is that the software can be executed to transfer inputs to outputs. The main idea of the proposed methodology is that the classes are identified by executing the classes on a user-input test program and the results are compared to the desired results.

This chapter detailed addresses the general concepts and methodology of the execution-based retrieval, and also describes each phase of the retrieval.

2.1 General Concepts of Retrieval by Execution

There are three main conditions for software development with reuse (Sommerville, 1997):

- It must be possible to find appropriate reusable components;
- The user of the components must have confidence that the components will behave as specified and will be reliable;
• The components must have associated documentation to help the user understand them and adapt them to a new application.

The above first and second conditions are closely related. Not only every retrieval methodology or retrieval tool should be efficiently used and executed, but it must also return the candidate components that behave exactly or closely to what the user expects.

Unfortunately, most retrieval tools, as they mainly depend on the specification or description of the components, separate the retrieval process and the behavior of the components. The software developers must know the specification of their system or know how to translate their requirements into specification, which is both difficult and ambiguous. After users locate the potential components in the library by using specification, it is still not very clear whether the potential components will behavior as they expected, as the specification is very abstract and different to the concrete behavior. Because users have no way to know the exact behavior prior to use these components, later they may find nothing, or some of these components are away from their expectation, and so they need go back to do the retrieval.

That is where the behavioral retrieval comes into consideration. It combine the first and second conditions by directly letting the user execute their test data on the components, and to see immediately the behavior of the components. This will guarantee that the retrieved components behave the same or close way to the users' expectation.
Execution-based retrieval takes advantage of the most essential property of a class — its **executability**. A class is defined as a collection of objects, where each object has a common set of operations and responds in a similar way to a given program. In this aspect, a test program can be viewed as a pseudo object that contains several predefined methods and their desired outputs. So the purpose of the retrieval aims to find a template which can accept the test program and generate the similar outputs. In order to get the similar outputs, the direct way is to execute the templates (classes) and compare the results with the desired ones.

The general concept of retrieval by execution is as follows. When a user tries to build a program, he or she may need to search an unfamiliar and huge-volume class library to find some classes for reuse rather than write from scratch. What the user only knows is the functionality or behavior of the program. Instead of randomly searching the class library or figuring out some ambiguous keywords or specifications, he or she can use some test data as an input, letting the retrieval system to locate the potential classes whose behaviors are closest to the desired behavior, by alternatively executing the user input data on each class in the library. If the retrieved classes are not specific, the system can narrow down the searching by only executing the retrieved classes. In this way, finally the user will find his or her desired classes, or the system returns nothing indicating no classes are close enough to the user expected.

The execution flow is shown in Fig. 2-1.
2.2 Approach to Execution-Based Retrieval

In the class library, if a class exists whose behavior exactly matches the desired behavior input by the user, the retrieval engine will yield this class. Otherwise, the retrieval engine yields those classes whose behaviors are similar to the desired one, as these classes are likely to be easily modified to behave as desired, or they can be inherited and overloaded to yield the desired behavior. The process of execution-based retrieval of classes can be divided into five phases:

- Construction of class library.
• **Organization of user inputs.** The user is responsible for several test messages, each of which includes the actual parameters and desired return result. The input information is stored in the retrieval engine as a test program including a desired behavior.

• **Execution.** By executing the test program on each class in the class library, and by comparing the actual behavior of the class and the desired behavior of the test program, a match information is achieved (including the matched methods and non-matched methods) for each class.

• **Retrieval.** Two strategies are used to select the classes which behave closest to the desired behavior. Strategy 1 first finds those classes which have maximum matched methods to the test program, within which locates those classes which have minimum non-matched methods. Strategy 2 finds those classes which have minimum non-matched methods to the test program, within which locates those classes which have maximum matched methods.

• **Union** the results of two strategies, eliminate the duplicates, and get the candidate classes.

The process of the whole execution-based retrieval is shown in Fig. 2-2.

### 2.3 Message and Test Program

A class is a template which contains a set of operations for a set of similar objects. The corresponding operation in every object responses a given message in a similar way.
Fig. 2-2 The process of the execution-based retrieval

In order to use the proposed methodology, we need to redefine the **message** in the following way:
A message is a name of operation and the set of all possible input and output values associated with this operation.

Here, besides the actual parameter values, we extend the meaning of message to include the user-desired return value from the message, called the desired output of this message. A message can have a name, a default name, or no name. The most important here is a message must have a set of actual input and output values to be sent to a class. More clearly, only the input values are to be sent to a class for execution, the desired output is used for comparison after the execution (See later). In actual programming, a value must have a type associated with it, so the type information of the values is also included in the message. Table 2-1 lists three typical messages.

| Test Program MyTestProgram has three messages: |
| Message1 { |
|   (String, "Specification and Implementation"), |
|   (character, 'C'), |
|   (integer, 5), |
|   (integer, 7) |
| Message2 { |
|   (float, 235.0034), |
|   (float, 56), |
|   (Void, "NULL") |
| Message3 { |
|   (double, -23.75E-5) |
| |
| Table 2-1 Three sample messages comprise a test program |
Message1 resembles a normal function, in which there are three parameters,
"Specification and Implementation", 'c' and 5, and the desired output is 7. In Message2, there are two parameters, 235.0034 and 56, and the desired output is NULL (no real output), which is a case of procedures in a class. In Message3, there is no parameter, and only a desired return value of -23.75e-5, which imitates some special functions, like RANDOM() .

Then a test program and a desired behavior is defined as:

A test program is a set of messages to be sent to a class. A desired behavior of a test program is a set of desired outputs from the messages in the test program.

Generally, a class contains several class methods. In order to discover its behavior thoroughly, several different messages are need to be sent to the class. These messages are combined into a single test program, from which all the desired outputs are combined into a desired behavior. This will facilitate the further execution. For example, the combination of the three messages in Table 2-1 is a test program MyTestProgram with the following desired behavior:

MyDesiredBehavior = {(float, 7), (Void, "NULL"), (double, -23.75e-5)}

2.4 Class and Class Behavior
Generally, class behavior is defined by description. For example, class “Stack” is described as “First in, last out”, and class “Queue” is described as “First in, first out”. But this definition is neither unambiguous nor formal, so is very difficult to implement into the retrieval system. In order to clearly understand the class behavior, a more accurate or even quantitative definition should be used. Here, a class behavior is defined as:

*A class behavior is the set of responses from the class methods when a test program is sent to the class and executed on these class methods.*

There are three aspects important in this definition.

First, *a class behavior mainly depends on the class methods rather than the class fields in the class.* Generally, the class variables are used to store information and to transfer information within the class. A class is classified by its object behavior, which can be represented by a state transition diagram (Booch, 1994). An object changes states when one or more of its operations are executed. Thus, executing a class’s methods demonstrates its object behavior. So class methods can be used to represent class object behaviors.

