2004-04-23

IFSO: A Integrated Framework For Automatic/ Semi-automatic Software Refactoring and Analysis

Yilei Zheng
Worcester Polytechnic Institute

IFSO: A Integrated Framework For Automatic/Semi-automatic Software Refactoring and Analysis

by

Yilei Zheng

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

Yilei Zheng
April 30, 2003

APPROVED:

Dr. George T. Heineman, Thesis Advisor

Professor Gary Pollice, Department Reader

Dr. Michael A. Gennert, Head of Department

Abstract

To automatically/semi-automatically improve internal structures of a legacy system, there are several challenges: most available software analysis algorithms focus on only one particular granularity level (e.g., method level, class level) without considering possible side effects on other levels during the process; the quality of a software system cannot be judged by a single algorithm; software analysis is a time-consuming process which typically requires lengthy interactions.

In this thesis, we present a framework, IFSO (Integrated Framework for automatic/semi-automatic Software refactoring and analysis), as a foundation for automatic/semi-automatic software refactoring and analysis. Our proposed conceptual model, LSR (Layered Software Representation Model), defines an abstract representation for software using a layered approach. Each layer corresponds to a granularity level. The IFSO framework, which is built upon the LSR model for component-based software, represents software at the system level, component level, class level, method level and logic unit level. Each level can be customized by different algorithms such as cohesion metrics, design heuristics, design problem detection and operations independently. Cooperating between levels together, a global view and an interactive environment for software refactoring and analysis are presented by IFSO.

A prototype was implemented for evaluation of our technology. Three case studies were developed based on the prototype: three metrics, dead code removing, low coupled unit detection.

Acknowledgements

I would like to thank my advisor, Prof. George Heineman, for his support, advice, and encouragement throughout my graduate studies. It is lucky for me to find an advisor giving me freedom and trust to come up and develop the idea. It’s also a great opportunity to escalate my knowledge level.

My thanks also go to Prof. Gary Pollice for being the reader of this thesis, Prof. Dave Brown teaching me a lot of valuable knowledge for this thesis and my friend, Hong Ao, spend time discuss with me.

Last, but not least, I would like to thank my husband Jun for his support, encouragement and love during the past two years. My parents receive my deepest gratitude and love for their dedication in past years.

Contents

1 Introduction 1
  1.1 Motivation 1
  1.2 Problems To Solve 4
  1.3 Requirement 5
  1.4 Contribution 7
    1.4.1 Contribution of the LSR model 7
    1.4.2 Contribution of IFSO 8
    1.4.3 Contribution of prototype implementation 9
  1.5 Related Work 10
    1.5.1 Metrics 10
    1.5.2 Reverse-engineering 11
    1.5.3 Refactoring technologies 12
    1.5.4 CMI (Component Model Implementation) 14
  1.6 Organization of this thesis 14
2 LSR Model 16
  2.1 Introduction 16
  2.2 Abstraction Mechanism 17
  2.3 LSR Entities 17
  2.4 Dependency between entities 19
  2.5 Syntax attribute 21
  2.6 Comparison to other research work 22
    2.6.1 AST (Abstract Syntax Tree) 22
    2.6.2 CIA 23
    2.6.3 Rigi 23
    2.6.4 Software architecture transformations 25
  2.7 Summary 26
3 IFSO framework 27
  3.1 Assumption 28
  3.2 Representation 29
    3.2.1 System Level 29
    3.2.2 Component Level 30
    3.2.3 Class Level 32
    3.2.4 Method Level 33
    3.2.5 Logic Unit Level 36
  3.3 System Architecture 38
    3.3.1 Detailed structure of levels 38
    3.3.2 Communication between levels 39
    3.3.3 Plug-and-play Model 39

3.3.4 Permanent Storage 40
4 Prototype Implementation 41
4.1 Design Decision 41
4.2 Initialize IFSO From Existing Code 43
4.3 Transfer Software System To Representation 45
4.3.1 System Level 45
4.3.2 Component Level 46
4.3.3 Class Level 47
4.3.4 Method Level 48
4.3.5 LogicUnit Level 51
4.4 Algorithms and Operations 52
4.5 Message Agent 53
4.6 GUI Browser 53
4.6 Scenario 56
5. Discussion and Evaluation 57
5.1 Case Study: Metrics 57
5.2 Case Study: Dead Code Removing 62
5.3 Case Study: Low Coupled Unit Detection 67
6 Conclusions and Future work 70
6.1 Conclusions 70
6.2 Future work 71
Appendix A: example source code 73
A.1 ComponentTestA 73
A.2 ComponentTestB 76
Appendix B: mini-tutorial 77
B.1 How to write an algorithm 77
B.2 How to write an operation 82
Appendix C: example code 87

Note: the author of this thesis drew all figures, which name is lead by “Figure”.

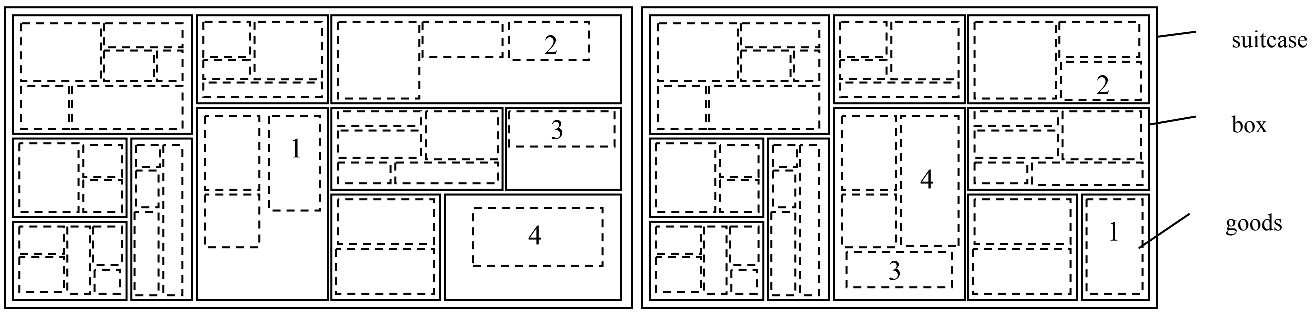
1 Introduction

1.1 Motivation

Refactoring is about improving the design of existing code. It is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. With refactoring you can even take a bad design and rework it into a good one. [FBB+99]

Refactoring becomes an important role in maintaining a legacy system because the original design of a system can be distorted in the implementation over time. This problem has been identified as "architectural drift" [HC01]. Also, complex code becomes unmanageable and only through changing the structure can it be simplified. Finally, future expansion of a system may be impossible without making some clear adjustments to the internal structure. The problem is that refactoring is a time-consuming process. Engineers need to understand structure and implementation details, locate desired code, analyze the structure, decide operations to improve the internal structure, and predict possible side effects of the modification. All these tasks are time-consuming and error-prone, especially for a complex legacy system. This is where automatic/semi-automatic software refactoring/analysis comes into play.

One of the key motivations of this thesis is that it requires cooperating several granularity levels during software refactoring/analysis to provide a global view for users. Most software is designed on several granularity levels, such as, component level, class level, or method level for component-based software. During refactoring process, we don't want a local modification within one granularity level to negatively affect other granularity levels. It requires to capture high-level structure information when apply a modification in lower level. For example, there are two granularity levels involved in object-oriented software system: class level and method/attribute level, e.g., split a method may cause decrease the cohesion of its class as an example described in appendix C. In the same way for software analysis, we want to get a global understanding not only focus on a particular granularity level. This problem is similar to the following suitcase reorganization example. Assume we had a suitcase containing numerous boxes, which also contained multiple goods, and we wanted to reorganize the contents of the suitcase to make the space usage most efficient. As shown below, optimal adjustment of goods inside one box might



a: The usage of space is bad. b: Adjust a, maximize the usage of space.

not necessarily lead to a better-organized suitcase as a whole. To achieve a final optimized result, adjustments on multiple levels (goods, boxes, suitcase) are almost always required. The same reason applying to software, focusing on one level is not enough for understanding and refactoring of software globally.

In addition to the key motivation, there are several motivations involved in the three basic steps to refactoring software: understand software, analyze, and carry over a set of refactoring operations. Following we will discuss issues involved in each step separately.

Software Understanding

One of the most promising approaches to the problem of software understanding is *reverse engineering[MJS+00]*. However capturing just the high-level structural information is not enough for refactoring code, which needs low-level code information. It is hard to know what effects are generated on structure during refactoring code by only capturing the low-level information, e.g., Abstract Syntax Tree. It is the second motivated problem addressed for our research work to capture both the high-level structure information and the low-level code information of software.

Software Analysis

There are many approaches to analyze software, such as cohesion/coupling measurement, detection of design defects, and locating similar code fragments [BDG+94]. The challenge is how to incorporate diverse metrics, design problem detection, and software analysis heuristics to understand software. First, it's a time-consuming task to implement these metrics, detections or heuristics for software. They need an abstract representation for software so these metrics, detections or heuristics can be reusable for various systems. Second, each analysis algorithm (in the rest of this paper, we will call all kinds of software measure metrics, detection of design defects or other kinds of software analysis technology as analysis algorithm.) focuses on a specific granularity level. To analyze software thoroughly, we need access to all granularity levels. So there are two problems that have to be addressed. One is how to define a standard representation for software and also how the representation can be adapted to meet the specific requirements of a specific software system. Another one is incorporating diverse analysis algorithms of different granularity levels.

Refactoring Software

Driving a set of refactoring operation based on analysis result is the fifth motivated problem. For example, deleting a method from a class, moving a method from a class to another class, or deleting a false case from switch statement.

The key issue is that the modification should be propagated to the whole system for further refactoring analysis. Fixing a detected design problem may cause side effects in the rest of the system. So the sixth motivated problem addressed is how to propagate a local change to the whole system and how to build an interactive environment for software refactoring/analysis.

1.2 Problem To Solve

The core-addressed problem by this thesis is how to integrate various refactoring technologies together to support the automatic/semi-automatic refactoring processing. Several detail problems are highlighted:

1. Provide a standard representation of hierarchical semantic knowledge for software. Based on the standard representation, we can develop algorithms and operations independent from concrete various software systems. That is, the implemented algorithms and operations can be reused. Furthermore, the hierarchical representation isolates the design issues, operations, and responsibilities from level to level. Each level focuses only on specific problems involved in current level.
2. Provide a framework for integrating algorithms and operations.
3. Algorithms can drive operations for refactoring. It is a good way to isolate analysis logic

from operations and a foundation for automatic software refactoring. An operation can be driven automatically based on different analysis results.

4. Provide a global view for software analysis and refactoring.
5. Provide an interactive environment. Users can get feedback from the framework after apply an operation upon the representation by propagating the modification to whole system.
6. Automatically extract information from existing code and reconstruct high-level structure information.

1.3 Requirement

All these problems drive us to define a framework as a foundation for automatic/semi-automatic software refactoring/analysis. The key requirement of this design is that the user of the framework should only focus on designing of software analysis algorithms or refactoring operations, much like the EJB developer only focuses on business logic design based on the services provided by the EJB container. We will discuss detailed requirements in the rest of this section.

First, the representation of software in the framework should be independent from a specific software system. The developed algorithms and operations based on this representation can then be reusable. Users don’t need to develop algorithms and operations again and again for specific systems.

Second, the software under consideration should be represented in multiple abstract levels. As the discussed key motivation in section 1.1, it requires capturing all information involved in each granularity level for software refactoring/analysis globally. Second, it is import to organize the information in hierarchical structure because it is more extensible and understandable than mix them together. Each level has independent related information and issues. Third important reason for multiple levels representation is that most available software analysis algorithms only focus on one granularity level such as class-level or method-level. To represent software in multiple levels, each level can be plugged in different algorithms and capture analysis information separately.

Third, the representation for software must be comprehensive. The representation should capture structure information and syntax information. The structure information is required for software analysis while the syntax information is required for software refactoring. Also, being comprehensive guarantees that software tools working at each level of abstract will perform correctly without missing information [CNR90].

Fourth, the framework should be extensible to meet various requirements of different systems. If the language changes, the framework can be customized to include different syntax information or other kinds of specific information. If the software design changes, the framework can be customized to represent software at different granularity levels as needed. For example, user can add more levels as needed; customizing each level to include specific information, i.e., inheritance relationship in class level, data flow in method level and so on.

Fifth, the framework should provide a global view for users by cooperating between levels, algorithms and operations. Algorithms calculation can be composed with regard to algorithms that execute lower in the hierarchy. Operation can be requested down to impact lower layer in the hierarchy and notifications of changes are then sent up the hierarchy. In this way, each level can focus on particular design issues or rules involved in current level but also can cooperate with other levels to archive global understanding/improvement.

Sixth, the framework should support plug-and-play algorithms and operations. As a foundation environment provided by the framework, the user can more focus on understanding system instead of spending time to implement algorithms or operations.

Seventh, the framework should enable existing code to be parsed to form the units of the abstract, multi-level hierarchy automatically.

1.4 Contribution

The primary contribution of this thesis is providing a platform for automatic/semi-automatic software refactoring/analysis. A conceptual model, LSR, is defined as an abstract standard representation for software. By introducing a framework, which is built upon the LSR model, various related refactoring technologies can be integrated together. We also have built a prototype system to realize the proposed framework, IFSO. This prototype system helps to validate the approach and provides the basis for conducting experimental studies.

Additionally, this thesis makes contribution in three categories: the LSR model, IFSO framework, and prototype implementation. In each category, the contributes are as follows.

1.4.1 Contribution of the LSR model

According to a design trait: design from an abstract level to a concrete level, we defined a hierarchical abstract representation, named LSR model, for software. Each level corresponds to a granularity level in design. The key contributes of defining the LSR model are given below:

1. The LSR model is abstract and extensible, that is, the LSR model can be customized for various software systems. Users can define how many level involved, what structural edges should be captured for each level, what syntax information is associated with the entities and so on. Each layer has the same abstract structure, guaranteeing an extensible model because it is easy to add or remove a level from the model.
2. A robust abstract description of objects involved in software is introduced. That is, objects have the same abstract description. Although similar semantic graph description has been discussed in [FH00, MK88], our description is more dedicate for software refactoring.
3. The dependencies between levels are well modeled.
4. The gap between low-level code information and high-level design structure are filled. The LSR model describes software from most abstract level to most concrete level. But the LSR model focuses more on semantic information, the high architecture design decisions aren't considered.

1.4.2 Contribution of IFSO

A customizable framework IFSO is built based on the LSR model. The framework provides a platform to integrate various technologies together to support software refactoring/analysis. The key contributes of IFSO are given below:

1. Provide a basic platform for automatic/semi-automatic software refactoring and software

analysis. Software analysis and refactoring are complex task. They are domain related, purpose related, etc. That is, various algorithms and operations may be required for various domain and purpose. Based on IFSO, users need only focus on design and developer algorithms and operations.

2. A global view is presented by cooperation between levels as a whole. Each level can communicate with its neighbor levels by using the communication mechanism provided by the IFSO framework. Operations can notify modification to upper levels. Algorithms can send requests to lower levels to retrieve detailed information. The operations and algorithms in one level can cooperate with each other and with other levels though the communication mechanism.
3. An interactive environment is presented. When an operation is executed in a level, the framework will propagate the change to whole system. Based on the effects, users can decide what to do next.
4. A plug-and-play model is designed for plugging algorithms and operations. Each level of the representation can be customized to plug in algorithms and operations independently.

IFSO customizes the LSR model for the component-based Java software. This process shows the adaptation of the LSR model.

1.4.3 Contribution of prototype implementation

A prototype is implemented to realize the proposed framework for Java systems. Several design decisions are valuable for future implementation:

1. Automatically initialize IFSO based on XML parsing. It includes automatically extract information out of source code and reconstructs high-level design information. We use JavaML[Bad00], which is an XML-based representation for Java source code, for initialing the low levels of IFSO. For high-level, an XML-based system description are defined. Based on XML, we can leverage the abundance of XML tools and techniques to parse and manipulate the source code easily.
2. Implement a simple case to directly write back the modification to source code. After running an operation upon IFSO, the modification can be written back to source code based on XML tools and techniques. The author of JavaML provides an XSLT for back-convert. An operation can manipulate the XML file by achieving the attribute of entities of the customized LSR model.
3. Provide a basis for experiment studies. Three case studies were developed by this thesis: several measurement metrics for different levels, dead code detection and removing and lower coupled unit detection.

1.5 Related Work

In this section we present an overview of related work. We will give detail comparison between the LSR model and other available research work in Chapter 2.

1.5.1 Metrics

Metrics provide a way to measure certain properties of a software system according to well-defined, objective measurement rules. The measurement results can be used to describe, judge or predict characteristics of the software. [LK94] describes a set of metrics for object-oriented software: meaning, graph of statistics collected from actual OO projects, affecting factors, related metrics, heuristic values and suggested actions. Upon this information, the designer or developer can make a better decision about software engineering tasks.

Many different software metrics appear based on different perspectives: project size, project complexity [LC94][MW89], software cohesion/coupling metrics [AK99][Bal96][BK98][BDW98][BO94][CK98][CZX02][Lak93][Mis00][OB95][ZX02]. Or based on different granularity levels: functional metrics [BO94], class metrics [Bal96][CK98][ZX02] or design level object metrics [BK98]. Briand et al provide a unified framework for cohesion measurement in OO systems [BDW98], discussing the shortcoming for each object cohesion metric and providing the framework that summarize a set of different conditions under which certain type of cohesion metrics can be applied.

Each kind of metrics addresses a specific perspective of software. It is hard to understand software, such a complex product, by using only a few metrics. Furthermore, it may need different set of metrics for different analysis purposes. IFSO provides aggregate info by collecting all kinds of metrics together.

1.5.2 Reverse-engineering

Reverse engineering is “analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information” [CC90].

Most reverse engineering research and practice is at the code level because the code is the only reliable source of information about the system [MJS+00]. CIA [CNR90], FAST [BG78], Cscope [Ste85] are tools for procedural program languages. They extract language dependent objects (e.g., macro, data type, function…) and reference relationship between them out of source code. TableGen [BBC+00] and Rational Rose [RAT] are for object-oriented language. Other kinds of model for capture high abstract level information for aiding the development or maintenance phases of the project are published in past years. We will discuss these models in Chapter 2.

The issue is that there is a gap between the low-level semantic information and the high-level structure information. In order to refactor code for improving internal structure of software, one needs to understand the structural effects while apply a refactoring operation. Part of models/tools [CNR90][Bad00] focus on extract object and reference relationship between them to ease burden of parsing source code without considering the structure level. Part of models/tools [MK88][FH00] focuses the high level structure information for aiding software understanding and maintenance but missing capture of the low-level semantic information.

1.5.3 Refactoring technologies

Heung Seok et al use the metrics to assess and restructure classes [CK96]. By analyzing the relationship pattern of a class, Heung Seok finds a way to create several high cohesion classes instead of the original low functional cohesion class.

The paper [LD99] discuss restructuring program fragments by break them into small, cohesive pieces. Given a function that performs several activities, they show how to "automatically" decompose that function into several functions, each performing only a single activity or a single set of related activities is the problem addressed in the paper. They introduced automatic approach is based on specified cohesion.

DUPLOC [BBC+00] tool detects occurrences of duplicated code in syntax level and very restricted automatic code refactoring.

Audit-RE provides [BBC+00] automatic detection of violations of "best-practice" heuristics in object-oriented re-engineering by given a set of pre-defined design problem.

ECLIPSE [ECL] provides a set of refactoring operations: move, change method signature, rename, convert anonymous class to nested class, extract method, extract local variable and so on. IntelliJ IDEA [INT] is a Java IDE, which provides 25 plus refactoring tools, such as renaming, move, introduce variable, extract interface/superclass, extract method, etc. But these individual refactoring tools/operations are independent with each other and no supports for structural analysis before or after apply a refactoring tools/operations. That is, these refactoring tools are only benefit after a developer knows what problem is and how to fix it.

EXTRACT [Cal03] is an extensible language for code manipulation. Based on AST, EXTRACT provides JPath, which is used to identify code to be transformed as XPath. This language let programmer transfer code more easily.

Various refactoring technologies are published. Each of them solves a specific problem. In order to refactoring software thoroughly, we need a platform to integrate various refactoring technologies together.

1.5.4 CMI (Component Model Implementation)

A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment [HC01]. The CMI [CMI] is a component model implementation used as a component model standard for our software engineering course project. The prototype implementation refers to it as our component model for the component level of IFSO. Each component has a set of required interfaces and provided interfaces, which are defined as Java interfaces. The required interfaces and provided interfaces are the contract between components, cannot be changed after deployed. The provided interfaces of a component are similar to local interface of an EJB in J2EE. Likewise, each component needs to provide a set of interfaces required by the CMI as home interface of an EJB. The CMI will manage life cycle of components.

Components interact together though well-defined interfaces. According to an application description, the CMI will instantiate all involved components and check the contract between them. If the instantiate is success, each component can get the required interface instance passed by the CMI.

1.6 Organization of this thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces the LSR model and comparison with other research work. Chapter 3 introduces IFSO, including assumption, customization of the LSR model for component-based software systems, system architecture. Chapter 4 represents the prototype implementation, including design decisions, initialization the

IFSO from existing code, transformation of a software system to the representation, plug-and-play of algorithms and operations, message agent and GUI browser. Several case studies will be introduced in chapter 5. Chapter 6 is for conclusions and future work.

2 LSR Model

The formally designed LSR (Layered Software Representation) model provides a consistent and robust mechanism to describe software artifact at any level of abstraction independent of programming languages. In this chapter we will introduce the definition of the LSR model and compare it with other representations of software.

