Slicing-Based Coupling Measurements

Wenjun Meng

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2003

© Wenjun Meng, 2003

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisisitons et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-83916-8
Our file Notre référence
ISBN: 0-612-83916-8

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

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

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

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

Canada

Abstract

Slicing-Based Coupling Measurements

Wenjun Meng

The market forces affecting today's software development have placed a greater emphasis on software quality. A variety of design measurements have been proposed to control the development of software systems and to evaluate the final software products after delivery. In this thesis, we propose a slicing-based coupling measurement framework to assess the complexity of different granularities in a slice. The proposed framework combines well-known and proven coupling measurements, namely CBO, RFC, and MPC, with program slicing-based source code analysis techniques. These slicing-based coupling measurements are further extended for different abstraction levels and application domains. The proposed measurements are implemented as part of the CONCEPT project to provide an aid for programmers comprehending and assessing source code and to give some objective heuristics for software engineers in selecting and prioritizing maintenance tasks.

Acknowledgements

I would like expressing my sincere gratitude to my supervisor Dr. Juergen Rilling for his support, guidance, patience, and valuable insight, which have made the completion of my thesis possible.

I am also grateful to Yonggang Zhang and other colleagues in CONCEPT project for sharing measurement issues and associated data.

Finally, I would like to thank my parents, husband and little daughter. Without their love and support, I could not accomplish this thesis.

Contents

LIST OF FIGURES VII
LIST OF TABLES VIII

1 INTRODUCTION 1
1.1 SOFTWARE MEASUREMENT 2
1.2 PROGRAM SLICING 3
1.3 GOALS 3
1.4 THESIS OUTLINE 4

2 BACKGROUND 5
2.1 PROGRAM SLICING 6
2.1.1 Static slicing and dynamic slicing 7
2.1.2 Forward and backward slicing 9
2.1.3 Executable and non-executable slicing 11
2.1.4 Union and intersection of program slices 12
2.2 SOFTWARE MEASUREMENT 14
2.2.1 Low Coupling and high cohesion 14
2.2.2 Coupling in OO program 15
2.2.3 Measuring OO program coupling 19
2.2.4 Measurement tools 24
2.3 CONCEPT FRAMEWORK 29

3 SLICING BASED COUPLING MEASUREMENT 31
3.1 SLICING BASED MEASUREMENT LITERATURE 31
3.2 MOTIVATION FOR THE MEASUREMENT FRAMEWORK 32
3.2.1 Identify and focus 33
3.2.2 Direct versus indirect coupling measurement 33
3.2.3 Measuring export coupling 35
3.3 GENERIC SLICING BASED COUPLING MEASUREMENT 36
3.3.1 Slicing based coupling measurement 37
3.3.2 Views on the slice based measurement 40
3.3.3 Measure export coupling 42
3.4 VALIDATION OF THE MEASUREMENTS 44

4 ABSTRACTION DRIVEN SLICING BASED MEASUREMENTS 46
4.1 SLICING HIERARCHY 46

4.2 ABSTRACTION ORIENTED SLICING BASE COUPLING MEASUREMENT 48

5 SLICING BASED COUPLING MEASUREMENT FOR CHANGE IMPACT ANALYSIS 50

6 IMPLEMENTATION 52

6.1 GOAL 52

6.2 BASIC WORKFLOW 53

6.2.1 Parsing 54

6.2.2 PostGreSQL Database 54

6.2.3 Coupling analysis and measurement derivation 56

6.3 DESIGN DETAILS 57

6.3.1 Reference identification 57

6.3.2 Class Diagram 64

6.3.3 Deriving traditional CBO, RFC and MPC 64

6.3.4 Deriving slicing based SCBO, SRFC, and SMPC 74

6.3.5 Comparison and analysis 76

6.3.6 Future work 79

7 CONCLUSION 81

BIBLIOGRAPHY 82

APPENDIX A ELEVATOR PROGRAM 86

APPENDIX B MEASUREMENT RESULT 94

List of Figures

Figure 1 Theoretical basis for the development of Object-Oriented measurements[Ema99] 14