As there are only description and implementation in a class method, a message is needed from the outside environment to be executed on this method to see how the method reacts
to it. So second, a class behavior is mainly achieved against a certain test program. If we say a class behavior, we refer to a certain test program this class behavior respects to. Also, this means that a class may have several different or unrelated class behaviors with respects to several different test programs. For example,

```java
int plus (int a, int b) -- return a + b
int multiple (int a, int b) -- return a * b
int indexOf (String string, char ch) -- return the position of first occurrence of the specific character in the string
int lastIndexOf (String string, char ch) -- return the position of last occurrence of the specific character in the string
```

If we send a message `{2, 2}` to both plus and multiple methods, we will get the same result of 4. If we send a message `"I am a student", 'd'` to both indexOf and lastIndexOf methods, we will get the same result of 10. One way to avoid this situation is to recursively execute different test programs on the same class, which is discussed in the later section.

Finally, the definition shows that the number of elements in a class behavior is the same as or less than that in the executed test program. If every message is accepted by the class, we will get the same number of elements in the class behavior. However, there are reasons that a message is rejected, such as no prototypes of methods match the message. In this case, some messages can not be executed on the class and no values returned for
these messages, so the number of elements in a class behavior will be less than that in the executed test program. For example, executing the above test program in Table 2-1 on the class in Table 2-2 will generate a class behavior shown in Table 2-2.

Class MyClass contains four public methods:

- integer index (String string, String substring, integer fromIndex)
  -- Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
- float add (float a, float b)
  -- Return the sum of two float values
- public integer getLength (String string)
  -- Returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.
- integer index (String string, String substring)
  -- Returns the index within this string of the first occurrence of the specified substring.

By Executing The test program MyTestProgram on this class, we get a class behavior:

```java
MyClassBehavior = { (integer, 7), (float, 291.0034) }
```

Table 2-2 A class and its class behavior
2.5 Match -- A Way to Compare Class Method and Test Message

The most basic operation in the whole execution-based retrieval is to execute a message on a class method to find a match. A match is defined as follows.

A match between a message and a class method is that when executing the message on the class method, we can get the same output as or similar output to the desired output of the message.

By the definition above, a match is achieved when the following conditions are met:

- The class method is a public method, rather than an abstract method, as only a public and concrete method can be called and executed from outside the class.
- The argument number in the message is the same as that in the class method.
- Each formal parameter type in the class method is the same as or the supertype of the corresponding actual parameter type in the message.
- When the message is executed on the class method, no error or exception occurs.
- The return type from the class method is the same as or the subtype of the message.
- The return value of the class method after the execution is the same as or in the acceptable tolerant range to that of the message.
In object-oriented concept, a class is actually a data type. The supertype requirement means that if the type of one formal parameter in the class method requires a type A, then the type of the corresponding actual parameter in the message should also be type A or the type of the subclasses of the class A. Most object-oriented programming languages adapt the concept of **inclusion polymorphism**, which means that if S is a subtype of T (S ⊆ T), then a value known to be in Type S can safely be used whenever a value of type T is expected. For example, in Java, an Integer type can be used wherever an Object type is needed without any error, as Object is the superclass of Integer. The details of type conversion will be discussed in next chapter.

In the above example, there is a match between message1 in MyTestProgram and indexOf method in MyClass, as they meet the above definition. There is no match for message2, because the return type and value are different from that of add method in MyClass, although it is executed. There is no match for message3, as it is rejected by MyClass.

### 2.6 Match Information of A Class – Degree of Closest to A Test Program

The core of the methodology is to compare a class behavior and a desired behavior to find a match information. A **match information** of a class is defined as following.
A match information of a class regarding to a test program is composed of three parts: the class name, matchInfo (the number of matches), and nonMatchInfo (the number of nonmatches when the test program is executed on the class and/or its superclasses.

The match information of a class is achieved by the following process (Table 2-3):

- Loop each message in the test program
- Execute the message on the class (loop each class method but with Check&Skip) to find a match. If a match is not found, go to the superclasses to find a match, until a match is found, or no further superclass is available.
- If a match is found, count one match.
- After executing all the messages in the test program, count the total matches as matchInfo, and the total methods in the class and/or its superclass(es) minus matchInfo as nonMatchInfo.

Table 2-3 Executing a test program on a class for the match information

In the second step above, before a message is executed on a class method, if this method name already appears in one of the subclasses of the class, then this method will be skipped and not be executed. This Check&Skip step is used to ensure that we will not execute a class method if it is overloaded in one of its subclasses. This is very important. If the loop reaches a class method which has the same name as one method in the subclasses of this class, this means this method is overloaded in the subclass and no
match is achieved in the subclasses (otherwise the loop will stop by the above operation). In this case, we cannot execute this class method for a match.

The other Check&Skip is to skip the method which is already matched to a previous message in the test program. If two or more messages match the same class method, we cannot count this class method more than once (otherwise we will get the wrong number of matches). So if a class method matches a message, the system will skip it when executing the following messages.

In the above example, supposed MyClass has no superclasses for the simplicity, after MyTestProgram is executed on MyClass, we will find 1 match (message1 and indexOf), and 3 (Total methods in MyClass is 4, minus 1 match) nonmatch. So the match information for MyClass is:

\[
\text{MyMatchInformation} = \{ \text{MyClass}, 1, 3 \}
\]

### 2.7 Retrieval Strategies

After we have the match information of all the classes in the class library, we will try to determine which classes are candidate classes. A candidate class is defined as follows.

A candidate class is such a class that has the maximum matches and minimum nonmatches to a test program, after the test program is executed on this class.
It is reasonable for the above definition, because the candidate classes by the above definition will be the classes closest to the user expected. If a class is exactly match to the test program, the methodology will locate it first. Two strategies are use here to locate the candidate classes from a class library:

Strategy 1 finds those classes which have maximum matches, within which locates the classes which have the minimum nonmatches.

Strategy 2 finds those classes which have minimum nonmatches, within which locates the classes which have the maximum matches.

Generally, the two strategies will not select identical classes, so both are used in order to get the complete set of classes wanted.

2.7.1 Max, min and union functions

The max function is used to extract the maximal sequences from a set of values. If we have a set of value sorted ascending, this function will return the last several values in the set. For example, if one set of values is \{3, 3, 6, 9, 2, 8\}, the max function will return \{8, 9\}. In the retrieval process, the max function is used to retrieve a subset of classes which have the maximum matches from a set of classes.
The **min** function is used to extract the minimal sequences from a set of values. If we have a set of value sorted ascending, this function will return the first several values in the set. For example, if a set of values is \( \{3, 3, 6, 9, 2, 8\} \), the min function will return \( \{2, 3\} \). In the retrieval process, the min function is used to retrieve a subset of classes which have the minimum nonmatches from a set of classes.

The **union** function accepts two sets of values, returns the distinct common values within these two sets. For example, if one set of values is \( \{3, 3, 6, 9, 2, 8\} \), and the other set is \( \{7, 8, 3, 5, 9\} \), the union function will return \( \{3, 9, 8\} \).

### 2.7.2 Strategy 1 -- maximize, then minimize

Strategy 1 is used in the following way:

- Call max function on all the classes and their matches, and get a subset of classes which have the maximum matches.
- Call min function on this subset of classes and their nonmatches, and get a set of candidate classes which have the minimum of nonmatches.