2.1 Introduction

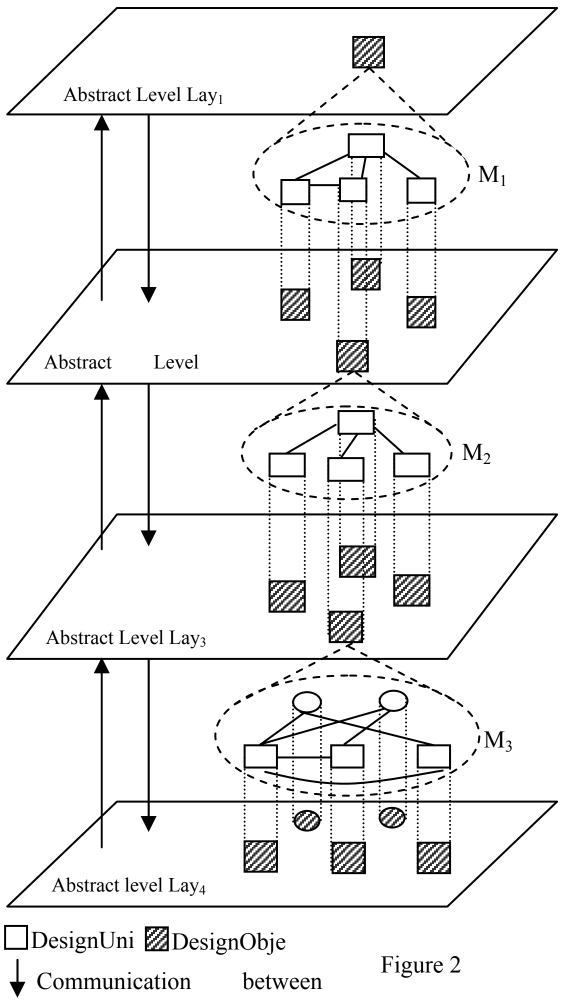


Figure 2

LSR model is a conceptual model for the representation of software. It defines an extensible-layered abstract representation as shown in Figure-2. Each level corresponds to a design granularity level in the system. Higher levels define the constraints, which should be preserved during the lower level design process, while the lower level refines the upper level design. User can add or remove levels according to specific requirements of software. The arrow link between levels means communication between levels. Modification in lower levels needs to be propagated to upper levels and upper levels can send change request to lower level.

The communication mechanism is not defined in the LSR model. The LSR model focuses on capturing required information for such a communication mechanism.

Each level includes a set of *DesignObjects,* which is designed based on *DesignUnit* provided by neighboring lower level. That is, an upper level design object is designed based upon design objects in the next lower level as a black box; the name for the black box is *DesignUnit*.

Each *DesignObject* is represented as a directed graph. The nodes and arcs of the graph represent the *DesignUnits* and their dependencies as shown in Figure-3.

2.2 Abstraction Mechanism

Abstraction plays a dominant role in any area of knowledge representation. In the realm of conceptual modeling, abstraction mechanisms serve as organizational axes and design methodologies for the development of conceptual models [BM84]. The abstraction mechanism used by LSR model is refinement: the entity in the up level is designed based on interfaces provided by the entity in the low level and the low level will fill in detail design of each entity in the up level. For example, object-oriented software is designed by defining interfaces of classes and relationship between classes and then filling in the detail design of each class.

2.3 LSR Entities

In this section, we will formally define some key concepts.

Layer(L): $L_i = \{DO_j\}$ and $L_i \cap L_j = \varnothing$. $M_{k,k+1}$ is a function mapping $DO_j \rightarrow \{DU_i\}$, where $DO_j \in L_k$ and $DU_i \in L_{k+1}$ (DU is a black box, a special view, of DO). Layer (entity) is a function returning the layer in which the entity resides.

DesignUnit(DU): DU = {RI, PI, Att} where RI = {RI_i}, PI = {PI_j} and Att = {Att_k}, is a black box of design object that exposes *Required Interface(RI), Provided Interface(PI)* and associated attributes (Att) (i.e., meta data) to outside. It is a special view for design object without revealing the internal structure.

Required Interface(RI): specifies external dependencies that must be "satisfied" for the design unit to be well-defined.

Provided Interface (PI): specifies the information that is externally visible for a design unit and guaranteed to be valid by the design unit if its required interfaces are satisfied.

Attributes(Att): a set of (N, V) metadata pairs, N is the name of a attribute, V is the value of the attribute.

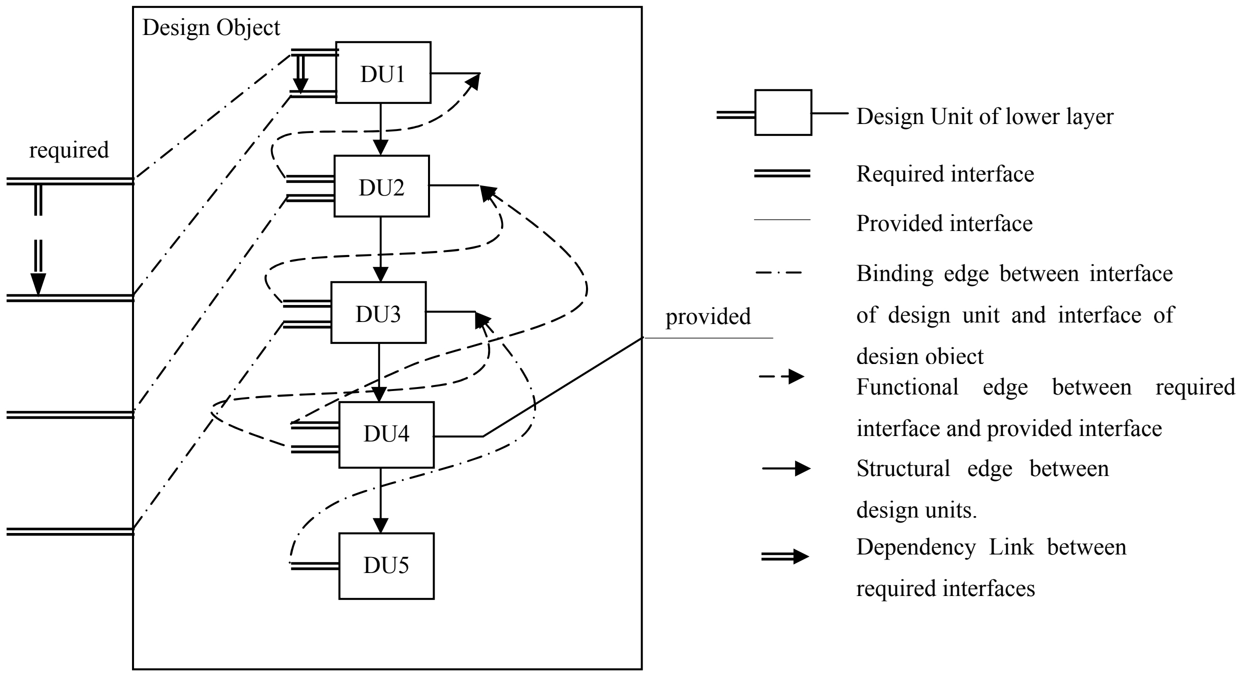

Design Object
DU1
DU2
DU3
DU4
DU5
required
provided
Design Unit of lower layer
Required interface
Provided interface
Binding edge between interface of design unit and interface of design object
Functional edge between required interface and provided interface
Structural edge between design units.
Dependency Link between required interfaces

Figure 3

A *Design Object* maintains structural information as we now describe.

DesignObject(DO): Each design object DO can be viewed as a graph G = (V, E) as shown in

Figure-3. $V = \{DO\} \cup \{DU_j\}$ such that (Layer (DO) = k) $\wedge$ (Layer (DU_j) = k+1) $\wedge$ ($DO \rightarrow \{DU_j\} \in M_{k,k+1}$). $E = \{Bindings \cup Links\}$ where Bindings = $\{BE_i\}$ and Links = $\{FE_p\} \cup \{SE_q\} \cup \{DE_p\}$.

Binding: a binding is an undirected edge (DU_i, DU_j, Att) such that Layer (DU_i)+1 = Layer (DU_j).

Link: is an directed edge (DU_i is the head , DU_j is the tail, Att) such that Layer(DU_i) = Layer(DU_j).

Binding Edge (BE): is a Binding (DU_i.Interface$_k$, DU_j.Interface$_p$, Att), Interface is a required interface or a provided interface.

Functional Edge (FE): is a Link (DU_i.RI$_k$ where DU_i is the head, DU_j.PI$_p$ where DU_j is the tail, Att).

Structural Edge (SE): is a Link (DU_i is source, DU_j is target, type, Att) that contains additional type information about the link between DU_i and DU_j.

After describing the graph, Figure-3, there is a special kind of link, called a *Dependency Link(DE)*, that must be exposed by a design unit. A dependency link is a directed edge (Head, Tail) between two required interfaces of a design unit. This means we must redefine a design unit to be DU = {RI, PI, Att, DE}, DE = $\{DE_k\}$, DE_k is (Head, Tail), where Head, Tail $\in$ RI.

2.4 Dependency between entities

There are four kinds of syntax dependencies involved in LSR model:

1. Dependency between provided interface and required interface (Functional Edge): a

provided interface provides the interface required by the required interface. It is equivalent to a reference relationship between objects.

2. Dependency between design units (Structural Edge): there are different syntax dependencies between design units within a level. For example, inheritance between classes or data flows between statements. The user can customize dependencies between design units by using structural edges. For refactoring code, this information is important. For example, to determine if a method can be deleted from a class, we check if the class implements an interface that defines the method. In this case, implementation should be part of extensible dependency between class design units.
3. Binding between interface of a design unit and interface of a design object. The binding information captures relationship between levels. For example, if one observes high-level unexpected edges between design units, the internal structure of the design unit at lower level reveals the reason for the dependencies. Conversely, modification in lower level can be found to play some role in the change of the software structure [FH00].
4. Dependency between required interfaces (Dependency link between required interfaces): a required interface depends on another required interface. For example, Integer.valueOf(i).toString(). There are two required interfaces: Integer.valueOf(i) and toString(). The required interface toString() is dependent on return type of Integer.valueOf(i). When a required interface changes, its dependent required interface is captured as well.

The purpose of capturing dependencies between entities is to effectively locate all related entities when applying an operation upon an entity and propagating the modification. LSR model only focus on direct dependency between entities because indirect dependencies can be inferred from it. For example, method A writes attribute B and method A1 reads attribute B to control its data flow. Even though A doesn't directly call A1, there is an implicit dependency between them. LSR model captures direct reference relationships between A and B, A1 and B as shown in Figure-4. Inside A1, we can find that the required interface (Labeled by "1") is bound with a required interface of an "IF" statement as shown in Figure-5. So it needs further semantic analysis to expose implicit dependency by giving a complete structure information and syntax information.

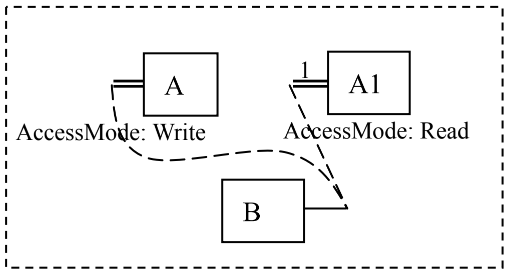


Figure 4

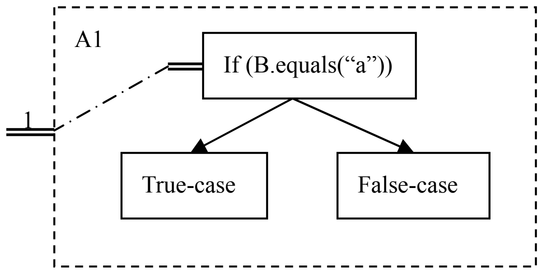


Figure 5

2.5 Syntax attribute

The LSR model doesn't define attributes (Att) for each entity because the conceptual model is independent from the programming language. Users need to customize the LSR model for a specific software system. In chapter 3, we will introduce how to customize the LSR model for component-based Java software systems.

In our thesis, as described in section Chapter 3, we define IFSO to have five layers of representation－system, component, class, method, logic unit. Other languages, such as C, may only have four－system, file, function, logic unit. The generic design of LSR and its implementation for IFSO is one of the contributions of this thesis.

2.6 Comparison to other research work

In existing research, various representations have been designed for software for various purposes, e.g., software understanding, code manipulation, reverse engineering, etc. We will compare our model to these proposed representations to exhibit issues that haven't been addressed by previous research work.

2.6.1 AST (Abstract Syntax Tree)

An Abstract Syntax Tree is a data structure representing abstract syntax, which is independent of machine-oriented structures and encodings and also of the physical representation of the data. An AST is a good abstract tree for code manipulation. EXTRACT [Cal03] is such a tool designed based on ASTs. The limitation of AST is it focuses only on syntax information of each source code file. There is no software structure information included. The developer has to keep the structure diagram in mind and transfer it to a set of code manipulate operations. The LSR model tries to fill the gap between code syntax representation and software structure representation. AST representation is flat while our model represents software in three-dimensional manner. The LSR model models relationship between granularity levels in additional to syntax dependencies between design units and syntax attributes of each entity. Moreover, LSR model captures high-level structure information, which information cannot get from source code.

2.6.2 CIA

CIA (C Information Abstraction System) is a system for analyzing program structures [CNR90]. It extracts global objects: files, macros, global variables, data types and functions out of C program. Each object has a set of attributes. And reference relationships between them are captured. All the information is saved to database. Programmers can invoke relational queries to analyze various aspects of their software [CNR90].

The conceptual model introduced in CIA is flat; there is only one level. And only reference relationships are captured. It is not enough for complex software that involved several granularity levels (e.g., object-oriented software system). There is an inheritance dependency between classes, an implementation dependency between a class and an interface. It requires capturing variegated dependencies. Additionally, only focus on program level is not enough for software structure understanding. The LSR model can extend to not only extract objects out of source code and variegated dependencies between them but also captures high design level objects by acquire more information from designer.

2.6.3 Rigi

*Rigi was designed to address three of the most difficult problems in the area of programming-in-the-large: the mastery of the structural complexity of large software systems, the effective presentation of development information, and the definition of procedures for checking and maintaining the completeness, consistency, and traceability of system descriptions. [*MK88*]*

The Rigi model is a special purpose semantic network (graph) data model for the representation and organization of the "bricks" and "mortar" of complex software systems. The nodes and arcs of the graph represent the components of a software system and their dependencies [MK88]. The LSR model dedicates to software refactoring by exposing precise and comprehensive implementation structure while the Rigi model dedicates to software understanding by capturing diverse dependencies and information. The common points of the Rigi model with the LSR model are:

1. Abstract software system as a graph data model. Both models represent software as a set of nodes, the components of a software system, and arcs, their dependencies.
2. Layered hierarchical representation. Both the Rigi and LSR model represents software in a hierarchical structure.

The differences between the Rigi model and LSR model are:

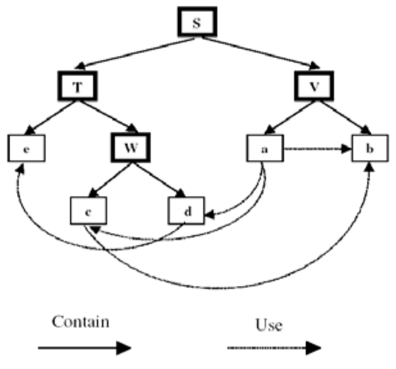
1. The major objective is different. The objective of the Rigi model is to effectively represent and manipulate the building blocks of a software system and their myriad dependencies, thereby aiding the development phases of the project [MK88]. The objective of the LSR model is defining an abstract representation for a software system, which captures both structure and syntax information, as a foundation for automatic/semi-automatic software refactoring/analysis. LSR model focus more on structure and syntax dependencies.
2. The abstraction mechanism is different. The Rigi model provides a set of abstraction mechanisms: aggregation, generalization, and set. These three abstraction mechanisms can be applied recursively to construct aggregation, generalization, and set hierarchies, respectively [MK88]. The LSR model only applies a abstraction mechanism: refinement.
3. The defined abstract objects in model are different. The Rigi model defines thirteen abstract object classes to model the "bricks" of a software system: sys (subsystem), rel (subsystem

release or variant), pro (program module), mod (module), def (definition), imp (implementation), gen (generic definition variant), alt (alternative implementation variant), rev (revision), doc (documentation), dat (data), pic (picture), and acc (accessory). The LSR model defines only one kinds of abstract object in each level: design object. In the LSR model, the abstract object involved in each level is similar except specific attributes and structural dependencies.

4. The defined dependence between abstract objects is different. The Rigi model defines three dependence classes to model the "mortar" of a software system: structure, change or compilation, and semantic [MK88]. A semantic dependency constitutes any relationship between two components a designer would like to express and document [MK88]. The LSR model only focuses on structure and syntax dependency between design units.

2.6.4 Software architecture transformations

Hoda Fahmy and Richard C. Holt categorized a set of useful architectural transformations and described them within the framework of graph transformations. They abstracted software as a directed typed graph as shown in Figure [FH00]. The idea behind Hoda and Richard's paper is very similar to LSR model but they describe in more general way. The transformations discussed in the paper are well modeled as part of the LSR model. For example, in LSR model, the binding edge captures dependencies between levels. It is corresponding to lifting transformation (lift low-level use edges



FH00 (b)

up the system hierarchy) for architecture understanding. In other words, the LSR model applies the transformations as part of the LSR model instead of conceptual discussion.

2.7 Summary

The LSR model defines a robust, extensible and abstract representation for software. It can be instantiated for all kinds of software designed by refinement approach. Objects have same structure by the high abstraction while it can be customized to include specific syntax or metadata information for a specific system. Levels have same abstract structure so it is easy to add or remove a level according to a specific system. So LSR can be customized to describe software from the most concrete level to the most abstract level. All design granularity levels' information is captured. This is an important contribution of the LSR model. Another important contribution of the LSR model is various dependencies are well modeled by LSR. When you access to a design object, comprehensive dependency information (functional dependency, structural dependency, binding dependency, etc) is provided.

In next Chapter, we will describe how IFSO applies the LSR model to concrete environment to support software refactoring.

3 IFSO framework

The IFSO (Integrated Framework for SOftware refactoring and analysis) framework is designed as a customizable platform for software refactoring/analysis. It represents software at multiple granularity levels, which is a concrete representation of the LSR model, and maintains the relationships between them as shown in Figure-5. Different algorithms and operations can be used to customize each granularity level independently. And a communication mechanism is provided to cooperate between levels or algorithms and operations. IFSO propagates the modifications between levels to ensure consistency for the whole system. Now we only focus on component-based software of Java. In future work, we can extend IFSO to support various kinds of software systems. In the following section of this chapter, we will introduce design assumptions for IFSO, then we will discuss the representation of IFSO and system architecture.

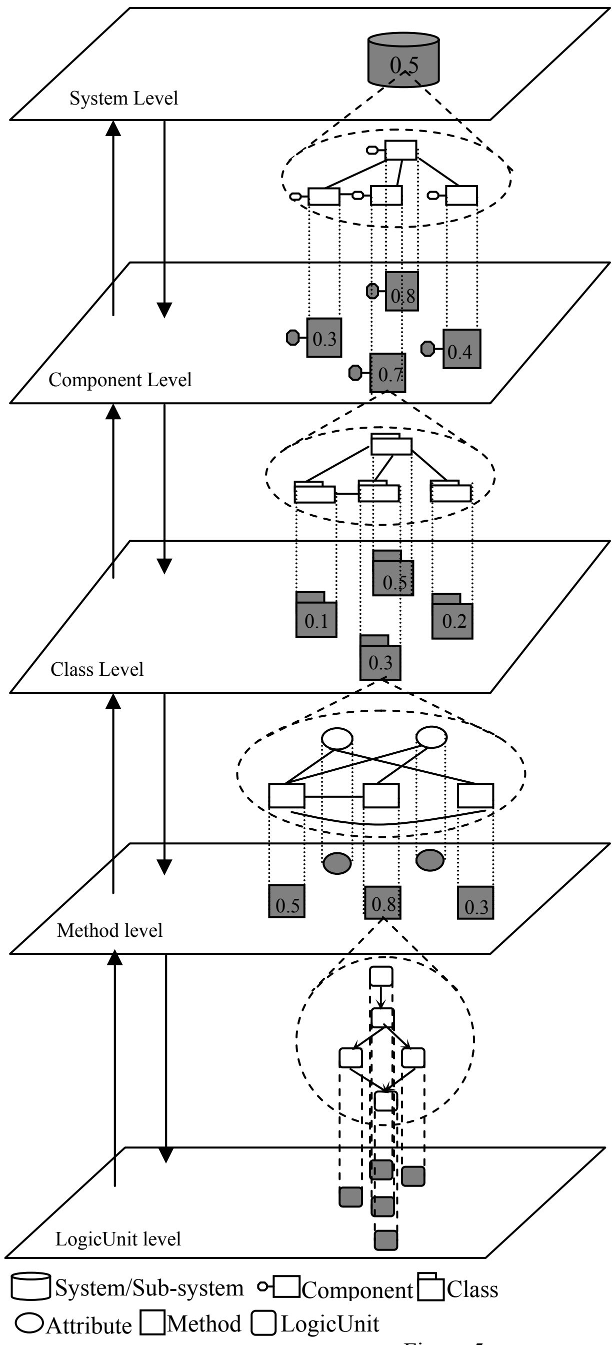


Figure 5

3.1 Assumption

We make the following assumptions for IFSO:

1. The most important assumption of IFSO is that the design process used in the target software is refinement. That is, the design of a lower granularity level is a refinement of the higher granularity level while the higher level defines the constraints, which should be preserved in the lower level design.
2. Currently IFSO supports component-based software by Java. There are only five granularity levels involved in IFSO: system, component, class, method and logic unit. The motivation for introducing a logic unit level under the method level is that the method level exposes control flow of a method while the logic unit level has no control flow. Though IFSO can be extended to more levels, we restrict the design to five levels for current research on Java source code. Also IFSO can be extended to support various kinds of software, it is discussed in section 6.2.
3. The system is a self-contained. There is no provided interface and required interface expose to outside of the system. We also ignore environment or operation system dependency. Some required interfaces of a design object might not find a provided interface in current system. As Figure-17 in section 4.3.2, there is no provider associated with a required interface because it is provided by system. For example, System.out.println. Currently, we don't cover this issue.
4. For the prototype implementation, we ignore some aspect of the Java language, such as synchronized blocks, inner classes, anonymous classes, assertions, etc.