Figure 2 Method overloading in ClassDef 18

Figure 3 CONCEPT framework 30

Figure 4 Direct and indirect coupling in a message chain 34

Figure 5 Example for indirect coupling 35

Figure 6 Impact analysis through program slicing 50

Figure 7 Workflow for measurement derivation 53

Figure 8 Static source code analysis [Zha03] 54

Figure 9 An AST example [Zha03] 55

Figure 10 Class Diagram for measurement derivation 65

List of Tables

Table 1 OO program Hierarchy 16

Table 2 Properties of the proposed measurements 39

Table 3 Slicing hierarchy 46

Table 4 Declaration reference 58

Table 5 Creation reference 60

Table 6 Method invocation reference 61

Table 7 Field accessing reference 62

Table 8 Other references 63

Table 9 CBO from elevator program 66

Table 10 MPC from elevator program 69

Table 11 RFC from elevator program 72

Table 12 SCBO for S<current_floor, Elevator L59> 75

Table 13 SRFC for S<current_floor, Elevator L59> 75

Table 14 SMPC for S<current_floor, Elevator L59> 76

Table 15 CBO, RFC and MPC in our system 77

Table 16 CBO, RFC and MPC in Sun ONE Measurement Tool 78

Table 17 SCBO, SRFC and SMPC for S<current_floor, Elevator L59> 78

1 INTRODUCTION

Any useful computer program is likely to require changes during its life in order to correct errors, to support new peripherals, and to adapt to changes in the domain served by the program [Bas95, Bas96, Bri93]. In fact, maintenance of existing source code is responsible for a substantial portion of a computer program's lifetime cost. Poor design, unstructured programming methods, and crisis-driven maintenance may lead to the poor code quality, which in turn affects program evolution costs. As software has become more pervasive and its life expectancy has increased, it has been subject to greater pressures to integrate and interact with other software and to evolve and adapt to uses in all manner of new and unanticipated contexts, both technological and sociological. Software systems have to be flexible in order to cope with evolving requirements [Bie95, Bri93]. Although good software engineering practice encourages programmers to plan for future modifications, not every future design change can be predicted. User requests for changes are often a consequence of using the system after delivery. With respect to quality control, well-designed modules should exhibit a high degree of cohesion and a low degree of coupling, such that each module addresses a specific, well-defined sub-function from a system structure view [Abr95, Bri93, Ede94], which might be one of the important ways to ensure the quality of the software during its evolution. Nowadays, the market forces affecting today's software development have placed a larger emphasis on software quality, which in turn has lead to an increasingly large body of work being performed in the area of software measurements, particularly for understanding and evaluating the quality of the existing software.

1.1 Software measurement

From the earliest days of the software engineering discipline there has been wide agreement on the need to measure software processes and products as a pre-condition for establishing control of the software quality during software evolution activities [Lit99]. Software measurement is "a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [IEE93]. One informal definition is that software measurement is a process of quantifying the attributes of software in order to characterize it according to clearly defined rules [Fen97, Orm03]. Some attributes such as size of the program, coupling of the program can be directly quantified, while the attributes such as maintainability, testability and fault-proneness cannot be directly accessed. To date, no measurement suite has claimed that it is complete for software quality assessment. Therefore, the emphasis of this thesis is to derive traditional CBO (coupling between object classes), RFC (response for a class) and MPC (message passing coupling) in our CONCEPT reverse engineering environment, and to further combine them with slicing techniques to create slicing based measurements to see how these measurements could benefit the software design and quality assessment as well as guide programmers' activities during software evolution. To our knowledge, little research work has been done so far in this field, combining good traditional measurements and various slicing techniques. Our current research is at an early stage especially with respect to the validation of usability and applicability of these measurements.

1.2 Program slicing

One of the key challenges in the process of understanding software is to have some aid for rapidly building a conceptual model of the system. It often suffices to obtain only a partial understanding that is sufficient to build a conceptual model one can trust when performing a particular assessment/change, or to develop a model for locating places where such an assessment/change should be applied. It is generally accepted that some effective automatic tools could speed up the creation of conceptual models and that these tools could help improve the quality of the programs being maintained [Mul00]. Slicing is one proposed automated technique for that purpose [Wei81, Wei82], which reduces the size of programs or decomposes a larger program into smaller components so that the total cognitive loads can be reduced. Program slicing isolates the other large code components that are not related to the particular computation. Variant slicing techniques such as forward and backward slicing, static and dynamic slicing, etc are proposed for different application purposes.

1.3 Goals

Since program slicing reduce the cognitive burden during software maintenance by identifying a particular computation associated part of the original source code and coupling measurements provide an aid in analyzing the inherent couplings in the source code, a natural combination of program slicing and measurement comes out to identify the interested part of the original program and further quantify the coupling complexity of the part from the original program. In this thesis, we combine useful measurements

[Li93, Chi94] with different slicing techniques to focus these measurements on particular program aspects rather than the whole system. This focused context is created through program slicing [Wei81], a program reduction technique that guarantees the same behavior with respect to the slicing criterion as the original program. Additionally, we propose a slicing hierarchy to further refine these measurements. These new slicing based coupling measures are implemented in our CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) prototype [Ril01] that was developed as a light-weight framework to guide programmers during the task of understanding large Object-Oriented programs and their executions. Therefore, the "hotspots" within the current design and the overall dependencies among the different program granularities will be automated to guide programmers in selecting and prioritizing maintenance tasks.