### 2.7.3 Strategy 2 -- minimize, then maximize

Strategy 2 is used in the following way:

- Call min function on all the classes and their nonmatches, and get a subset of classes which have the minimum of nonmatches.
• Call max function on this subset of classes and their matches, and get a set of
candidate classes which have the maximum matches.

2.7.4 Union for retrieving classes

Together, Strategies 1 and 2 yield the candidate classes which are most easily modified to
obtain the desired behavior when they are executed. Then we need to union the two sets
of classes, eliminate the duplicate, and get the desired candidate classes.

Sometimes the candidate classes do not satisfy the user by the following reasons:

• Low precision: the candidate classes are too many, some of which may not closely
related to the desired behavior.

• Low recall: the candidate classes are too few, which means some good candidates
may be missed.

• There are no candidate classes at all.

• The candidate classes are not what the user expected.

For the first reason, the user may choose either to browse the candidate classes to find
what they wanted, or let the system to filter out the candidate classes to get more specific
classes, by providing some more test data. For the rest reasons, the user may simply
execute again the program by specifying different set of test data.
2.8 A simple Example

In this section, a small and arbitrary class library is constructed to demonstrate the methodology.

Consider a sample class library `MyClassLibrary` that consists of only six string processing classes named `StringProcess`, `StringProcessA`, `StringProcessB`, `StringProcessC`, `StringProcessD`, and `StringProcessE` (Fig. 2-3).

![Diagram of class library](image)

**Fig. 2-3** A sample class library. The arrows show the inheritance among classes.

The descriptions of the methods are as follows:
Class StringProcess:

- int index (String string, String substring) returns the index within this string of the first occurrence of the specified character.

Class StringProcessA inherits StringProcess:

- int index (String string, character ch) returns the index within this string of the first occurrence of the specified substring.

Class StringProcessB inherits StringProcess:

- int index(String string, String substring, int fromIndex) returns the index within this string of the first occurrence of the specified substring, starting at the specified index;
- int lastIndex (String string, String substring) returns the index within this string of the rightmost occurrence of the specified substring;
- int lastIndex (String string, String substring, int fromIndex) returns the index within this string of the last occurrence of the specified substring. The returned index indicates the start of the substring, and it must be equal to or less than fromIndex;
- String concat (String string1, character ch) adds the character to the end of the string.

Class StringProcessC:
- int index (String string, String substring, int fromIndex) returns the index within this string of the first occurrence of the specified substring, starting at the specified index;

- int lastIndex (String string, String substring) returns the index within this string of the rightmost occurrence of the specified substring.

Class StringProcessD inherits StringProcessA:

- int indexOf (String string, String substring, int fromIndex) returns the index within this string of the first occurrence of the specified substring, starting at the specified index;

- String getSubstring (String string, character ch, int beginIndex) returns a new string that is a substring of this string. The substring begins at the specified index and extends to the end of this string, and excludes the specific character;

- int getLength (String string) returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.

Class StringProcessE inherits StringProcessA:

- int indexOf (String string, String substring, int fromIndex) returns the index within this string of the first occurrence of the specified substring, starting at the specified index;

- boolean stringStartsWith(String string, String prefix) tests if this string starts with the specified prefix;
• boolean stringEndsWith(String string, String suffix) tests if this string ends with the specified suffix;

• String getSubstring (String string, int beginIndex) returns a new string that is a substring of this string. The substring begins at the specified index and extends to the end of this string.

Suppose the user is looking for a string processing class that can perform four basic operations of finding the specific index of a string. The methods available in the desired class are:

• int method1 (String string, character ch): Returns the index within this string of the first occurrence of the specified character;

• int method2 (String string, char ch, int fromIndex): Returns the index within this string of the first occurrence of the specified character, starting at the specified index;

• int method3 (String string, String substring): Returns the index within this string of the first occurrence of the specified substring;

• int method4 (String string, String substring, int fromIndex): Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

A mechanism is provided for the user to input a set of test data for each test method, including the parameters and return value. Then the system will construct the test data
into a test program (list of parameters) and a desired behavior (list of desired return values from these test methods) as follows:

\[
\text{MyTestProgram} = \{ ("Specification and Implementation", 'c'),
                     ("Specification and Implementation", 'c', 5),
                     ("Specification and Implementation", "and"),
                     ("Specification and Implementation", "and", 15) \}
\]

\[
\text{MyDesiredBehavior} = \{3, 2, 14, -1\}
\]

The retrieval system executes the test program on each class (including its superclass) in the library. The class behaviors of all classes are obtained as follows:

\[
\begin{align*}
\text{MyClassBehavior} & = \text{execute(MyTestProgram, StringProcess)} = \{14\} \\
\text{MyClassBehaviorA} & = \text{execute(MyTestProgram, StringProcessA)} = \{3, 14\} \\
\text{MyClassBehaviorB} & = \text{execute(MyTestProgram, StringProcessB)} \\
& = \{"Specification and Implementation", 14, -1\} \\
\text{MyClassBehaviorC} & = \text{execute(MyTestProgram, StringProcessC)} \\
& = \{14, "SPECIFICATION AND IMPLEMENTATION" \} \\
\text{MyClassBehaviorD} & = \text{execute(MyTestProgram, StringProcessD)} \\
& = \{3, "ifiation and Implementation", 14, -1\} \\
\text{MyClassBehaviorE} & = \text{execute(MyTestProgram, StringProcessE)} \\
& = \{3, 14, -1\}
\end{align*}
\]

The match information for all the classes is listed in Table 2-4 (Note that the statistics below for each class includes the number of methods in its superclass):
<table>
<thead>
<tr>
<th>Class Names</th>
<th>Number of Methods</th>
<th>Matched Methods</th>
<th>Unmatched Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringProcess</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>StringProcessA</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>StringProcessB</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>StringProcessC</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>StringProcessD</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>StringProcessE</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2-4 Match Information of the classes in class library

Using Strategy 1, first StringProcessD and StringProcessE are selected as they have the maximum number (3) of matched methods, then from these two classes StringProcessD is selected as it has the minimum number of unmatched methods (2).

Using Strategy 2, first StringProcess and StringProcessA are selected as they have the minimum number (0) of unmatched methods, from these two classes StringProcessA is selected as it has the maximum number of matched methods (2).

Finally, we get StringProcessA and StringProcessD as the candidate classes. It is clear that both StringProcessA and StringProcessD can be modified easily to get the desired class. To use StringProcessA, we need to extend two more methods. To use StringProcessD, we only need add one method.
Chapter Three
A Prototype System

A prototype retrieval system is implemented by Java, and a class library is constructed to test the system.

3.1 System Design

A number of features are considered during the system design. The design of the retrieval system focus on the following aspects:

- The retrieval system should be easily used, and a user-friendly GUI interface should be provided.
- The retrieval system should be multithreading, which permits several users to use at the same time.
- The retrieval system should be available through the Internet, which provides the user to interact with the remote retrieval system or remote class library.
- Since classes are evaluated by executing them on test program, so ready-to-execute class libraries should be maintained by the system.
• To help the users to familiar and understand the class library, and to examine the retrieved candidate classes, the system should provide a way to browse the classes in the class library.