5. We refer to CMI 2.0 [CMI] as our component model specification for component design.
6. While parsing source code into IFSO, we ignore all syntax checks by assuming that the system compiles successfully.

3.2 Representation

This section will describe how IFSO customizes the LSR model to meet requirements for component-based Java software systems. Design objects, associated attributes, structural edges, types of required interfaces and provided interfaces will be described for each level separately.

3.2.1 System Level

The system level only includes one design object: system (Figure-6). The system is a design based on a set of components. Each component has required interfaces and provided interfaces. A required interface of a component should be provided by another component in the system according to CMI 2.0 specification. The provided interface of a component should be there even

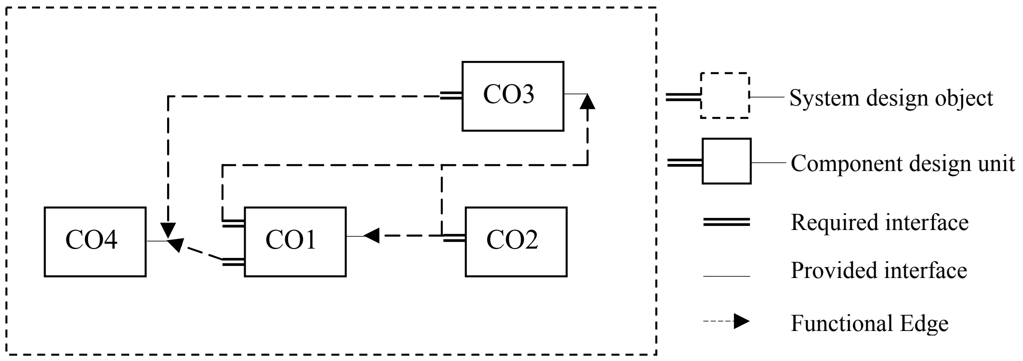


Figure 6: System Design Object

though there may be no required interfaces that depend on it. An interface of a component is a contract with other components. The representation of a system design object captures the software architecture.

3.2.2 Component Level

The design object in the component level is component (Figure-7). Each component is designed based on a set of classes. The types of structural edge between classes are: inheritance and implementation.

According to the specification of CMI [CMI], all classes of a component can only communicate outside of the component by using the defined required interfaces. A provided interface of a component is defined in an interface and a class within the component should implement the interface. As shown in Figure-7, the provided interface of the component defined in interface I6 is linked by C3 using a structural edge and I6 is bound to the design object provided interface of the component. While it is true that C3 actually implements the interface

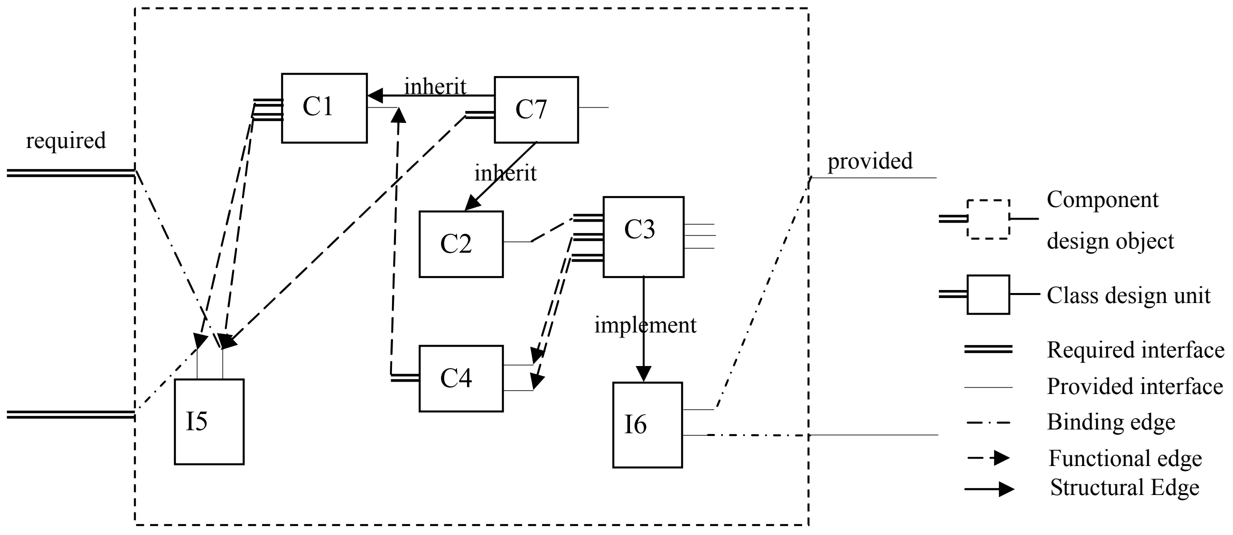


Figure 7: Component Design Object

exposed by the component. A required interface of a component is defined in an interface I5 as well. Internal to the component, the functional edges between class units capture the actual use of an interface; for example, class C1 uses two methods defined by the I5 interface while C7 uses the second one. By binding the external required interface to the interface definition, we accurately capture the real dependence by interface not just by methods.

Attributes defined in the interface of a component design object are listed in Form-1 (C).

REQUIRED INTERFACE OF A COMPONENT (FORM-1 (A))			
Type Name	**Access Mode**	**Is independent**	**Example**
Interface Method	Call	No	The method defined in an interface.
Interface Attribute	Read	No	The attribute defined in an interface.

"Is independent" means if the required interface is dependent on other required interfaces. For example, s.getLength(), function geLength() is dependent on variable s. Yes/No means it may/may not be independent.
AccessMode: there are four kinds of access mode, Read, Write, Call, Access. Read: read value; Write: write value; Call: function call; Access: variable reference.

PROVIDED INTERFACE OF A COMPONENT (FORM-1 (B))	
Type Name	**Example**
Interface Method/Attribute	I5.methodA(int, int)

ATTRIBUTES FOR ENTITIES IN COMPONENT LEVEL (FORM-1(C))		
Entity Name	**Attribute Name**	**Attribute Value**
Required Interface of the Component Design Object	Interface Name	For example, I5, as in Figure-7
	Name	Interface name. For example I5.methodA, where I5 is the interface name, methodA is the method name.
	Type	Interface Type: Function;Attribute;
	Target	Provider name
Provided Interface of the Design Object	Interface Name	For example, I6, as in Figure-7
	Name	Interface Name, For example I6.methodA, where I6 is the interface name, methodA is the method name
	Return Type	Return type

3.2.3 Class Level

The design object in the class level is a class design object (Figure-8). A class design object is designed based on methods and attributes. Only the functional edges are captured within a class.

The required interface of a class is the required interface of a method/attribute, which accesses other objects outside of the class. Form-2 (A) lists types of required interface for a class. A provided interface of a class is a method/attribute, which can be accessed by outside, that is, the visibility is public or protected in Java. For protected method or attribute, a class includes package name as its attribute for precise visibility analysis. The attributes associated with entities in class level are described in Form-2(C).

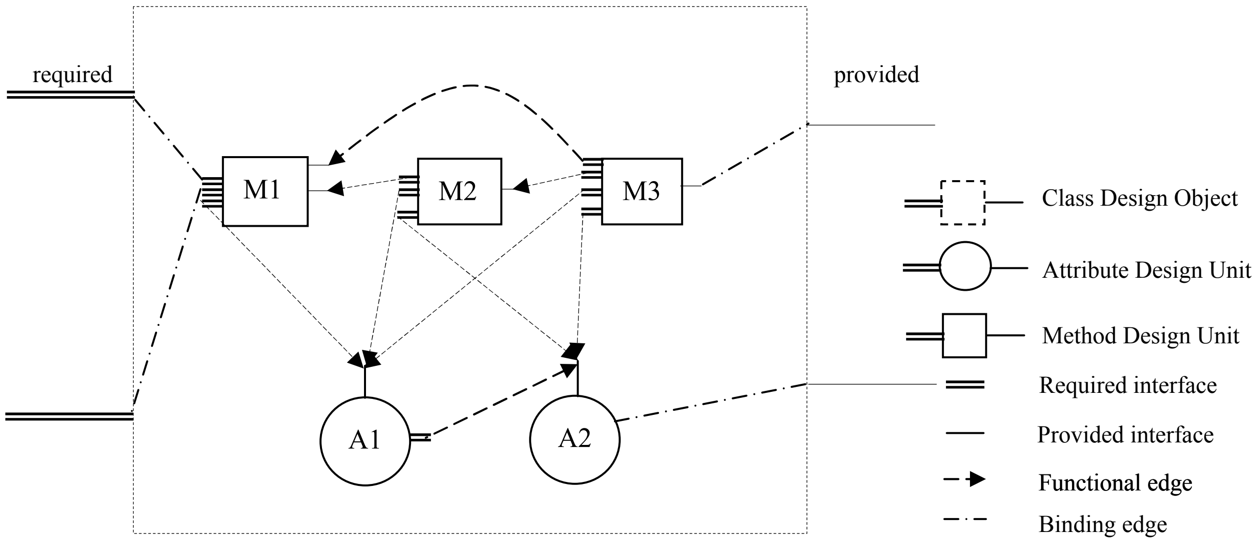


Figure 8: Class Design Object

REQUIRED INTERFACE OF A CLASS (FORM-2 (A))			
Type Name	**Access Mode**	**Is independent**	**Example**
Object's Method	Call	Yes/No	s.getLength(), type of s is String. Integer.valueOf(i).toString(), toString is dependent on Integer.valueOf(i)

Object's Attribute	Read, Write, Access	Yes/No	if (v.elementCount > 0), type of v is Vector.
Static Attribute	Read, Write, Access	Yes/No	System.out.println(System.err)
Static Method	Call	Yes/No	System.out.println("test")

"Is independent" means if the required interface is dependent on other required interfaces. For example, if s.getLength(), then function getLength() is dependent on variable s. Yes/No means it may/may not be independent.
AccessMode: there are four kinds of access mode, Read, Write, Call, Access. Read: read value; Write: write value; Call: function call; Access: variable reference.

PROVIDED INTERFACE OF A CLASS (FORM-2 (B))	
Type Name	**Example**
Public/Protected Method	None
Public/Protected Attribute	None

ATTRIBUTES FOR ENTITIES IN CLASS LEVEL (FORM-2 (C))		
Name	**Attribute Name**	**Attribute Value**
Class DesignUnit	Name	Class Name
	Visibility	Public;Protected;Private
	StaticFlag	True/false
	AbstractFlag	True/false
	Implements	A list of implemented interfaces
	Extends	A list of parent classes
	PackageName	Package name
Provided Interface of a Class	Interface Name	Function or attribute name.
	Type	Function; Attribute;
Required Interface of a Class	Interface Name	Function or attribute name
	Type	Function; Attribute
	Target Name	Class name

3.2.4 Method Level

There are two kinds of design objects involved in the method level: method (Figure-9) and attribute (Figure-10).

A method is designed based on a set of logic units. The type of structural edge between logic units is data flow. The required interface of a method is the required interface of logic unit, which accesses outside of the method. The provided interface of a method is the method name, the parameter list and return type. Form-3 (A) lists all possible types of required interface in method level for a Java program.

An attribute design object only has required interfaces and provided interfaces, there is no

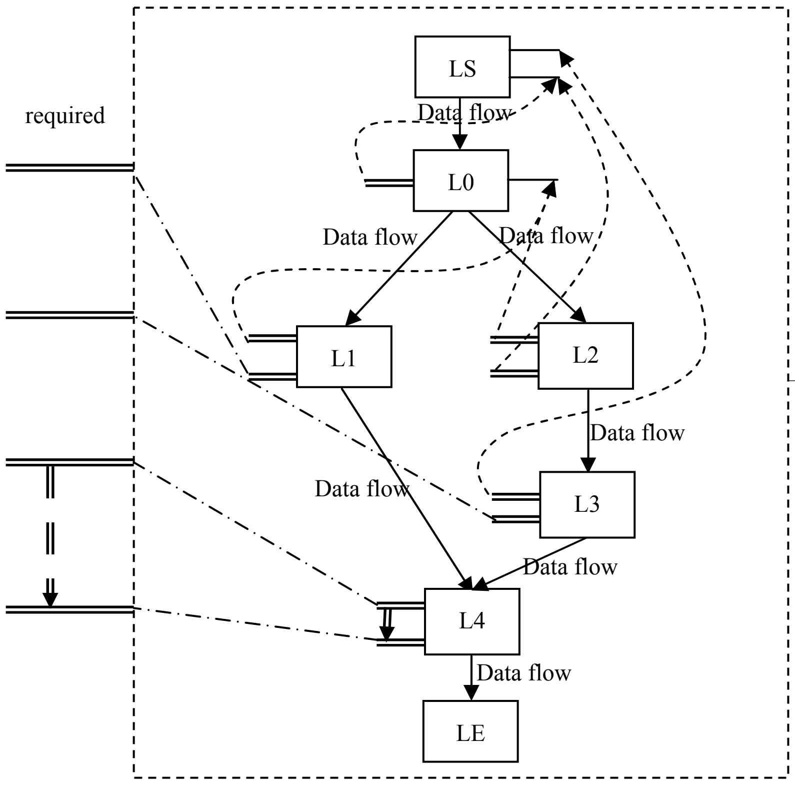


Figure 9: Method Design Object

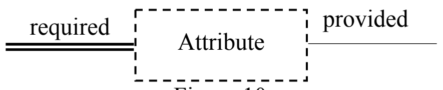
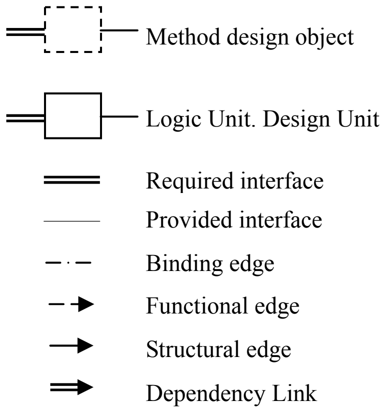


Figure 10



internal structure as shown in Figure-10. We only need to parse the attribute definition expression to get required interfaces and provided interfaces as done for statements in LogicUnit Level. The types of required interfaces and provided interfaces for a statement are listed in Form-4 (A).

REQUIRED INTERFACE OF A METHOD (FORM-3 (A))			
Type Name	**Access Mode**	**Is independent**	**Example**
Method	Call	Yes/No	add(i,j)
Object Variable	Read, Write, Access	Yes	v = new Vector()
Object's Method	Call	Yes/No	s.getLength(), type of s is String.

			Integer.valueOf(i).toString(), toString is dependent on String.valueOf(i)
Object's Attribute	Read, Write, Access	Yes/No	if (v.elementCount() > 0), type of v is Vector.
Static Attribute	Read, Write, Access	Yes/No	System.out.println(“test”)
Static Method	Call	Yes/No	System.out.println("test")

"Is independent" means if the required interface is dependent on other required interfaces. For example, s.getLength(), function getLength() is dependent on variable s. Yes/No means it may/may not be independent.

AccessMode: there are four kinds of access mode, Read, Write, Call, Access. Read: read value; Write: write value; Call: function call; Access: variable reference.

PROVIDED INTERFACE OF A METHOD (FORM-3 (B))	
Type Name	**Example**
Function	None

ATTRIBUTES FOR ENTITIES IN METHOD LEVEL (FORM-3 (C))		
NAME	**ATTRIBUTE NAME**	**ATTRIBUTE VALUE**
Method DesignUnit	Visibility	Public;Protected;Private
	StaticFlag	True/false
	AbstractFlag	True/false
	Return Type	Return type
Attribute Design Unit	Name	Attribute name
	Visibility	Public;Protected;Private
	StaticFlag	True/false
	AbstractFlag	True/false
	DataType	Data type
Provided Interface of a Method	Interface Name	Function name, include a list of parameter type. For example, calculate(int, int)
	Return type	Return type.
Required Interface of a Method	Interface Name	Function, attribute variable name
	Type	Function, Attribute variable
	Target Name	Function, Attribute variable or Class name
Provided Interface of a Class	Interface Name	Attribute variable name
	Data type	Type name
Required Interface of a	Interface Name	Function, attribute variable, Class name
	Type	Function, attribute variable, Class

Class	Target Name	Function, Attribute variable or Class name

3.2.5 Logic Unit Level

Logic Unit Level is the lowest level in IFSO. The design object in this level is a Logic Unit, which is designed based on a set of statements, as shown in Figure-11. Definition for Logic Unit is:

LogicUnit is a cohesive unit, composed by a set of statements with no control flow. The statements are logically related together. There is similar to Basic Blocks [ARJ86].

The purpose of inserting a logic unit level between the method and the statement is to simplify the responsibility of the method level. Without the logic unit level, the method level needs to analyze data flow structure and logic relationship between statements. Now, the logic unit level will take care of the logic relationship between statements. The method level only needs

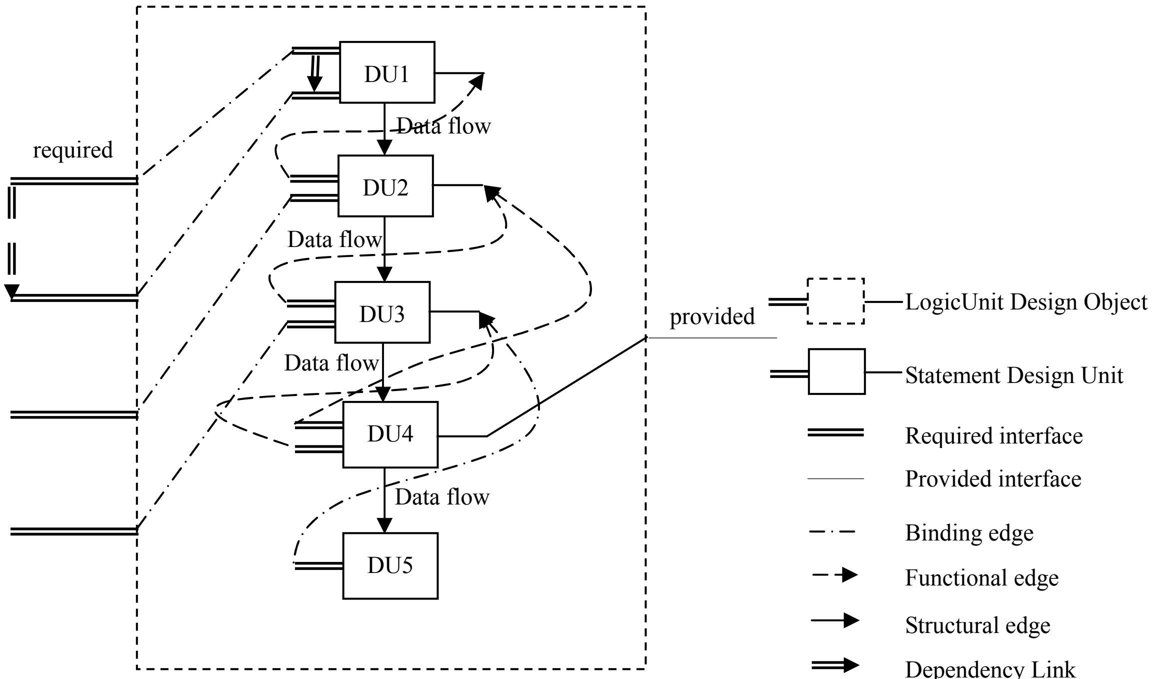


Figure 11: LogicUnit DesignObject

to focus on data flow structure between logic units.

The required interface of a logic unit is a required interface of a statement, which accesses outside of the logic unit. Form-4 (A) lists the types of required interface for logic units and statements. The provided interface of a logic unit is the provided interface of a statement, which can be accessed by outside. The types of provided interface are listed in Form-4 (B).

REQUIRED INTERFACE OF A STATEMENT/LOGICUNIT/ATTRIBUTE(FORM-4(A))			
Type Name	**Access Mode**	**Is independent**	**Example**
Primitive Type Variable	Read, Write	Yes	if (i>0)
Object Variable	Read, Write	Yes	if (s instanceof String)
Method	Call	Yes	add(i, j)
Object's Method	Call	Yes/No	s.getLength(), getLength() is dependent on object variable s.
Attribute	Read, Write, Access	No	if (v.elementCount > 0), elementCount is dependent on object variable v.
Object’s Attribute	Read, Write, Access	Yes/No	v.size
Static Attribute	Read, Write, Access	Yes/No	System.out.println(System.err)
Static Method	Call	Yes/No	System.out.println("test")

PROVIDED INTERFACE OF A STATEMENT/LOGICUNIT/ATTRIBUTE(FORM-4(B))	
Type Name	**Example**
Primitive Type Local Variable	int i;
Object Local Variable	String s;

ATTRIBUTES FOR ENTITIES IN LOGICUNIT LEVEL (FORM-4(C))		
Name	**Attribute Name**	**Attribute Value**
LogicUnit DesignUnit	Type	If(begin/end); switch(begin/end); while(begin/end); for(begin/end); expression;
	IsReturn	Yes/No (if the design unit include a return statement, IsReturn=yes, otherwise, IsReturn=no
	Expression	The string of the statement.

Statement Design Unit	Expression	The string of the statement.
	IsReturn	Yes/no. If the statement is a return statement, IsReturn=yes, otherwise, IsReturn=no.
Provided Interface of a LogicUnit/StatementUnit	Interface Name	Variable name
	Data type	Data type
Required Interface of a LogicUnit/StatementUnit	Interface Name	Function, Attribute variable, Class, Local variable name
	Type	Function; Attribute Variable; Local Variable; Class
	Target Name	Function, Attribute variable, Class, Local Variable name
	Function Name	If the required interface is a parameter of a function call, this attribute is the name of the function
	CastType	If the expression is casted, this value is the type name of the cast type.