1.4 Thesis outline

The remainder of this thesis is organized as follows: Chapter 2 discusses some background concepts related to software measurement and program slicing. Chapter 3 introduces our program slicing based coupling measurements, their definitions and justifications. Chapter 4 introduces a slicing hierarchy on which the measurements are applied. Chapter 5 discusses one possible application of the slicing based coupling measurements: change impact analysis. Chapter 6 discusses the implementation issues and experimental results. Finally, Chapter 7 presents the conclusion and outlines future work.

2 BACKGROUND

Software maintenance may degrade the structure of software, ultimately making maintenance more costly. The longer software systems are in use, the more likely it is that these systems have to be maintained. They have to be changed to reflect new features (perfective maintenance), fix identified defects (corrective maintenance), and adjusted for a changing environment (adaptive maintenance). Enhancing the maintainability requires software developers and designers to enhance the quality, and therefore the design of existing systems. The quality assessment of software systems can be conducted by interviewing the architects and designers of a product and reviewing the analysis and requirement documents. However, these approaches might only be feasible if the software architects are still available and the existing documentation reflects the real source code. The approach presented in this paper is to conduct a software design assessment based on source code analysis [Har97, Ott89, and Ott93] and software measurement computation. These design measurements can be applied to identify "hot spots" which are locations in the design that represent good candidates for potential redesign and enhancement, as well as to help to prioritize comprehension and maintenance activities.

In what follows, the background knowledge about program slicing and software measurement associated to this research is discussed.

2.1 Program slicing

Basically, a program slice is those parts of a program that directly or indirectly affect the values of the variable computed at some point of interest, referred to as a slicing criterion. Program slicing is a low level source code based program transformation. Essentially, the automation of program slicing is based on some particular slicing criterion and program dependence analysis. A slicing criterion typically has two important parts<i, v>, where i is some interest point as a statement number in the source code and v is the variable of interest. Mark Weiser[Wei81] first proposed program slicing based on the observation that program slicing relates closely to the mental process used by humans during debugging software systems. Program slicing removes those parts of the program that are not relevant to a certain computation in the form of slicing criterion, which speed up the process of identifying relevant source and of building mental models of the program. According to the Mark Weiser's slicing definition, two key properties of a slice are intuitively desirable. First, the slice must be derived from the original program by deleting statements. Second, the behavior of the slice must correspond to the behavior of the original program as observed through the window of the slicing criterion. However, many extensions of the original program slicing notion are not strictly abiding by these two constraints due to the requests from different application domains.

Program slicing have many applications in debugging [Wei82, Tip95], software comprehension [Har01], testing [Hart95, Gup92, Gop91], and reuse etc. Various notions of program slices and slicing algorithms have been proposed for different application purposes. The diversity of slicing techniques results from the understanding of the

concept of "slicing", or the transformation rules of the program slicing. For example, amorphous program slicing of Harman and Danicic[Har95] is computed based on any program transformation which simplifies the program and which preserves the effect of the program with respect to the slicing criterion. Thus, the obtained slice might not be a subset of the original source code since the transformation is only consistent to the semantics of the original source code.

2.1.1 Static slicing and dynamic slicing

The notion of program slicing was originally proposed by Mark Weiser in 1981 [Wei81], which is also referred to as "static backward" slicing. Based on the original definition of a static program slice, a slice S consists of these parts of a program P that potentially could affect the value of a variable v or a set of variables vs at a point of interest i. This slice is obtained from P by deleting zero or more statements. Whenever P halts on an input I with state trajectory T, then slice S halts on input I with state trajectory T', and $PROJ(T)=PROJ(T')$, where $PROJ$ is the projection function associated with criterion C[Wei82]. Weiser's static slicing is derived by computing consecutive sets of relevant statements according to the data flow and control flow dependence within the source code P.

Korel and Laski [Kor88] first introduced the notion of dynamic slicing that can be seen as a refinement of the static approach by utilizing additional information derived from program executions on some specific program input. A dynamic slicing criterion of program P executed on input x is a tuple $C=<x, y^q>$ where y^q is a variable y at execution

position q. An executable dynamic slice of program P on slicing criterion C is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and when executed on program input x produces an execution trace $T'x$ for which there exists the corresponding execution position q' such that the value of y^q in T_x equals the value of $y^{q'}$ in T'_x. A dynamic slice P' preserves the value of y for a given program input x. In dynamic program slicing, only the dependences that occur in a specific execution of the program are taken into account. Ideally, a dynamic slice is an executable part of a program P whose behavior is identical, for the same program input with fixed values, to that of the original program with respect to a variable v at some execution position. Only the subsets of the source codes that occur in a specific execution of the program are taken into account.