• The system design relies on the concept of maintaining a growing class library, so the associated management facilities should be provided for supporting this design idea. This including the insertion of the class into the class library, deletion of the class from the class library, and a browser to examine the information of the classes in the class library.

• Since the system is execution based, a run-time exception handling is a must. This includes handling three run-time problems: error termination, non-termination, and long run-time.

• The system should support recursive retrieval, which means that the retrieval can be done several times and each time is performed based on the current results. In this way, the number of candidate classes can be further narrowed down until a satisfied set of candidates is achieved.

3.2 Client / Server Application

The prototype is implemented in a client/server module. The server side consists of the following modules:
• A server thread module ensures the functionality of multithreading. As the main
gate connected to the client side, it accepts the information from client side, calls the
corresponding modules to perform the related operation, and sends the results back to
the client side.

• A class library module stores the information of class libraries, such as the class
library list, the current (user selected) class library, its full path and name. Also, the
class library module performs operations of adding class to the class library and
deleting a class from the class library.

• A message information module stores the information of a test message, including
the number of parameters, parameter types and values, and the return type and value.

• A test program module stores all the user input test data, consisting of several test
messages.

• A class information module dynamically examines a given class, and returns its
class descriptor, superclass, field and method information.

• A retrieval module performs the core operation of the execution-based retrieval. It
accepts a test program, a desired behavior, and a class list, alternatively executes the
test program on each class in the class list, and compares the executed results with the
desired behavior. For each class, a match information is sent to the match information
module.

• A match information module accepts the match information of the classes in the
class library, uses two different strategies to get two sets of potential classes, unions
them to get the candidate classes.
The client side is mainly composed of several user interfaces for the user to input test data, and to display the corresponding results upon the user request.

- A **main interface** controls the whole process, and connects to the different interfaces for different operation. It lists the names of the class libraries for the user to select a class library for processing.

- An **input interface** provides the user a way to input the information of a test message, including the parameter types and values, and the desired return type and value. The user can input several test messages, also can lets the system perform the execution-base retrieval at any stages. Also, this interface performs the validation of the input data.

- A **class-browser interface** displays the class list in the specified class library. When the user select a class, the system will display the corresponding class information, including its class descriptor, superclass, field and method information.

- A **class-addition interface** provides the user to add a class to the class library, provided he or she gets the permission from the system administrator.

- A **class-deletion interface** provides the user to delete a class from the class library, provided he or she gets the permission from the system administrator.

- A **candidate class interface** displays the candidate classes retrieved from the class library by the retrieval system. The user can browse these classes for detailed information, save the classes, or let the system recursively perform execution on these candidate classes to narrow down the results for more specific candidate classes.
The interaction between the client and server is shown at Fig. 3-1.

![Diagram showing the interaction between Server Side and Client Side](image)

**Fig. 3-1** Client/server module of execution-based retrieval from class library. The single arrow means sending information, and the double arrow means exchanging information.

### 3.3 Construction of Class Library

Several sample class libraries are provided for the test of the system. Each class library has an entry in a **class library list file**, which is used by the system to get the class library information. Each class library is constructed as a subfolder, within which are the classes in this class library and a **class library information file** listing all the class names.
without any specific order. The file is used by the system to keep tracks on the classes when the retrieval, insertion and deletion are performed.

In order to focus on the retrieval methodology, the class library is organized in the following way:

- Each class is a standard Java class, and can only inherit classes from Java APIs or classes in the same class library, rather than from some other third-party packages. This is to ensure that if a superclass is needed for the retrieval, the system can find it.
- The majority of the classes are functioned to perform mathematical calculations, or to process strings and data structures, rather than to perform some complicated tasks, such as graphic drawing, input/output operation or networking. As discussed later, the system only accepts the primitive data types from the test program, so the above requirement will make the methods in the class take mainly primitive data types, and generally will return a value rather than nothing.
- The classes perform different tasks, but are related somehow. For example, some classes contain exactly same methods. In one class, there may be several versions of the same method. This is especially designed to test the functionality of the retrieval system.

The information of the specific class library, including the full path and name of the class library and the class list in this library is stored at the class library module. The current class library name is selected by the user from the Main Interface (Fig. 3-2).
Execution-based Retrieval System from Class Library

Welcome to the **Execution-based Retrieval System from Class Library**. The system provides you a way to execute a test program on a user-defined or default class library to retrieve the candidate classes.

**Class Library:**

- C:/Engine/Source/SourceProcess.txt
- C:/Engine/Calculation/Calculation.txt

---

**Browse Class Library** | **Execution-based Retrieval**

**Add Class to Class Library** | **Delete Class from Class Library**

**Save Class Library** | **Exit Retrieval System**

---

Fig. 3-2 The Main interface displays the class library list for the user to select the class library for execution.

### 3.4 Organization of Test Program

The user inputs the test data at the **Input Interface** (Fig. 3-2), and the data are organized into a test program by **test message** and **test program modules**.
In input interface, the user is responsible for inputting in the information of the test messages. For each test message, the user should select the type for each parameter, and enter its value. Also, the user should enter the desired return type and return value for this test message. In order to simplify the process, only the following data types are allowed.
for the types of parameters and return value, the \texttt{Void} data type is only allowed for the return value:

\begin{itemize}
\item \texttt{Boolean, Byte, Short, Integer, Long, Float, Double, Character, String, Void, Object}.
\end{itemize}

After the user types in a message, a \textbf{validation} is performed for the input. This is to validate:

\begin{itemize}
\item If the user types in some values rather than leaves every field empty.
\item If the input order is enforced. The user cannot enter a value without enter its previous value for the parameters.
\item If each value matches its corresponding type (Table 3-1).
\end{itemize}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Type} & \textbf{Validation Rule} \\
\hline
\texttt{Boolean} & Only values allowed are \texttt{true} and \texttt{false} \\
\hline
\texttt{Byte, Short, Integer, Long} & The value can be convert to the specific type without any errors \\
\hline
\texttt{Float, Double} & The value can be convert to the specific type without any errors \\
\hline
\texttt{Character} & The length of the value is 1 \\
\hline
\texttt{String} & Any value is valid as long as its length is great than 0 \\
\hline
\texttt{Void} & The length of the value is 0 \\
\hline
\texttt{Object} & any values \\
\hline
\end{tabular}
\caption{The validation for the user-input values and types}
\end{table}
After the validation, the message information is sent to the server and stores as an object of message information module. Also, the message information is added as an entry into the test program module.

In the input interface, the user may select to input the information of next test message, or he or she can inform the system to perform the execution based on the input at any time. When all the test messages are received by the server, they are organized into a test program, which is a list of test messages.

3.5 Execution

The retrieval engine is controlled by Retrieval module, which perform the core operations in the retrieval process (see Fig. 2-2).