3.3 System Architecture

IFSO is a layered framework of cooperating levels. In this section, we describe the detailed structure of each level and communication mechanism between the levels.

3.3.1 Detailed structure of levels

The internal structure of each level is shown in Figure 12. Each level has plugged operation components and algorithm components. The representation provides a set of standard interfaces to these operation components and algorithm components. Also, the representation and the

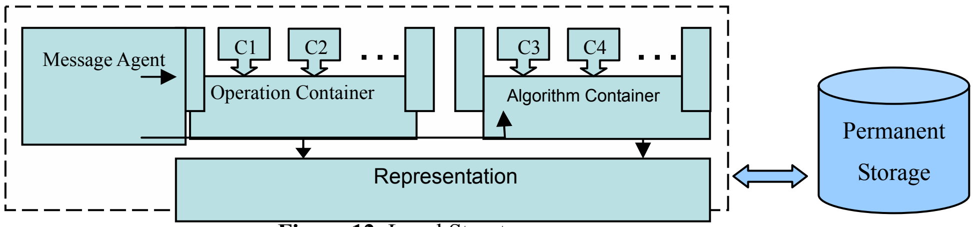


Figure 12: Level Structure

containers store information to database or file system for permanent storage. A message agent is attached to each level. It communicates with the operation container and the algorithm container by sending and receiving messages.

3.3.2 Communication between levels

Each level has a Message Agent to communicate with its neighboring levels. A Message Agent receives the requests from its neighboring levels and evaluates the request in its level. Each level issues notification events up to its upper level upon the cooperation of an operation. The message agent will forward the message to the operation container and algorithm container and return the result, which is passed back from the containers, to the neighboring level. C2 architectural style [RNK++95] is an inspiration for this message-based structural design. Plugged operation components and algorithm components can all be used by other levels by registering with the container. Using the message agent, the level can be customized independently but also can cooperate during analysis processing.

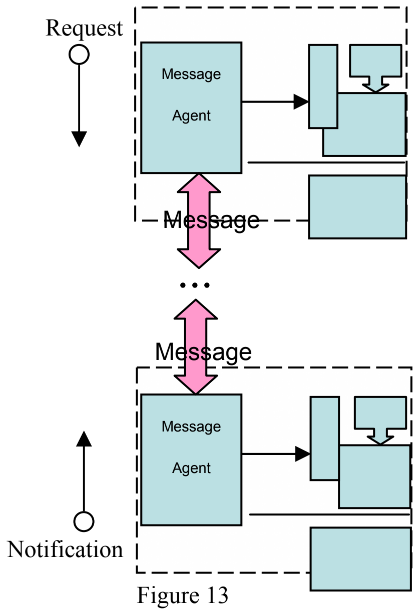


Figure 13
Level Communication

3.3.3 Plug-and-play Model

Each level has an operation container and an algorithm container. The operation container manages operation plug-ins, which can manipulate the representation. The algorithm container manages algorithm plug-ins, which only get information from the representation for analysis, i.e., cohesion/coupling metrics calculation.

As shown in Figure-14, each plugged component needs to provide a set of interfaces required by the container. The component can access the representation by the interface provided by the representation. Users plug the component by calling register interface of the container. After plugging the component, user can invoke it by calling send/receive interfaces provided by the message agent, which will forward the message to the target container (operation container or algorithm container) by interpret the message. A plugged component may invoke send interface of the message agent to other levels to propagate changes or acquire information.

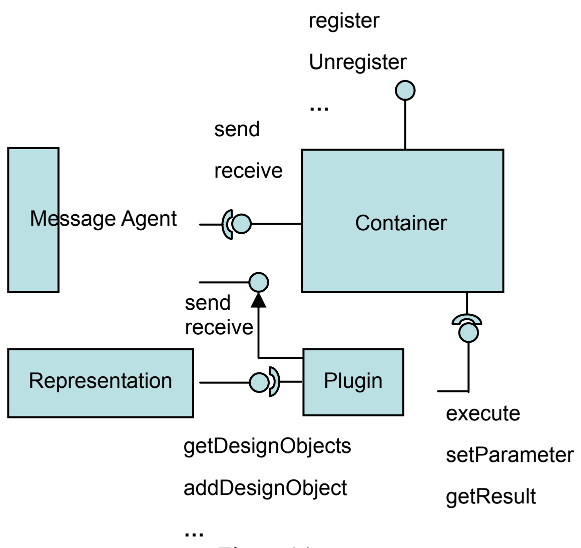


Figure 14

3.3.4 Permanent Storage

The representation and customized information need to be saved for further reloading. The purposes of permanent storage are:

1. Quick reload. Loading from software system source code is a time-consuming process. There are two phases: extracting design objects and dependencies out of source code files and reconstructing high level structure from XML system description file input by users. Both phases require parsing files and semantic analysis for various dependencies information (e.g., reference dependency between design objects, inheritance dependency, control flow

dependency). Additionally, it includes a large number of source code files for a system. It needs only to running this process one time after provide permanent storage feature.

2. Save current status of IFSO. User can apply operations and algorithms upon the representation. By saving current representation to permanent storage, we can reload it next time.
3. Save customization information. For example, plugged operations and algorithms.

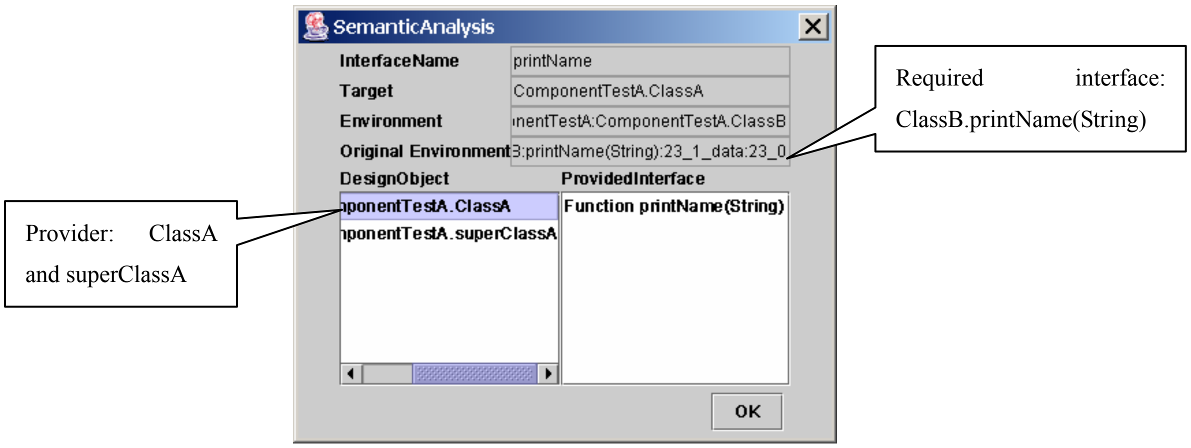
4 Prototype Implementation

The key functionalities of the implementation include: building up IFSO, automatically loading Java software system to IFSO, algorithms and operations plug-and-play model, saving IFSO to files, reload IFSO from the files, running selected algorithms or operations and a framework browser.

4.1 Design Decision

In order to leverage the abundance of XML tools and technique, we decide to use JavaML, an XML-based representation for Java code [Bad00]. After transferring source code files to XML files, we use XPath to parse XML files to IFSO. Conversely, we update XML file directly when refactoring code. The author of JavaML provides an XSLT-based back-converter to transfer XML file return to source code file.

Because JavaML doesn't cover reference relationship cross-files and method invocation relationship, it needs additional semantic analysis to get the information. The solution of our implementation for this problem is: simple syntax analysis + user input. Because a method can be overridden or overloaded, it needs complex semantic analysis to find required method. For example, ClassB has two required interfaces: ClassA.PrintName(), ClassA.PrintName(name). ClassA.PrintName() is provided by superClassA.printName() which is super class of ClassA. The solution for this issue of current implementation is let user provide this information. During parsing process, the parser will pop a dialog (Figure-18) if the parser detect ambiguous syntax situation. We will automate this process in further work.


SemanticAnalysis
Required interface: ClassB.printName(String)
Provider: ClassA and superClassA
OK

InterfaceName	printName
Target	ComponentTestA.ClassA
Environment	nentTestA:ComponentTestA.ClassB
Original Environment	3:printName(String):23_1_data:23_0

DesignObject	ProvidedInterface
nponentTestA.ClassA	Function printName(String)
nponentTestA.superClassA	

Figure-18

High-level information is provided as XML file as well. There is a system description XML file that describes component and relationship between components.

Because each level of the LSR model has similar structure except the concrete object is different. Abstract factory design pattern [GJV95] is such a pattern, which provides an interface to create families of related or dependent objects without specifying their concrete classes. The interface for creating level is same but it can create different level by given different concrete factory. And implementation for each level is independent with each other. Adding a level or implementation modification of a level will not influence other levels.

IFSO can be loaded from source code, files, database or other kinds of sources. Also it can be saved to files, database. In order to provide an extensible environment, we use visit design pattern [GJV95]. We can implement a new concrete visitor if a new source available without modifying original code. Now, there are two concrete visitors for loading: XML loading and object file loading; and one concrete visitor for saving: object file saving.

4.2 Initialize IFSO From Existing Code

Load software to IFSO is a time-consuming process. Version 1.0 of IFSO will automatically process this step. There are two parts involved. One is loading source code to logic unit level, method level, class level and component level. Another part is for component level and system level which information cannot find in source code. The user needs to provide that information.

Our solution is shown in Figure-15. We use an available tool named JavaML [Bad00] to transfer the source files to XML files. The user provides a system description XML file for system level and component level. The parser will accept these two kinds of XML files parse them to the representation of IFSO by using XPath technology. Example-1 shows XML format for the java source code. Example-2 shows XML format for system description file.

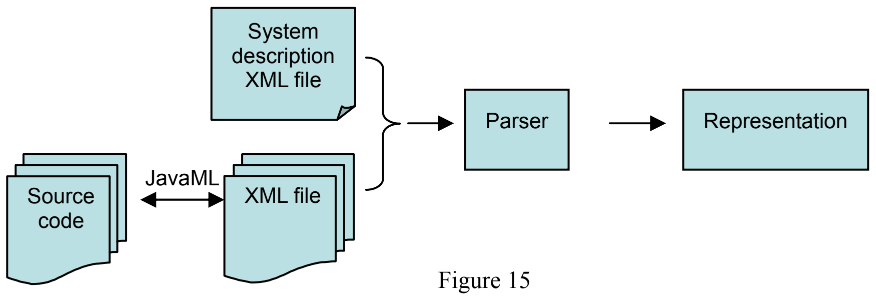


Figure 15

```
package ComponentTestA;

import ComponentTestB.*;

public class ClassA extends ComponentTestA.superClassA {
        public void printName(String name){
                java.lang.System.out.println(name + String.valueOf(k));
        }
}
```

Example 1(a): Java Source Code

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-source-program SYSTEM "java-ml.dtd">

<java-source-program>
<java-class-file name="E:/ZhengYiLei/thesis/implementation/ComponentTestA/ClassA.java">
<package-decl name="ComponentTestA"/>
<import module="ComponentTestB.*"/>
<class name="ClassA" visibility="public" line="19" col="0" end-line="40" end-col="0">
   <superclass name="ComponentTestA.superClassA"/>
   <method name="printName" visibility="public" id="ClassA:mth-33" line="23" col="8" end-line="25"
end-col="8">
      <type name="void" primitive="true"/>
      <formal-arguments>
         <formal-argument name="name" id="ClassA:frm-31"><type
name="String"/></formal-argument>
      </formal-arguments>
      <block line="23" col="42" end-line="25" end-col="8">
         <send message="println">
            <target><field-access field="out"><field-access field="System"><field-access
field="lang"><var-ref name="java"/></field-access></field-access></field-access></target>
            <arguments><binary-expr op="+"><var-ref name="name" idref="ClassA:frm-31"/><send
message="valueOf">
                  <target><var-ref name="String"/></target>
                  <arguments><var-ref name="k"/></arguments>
               </send>
            </binary-expr></arguments>
         </send>
      </block>
   </method>
</class>
</java-class-file>
</java-source-program>
```

Example 1(b): XML File for (a)

```
<system name="test">
        <components>
                <component       name="ComponentTestA"
root="e:\zhengyilei\thesis\implementation\ComponentTestA\xml-unparsed"
jarfile="e:\zhengyilei\thesis\implementation\ComponentTestA.jar">
                        <providedinterface name="Interfaces.InterfaceA" provider="ComponentTestA.ClassA">
                        </providedinterface>
                        <requiredinterface name="Interfaces.InterfaceB" provider="ComponentTestB">
                        </requiredinterface>
                </component>
                <component       name="ComponentTestB"
root="e:\zhengyilei\thesis\implementation\ComponentTestB\xml-unparsed"
jarfile="e:\zhengyilei\thesis\implementation\ComponentTestB.jar">
                        <providedinterface name="Interfaces.InterfaceB" provider="ComponentTestB.TestA">
                        </providedinterface>
                        <requiredinterface name="Interfaces.InterfaceA" provider="ComponentTestA">
                        </requiredinterface>
                </component>
        </components>
</system>
```

Example 2: System description XML file

4.3 Transfer Software System To Representation

By given a set of XML files, how does the parser map the system to the representation of IFSO as describe in section 3.2? In this section we will introduce a set of examples to expose details involved in this step.

4.3.1 System Level

As shown in Example-2, a system description XML file describes what components involved in a system, required interfaces and provided interfaces of a component, and the dependency between components. Different component model specifications need different system description files. Current version of IFSO implementation supports CMI 2.0. The representation for Example-2 is shown in Figure-16.

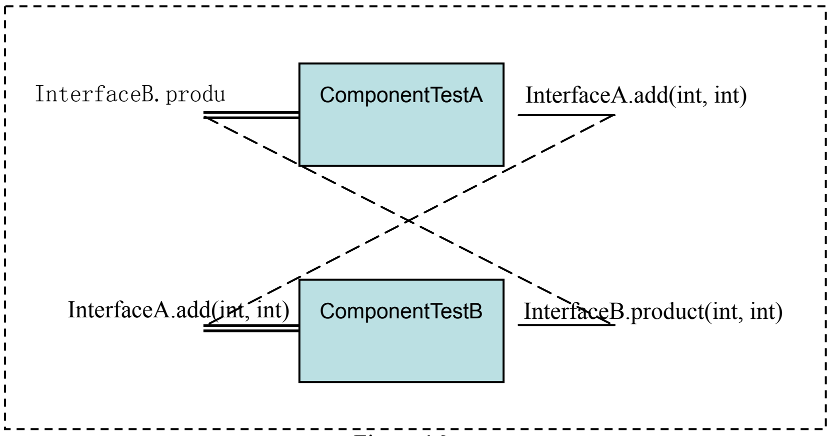


Figure 16

4.3.2 Component Level

Part of component level information comes from system description XML file, that is, what classes are involved in a component. All classes of a component are packaged in a jar file and the jar file name is as an attribute of component described system description. Figure-17 describes the representation for ComponentTestA described in Eample-2. All related class source code files

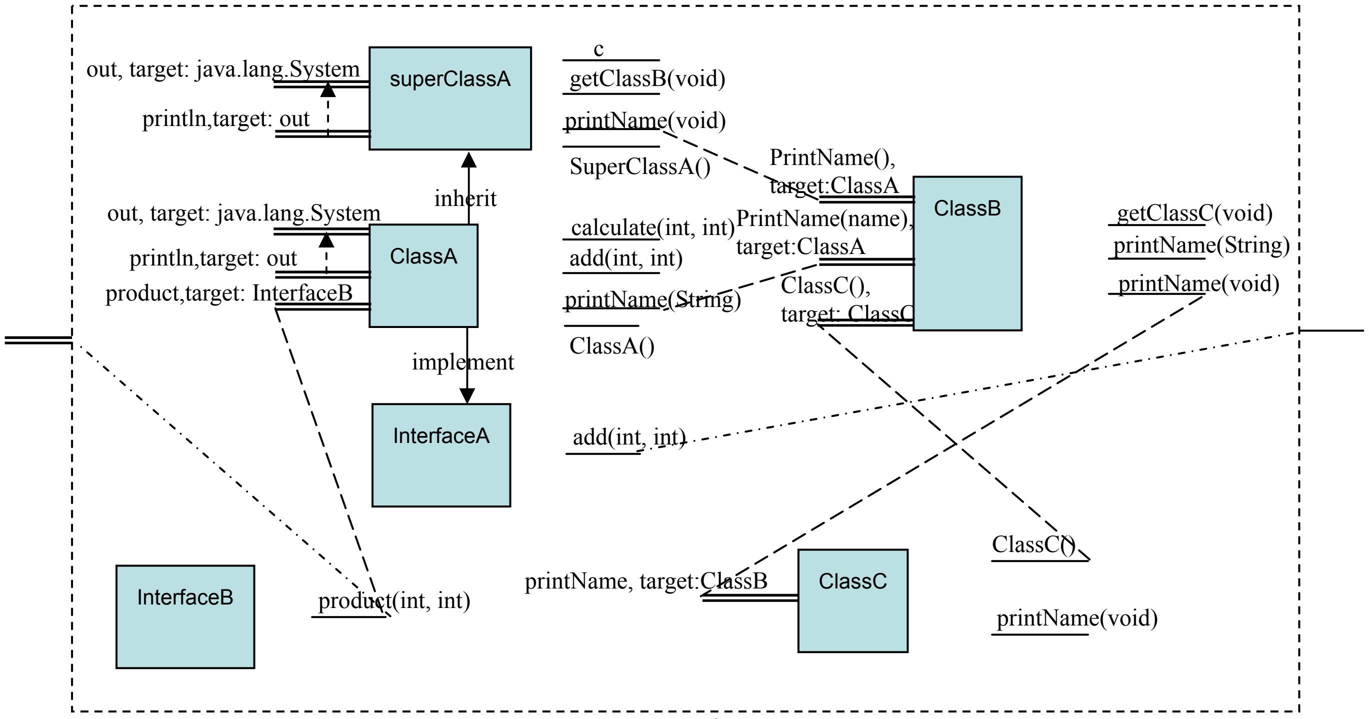


Figure-17

could be found in appendix A. InterfaceA is the provided interface of ComponentTestA and InterfaceB is the required interface.

In Fgure-17, the interface println, which target is out, is dependent on the interface out, which target is System. The source code expression is “System.out.println”. Function println is dependent on variable out. There is no provider associated with them because it is provided by system. Currently, we don’t cover this issue.

4.3.3 Class Level

All information involved in class level is gained from source code. Several aspects of Java programs (e.g., anonymous class, inner class), aren’t supported by the current implementation. Figure-19 is the representation for Eample-3. All public and protected methods/attributes are exposed as provided interfaces of the class. If there is no constructor available, a default constructor will be created as ClassB(void).

```
public class ClassB {
    private Test.ClassA a = new Test.ClassA();

    public Test.ClassC getClassC(){
        return new Test.ClassC();
    }

    public void printName(String name){
        a.printName(name);
    }

    public void printName(){
        a.printName();
    }
}
```

Example 3

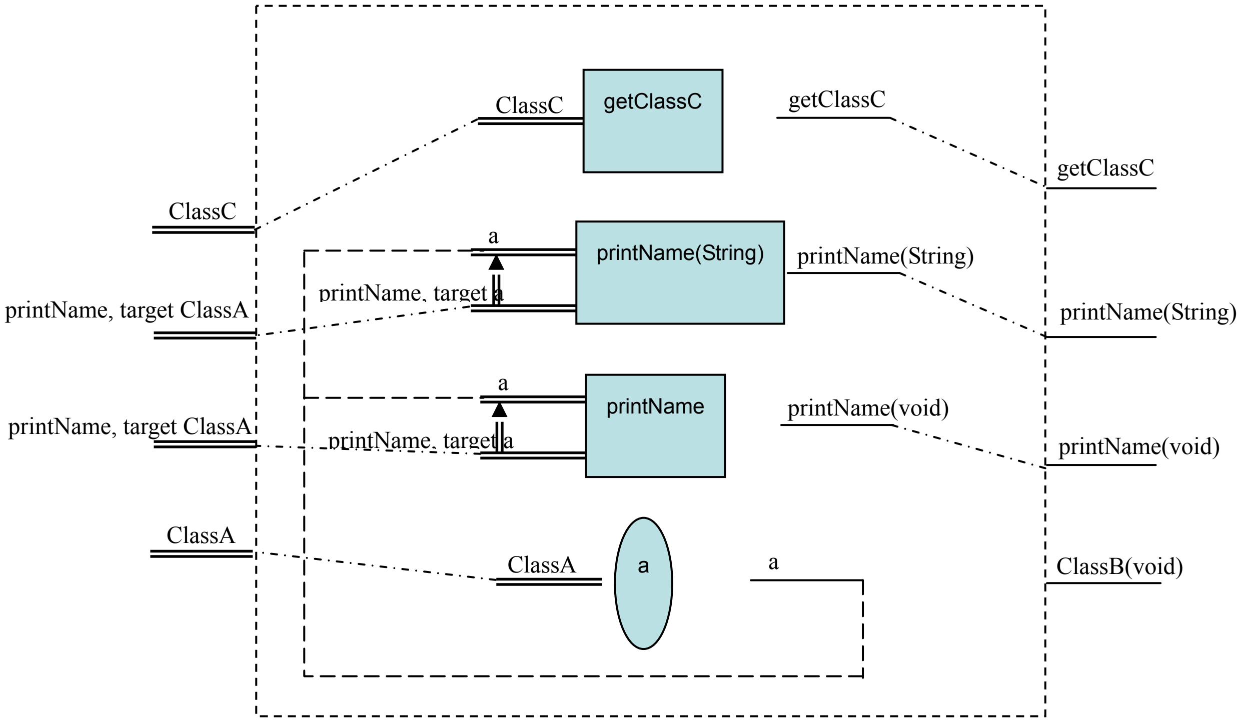


Figure 19

4.3.4 Method Level

A method includes a set of logic units. The processing is similar to class level: analysis the relationship between logic units; expose required interfaces of the logic units to the method; create provided interface for the method. In this level, syntax analysis is decided. We can find provided interface for each required interface unambiguously.

Figure-20 is the representation of Example-4. We can see that a logic unit is generated when an if/loop/while/switch statement occurs except for some special cases. In the next section will discuss detail about generation of a logic unit. “-1_0_start” is a special logic unit. Every method has such a start logic unit; there is no statement involved in a start logic unit. The provided interface of the start logic unit is the parameters of the method.