3.5.1 Inputs to the retrieval process

The retrieval module accepts two inputs. One is a test program (TestProgram), which consists of test messages from the user input, and the other is a list of classes (ClassList) to be executed on. Because the retrieval module handles the retrieval from both the class library and the candidate classes, so there is no need to distinguish them in the retrieval engine. Rather, the system checks whether the classes come from class library or candidate classes by examining the interfaces in the client side. If the
execution is triggered by Retrieval Interface, the class library is used for the retrieval. Otherwise, if the execution is triggered from Candidate Interface, the candidate classes are used for the retrieval.

3.5.2 The definition of a type match

A type match does not mean that two types have to be exact the same. Most programming languages allow explicit or implicit widening type conversion (inclusion polymorphism), which converts a type to its widen scope type without any error. For example, a Short type can be converted to Integer type safely. So if an Integer type is required for a formal parameter, we can use Integer, Short or Byte safely.

The implementation accepts the widening type conversion in the type match comparison, which is performed in matchType function. The matchType function will return true if a former parameter in a class method and an actual parameter in a test message meet the following widening type conversion in Table 3-2.
<table>
<thead>
<tr>
<th>Formal Parameter Type</th>
<th>Allowed Actual Parameter Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>Byte</td>
<td>Byte</td>
</tr>
<tr>
<td>Short</td>
<td>Byte, Short</td>
</tr>
<tr>
<td>Integer</td>
<td>Byte, Short, Integer</td>
</tr>
<tr>
<td>Long</td>
<td>Byte, Short, Integer, Long</td>
</tr>
<tr>
<td>Float</td>
<td>Byte, Short, Integer, Long, Float</td>
</tr>
<tr>
<td>Double</td>
<td>Byte, Short, Integer, Long, Float, Double</td>
</tr>
<tr>
<td>Character</td>
<td>Character</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Object</td>
<td>Any types</td>
</tr>
<tr>
<td>Void</td>
<td>Void</td>
</tr>
</tbody>
</table>

Table 3-2 The former parameter types and their allowed actual parameter types

3.5.3 The definition of a method match

The whole aim of the execution is to find the match between a test program and the classes in a class library. The very basic operation is to execute a test message on a class method to see if a match can be found between them. A match between a test message (TestMessage) and a class method (ClassMethod) is defined in match function (Table 3-3):
**match** function return *boolean*

- If the number of formal parameters in *ClassMethod* is different from the number of actual parameters in *TestMessage*, return *false*
- Loop each formal paramemters
  - If **matchType** (formal parameter in *ClassMethod*, its corresponding actual parameter in *TestMessage*) return *false*, return *false*
- Execute the *ClassMethod* by using the actual parameters in *TestMessage*
- If **matchType** (the return type form executing *ClassMethod*, the desired return type in *TestMessage*) return *false*, return *false*
- If the return value form executing *ClassMethod* equals the desired return value in *TestMessage* or in the tolerant range, return *true*, else return *false*. For the *Boolean*, *Byte*, *Integer*, *Long*, *Character*, *String* and *Void* data types, an exact match is required. For the *Float* and *Double* data types, a tolerance of 10E-4 is allowed, as it is impractical to find the exact match for float and double values.

| Table 3-3 The **match** function |

After match function is defined, it is very easy to define the **executeTestMessage** function (Table 3-4). In the function, it is required the parameter orders are also matched.
**executeTestMessage** function accepts a test message *TestMessage* and a class *Class*,

```plaintext
return boolean
```

- Loop each class method *ClassMethod* in the *Class*
  - If match(*TestMessage, ClassMethod*) return true, return true
- Return false

Table 3-4 The **executeTestMessage** function

### 3.5.4 Executing a test program on a class

The core operation of the execution-based retrieval is to execute a test program (*TestProgram*) on a class (*Class*) to find the match information (*MatchInfo*). This is performed by **executeTestProgram** function (Table 3-5).

The **Check&Skip** step is very important. It will check two situations.

If the loop reaches a class method which has the same name as one method in the subclasses of this class, this means this method is overloaded in the subclass and no match is achieved in the subclasses (otherwise the loop will stop by the above operation). In this case, the class method cannot be executed for a match, so it will be skipped to next method.
**executeTestProgram** function return **void**

- Let the current class **CurrentClass** be **Class**
- Loop till **CurrentClass** is null
  - Get the test messages of **TestProgram**
  - Loop each test message **TestMessage** in **TestProgram**
    - **Check&Skip**
      - If **executeTestMessage**(**TestMessage**, **Class**) return **true**
        - Increase **matchMethods** by 1
      - Else If **match**(TestM, ClasssM) return **false**
        - Get the superclass of this class
        - Recursively call **executeTestMessage** function till either a match is found or no superclasses are available
- Count the total **matchMethods**
- Use the different between the total methods from all classes executed (may include **Class** and some or all of its superclasses) and the **matchMethods** as **nonMatchMethods**
- Put the **Class, matchMethods, nonMatchMethods** as an entry to the **MatchInfo** module if at least one **matchMethods** is gained

Table 3-5 Executing a test program on a class for the match information

The other is to skip the method which already matches to a previous message in the test program. If two or more messages match the same class method, it can not be counted more than once (otherwise we will get the wrong number of matches). So if a class
method matches a message, the engine will skip it when executing the following messages.

3.6 Finding Candidate Classes

The final step – retrieval of the candidate class – is performed by MatchInfo module. This module stores the name of the classes and their match info. Only the classes with at least one match method are stored. The system tries to locate the candidate classes by using the match information of the classes. In order to understand the retrieval process, three functions are needed.

3.6.1 Three assistant functions

The max function is used to extract the maximal sequences from a set of values. The min function is used to extract the minimal sequences from a set of values. The union function accepts two sets of values, returns all the values within these two sets without duplication.

max function: accepts a set of classes, return a subset

- Let an integer value NUM to be large enough
- Loop the classes
- If the match info of a class equals to NUM, add this class to a list LIST
• If the size of LIST is enough, return LIST

• Else decrease NUM by 1, continue loop till either the size of LIST is large enough, or NUM is decreased to 0.

**min** function: accepts a set of classes, return a subset

• Let an integer value NUM to be 0

• Loop the classes

• If the match info of a class equals to NUM, add this class to a list LIST

• If the size of LIST is enough, return LIST

• Else increase NUM by 1, continue loop till either the size of LIST is large enough, or NUM is high enough.

**union** function: accepts two sets of values, returns one set of values

• Declare a empty set MySet

• Loop the first set of values

  • If a value in the first set does not appear in MySet, add it to Myset

• Loop the second set of values

  • If a value in the second set does not appear in MySet, add it to Myset

• Return MySet

**3.6.2 Strategy One**

-51-
The retrieval engine uses two strategies to get the candidate classes, which is performed by \texttt{strategy1} and \texttt{strategy2} functions.

The \texttt{strategy1} first locates the classes with the most matched methods with the test program by using the \texttt{max} function, within which locates the classes with the least unmatched methods with the test program by using the \texttt{min} function (Table 3-6).

\begin{table}

\textbf{strategy1 function}

- Locate the class list \texttt{ClasssList} and their match info \texttt{MatchInfo}

- call \texttt{max(MatchInfo)} to locate the classes \texttt{FindClasses} with the maximal \texttt{matchInfo}

- call \texttt{min(FindClasses)} to locate the classes \texttt{CandidateClasses1} with the minimal \texttt{nonMatchInfo}

\end{table}

\textbf{Table 3-6 Strategy 1}

\subsection{3.6.3 Strategy Two}

The \texttt{strategy2} first locates the classes with the least unmatched methods with the test program, within which locates the classes with the most matched methods with the test program (Table 3-7).
strategy2 function

- Locate the class list ClasssList and their match info MatchInfo
- call min(MatchInfo) to locate the classes FindClasses with the minimal nonMatchInfo
- call max(FindClasses) to locate the classes CandidateClasses2 with the maximal matchInfo

Table 3-7 Strategy 2

3.6.4 Union two sets of candidate classes to get candidate classes

Because generally these two strategies yield different classes, union function is used to combine these two lists, and eliminate the duplicate ones. Finally, the classes after union operation will be returned as candidate classes:

CandidateClasses = union(CandidateClasses1, CandidateClasses2)

3.6.5 Filtering the candidate classes

The retrieved candidate classes are displayed in the Candidate Interface (Fig. 3-4). The user can view the detailed information of a class in the candidate classes, and save the classes. If the result is not very satisfied, the user can let the system recursively execute on the candidate classes to get more specific candidate classes. For the system, it is easy to recursively execute, only thing it needs to do is use the class list from the candidate
Candidate Classes

List of Candidate Classes:
- StringProcess
- StringProcessA
- StringProcessC
- StringProcessB
- StringProcessE

Information of the Selected Class:
- Name: StringProcessD
- Class/Interface: Class
- Superclass: StringProcess
- Primitive type: false
- Interfaces: None

Browse Class  Save Class
Filter Candidate Classes  Reset

Back to Main

done

Fig. 3-4 Candidate Interface displays the candidate classes after the retrieval process, and provides the user to filter the candidate classes to get more specific classes.
classes rather than the classes form class library as the class list when calling the execution retrieval module again.

3.7 Class Browser – A Way to Specification-Based Retrieval

The most difficult situation that could appear and the most unlikely is that the class library users have no concrete knowledge of the system and its components, and without a concrete knowledge what they are looking for. For these users, a way to browse all the components in the class library should be provided. It is most likely that after a first browsing step, the users have an idea of what they might reuse, and they can then switch their strategy to search the things.

In order to help the user to better understand the classes in the class library or the retrieved candidate classes, a class browser is provided with the system to show the detailed information of a given class (Fig. 3-5).

The displayed detailed information of a class includes:

- Class name, descriptor, if it is a class or interface
- Superclass name
- For each field, its name, descriptor
- For each method, the parameter type list
Fig. 3-5 The browse interface displays the detailed information of a selected class

### 3.8 Class Library Maintenance

It is very important that the class library is kept in a dynamic way. Also, at the very beginning, there should be a way to add classes into the class library. An Add Interface (Fig. 3-6) provides the functionality to add a class into the class library. When the user
open a class file, first the system will validate to see if the file exists, and if it is a valid Java class. Then the system will copy this file into the folder where the classes of class library are located. Then it will add one entry into the class library file. Of course, the Add Interface is mainly used by the system administrator. The user should get permission from the system administrator to add a class.

Fig. 3-6 Add Interface lets the user add a class into the class library
Class deletion is also very important. A **Delete Interface** (Fig. 3-7) is used to delete a class from the class library. For the purpose of keeping tracks, the real class file will not be deleted. Rather, a mark is added at the entry of this class in the class library file, thus just marking this class to be deleted. Of course, the Delete Interface is mainly used by the system administrator. The user should get permission from the system administrator to delete a class.

![Delete Interface](image.png)

Fig. 3-7 The Delete Interface lets the user delete a class from the class library
3.9 Error Handling

Since the retrieval system is execution-based, it is important to handle the run-time problems. Three problems arise immediately: *error termination, non-termination and long-run-time*. The system handles these problems in two aspects: **validation check and time-out interval**.

Validation check is mainly applied to input data to avoid invalid input leading run-time problem. In all the interfaces, the user-input data and user-selected data are validated against some rules. A **message box module** is implemented to prompt the user about the information, warnings, and error messages (Fig. 3-8).

![Input Interface](image1)
![Delete User](image2)

*Warning: Applet Window*

*Warning: Applet Window*

Fig. 3-8 A sample error message box and a sample warning message box

- If an interface can not be connected to the server for any reason, a message will tell the use to try again or check the connection.
- In Main interface, if the user does not specify a class library, the system will use the default one, which avoids problems associated with non-class library.
• In Browse Interface, if a user tries to browse or save a class without selecting one, an error message will prompt him or her to select a class first.

• In Input Interface, when the user clicks “Retrieve Classes” or “Next Method” buttons, before the input information is sent to the server, a series checks are performed:
  • If the user enters nothing, prompt he or she to input some values.
  • If the user enters a value for a parameter but does not enter the previous one, prompt he or she to do so.
  • If the type and value do not match for each parameter or the desired return result, an error message is issued; This includes the user does not select a type for a value.
  • For the non-termination and long-run-time problems, the interface will receive information form the server, and prompt to the user.

• In Add Interface, if the user specify an invalid class file, including wrong path, wrong file type, wrong file name extension, an error message is issued to let the user open another file.

• In Delete Interface, if a user tries to perform the deletion without selecting one class, an error message will prompt him or her to select one first.

• In Candidate Interface, if a user tries to browse or save a class without selecting one, an error message will prompt him or her to select a class first.

The server side mainly deals with the run-time errors and exceptions. Some exceptions are used by the system to perform matching operation. For example, when a message is
executed on a class method, the *IllegalArgument.Exception* is used to exclude this method for a match. Other exceptions, such as *FileNotFoundException* exception, will be used to send the corresponding messages to the client side.

A **Time-out Interval Module** is used to handle the non-termination and long-run-time problems. A good retrieval system should provide the user with results as soon as possible. If running a component takes a long time, the system will interrupt the execution before the process is completed and the corresponding message will inform the user to simplify the test data, to specify the different test data, or to split the execution into several steps by filtering the candidate classes.

### 3.10 A Scenario

Here a simple example is provided to show how the retrieval system works and how to evaluate and use the retrieved candidate classes.

There are some situations where we need to treat all data as strings. For example, in client/server interaction in Java, all the data flowed between the server and the client are packed as strings, regardless of their original data types. In order to process the strings, we may search the class library for some classes which can perform the operations between strings and other data types. In specific, we need a class which can perform the following 4 operations:
• Convert every data type to string;
• Compare different data types as they are strings;
• Concatenate different data types as they are strings;
• Calculate the length of every data type as it is string.