```
1public void keyReleased(KeyEvent e) {
2   int i;
3    boolean found = false;
4    char key = e.getKeyChar();

    // start new game if user has already won or lost.
 5   if (secretWordLen == wordLen || wrongLettersCount == maxTries)
{
 6       newGame();
 7       e.consume();
 8       return;
    }

    // check if valid letter
9    if (key < 'a' || key > 'z') {
10        play(getDocumentBase(), "audio/beep.au");
11        e.consume();
12       return;
    }
    // check if already in secret word
13    for (i=0; i<secretWordLen; i++) {
14        if (key == word[i]) {
15            found = true;
16            play(getDocumentBase(), "audio/ding.au");
17            e.consume();
18            return;
        }
    }
}
```

Example 4

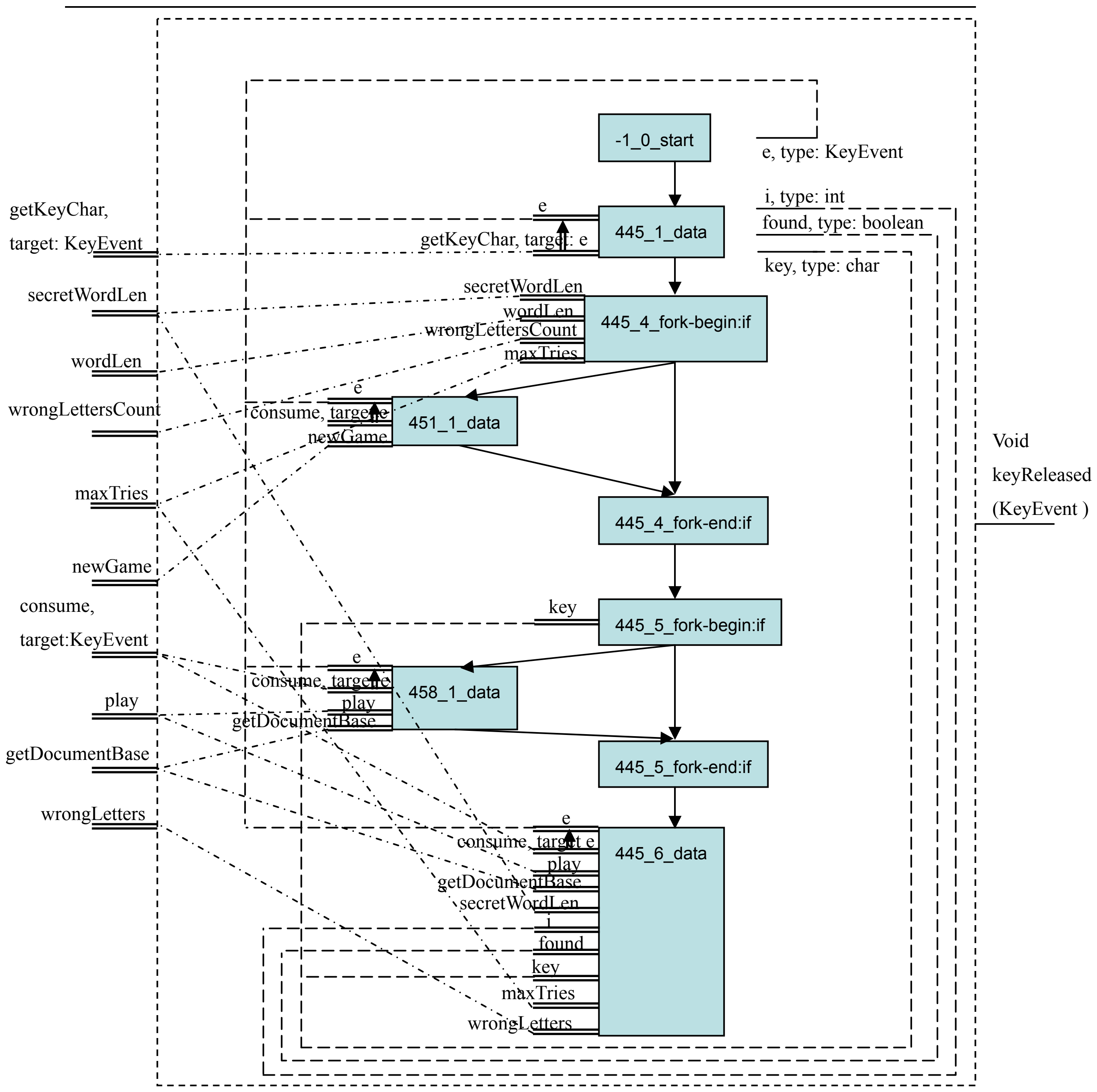


Figure 20, the following describes what lines of code included in each logic unit.

-1_0_start: is a start logic unit. The provided interfaces of a start logic unit are parameters of the method.

445_1_data: 2,3,4.

445_4_fork-begin:if: 5.

451_1_data: 6,7,8.

445_4_fork-end: is empty. There is no code included in a fork-end type logic unit.

445_5_fork-begin:if: 9

458_1_data: 10,11,12

445_5_fork-end:if: is empty.

445_6_data: 13,14,15,16,17,18. (This logic unit is a special case 2 discussed in section 4.3.5).

4.3.5 LogicUnit Level

Logic is a domain dependent concept. There is no standard definition for logic unit. The approach used in current implementation is very straightforward. When an if/loop/while statement occurs, then generate a new logic unit for the inside paragraph except several special cases. As shown in Figure-20, “451_1_data” logic unit is for 6,7,8 lines of code in Eample-4. The special cases are:

1. There is an if/loop/while/switch statement immediately following an if/loop/while/switch. As in Eample-4, there is an “if” statement after line number 13. 445_6_data logic unit corresponds to this part of code (from line number 13 to 18). The reason for this case is to simplify the implementation.
2. Only look into two levels of nesting (if/while/loop/switch includes another if/while/loop/switch). Some logic units may include control flow because this coarse-grained calculation.

The reason for above two special cases is to simplify the implementation. Future work will investigate the analysis of control flow.

```
0 int i;
// create tracker
1 tracker = new MediaTracker(this);
// load in dance animation
2 danceMusic = getAudioClip(getDocumentBase(),"audio/dance.au");
3 danceImages = new Image[40];
4 for (i = 1; i<8; i++) {
5   Image im = getImage(getDocumentBase(), "images/dancing-duke/T" + i + ".gif");
6   tracker.addImage(im, DANCECLASS);
7   danceImages[danceImagesLen++] = im;
}
```

Example 5

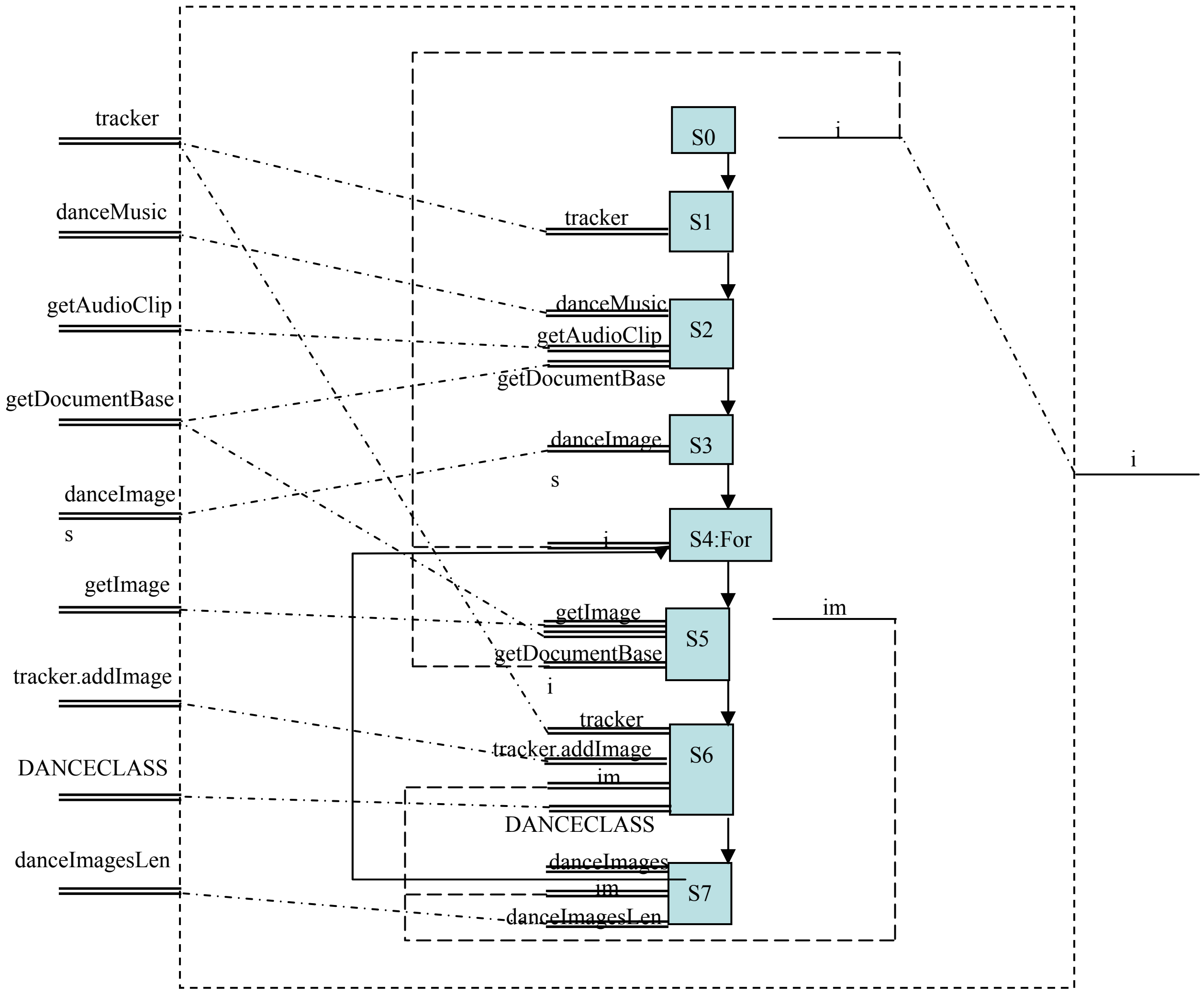


Figure 21, the number included in the name of each logic unit is the line number of the code in Example-5

4.4 Algorithms and Operations

We use XML file to describe plug-ins. Example-6 is an example for algorithm plug-ins. Exapmle-7 is an example for operation plug-ins. By given XML file and selected level, IFSO will register plug-ins to the container of the target level.

```
<algorithms>
    <algorithm name="McCabeMetrics" jarfile="e:\zhengyilei\thesis\implementation\algorithm.jar">
        <instance name="algorithm.AlgorithmTest">
        </instance>
    </algorithm>
</algorithms>
```

Example 6

```
<operations>
    <operation name="moveoperationcl" jarfile="e:\zhengyilei\thesis\implementation\operation.jar">
        <instance name="operation.MoveOperationCL">
        </instance>
    </operation>
</operations>
```

Example 7

4.5 Message Agent

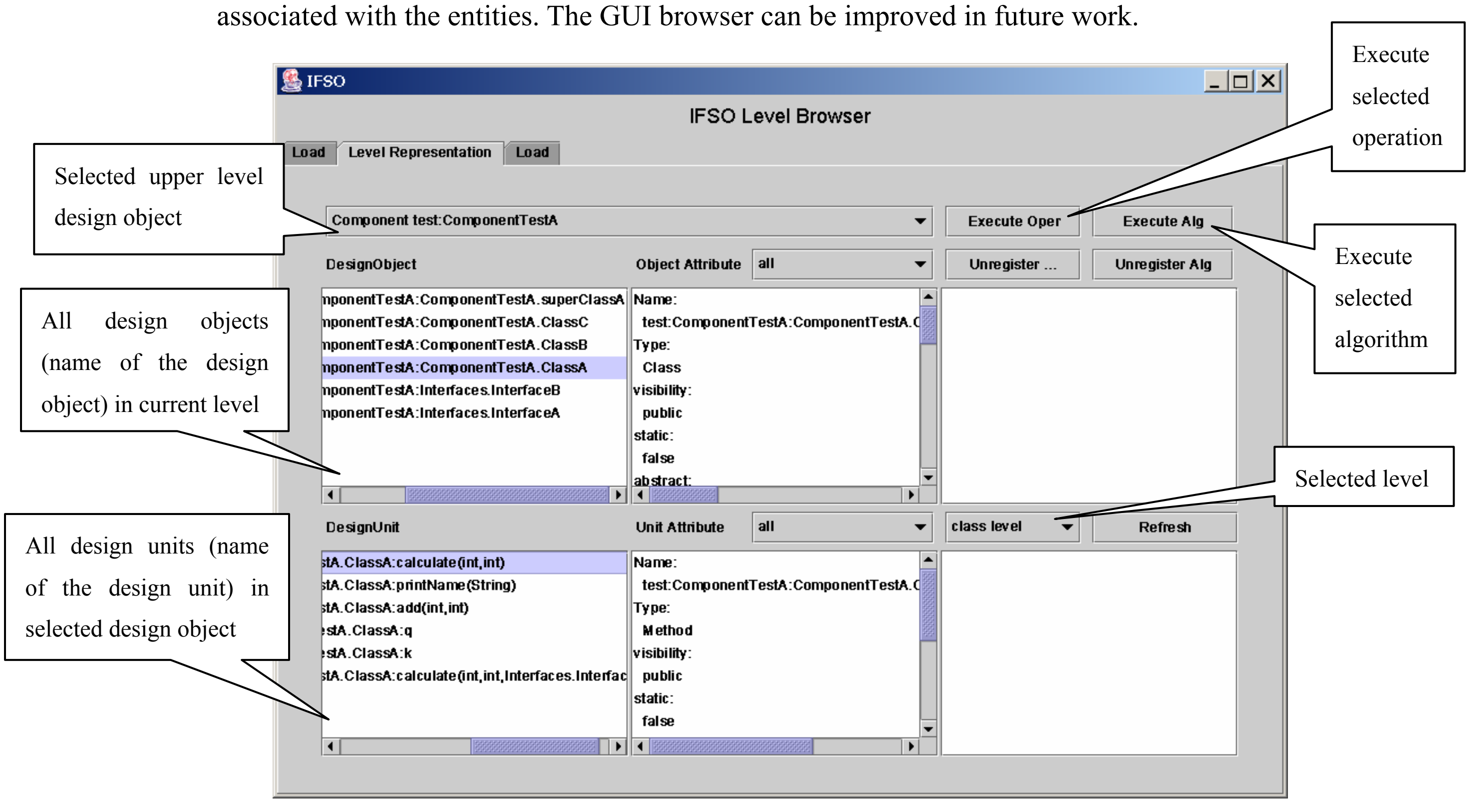
As discussed in section 3.3, there are two types of messages. One is sent to the message agent, which interprets the message to decide forward the message to which container in current level. The message format used in the message agent is: type [algorithm/operation]; data[@value]. Another one is sent to the container, which interprets the message to decide which plug-in should be executed. The message format used in the container is: name{@algorithmName};object{@objectName};type{@objectType};parameter{@name;@value};parameter{@name;@value}…. Here is an example for executing a removing operation:

`type[operation];data[``name{moveoperationcl};object{test:ComponentTestA:ComponentTestA.ClassA};type{Class};parameter{target;test:ComponentTestA:ComponentTestA.ClassB};parameter{designunitname;test:ComponentTestA:ComponentTestA.ClassA:calculate(int,int)};parameter{designunittype;Method}`**`]`**

The message is sent to a message agent and the bold part is interpreted by the message agents. After parsing the message, the message agent knows the message type is operation and then forward the rest part of the message to the operation container. The operation container will parse the rest message to decide which operation need to be executed, which design object is required, what parameters should be passed into the operation.

4.6 GUI Browser

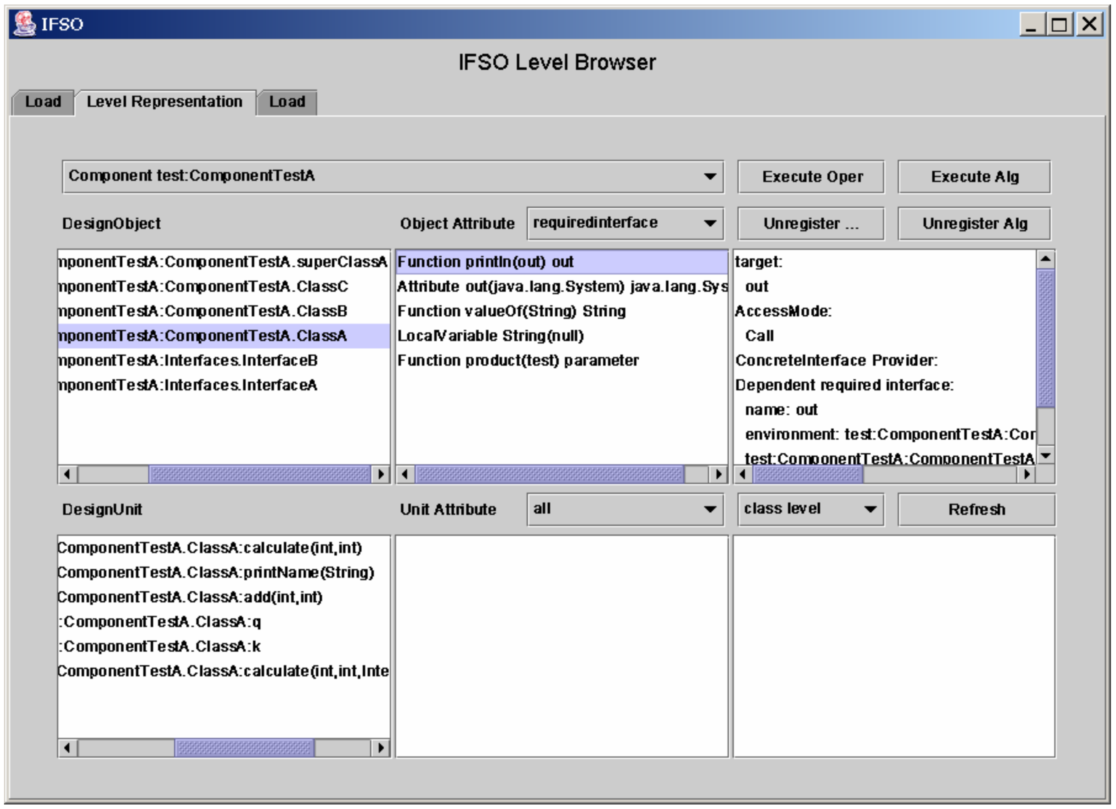
The implementation provides a simple GUI browser, which is functional, built upon a set of interfaces provided by the representation. After loading, users can browse design object of each level, required interfaces of the design object, provided interfaces of the design object, all kinds of links associated with the design object, design units within the selected design object, required interfaces of each design unit, provided interfaces of each design unit, all kinds of other attributes associated with the entities. The GUI browser can be improved in future work.



Execute selected operation
Selected upper level design object
Execute selected algorithm
All design objects (name of the design object) in current level
Selected level
All design units (name of the design unit) in selected design object
IFSO
IFSO Level Browser
Load
Level Representation
Load
Execute Oper
Execute Alg
DesignObject
Object Attribute
all
Unregister ...
Unregister Alg
DesignUnit
Unit Attribute
all
class level
Refresh

```
Component test:ComponentTestA

mponentTestA:ComponentTestA.superClassA
mponentTestA:ComponentTestA.ClassC
mponentTestA:ComponentTestA.ClassB
mponentTestA:ComponentTestA.ClassA
mponentTestA:Interfaces.InterfaceB
mponentTestA:Interfaces.InterfaceA

Name:
  test:ComponentTestA:ComponentTestA.C
Type:
  Class
visibility:
  public
static:
  false
abstract:

stA.ClassA:calculate(int,int)
stA.ClassA:printName(String)
stA.ClassA:add(int,int)
estA.ClassA:q
estA.ClassA:k
stA.ClassA:calculate(int,int,Interfaces.Interfac

Name:
  test:ComponentTestA:ComponentTestA.C
Type:
  Method
visibility:
  public
static:
  false
```

GUI Browser of IFSO: list all attributes for selected design object and attributes for selected design unit.



Component test:ComponentTestA

DesignObject

Object Attribute: requiredinterface

```
nponentTestA:ComponentTestA.superClassA
nponentTestA:ComponentTestA.ClassC
nponentTestA:ComponentTestA.ClassB
nponentTestA:ComponentTestA.ClassA
nponentTestA:Interfaces.InterfaceB
nponentTestA:Interfaces.InterfaceA
```

```
Function println(out) out
Attribute out(java.lang.System) java.lang.Sys
Function valueOf(String) String
LocalVariable String(null)
Function product(test) parameter
```