First, we construct a test program which contains 4 test messages corresponding to the above 4 operations:

• message1 = {(Float, -34.76), (String, "-34.76")}
• message2 = {(Int, 3427), (Object, 3427), (Boolean, true)}
• message3 = {(String, "abcd"), (Int, 234), (String, "abcd234")}
• message4 = {(Long, 986754), (Int, 6)}

After execution, the retrieval system returns the following candidate classes with match info:
ObjectOperation (2, 1), ObjectCompare (3, 4), ObjectEquals (3, 4).

Here we list the code of these 3 classes for explanation:

public class ObjectOperation {
    public void getObject (Object arg) {}

    public Object length ( Object object ) {
        return new Integer( object.toString().length() );
    }

    public Object concat ( Object first, Object second ) {
        return first.toString() + second.toString();
    }
}
public class ObjectCompare extends ObjectOperation {
    public boolean greater (Object arg1, Object arg2) {
        return arg1.hashCode() > arg2.hashCode();
    }

    public boolean greaterEqual (Object arg1, Object arg2) {
        return arg1.hashCode() >= arg2.hashCode();
    }

    public boolean less (Object arg1, Object arg2) {
        return arg1.hashCode() < arg2.hashCode();
    }

    public boolean lessEqual (Object arg1, Object arg2) {
        return arg1.hashCode() <= arg2.hashCode();
    }
}

public class ObjectEquals extends ObjectOperation {
    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);
    }
};

Two matches are found in Class ObjectOperation. Method length matches message4, and Method concat matches message3. Both Class ObjectCompare and ObjectEquals inherit from Class ObjectOperation, so they both have at least 2 matches. Also, Method greaterEqual in Class ObjectCompare matches message2, and Method notEquals in Class ObjectEquals matches message2.
But neither Method `greaterEqual` nor Method `notEquals` is what we want, as they only check if two objects do not equal. It is clear that we can get our desired class by creating a class which takes one of these 3 classes as a superclass, and extending two methods which perform the functionality of `message1` or `message2`. 
4.1 Evaluation of the Methodology and Implementation

The methodology and retrieval technique is primarily for improving software productivity in software reuse. This improvement can be evaluated according to the following criteria:

- Generosity. The technique should be suitable for use in as many situations as possible.
- Large-scale reuse. Reusing a large software component may save more time than reusing several smaller ones. Thus, large-scale reuse can save system analysis time.
- Retrieval precision and recall. Execution-based retrieval should be precise. Otherwise, a user may spend much time to understand the retrieved classes that are not reusable. Moreover, retrieval recall should be high. Otherwise, a user may specify a class from scratch, instead of reusing existing classes.
- Retrieval efficiency. Retrieving candidate reusable classes should be efficient. Otherwise, a user may spend much time waiting for retrievals.
- Reusable class selection. Guidance should be available to facilitate selection of appropriate candidate reusable classes, because many candidates may be retrieved.
We evaluate the methodology and retrieval technique according to the five criteria depicted above.

Typically, some experiments should be conducted to compare the use of the technique and building from scratch. Here, only a brief and non-qualitative evaluation is carried out.

4.1.1 Generosity

Theoretically, the methodology only needs a class library and a test program to execute and retrieve some class components, regardless of programming languages (as long as the language has the concept of modules, classes or templates) and development environment (no restriction to specific operating system). But in practice, as the class concept is so diverse in the different languages, the methodology may need to be modified to fit the specific language. For example, for the multi-inheritance in C++, the methodology must have a specific way to deal with it when it is implemented in C++. In different operating systems, the interactions between the user and the system are so different that they deserve individual-based consideration.

The implementation is written in Java, which is declared as a language of “writing once, run everywhere”. In this sense, the implementation can be used in several different platforms, and can be used through the Internet, which makes it possible for remote search and retrieval.
4.1.2 Large-scale reuse

As described in Chapter One, four levels of software reuse exist, in which the retrieval from class library belongs to the third level—module or object reuse. It is better than the function reuse in the sense of reuse scale. The user can get several classes rather than several functions from a same set of test data. Also, classes have a general concept and are especially constructed for reuse, and can be closely related by inheritance or semantics. This will save the user some time to figure out the relations and functionality.

It may be argued that class reuse is smaller in scale than that of system or subsystem reuse. It is ideal that a whole system or subsystem exists for reuse. The problem here is that there are very few general-purpose systems or subsystems available for reuse. Not only they are too difficult to construct, but also each system has so many specific aspects to be considered. It is very rare that general components in the reusable system or subsystem can meet the functional or non-functional requirements of different potential users. So till now, the system and subsystem reuse is mainly in internal delivery in some companies. For this aspect, the generosity of the proposed methodology will benefit its reuse scale.

4.1.3 Precision and recall

The technique retrieves classes based on their behaviours as represented by their
responses to the given test programs. Since classes that behave similarly are considered for reuse, a behavioral technique can improve retrieval precision. Moreover, filtering candidate classes gives the user a way to get more specific classes from a retrieved set of classes, thus can greatly improve the precision.

To increase retrieval recall, each class with at least one match to the given test program is put in the retrieval queue for later retrieval execution, which may decrease precision. Basically, the system looks a similar behavior rather than an exact match, so the recall is relatively high.

However, in the implementation for the methodology, the retrieval recall is adversely affected by some limitation of the Java language. In the execution process, the object of a class in the class library needs to be declared. As the class name is unknown, a dynamic declaration is needed. But in Java language, when we dynamically declare an object of a class, we can only use the default constructor of the class without parameters. As some classes have no default constructors, some candidate classes may be missed by using the Java implementation. In this sense the retrieval recall is not so high as expected in the methodology.

4.1.4 Efficiency

In order to get the high recall, the system loops each class in a class library to find the match information, which seems to decrease the efficiency. However, the technique
improves retrieval efficiency by doing type-checking before real execution, thus save some execution time. Also, the repeated execution will only focus on the previously retrieved candidate classes rather than the whole classes. So in general, the retrieval efficiency is high.

4.1.5 Reusable class selection

Before the retrieval process, a class browser is provided for a user to browse the detailed information of a class, which enables the user to have a general understanding for the class library. After the retrieval process, a user can also browse a class in the candidate class list, which can be used as a guide to select the appropriate candidates for reuse.

4.2 Comparison to Other Retrieval Methods

There exists a spectrum of retrieval methods, based upon the nature of the indices used as representations of components (Atkinson, 1995):

- **External Indices**: Keywords, faceted and feature-based classification techniques all seek to find relevant components based upon controlled vocabularies, properties and ontologies external to the class;
- **Internal Static Indices**: Structural-matching techniques, most notably signature and specification matching techniques, seek to find relevant components based upon...
elements of the structure of software components;

- **Internal Dynamic Indices**: Behavioral retrieval techniques seek to take advantage of the distinguishing property of software --- executability. Behaviour-based techniques seek to find relevant components by comparing input and output spaces of components.

The proposed methodology belongs to the third retrieval regime. Compared to other two methods, it has both similarities and differences.

### 4.2.1 Similarities

There are three major similarities between the proposed methodology and the other two retrieval methods.