```
target:
  out
AccessMode:
  Call
ConcreteInterface Provider:
Dependent required interface:
  name: out
  environment: test:ComponentTestA:Cor
  test:ComponentTestA:ComponentTestA
```

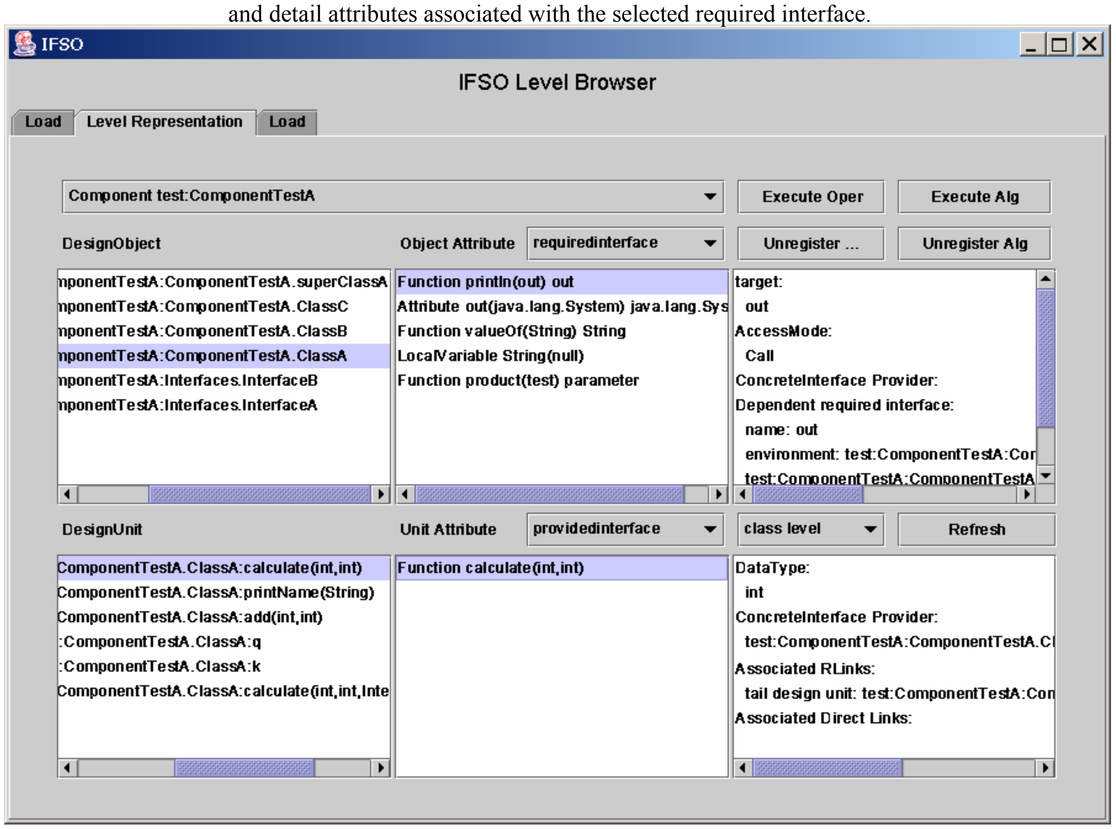
DesignUnit

Unit Attribute: all

class level

```
ComponentTestA.ClassA:calculate(int,int)
ComponentTestA.ClassA:printName(String)
ComponentTestA.ClassA:add(int,int)
:ComponentTestA.ClassA:q
:ComponentTestA.ClassA:k
ComponentTestA.ClassA:calculate(int,int,Inte
```

GUI Browser of IFSO: lists required interface of selected design object and detail attributes associated with the selected required interface.



Component test:ComponentTestA

DesignObject

Object Attribute: requiredinterface

```
nponentTestA:ComponentTestA.superClassA
nponentTestA:ComponentTestA.ClassC
nponentTestA:ComponentTestA.ClassB
nponentTestA:ComponentTestA.ClassA
nponentTestA:Interfaces.InterfaceB
nponentTestA:Interfaces.InterfaceA
```

```
Function println(out) out
Attribute out(java.lang.System) java.lang.Sys
Function valueOf(String) String
LocalVariable String(null)
Function product(test) parameter
```

```
target:
  out
AccessMode:
  Call
ConcreteInterface Provider:
Dependent required interface:
  name: out
  environment: test:ComponentTestA:Cor
  test:ComponentTestA:ComponentTestA
```

DesignUnit

Unit Attribute: providedinterface

class level

```
ComponentTestA.ClassA:calculate(int,int)
ComponentTestA.ClassA:printName(String)
ComponentTestA.ClassA:add(int,int)
:ComponentTestA.ClassA:q
:ComponentTestA.ClassA:k
ComponentTestA.ClassA:calculate(int,int,Inte
```

```
Function calculate(int,int)
```

```
DataType:
  int
ConcreteInterface Provider:
  test:ComponentTestA:ComponentTestA.Cl
Associated RLinks:
  tail design unit: test:ComponentTestA:Con
Associated Direct Links:
```

GUI Browser of IFSO: list provided interface of selected design unit and detail attributes associated with the selected provided interface.

4.6 Scenario

The following is the scenario for using our prototype.

Initialize IFSO from existing code:

Step 1: JavaML-for-jikes to transfer Java source code to XML file. Make sure package name part of each expression in the Java source code is complete. For example, system.out.println("Test") should be java.lang.system.out.println("Test"). Now, we use EXTRACT [Cal03] to transform the source code to meet this requirement.

Step 2: edit a system description XML file for the system. Package all source code as a jar file, put it under the directory described in the description file.

Step 3: run IFSO, load the system by given the system description file and work directory.

Plug in operations and algorithms

Step 1: edit a plug-in description XML file. Package all class file as a jar file, put it under the directory described in the description file.

Step 2: Register the plug-in to IFSO by given the description file and selected level.

Step 3: repeat step 1 and 2, until all required operations and algorithms are registered.

Running operations and algorithms

Go to IFSO browser, select a design object and algorithms/operations to run.

5 Discussion and Evaluation

To evaluate IFSO, we chose three case studies. In this chapter we will examine these three case studies. Each case study has two parts: case description, the features of IFSO presented by the case.

5.1 Case Study: Metrics

In our first case study, we developed three metrics (all source code can be found in Appendix C):

1. Lack of Cohesion in Method, LCOM [SC91] for class level: is a cohesion metrics for classes. A small value means high cohesion. The algorithm description is shown in Example-8.
2. Average number of parameters for class level [LK94]: is used to calculate average number of parameters for methods in a class.
3. Cyclomatic complexity metrics [MW89] for logic unit level: is used to calculate number of executable path in a module.

All algorithms need to implement the IAlgorithm (Example-9) interface for plug-and-play. After completing the implementation, users need to provide an XML algorithm description file as shown in Example-10. Given the algorithm description file, users can register the three algorithms to the target level though a GUI as shown in Figure-22. Now, we can run the algorithms to analyze the code by using GUI browser (Figure-23). IFSO provides interface to save the results as metadata of entities in the representation for future retrieving.

This case study shows three things. First, it is easy to implement algorithms based on the standard representation. As the captured functional edges between methods and attributes, LCOM algorithm can easily determine the attributes accessed by a method by scanning all associated functional edges of the method design unit. The number of parameters of each method design unit is calculated by parsing the signature attribute of the method design unit, as example, printName (String). The cyclomatic complexity metric is calculated based on the structural edges captured in logic unit level between statements. Each statement design unit knows how many outbound data flow structural edges, which is equal to the edge of the flow control graph [MW89], associated with it.

Second, levels can be customized independently. Each level focuses on specific design issues involved without disturbing other levels. As this case study shows, we calculate the number of executable paths within the logic unit level while cohesion is calculated within the class level.

Third, various algorithms can be integrated together supply a gap with each. For example, the max value of LCOM varies from class to class as the number of methods varies. Only relying on LCOM cannot understand a class correctly. It requires other algorithms to analyze the objects by using different approaches.

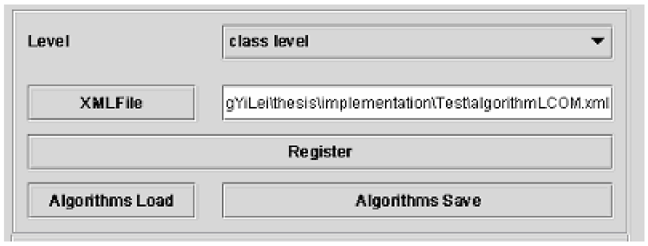
In this case study, each algorithm needs to be executed manually. In fact, the user can develop an algorithm to automate this process. It will be discussed in the next case study. Additionally, algorithms in different levels can cooperate with each other. In the third case study, we will show an example.

```
Get the target design object from current level;
For each design units in the design object {
    If the design unit is a class attribute then{
        Get provided interfaces of the design unit;
        For each provided interface of the design unit {
            Get associated direct links;
            For each direct links {
                Get the head of the direct link;
                If the head is a method then {
                    Add the attribute name to the accessing set of the method.
                }
            }
        }
    }
}
int p = 0; //P = |{(Ii, Ij) | Ii ∩ Ij = Φ}|
int q = 0;   //q = |Q = {(Ii, Ij) | Ii ∩ Ij ≠ Φ}|
For each method in the class {
    Compare the accessing set of the method with other methods {
        If there is shared attribute then
            q++;
        else
            p++;
    }
}
int result = 0;
if (p > q){
    result = p – q;
}
```

Example-8: algorithm implementation description for LCOM

```
public interface IAlgorithm extends IPlugger, Serializable{
    /**
     *
     * @return The name of the algorithm
     */
    public String getAlgorithmName();
    /**Execute the algorithm
     *
     * @param objectName The design object upon which the algorithm is required to run.
     *                   This parameter can be null, it depends on different
algorithms.
     * @param objectType The type of the design object.
     * @param il        Current level.
     * @param iar       The representation for current level.
     * @return  A string
     * @throws AlgorithmException
     */
    public String execute(String objectName, String objectType, ILevel il,
IAccessRepresentation iar) throws AlgorithmException;
    /**Set required parameters.
     *
     * @param name  Name of the parameter.
     * @param value Value of the parameter.
     * @throws AlgorithmException
     */
    public void setParameter(String name, String value) throws AlgorithmException;
    /**Get results.
     *
     * @param name Name of the result.
     * @return value of the named result.
     * @throws AlgorithmException
     */
    public String getResult(String name) throws AlgorithmException;
    /**Get the running result.
     *
     * @return A string result value.
     * @throws AlgorithmException
     */
    public String getResult() throws AlgorithmException;
}
```

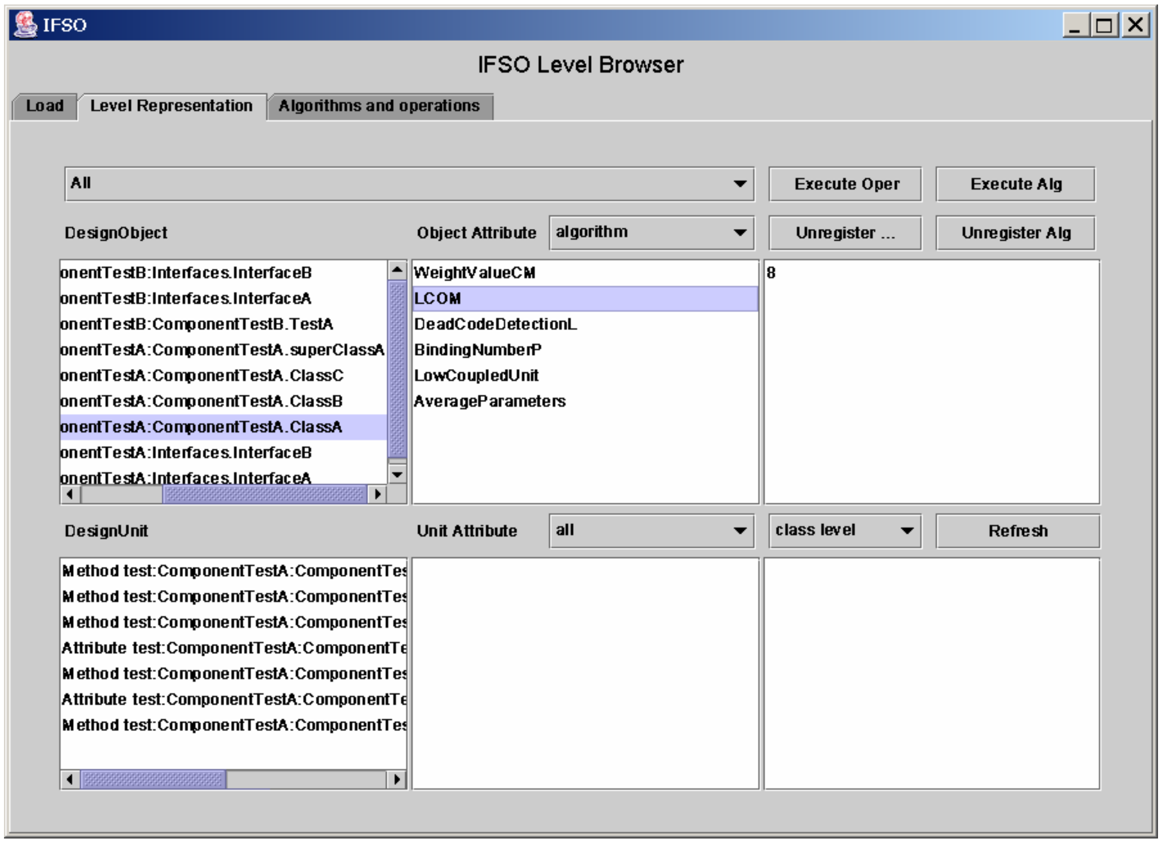
Example-9


Level: class level

XMLFile: gYiLei\thesis\implementation\TestalgorithmLCOM.xml

```
<algorithms>
        <algorithm name="LCOM" jarfile="e:\zhengyilei\thesis\implementation\algorithm.jar">
                <instance name="algorithm.LCOM">
                </instance>
        </algorithm>
</algorithms>
```

Example-10



```
onentTestB:Interfaces.InterfaceB
onentTestB:Interfaces.InterfaceA
onentTestB:ComponentTestB.TestA
onentTestA:ComponentTestA.superClassA
onentTestA:ComponentTestA.ClassC
onentTestA:ComponentTestA.ClassB
onentTestA:ComponentTestA.ClassA
onentTestA:Interfaces.InterfaceB
onentTestA:Interfaces.InterfaceA
```

```
WeightValueCM
LCOM
DeadCodeDetectionL
BindingNumberP
LowCoupledUnit
AverageParameters
```

```
8
```

```
Method test:ComponentTestA:ComponentTes
Method test:ComponentTestA:ComponentTes
Method test:ComponentTestA:ComponentTes
Attribute test:ComponentTestA:ComponentTe
Method test:ComponentTestA:ComponentTes
Attribute test:ComponentTestA:ComponentTe
Method test:ComponentTestA:ComponentTes
```

Figure 22

5.2 Case Study: Dead Code Removing

In our second case study, we developed an algorithm for dead code detection and an operation for removing the dead code. The key points of this case study are the propagation after running an operation and cooperation between levels.

The dead code algorithm detects dead classes in the class level and dead method/attribute for the method level. If the provided interfaces of a design object aren't used, the class/method/attribute object is considered to be dead. To check if the provided interface is used, a request needs to be sent to the upper level to check the number of related direct links and bindings. The provided interface is used if the number of related direct links is larger than 0 except for component level. In the component level, a provided interface is used if the provided interface is bound as a provided interface of the component no matter how many direct links are associated. Because a provided interface of a component is a contract with outside, it should always be there no matter whether it is used now. The request needs to be forwarded to further upper level if the number of related bindings is larger than 0 but the number of related links is equal to 0. We developed a DeadCodeDetection algorithm to drive dead code detection on whole systems (i.e., both class level and method level). Figure 24 shows detailed communication between algorithms located in different levels.

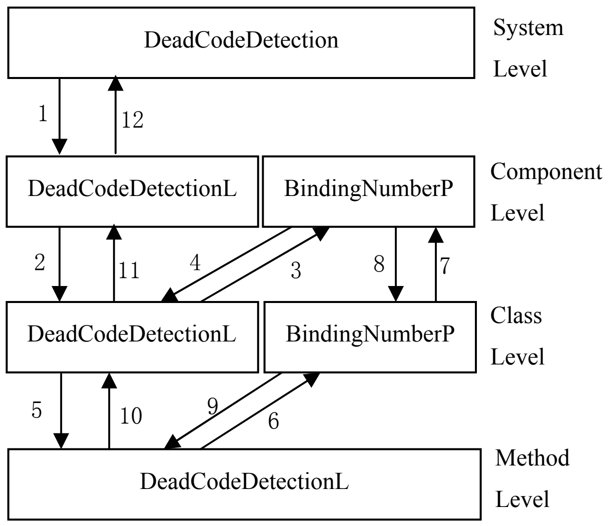


Figure 24

DeadCodeDetection: detect dead code in whole system.
DeadCodeDetectionL: detect dead code of current level and send dead code detection request to the low level.
BindingNumberP: calculate how many direct links or bindings associated with the given provided interface.

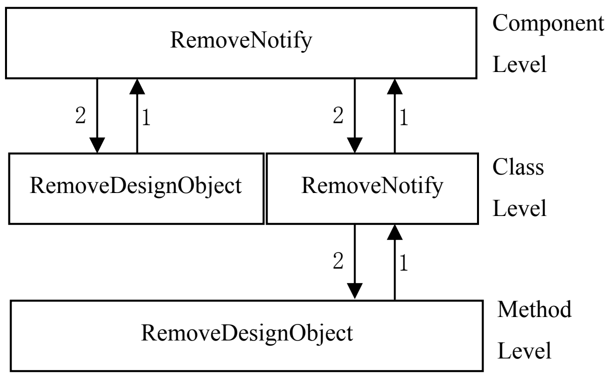
1. Send dead code detection request to component level.
2. Forward the request to class level.
3. Send a request to BindingNumberP in component level.
4. Return the result. If the number of related direct links > 0, the provided interface is used by others. If the number of related direct links = 0 but the number of bindings > 0, the provided interface is a provided interface of a component. We will regard the provided interface is used because a provided interface of a component is a contact with outside. No matter if there is used now, the interface should always be there.
5. Send dead code detection request to method level.
6. Send a request to BindingNumberP in class level.
7. If the number of related direct links = 0 but the number of bindings > 0, forward the request to component level.
8. Return the result.
9. Return the result.
10. Return detected dead method/attribute name to class level.
11. Return detected dead class/method/attribute to component level.
12. Forward the return value to system level.

The communication between levels is hosted by the message agent of each level. When an algorithm or operation requires support from other algorithms or operations, it only needs to send a request to the message agent as shown in Eample-11.

```
String message = "type[algorithm]";
String data = "name{DeadCodeDetectionL}";
message = message + ";data[" + data + "]";
String result = il.getMessageAgent().send(message, "down", null);
```

Example 11

The remove dead code operation has two parts: remove a design object in current level and send notification to upper levels. Two operations are developed: RemoveDesignObject and RemoveNotify as shown in Figure-25.



RemoveDesignObject: remove the selected design object from current level.
RemoveNotify: Accept notifications from the low level to remove all related direct links and bindings of the removed design object.

1. Send a notification to the upper level.
2. Return the result: success or fail.

Figure 25

After running the dead code detection algorithm, we can use RemoveDesignObject operation to remove the dead code. The point is that it may be found a new dead code after removing a dead code as Eample-11.

```
public class ClassB {
      private ComponentTestA.ClassA classA = new
      ComponentTestA.ClassA();
      …
      public void removeOperationTest(){
            System.out.println(classA.getName());
      }

}
```

Example 11(a)

```
public class ClassA extends
ComponentTestA.superClassA implements
Interfaces.InterfaceA{
      ….
      public String getName(){
            String s = new String("this is for remove
operation");
            return s;
      }
}
```

Example 11(b)

removeOperationTest is a dead method. Method getName is only called by the method removeOperationTest. After remove the method removeOperationTest, the method getName becomes a dead code as well.

In Fgure-26 shows the detail representation of ClassA and ClassB in class level and ComponentTestA, which include ClassA and ClassB, in component level. When the target object removeOperationTest is removed, all related links are removed. The modification is naturally propagated to the whole system.

In this case study, several advantages of our framework are exhibited. First, the LSR model provides a novel abstract representation for software. Levels have similar structure and design objects have same representation. An operation or algorithm can be reused in different levels (but not all operations and algorithms can be reused in multiple levels). RemoveDesignObject, RemoveNotify, BindingNumberP and DeadCodeDetectionL are reused in several levels. The burden of development for operations and algorithms is minimized. Second, a modification can be propagated easily. The LSR model captures dependencies between levels. An operation gets all kinds of related entities by checking all related links. Third, IFSO provides a robust communicate mechanism between levels. By sending a message, all kinds of operations and algorithms can cooperate together.

The issue, which isn’t addressed in this cast study, is how to write back the modification to source code. We will discuss this issue in next cast study.

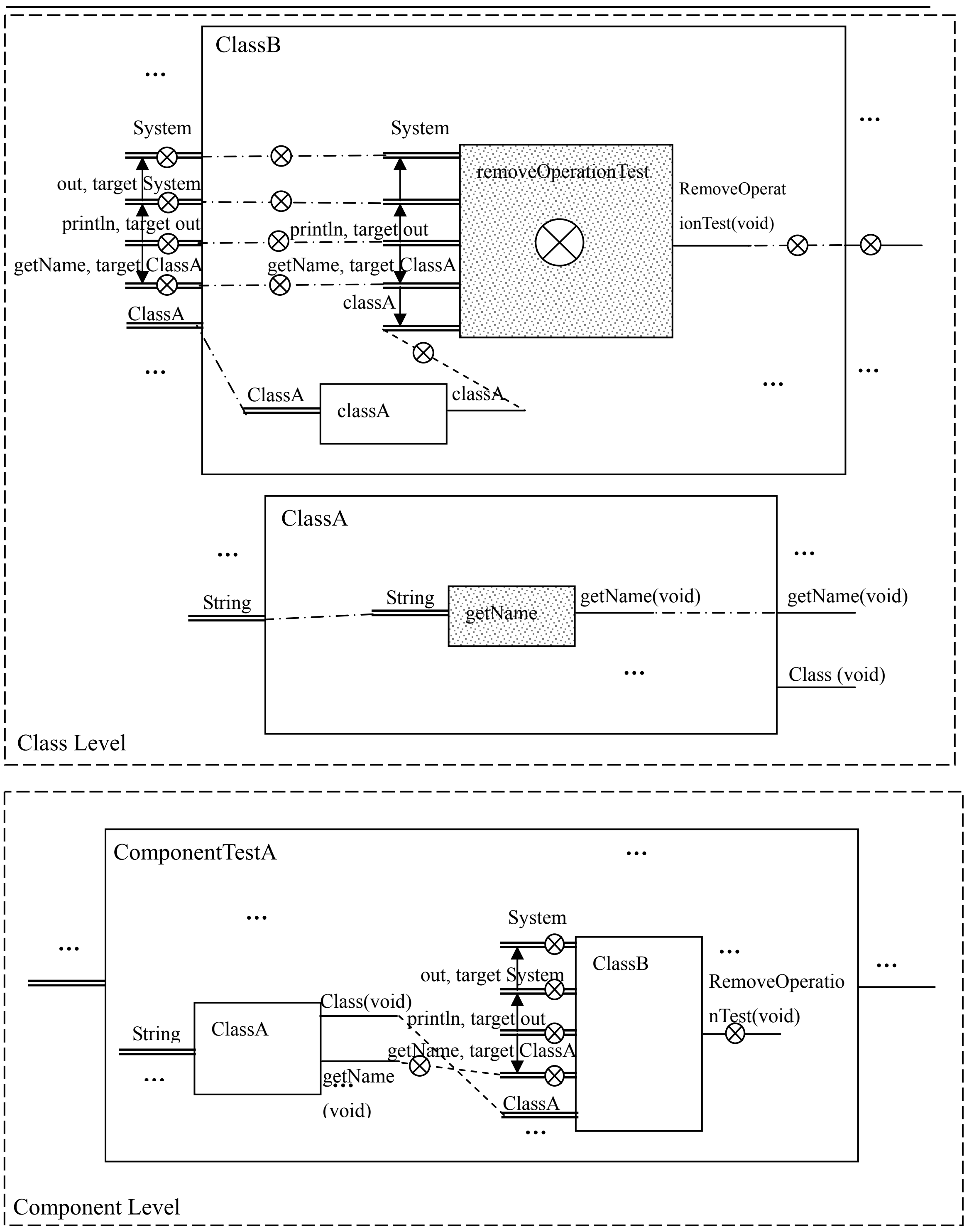


Figure 26: the representation of the component level and class level for dead code removing.
The gray object is the dead method. All entities labeled with ⊗ will be removed.

5.3 Case Study: Low Coupled Unit Detection

In our third case study, we developed an algorithm to detect lower coupled unit in the class level and an operation to move a method from this class to another class.

We developed an algorithm named LowCoupledUnit for the class level, an operation named MoveOperationCL for the class level, an operation named MoveNotifyComponentLevel for the component level and an operation named UpdateEnvironment for the levels under the class level.

If a design unit is higher coupled with other design object, this design unit is detected by LowCoupledUnit. The approach of detection is: calculate three numbers as described in Figure-27. NB means the number of dependencies upon another design object. NBDL means the number of dependencies of another design object upon the design unit. NDL means the number of dependencies of the design unit on the design object. If NB or NBDL > NDL, the design unit is low coupled with current design object. It may need to be moved to another design object.

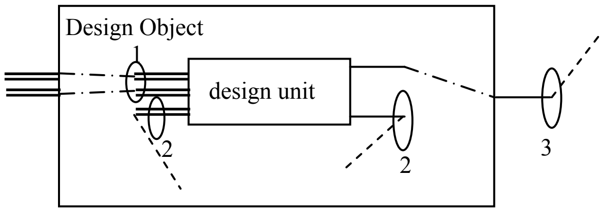


Figure 27

1. is number of binding, which is related to the required interfaces of the design object. (NB)
2. is number of direct links associated with the provided interfaces of the design unit. (NDL)
3. is the number of direct links associated with the provided interfaces of the design object, which is bound with the provided interface of the design unit.(NBDL)

As shown in Figure-28, method calculate (int, int) is a lower coupled design unit with ClassA. The NB and NDL values are equal to zero while NBDL is equal to 1. It is better to move calculate from ClassA to ClassB.

After detect a lower coupled design unit, the next step is moving the design unit. It is a complex operation. For a class, we need to check if the method is an inherited method or the method is defined in the implemented interface. Modification of the directed graph in IFSO is quite straightforward. The key point is syntax analysis. For example, all expressions, which access the design unit, should be changed. It may need to add a new instance variable definition. The implementation of MoveOperationCL only considers a simple situation: move the design unit from source design object to target design object without consider syntax effects.

Another issue is modification of source code file. After the representation of IFSO changed, we need to write back to source code file. In current work, we just provide a rough solution to prove it can be done without much effort. The solution is that we save org.w3c.dom.Node instance, which is retrieved during parsing the XML-based source code file to IFSO, as metadata of the entity of the representation. When an operation modifies the representation, it also update the XML-based source code file though the Node metadata. Comprehensive solution for consistency between the representation of IFSO and source code file isn't addressed in current research work.

```
//get the Node metadata associated with the design unit.
Node nObject = idu.getNode();
//get the parent node.
Node nParentSource = nObject.getParentNode();
//remove the method node from parent class node.
nParentSource.removeChild(nObject);
```

Example 12

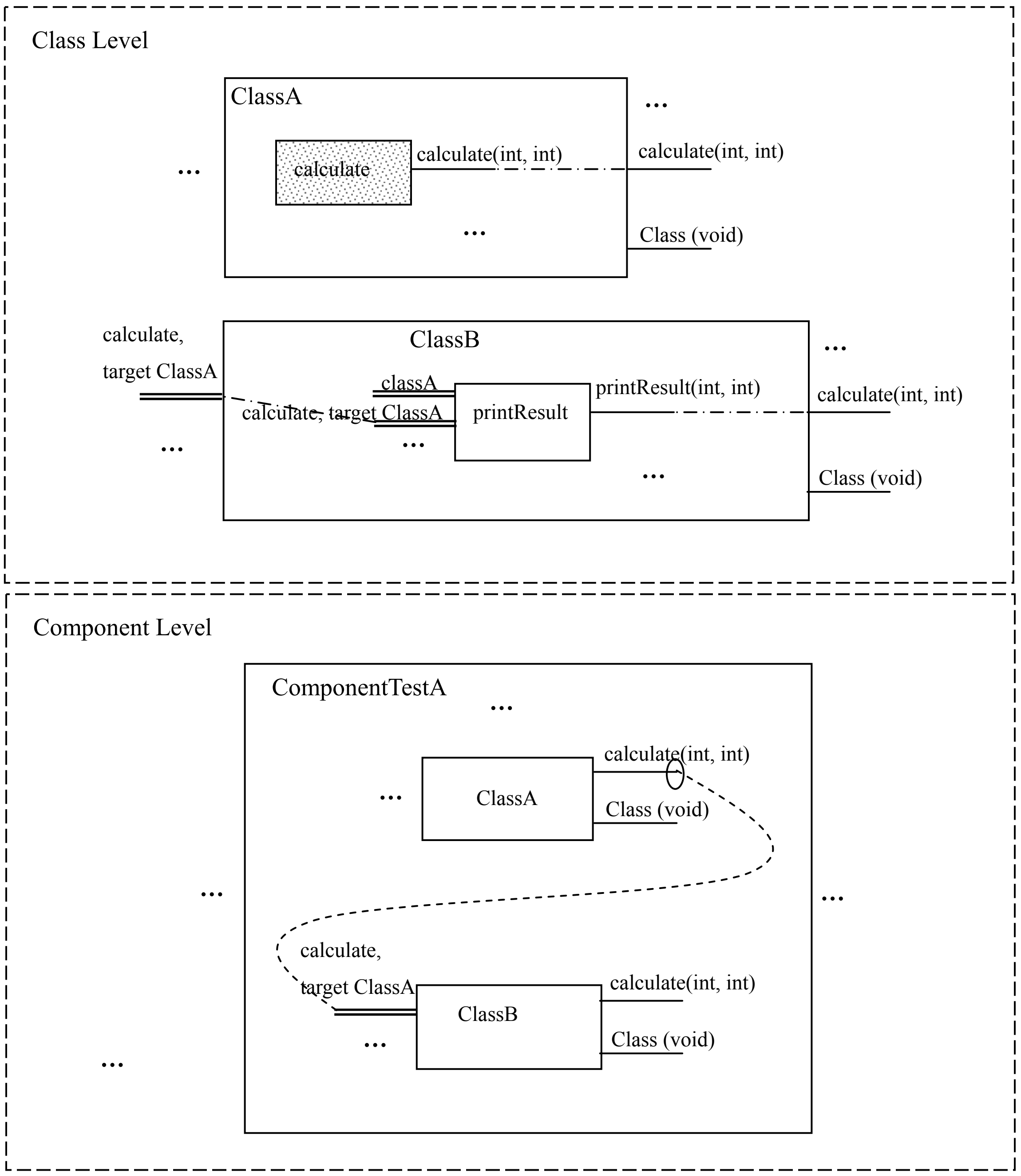


Figure 28: the representation of the component level and class level for Lower coupled unit detection.

6 Conclusions and Future work

6.1 Conclusions

Due to the growing complexity of software system, it has become increasingly critical to improve efficiency of software refactoring. One key step towards achieving such a goal is to provide a customizable framework as foundation to integrate various software refactoring technologies together.

In past, a lot of research work focused on software understanding, software measurement, code manipulating, code analysis, etc. These technologies are involved in different stages of software refactoring. On the other hand, a software system is designed in several granularity levels. Each granularity level has its own design issues, measurements, and code manipulation. Cooperating various technologies in various levels (that is, design algorithms and operations involved in each level) together is one way to support automatic/semi-automatic software analysis as proposed by this thesis.

A robust conceptual model, LSR model, is developed as standard representation for software. The model represents the software system in a hierarchical manner. Each design granularity level involved in the software system is captured as a level in the model. The IFSO framework is designed based on the LSR model. It instantiated the LSR model for a specific kind of software, e.g., component-based software. Additionally, the framework provides a communication mechanism to cooperate between levels as a whole. Each level can be customized independently based on the plug-and-play model of IFSO. Code transformation, from source code to the LSR model, permanent storage and code refactoring are addressed also.

Provided with such a framework, users can plug in algorithms and/or operations to levels according to their specific problems. Each granularity level can be analyzed and manipulated separately while users get a global view and feedback from IFSO. An algorithm or operation can be developed and reused easily based on the standard representation defined in the LSR model. As the case studies discussed in chapter 5, IFSO clearly isolated the responsibility for each level. Cooperating a set of simple algorithms to achieve a complex task is a more robust way for software refactoring presented by this thesis.

6.2 Future work

There are couple directions about the future work.

1. Currently, IFSO focuses only on component-based software system. There are fixed number of levels and pre-defined attributes for each entity, that is, the customization processing of the LSR model for the component-based software systems is hard coded in current version IFSO framework. In future works, the framework can be extended to provide a customization tool to support this process. Users can possibly define the number of levels, entities in each level, attributes of each. The LSR model guarantees such kind of flexibility. The key issue of this extension is to design a plug-in model and interfaces for the code transformation. After finishing the customization, users can plug in a model, which is developed by the standard interface defined by IFSO, to transform the source code to the customized framework.
2. To store the information in a database. Leveraging the ability of databases provides more

power query ability to users. Also extend IFSO to support large-scale software system.

3. To provide a script running feature for executing algorithms and operations in batches. Currently, users have to execute an operation or algorithm manually though the GUI browser. If IFSO can accept a script to run the operations or algorithms in batches, users can run analysis or software refactoring task more effectively.
4. To better support plugged operations. As discussed in the three cases in chapter 5, an operation has to maintain the consistence between the representation of IFSO and source code. How to maintain the consistence is a key issue need to be addressed in future work. A set of basic code refactoring operations needs to be integrated as part of the framework. When the representation is changed by a plugged operation, a corresponding basic code refactoring operation should be executed automatically. An alternative way can be executing code-refactoring operations in batches according to the operations log, which records a set of the representation modification operations generated during the process. It requires further research work to observe which approach is better.
5. Integrating available code manipulate tools to IFSO to support the basic code refactoring operations.
6. To develop a set of meaningful algorithms as basic algorithm package associated with the framework.

Appendix A: example source code

A.1 ComponentTestA

1 SuperClassA
2 ClassA
3 ClassB
4 ClassC

```
package ComponentTestA;

public class superClassA {
	protected Test.ClassC c = new Test.ClassC();
	private java.lang.String type = "super classA";

	public Test.ClassB getClassB(){
		return new Test.ClassB();
	}

	public void printName(){
		java.lang.System.out.println(type);
	}
}
```

```
package ComponentTestA;

import Interfaces.*;
import ComponentTestB.*;

public class ClassA extends ComponentTestA.superClassA implements Interfaces.InterfaceA{
	private int k = 2;
	private int q = 3;

	public void printName(String name){
		java.lang.System.out.println(name + String.valueOf(k));
	}

	public int add(int i, int j){
		return i+j + k + q;
	}

	public int calculate(int i, int j){
		return i+j;
	}

	public int calculate(int i, int j, Interfaces.InterfaceB test){
		//ComponentTestB.TestA test = new ComponentTestB.TestA();
		return test.product(i, j);
	}

	public String getName(){
		String s = new String("this is for remove operation");
		return s;
	}

}
```

```
package ComponentTestA;

public class ClassB {
        private ComponentTestA.ClassA classA = new ComponentTestA.ClassA();

        public Test.ClassC getClassC(){
                return new Test.ClassC();
        }

        public void printName(String name){
                classA.printName(name);
        }

        public void printResult(int i, int j){
                int result = classA.calculate(i, j);
                classA.printName(String.valueOf(result));
        }

        public void printName(){
                classA.printName();
        }

        public void RemoveOperationTest(){
                System.out.println(classA.getName());
        }

}
```

```
package ComponentTestA;

public class ClassC {
	private ComponentTestA.ClassB classB = new ComponentTestA.ClassB();
	private ComponentTestA.ClassA classA = new ComponentTestA.ClassA();

	public void printName(){
		classB.printName();
	}

	public int calculate(int i, int j){
		int result = classA.calculate(i,j);
		result = result + j;
		return result;
	}
}
```

A.2 ComponentTestB

1 TestA

```
package ComponentTestB;

import Interfaces.*;

public class TestA implements Interfaces.InterfaceB{
	public int product(int i, int j){
		return i*j;
	}
}
```

Appendix B: mini-tutorial

B.1 How to write an algorithm

```
public interface IAlgorithm extends IPlugger, Serializable{
     /**
      *
      * @return The name of the algorithm
      */
     public String getAlgorithmName();
     /**Execute the algorithm
      *
      * @param objectName The design object upon which the algorithm is required to run.
      *                     This parameter can be null, it depends on different algorithms.
      * @param objectType The type of the design object.
      * @param il         Current level.
      * @param iar        The representation for current level.
      * @return   A string
      * @throws AlgorithmException
      */
     public String execute(String objectName, String objectType, ILevel il,
IAccessRepresentation iar) throws AlgorithmException;
     /**Set required parameters.
      *
      * @param name  Name of the parameter.
      * @param value Value of the parameter.
      * @throws AlgorithmException
      */
     public void setParameter(String name, String value) throws AlgorithmException;
     /**Get results.
      *
      * @param name Name of the result.
      * @return value of the named result.
      * @throws AlgorithmException
      */
     public String getResult(String name) throws AlgorithmException;
     /**Get the running result.
      *
      * @return A string result value.
      * @throws AlgorithmException
      */
     public String getResult() throws AlgorithmException;
}
```

Step 1: develop an algorithm based on the interface provided by IFSO. Each algorithm plug-ins

```
public interface IAccessRepresentation {
    /**Return a list of design objects in current level.
      */
    public Enumeration getDesignObjects();
    /**Return a design object interface by given the name and type of the design object.
      */
    public IDesignObject getDesignObject(String name, String type);
    /**Return a design object interface by given the name and type of the design object.
    */
    public IDesignObject getDesignObject(String name, DesignElementType type);
    /**Return a list of design objects by given the parent name.
    */
    public Enumeration getDesignObjects(String parent);
}
```

must implement the interface IAlgorithm. The IAlgorithm.execute method will be called when an algorithm executed. Users can get a list of design object interfaces in current level though the interface IAccessRepresentation. IDesignObject provides a set of methods to get associated entity as described in the LSR model. Each entity has a corresponding interface, which can be used to get its attributes or associated other entities. Based on these interfaces, users can develop an algorithms or operations.

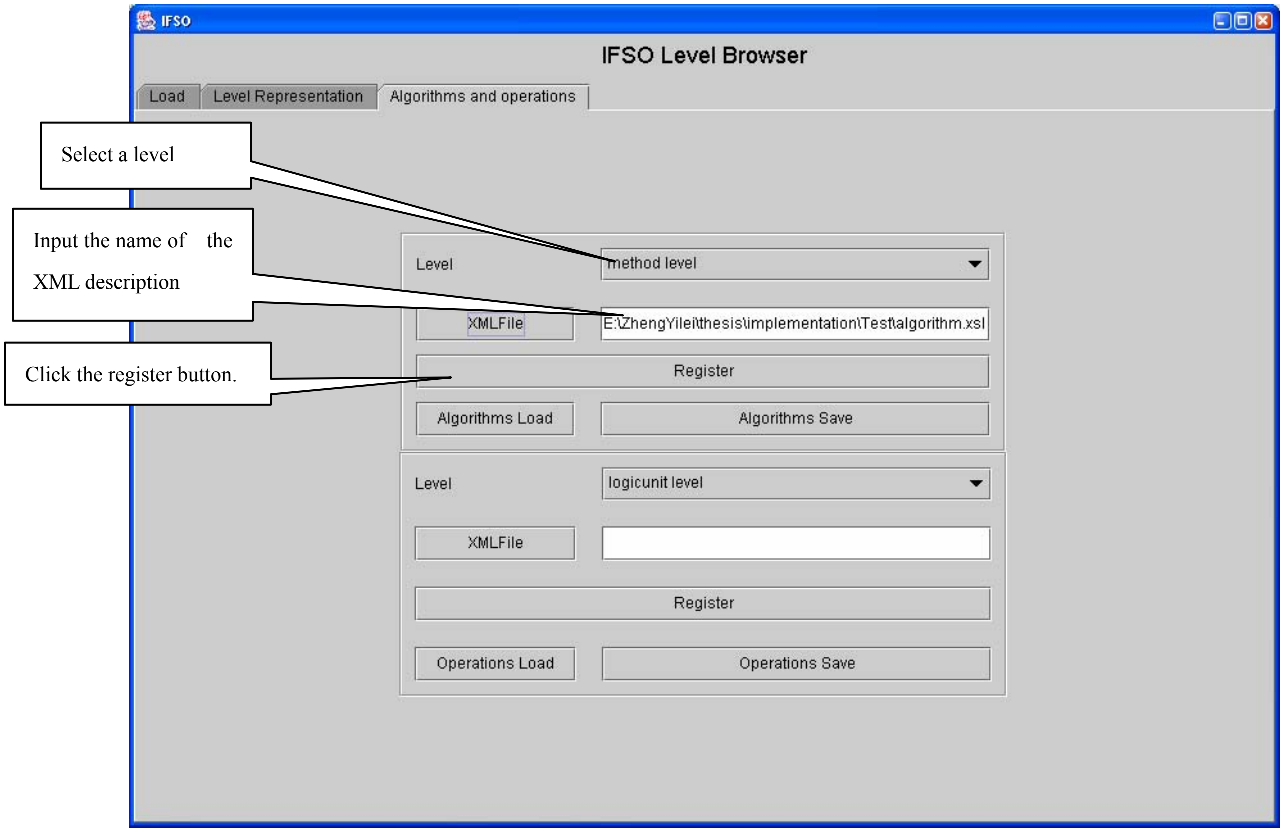
Step 2: pack the code to a jar file.

Step 3: write a XML description for the algorithm. The instance name should be the class name,

```
<algorithms>
        <algorithm name="McCabeMetrics" jarfile="e:\zhengyilei\thesis\implementation\algorithm.jar">
                <instance name="algorithm.AlgorithmTest">
                </instance>
        </algorithm>
</algorithms>
```

which implements the IAlgorithm interface. The algorithm name should be same as the return value of IAlgorithm.getAlgorithmName ().

Step 4: register the algorithm.



Example: the algorithm LCOM [] is described in Eample-8. The source code is following:

```
/*
 * Created on 2004-2-11
 *
 * Thesis Project Name: IFSO (Integrated Framework for SOftware analysis and refactoring
 * School: Worcester Polytechnique Institution
 * Advice: Professor George T. Heineman
 *
 */
package algorithm;

import java.util.Properties;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;
import edu.wpi.cs.ifso.common.*;
import edu.wpi.cs.ifso.algorithm.exceptions.AlgorithmException;
import edu.wpi.cs.ifso.algorithm.interfaces.IAlgorithm;
import edu.wpi.cs.ifso.interfaces.*;

/**
 * @author Yilei Zheng
 *
 *
 */
public class LCOM implements IAlgorithm {
    private String name = new String("LCOM");
    private Properties properties = new Properties();

    public String execute(
        String objectName,
        String objectType,
        ILevel il,
        IAccessRepresentation iar)
        throws AlgorithmException {
        if (il == null || iar == null){
```

```
        throw new AlgorithmException("LCOM.execute: input null parameter.");
    }

    if (objectName == null || objectType == null){
        throw new AlgorithmException("LCOM.execute: input null object name or object
        type.");
    }

    //get the design object
    IDesignObject ido = iar.getDesignObject(objectName, objectType);
    if (ido == null){
        throw new AlgorithmException("LCOM.execute: cannot get design object: " +
        objectName + " type: " + objectType + " from representation.");
    }

    try{
        Hashtable hResult = new Hashtable();
        Vector vMethod = new Vector();