Firstly, the execution-based retrieval method is defined by providing representations for test data (query), component, and match, so it has similar surface structure with that of others. An advantage of taking this uniform view of retrieval methods is to permit multiple retrieval methods to co-exist and plug together, thus providing a framework for a multiple-domain retrieval tool. Queries and test data can be pipelined because the result of a query is a library, which can be used with a different retrieval method to further narrow the search space.

Secondly, besides the execution-based retrieval which uses a very explicit input-output
model of component behaviour, the other retrieval methods all depend upon the
behaviour of the components in a library, even though the behaviour is represented
differently in each. For faceted-classification retrieval, the behaviour of the components
is concentrated implicitly into the terms used in each facet. For signature-matching
retrieval, the behaviour of the components is more dilute, residing in the type-signature of
each function of a module.

Thirdly, the methodology allows results of retrieval to be used as inputs to a component
adaptation engine. Variances between the test data and the retrieved candidate classes can
be analyzed (either in a pre-computed or dynamic fashion) to assist the library user to
adapt the retrieved component(s) into the context implied by the test program. This is
similar to other two retrieval methods. However, because behaviour is represented in
varying degrees of concentration, the adaptation engine will vary in the precision of the
advice given.

### 4.2.2 Differences

There are two major differences between the three retrieval methods.

Firstly, each retrieval method makes a different assumption about the nature of the
software components and indices used to represent them. The third column of Table 4-1
shows the different component indices for each retrieval method.
<table>
<thead>
<tr>
<th>Retrieval Method</th>
<th>Query</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faceted (Specification-based)</td>
<td>Name → Term</td>
<td>Name → Term</td>
</tr>
<tr>
<td>Signature (Structure-based)</td>
<td>bag Signature</td>
<td>bag Signature</td>
</tr>
<tr>
<td>Behavioral (execution-based)</td>
<td>Program * Behaviour</td>
<td>Class</td>
</tr>
</tbody>
</table>

Table 4-1 Different Query and Component Representations (After Atkinson, 1995)

For faceted retrieval, names and terms are chosen from a vocabulary that is constructed from the domain, not from the component itself. Signatures are constructed from types used in the components themselves. A class represents the dynamic behaviour of a component responding to program executions.

Secondly, Table 4-1 shows faceted and signature matching retrieval methods have the same query and component representations. This reflects the fact that both terms and signatures are static properties associated with components, and so, for ease of matching, queries and components can be statically represented in the same manner.

Execution-based retrieval sees the representation of the test data differs from the representation of the component, because behaviour depends on the program contained in the test data. As a result, the process of matching and selection becomes more computationally expensive, because the behaviour of each component is dynamically calculated using the program part of the query.

4.2.3 Suitability
These differences indicate that no single retrieval method can be expected to be best in all situations. For any given library search, some retrieval methods are more potentially profitable than others. The proposed methodology of execution-based retrieval is more appropriate for domains where components conform to a similar protocol, in order to maximize components with meaningful responses. Near-automatic adaptation and system construction can take place, because complete response differences can be derived from queries and components.

Classification retrieval methods are more appropriate for well-defined domains with standardized vocabularies or ontologies. Adaptation and system construction engines based on classification retrieval results are expected to require manual input, since the concentration of behaviour in keywords and terms is high. Structural retrieval methods are more appropriate for multiple domains, as long as the component representation method has the descriptive power to describe components in many domains. Adaptation and system construction can take place semi-automatically, with some adaptations being determined by the internal structural variance of queries and components.

4.3 Conclusion

In this thesis, the feasibility of retrieving classes is investigated by executing them on some test data input by users of class library. Based on the methodology, a prototype
system is designed and developed in Java language. Various tools are also provided by the retrieval system to allow users to insert, delete, and browse the classes. The execution-based retrieval is divided into three phases: the building of class library, the construction of test program, and the execution. The retrieval system is designed to support repeatable retrieval which means retrieval could be done several times. Also, a GUI is designed and developed for the system.

It is the first time for a methodology and its full implementation of the execution-based retrieval from class library. The methodology and implementation are more practical and reusable as the retrieval engine only searches for a similar behaviour rather than an exact match. Also, users have many different selections in almost every stage.

The previous methodology of class retrieval concentrates on the specification of the theoretical frameworks for behavioral retrieval, rather than giving optimized algorithm for the practical implementations (Atkinson and Duke, 1995). Considering the implementation of a methodology, there are number of practical considerations. Among them are the questions of how to determine correspondences between messages in the test program and operations of library classes, how much of the burden of calculation of desired outputs in behavior should rest with the library user, and how to provide efficient outputs of taking the behavioral projections. These questions are carefully considered at implementation.

Based on the work carried out in the thesis, a number of conclusions can be formed:
• The retrieval system provides a useful tool for program development based on reusable object-oriented components.

• To some degree, the execution-based retrieval proposed here improves the precision of retrieval in terms of the behavior of classes.

• The retrieval system supports repeatable retrieval. Each time the retrieval can be done based on the current results, so the number of candidates can be further narrowed down.

• The system allows users to update library classes by providing insertion and deletion features.

• The system is opened to the Internet by using Java programming language and client/server module.

There are still some things left both in the methodology and implementation.

• The methodology is not pure language-unrelated. As discussed earlier, for some programming languages with multi-inheritance, we need to modify the methodology.

• The methodology does not specify any data structures in the class library, which may affect the efficiency.

• In implementation, only primitive data and very few non-primitive data types are allowed as input data and output data type, which simplifies the processing but narrows its usability.

• The relative order of the arguments within a test message is required to be the same as
that of a class method for a match. Sometimes the relative order is not important. In this case the system should test each different combination of the arguments to find a match.

- Automatic test data generation is very useful for the users who are unfamiliar to the class library. It is not implemented here.

### 4.4 Further Work

There are a number of ways in which the methodology and the implementation presented in the previous two chapters can be extended and applied.

The methodology needs to be more general-purposed. Its class concept here should be extended to include abstract classes, interfaces, templates, modules, and other object-oriented components.

A simple extension allows the process to cater for a set of test programs as input, rather than a single program. Additionally, a scheme to return those classes which perform sub-tasks of the desired behaviour is needed, since more classes are relevant to solving the retrieval query at hand than those that solve it directly. Also, an automatic test data generation mechanism is needed to help users.

In implementation, non-primitive data types and user-defined data types need to be
concerned. A mechanism to let users select some classes for execution rather than all classes should be provided for increasing the efficiency. The structure of class library, the order of test messages, the order of arguments in a test message should be considered.

The eventual goal of this research is to specify a class-based software reuse repository by integrating a number of retrieval methods with other repository operations.


NAME: Hongjian Niu

PLACE OF BIRTH: Henan Province, P. R. China

DATE OF BIRTH: July 29, 1964

EDUCATION:

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

1984 – 1987 M. Sc., Department of Geology, Peking University, Beijing, China

1987 – 1991 Ph. D., Institute of Geology, Chinese Academy of Sciences, Beijing, China

1995 - 1996 Postdoctoral Research Fellow, Department of Earth and Planetary Sciences, Harvard University, Cambridge, Massachusetts, USA

1997 – 1999 M. Sc., School of Computer Science, University of Windsor, Windsor, Ontario, Canada