        /*get accessing instance variables set for each method, and save them to a
        hashtable. The key of the hashtable is the method name.*/
        //1. get all design units within the design object.
        Enumeration eDesignUnit = ido.getDesignUnits();
        if (eDesignUnit != null){
            while(eDesignUnit.hasMoreElements()){
                IDesignUnit idu = (IDesignUnit)eDesignUnit.nextElement();
                //2. find all class attributes and then get all associated direct links.
                if (idu.getType().equals(DesignElementType.Attribute)){

                    Enumeration eProvidedInterface = idu.getProvidedInterfaces();
                    if (eProvidedInterface != null){
                        while(eProvidedInterface.hasMoreElements()){
                            IProvidedInterface ipif =
                            (IProvidedInterface)eProvidedInterface.nextElement();
                            Enumeration eDirectLink =
                            ido.getDirectLinkT((IConnectElement)ipif);
                            if (eDirectLink != null){
                                //3. get head of the direct link, if it is a method,
                                //save it to the hashtable.
                                while(eDirectLink.hasMoreElements()){
                                    IDirectLink idl =
                                    (IDirectLink)eDirectLink.nextElement();
                                    IConnectElement ice =
                                    (IConnectElement)idl.getHead();
                                    if
                                    (ice.getEnvironment().getObjectType().equals(D
                                    esignElementType.Method)){
                                        Vector vAccess = null;
                                        if
                                        (hResult.containsKey(ice.getEnvironment()
                                        .toString())){
                                            vAccess =
                                            (Vector)hResult.get(ice.getEnvironmen
                                            t().toString());
                                        }else{
                                            vAccess = new Vector();
                                        }

                                        if (
                                        vAccess.indexOf(idu.getEnvironment().getU
                                        nitName()) < 0 ){

                                            vAccess.addElement(idu.
                                            getEnvironment().getUnitName());
                                        }

                                        hResult.put(ice.getEnvironment().
                                        toString(), vAccess);

                                    }
                                }
                            }
                        }
                    }
                }else if (idu.getType().equals(DesignElementType.Method)){
                    vMethod.addElement(idu);
                }
            }
        }

        //P = |{(Ii, Ij) | Ii ∩ Ij = Φ}|
        //q = |Q = {(Ii, Ij) | Ii ∩ Ij ≠ Φ}|
        int p = 0; //Ii^Ij = null
        int q = 0;   //Ij^Ij != null
        for(int i=0; i<vMethod.size(); i++){
            for(int j=i+1; j<vMethod.size(); j++){
                Vector vAccess1 =
                (Vector)hResult.get(((IDesignUnit)vMethod.elementAt(i)).getEnvironme
                nt().toString());
                Vector vAccess2 =
                (Vector)hResult.get(((IDesignUnit)vMethod.elementAt(j)).getEnvironme
                nt().toString());
                if (vAccess1 == null || vAccess2 == null){
                    p++;
                }else{
                    boolean flag = false;
                    for(int k=0; k<vAccess1.size(); k++){
                        if (vAccess2.indexOf(vAccess1.elementAt(k)) >= 0){
                            q++;
                            flag = true;
                            break;
                        }
                    }
                    if (flag == false){
                        p++;
                    }
                }
            }
        }

        int result = 0;
        if (p >= q){
            result = p - q;
        }

        //save the result to the design object as metadata.
        ido.saveAlgorithmResult(name, String.valueOf(result));
        properties.setProperty("result", String.valueOf(result));
        return properties.getProperty("result");

    }catch(Exception e){
        throw new AlgorithmException(e.getMessage());
    }
}

public void setParameter(String name, String value)
    throws AlgorithmException {
    // TODO Auto-generated method stub
    properties.setProperty(name, value);
}

public String getResult(String name) throws AlgorithmException {
    // TODO Auto-generated method stub
    return properties.getProperty(name);
}

public String getResult() throws AlgorithmException {
    // TODO Auto-generated method stub
    return properties.getProperty("result");
}

public String getAlgorithmName() {
    return name;
}

}
```

B.2 How to write an operation

Step 1: develop an operation based on the interface provided by IFSO. Each operation plug-ins must implement the interface IOperation. The IOperation.execute method will be called when an operation executed. Users can get a list of design object interfaces in current level though the

```
public interface IOperation extends IPlugger, Serializable{
    /**Return the name of the operation.
     *
     * @return Operation name.
     */
    public String getOperationName();
    /**Execute the operation.
     *
     * @param objectName Name of the target design object.
     * @param objectType Type of the target design object.
     * @param itran      Transaction management.
     * @param il      Level interface.
     * @param iar     Representation interface.
     * @param iur     Representation update interface.
     * @return             Execute result. The format of the return value is defined
     *                     by users.
     * @throws OperationException
     */
    public String execute(String objectName, String objectType, ITransaction itran, ILevel
il, IAccessRepresentation iar, IUpdateRepresentation iur) throws OperationException;
    /**Set up required parameter pairs.
     *
     * @param name   Name of the parameter.
     * @param value  Value of the parameter.
     * @throws OperationException
     */
    public void setParameter(String name, String value) throws OperationException;
    /**Get the running result.
     *
     * @param name Name of the result.
     * @return value of the result.
     * @throws OperationException
     */
    public String getResult(String name) throws OperationException;
    /**Get the running result string as the return value of the execute method.
     *
     * @return is same as execute method.
     * @throws OperationException
     */
    public String getResult() throws OperationException;

}
```

interface IAccessRepresentation. IDesignObject provides a set of methods to get associated entity as described in the LSR model. Each entity has a corresponding interface, which can be used to get its attributes or associated other entities. Based on these interfaces, users can develop algorithms or operations. The key issue of development of an operation is consistency control. Same as database, we use a transaction to control the consistency. ITransaction interface provides beginTransaction, commitTransaction, rollbackTransaction and action method. The method action is used to add a action to current transaction. All actions will be executed in batches when the commitTransaction is executed. If occurred any error during the operation, users can call rollbackTransaction to cancel the modification. In fact, the rollbackTransaction method just removes the saved action list.

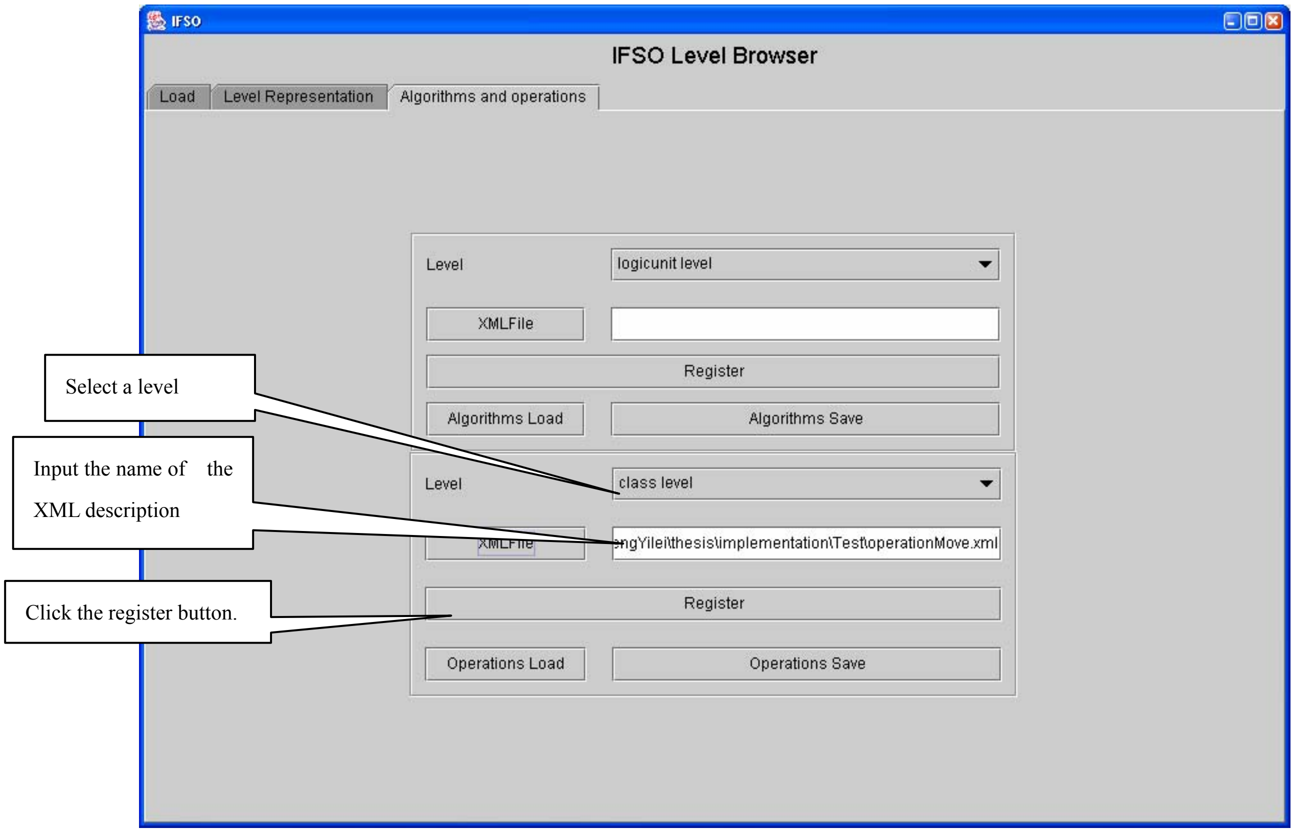
Step 2: pack the code to a jar file.

Step 3: write a XML description for the operation. The instance name should be the class name,

```
<operations>
        <operation name="moveoperationcl" jarfile="e:\zhengyilei\thesis\implementation\operation.jar">
                <instance name="operation.MoveOperationCL">
                </instance>
        </operation>
</operations>
```

which implements the IOperation interface. The operation name should be same as the return value of IOperation.getOperationName ().

Step 4: register the operation.



Example: the algorithm LCOM [] is described in Eample-8. The source code is following:

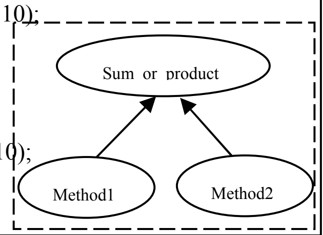
```
/*
 * Created on 2004-2-26
 *
 * Thesis Project Name: IFSO (Integrated Framework for SOftware analysis and refactoring
 * School: Worcester Polytechnique Institution
 * Advice: Professor George T. Heineman
 *
 */
package operation;

import java.util.Properties;

import edu.wpi.cs.ifso.interfaces.*;
import edu.wpi.cs.ifso.operation.exceptions.OperationException;
import edu.wpi.cs.ifso.operation.interfaces.IOperation;

/**Remove selected design object.
 * But current this operation only support remove the design object which no direct link
associated with
 * the provided interfaces of it.
 * Work Level: method level, class level
 *
 * @author Yilei Zheng
 *
 *
 */
public class RemoveDesignObject implements IOperation {

    private String name = new String("RemoveDesignObject");
    private Properties properties = new Properties();

    public String getOperationName() {
        return name;
    }
```

```
public String execute(
     String objectName,
     String objectType,
     ITransaction itran,
     ILevel il,
     IAccessRepresentation iar,
     IUpdateRepresentation iur)
     throws OperationException {

     if (objectName == null || objectType == null){
                  throw new OperationException("RemoveDesignObject.execute: input null
                  parameters.");
     }

            IDesignObject    ido    =    (IDesignObject)iar.getDesignObject(objectName,
            objectType);
     if (ido == null){
                  throw new OperationException("RemoveDesignObject.execute: cannot find
                  design object: " + objectName + " in the representation.");
     }

     if (itran == null){
                  throw new OperationException("RemoveDesignObject.execute: input null
                  parameters.");
     }

     try{
          itran.beginTransaction();

                  if        (il.getLevelName().equals("method        level")        ||
                  il.getLevelName().equals("class level")){
                      //add an action to the transaction
               itran.action(iur, ido, "remove");
               String message = "type[operation]";
                            //format                                      name{@algorithm
                            //name};object{@objectname};type{@objecttype};parameter{@na/
                            /me;@value};
                            //send a request to upper level to notify the modification
                            //happened in current level.
                            String     data     =     "name{RemoveNotify};object{"     +
                            ido.getEnvironment().getParent().toString()
                            +                        "};type{"                        +
                            ido.getEnvironment().getParent().getObjectType().toString()
                            +               "};parameter{designunitname;"            +
                            ido.getEnvironment().toString()
                            +               "};parameter{designunittype;"            +
                            ido.getEnvironment().getObjectType().toString()
               + "}";
               message = message + ";data[" + data + "]";
                            String  result  =  il.getMessageAgent().send(message,  "up",
                            itran);

               if (result.equals("success")){
                    if (itran.commitTransaction() == true){
                         result = "success";
                    }else{
                         result = "fail";
                    }
               }
               properties.setProperty("result", result);
               return properties.getProperty("result");
          }

          properties.setProperty("result", "not support");
          return properties.getProperty("result");
     }catch(Exception e){
          throw new OperationException(e.getMessage());
     }

}

public void setParameter(String name, String value)
     throws OperationException {
        properties.getProperty(name, value);
    }

    public String getResult(String name) throws OperationException {
        return properties.getProperty(name);
    }

    public String getResult() throws OperationException {
        return properties.getProperty("result");
    }
}
```

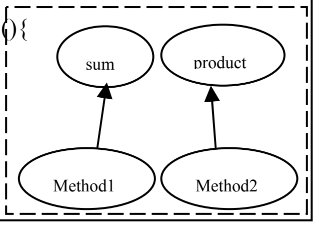
Appendix C: example code

In the (b), the method sum_or_product is split to two methods: sum and product. It is obvious that the cohesion of the right side class is lower than the left side. So the cohesion increase in method level can cause cohesion decrease in class level.

```
Class calculator {
    int sum;
    int product;

    public void sum_or_product(int flag, int m, int n){
        if flag = 1 then{
            int i = 1;
            sum = 0;
            while ( i <= m ){
                sum = sum + i;
                i = i + 1;
            }
        }
        else{
            int j = 1;
            product = 1;
            while (j <= n){
                product = product * j;
                j = j + 1;
            }
        }
    }

    public void method1(){
        …
        sum_or_product(1, 10);
        …
    }
    public void method2(){
        …
       sum_or_product(2, 10);
        …
    }}
```



```
Class calculator {
    int sum;
    int product;

    public void sum (int m){
        int i = 1;
        sum = 0;
        while ( i <= m ){
            sum = sum + i;
            i = i + 1;
        }
    }
    public void product(int n)
        int j = 1;
        product = 0;
        while (j <= n){
            product = product * j;
            j = j + 1;
        }
    }

    public void method1(){
        …
        sum ( 10);
        …
    }
    public void method2(){
        …
        product(10);
        …
    }}
```



(a) (b)

Assume that there is no relationship between calculator.method1 and calculator.method2 except they all invocate sum_or_product function.

```
Class calculator_sum {
    int sum;

    public void sum (int m){
        int i = 1;
        sum = 0;
        while ( i <= m ){
            sum = sum + i;
            i = i + 1;
            }
    }
    public void method1(){
        …
        sum ( 10);
        …
    }}
```

```
Class calculator_product {
    int product;

    public void product (int m){
        int i = 1;
        product = 1;
        while ( i <= m ){
            product = product * i;
            i = i + 1;
            }
    }
    public void method2(){
        …
        product ( 10);
        …
    }}
```

(c)

We find that the class can be split into two high cohesion classes in (b) situation as show in (c). It indicates that low design quality of high level may be caused by design problem of lower level. There is no way to improve cohesion of the class without inspecting the method.

References

[AK99] E. B. Allen and T. M. Khoshgoftaar. “Measuring Coupling and Cohesion: An Information-Theory Approach”. Sixth IEEE International Symposium on Software Metrics, Boca Raton, Florida, November 1999.

[ARJ86] A. V. Aho, R. Sethi and J. D. Ullman. “Compilers Principles, Techniques and Tools”. Bell Telephone Laboratories, 1986.

[Bad00] G. J. Badros. “JavaML”. http://www.cs.washington.edu/homes/gjb/JavaML/, 2000.

[Bal96] N V Balasubramanian. “Object-oriented Metrics”. 3rd Asia-Pacific Software Engineering Conference (APSEC ’96), Seoul, South Korea, December 1996.

[BBC+00] H. Bar, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. –M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, J. Weisbrod. “The FAMOOS Object-Oriented Rengineering Handbook”. http://www.rspa.com/reflib/Reengineering.html, October 1999

[BDG+94]E.Buss, R. De Mori, M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Muller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster and K. Wong. “Investigating Reverse Engineering Technologies: The CAS Program Understanding Project”. IBM Systems Journal, 33(3): 477-500, August 1994.

[BDW98]L. C. Briand, J. W. Daly and J. Wüst. “A Unified Framework for Cohesion Measurement in Object-Oriented Systems”. Empirical Software Engineering, 3(1): 65-117, 1998.

[BJ78]J. C. Browne and D. B. Johnson. “FAST: A second generation program analysis system”. In Proceedings of the 3rd international conference on Software Engineering, pages142-148, Atlanta, Georgia, United States, May 1978.

[BK98] J. M. Bieman and B. –K. Kang. “Measuring Design-level Cohesion”. IEEE Transaction on Software Engineering, 24(2) P111-124, February 1998.

[BM84] M. L. Brodie, J. Mylopoulos and J.W. Schmidt. “On Conceptual Modeling Topic in Information Systems”. Springer-Verlag, 1984.

[BO94] J. M. Bieman and L. M. Ott. “Measuring Functional Cohesion”. IEEE Transactions on Software Engineering, 20(8):644-657, August 1994.

[Cal03] P. W. Calnan. “EXTRACT: Extensible Transformation and Compiler Technology”. WPI master thesis, 2003.

[CC90] E. Chikofsky and J. Cross. “Reverse engineering and design recovery: A taxonomy”. IEEE Software, 7(1): 13-17, January 1990.

[CK96] H. S. Chae and Y. R. Kwon “Assessing and Restructuring of Classes Based on Cohesion”. 3rd Asia-Pacific Software Engineering Conference (APSEC’96), Seoul, South Korea, page76, December 1996.

[CK98] H. S. Chae and Y. R. Kwon. “A Cohesion Measure for Classes in Object-Oriented Systems”. 5th International Symposium on Software Metrics, Bethesda, Maryland, page158, March 1998.

[CMI] CMI http://www.cs.wpi.edu/~heineman/classes/cs509/

[CNR90] Y. Chen, M. Y. Nishimoto and C. V. Ramamoorthy. “The C Information Abstraction System”. IEEE Transactions on Software Engineering, 16(3): 325-334, 1990.

[CZX02] Z. Chen, Y. Zhou and B. Xu. “A Novel Approach to Measuring Class Cohesion Based on Dependence Analysis”. International Conference on Software Maintenance (ICSM’02), Montreal, Quebec, Canada, October 2002.

[ECL] Eclipse http://www.eclipse.org/

[FBB+99]M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. “Refactoring: Improving the Design of Existing Code”. Addison-Wesley, 1999.

[FH00]H. Fahmy and R. C. Holt. “Software Architecture Transformations”. In Proceedings of the International Conference on Software Maintenance (ICSM’ 00), San Jose, California, page88, October 2000.

[GJV95]E. Gamma, R. Helm, R. Johnson and J. Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison Wesley Longman, Inc., 1995.

[HC01] George T. Heineman and William T. Councill. “Component-based Software Engineering”. Addison-Wesley, 2001.

[INT] IntelliJ IDEA http://www.jetbrains.com/idea/

[Lak93] A. Lakhotia. “Rule-based Approach to Computing Module Cohesion”. In Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, United States, pages34-44, 1993.

[LC94] A. Lake and C. Cook. “Use Of Factor Analysis to Develop OOP Software Complexity Metrics”. In proceedings 6th Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon, 1994.

[LD99] A. Lakhotia and J. –C. Deprez. “Restructuring Functions with Low Cohesion”. Sixth Working Conference on Reverse Engineering, Atlanta, Georgia, page36, October 1999

[LK94]M. Lorenz and J. Kidd. “Object-Oriented Software Metrics”. Amazon, 1994.

[Mis00] V. B. Misic. “Coherence Equals Cohesion – Or Does It?”. Seventh Asia-Pacific Software Engineering Conference (APSEC’00), Singapore, December 2000.

[MJS+00] H. A. Muller, J. H. Jahnke, D. B. Smith, M. –A. Storey, S. R. Tilley and K. Wong. “Reverse Engineering: A Roadmap”. In proceedings of the conference on The Future of Software Engineering, Limerick, Ireland, Pages 47-60, 2000.

[MK88] H. A. Muller and K. Klashinsky. “Rigi: A System for Programming-in-the-large”. In Proceedings of the 10th international conference on Software Engineering, Singapore, pages80-86, 1988.

[MW89] T. J. McCabe and C. W. Butler. “Design Complexity Measurement and Testing”. Communications of the ACM, 32(12): 1415-1425, December 1989.

[OB95] L. Ott and J. M. Bieman. “Developing Measures of Class Cohesion for Object-Oriented Software”. In Proceedings of the Annual Oregon Workshop on Software Metrics (AOWSM’95), Oregon, June 1995.

[RAT] Rational Rose http://www-306.ibm.com/software/rational/

[RNK++95] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr. and J. E. Robbins. “A component- and message-based architectural style for GUI software”. Proceeding of the 17the international conference on Software engineering, Seattle, Washington, United States, pages 295 – 304, 1995.

[SC91] S.R Chidamber and C.F. Kemerer. “Towards a metric suite for object-oriented design”. Proceedings : OOPSLA ’91, Phoenix, AZ, pp. 197-211, July 1991.

[Ste85]J. L. Steffen. “Interactive examiniation of a C program with Cscope”. Winter USENIX Technical Conference, pages170-175, 1985.

[ZX02] Y. Zhou and B. Xu. “ICBMC: An Improved Cohesion Measure for Classes”. International

Conference on Software Maintenance (ICSM'02), Montreal, Quebec, Canada, October 2002.