Mark Weiser's original "static backward" slicing is based on the assumption that any statement, which is deleted can have no effect upon the slicing criterion when the program is executed in any initial state, without considering the execution related input details. That is, one major difference between dynamic slicing and static slicing. Dynamic slicing relies on a particular program execution, which considers only a certain set of inputs whereas a static slice preserves the program behavior for all set of inputs. Hence, static slices will be relatively larger and more conservative than dynamic slices. However, the computation of dynamic slices is in general more expensive and precise since it monitors the execution trace of a particular input. Dynamic slicing is good for testing and debugging, while static slicing is good for identifying the part of the source code for reuse.

Some existing coupling measurements can be applied on dynamic slice or static slice. In order to study the program behavior, dynamic slicing based coupling measurements are provided to quantify the some behavioral aspects of the program; for a more generally analysis of the program structure or other static aspects, static slicing-based coupling measurements can be used. The details of slicing based coupling measurements will be discussed in Chapter 3.

2.1.2 Forward and backward slicing

The sense of "backward" and "forward" is due to the process of constructing the slice. "Backward" or "forward" slice relies on whether the dependence flow of the program is in reverse or forward way to ascertain the statements and predicates which can affect the slicing criterion. Mark Weiser's original slicing falls as mentioned earlier into the category of "static backward" slicing.

In general, a backward slice is a set that consists of all statements and control predicates, which have some affect on the slicing criterion, whereas a forward slice [Hor90, Har01] is a set that consists of all statements and control predicates, which are affected by the slicing criterion. This definition of forward slice is widely used in slicing literature [Hor90, Har01, and Wan96]. Backward slicing and forward slicing are first distinguished in [Hor90] while the dependence graph of the program is analyzed.

In addition to the popular definition of forward slicing, there is yet another kind of forward slicing [Kor88] proposed for performing dynamic slicing. In dynamic slicing, two major algorithms have been identified such as backward algorithm and forward algorithm. Backward algorithms trace backwards [Hit96] a recorded execution trace to derive data and control dependencies that are then used for the computation of the dynamic slice. In contrast, forward algorithms [Luc01] aim to overcome a major weakness of the backward approach - the necessity of recording the execution trace during program execution. Forward slicing starts the computation at the beginning of the program and it analyze all the related variables in the computation, so it keeps the state of each variable in the program instead of preserving the execution trace. However, the slices with a same criterion derived by the above two dynamic slicing algorithms respectively should be same.

In this thesis, the popular forward slicing is taken into account since it produces a different slice with respect to the backward slicing. However, the forward slicing might not be executable so it is difficult to ensure the correctness of the slice. One conservative way to make forward slice executable is to backward slice all associated variables of the forward slice. Thus, the final slice is obtained by one forward slicing and several backward slicing, which ensures the final slice can be executable, but the resulting slice might be too conservative and too large. Therefore, the tradeoff to get executable or non-executable program slicing is based on the application purpose. For example, for impact analysis, it may be not necessary to derive executable forward slicing.

Traditional coupling measurements can be applied on the forward and backward slice. Backward slicing based coupling measurements are applicable for understanding the slice associated coupling during testing and debugging tasks. The goal is to completely understand how the program reaches a particular point of interest. Forward slicing based coupling measurements on the other hand have advantage of being able to measure issues related to change impact analysis, indicating the change complexity and effort involved to perform a modification.

2.1.3 Executable and non-executable slicing

Considering the notion of a slice in general, an important property of a slice is, whether the slice is executable or non-executable. "Executable" means in this context that the obtained slice itself is again an executable program. It preserves the syntactical property of the original program, and produces same output with the same input as the original program. There are many obvious benefits of the executable slices. Above all, an executable slice is a precise miniature of the original executable program with respect to the particular slicing criteria. It facilitates the understanding process of software engineers. In addition, an executable slice provides an effective way to validate whether the slicing algorithm is correct or not [Ril93]. No matter if the resulting executable slice was computed statically or dynamically, the slice should reflect a behavior that is identical to the original program execution.

However, many applications may not strictly require the slice to be executable. For example, the forward slicing as we discussed previously, may not be executable since a

forward slice [Hor90] is a set that consists of all statements and control predicates which are affected by the slicing criterion. For the forward slice to be executable, each statement in the forward slice needs to be further backward sliced. Thus, the final slice might be very large and less precise. A non-executable slice only retains the semantic feature of the original software. Compared with executable slices, non-executable slices might be significantly smaller and simpler and good for programmers to locate some faulty points during debugging [Ril93], or for general program understanding. But, its side effects [Ril93] may create some problems for software engineers to ensure and verify the correctness of the computed slice.

2.1.4 Union and intersection of program slices

One informal way to look at a slice might be that a slice is a set of a particular computation associated statement numbers. Thus, most set operations are still applicable to slices. In order to derive different abstraction slicing-based coupling measurements, union and intersection operations are used to identify some hot spots for analysis.

Definition 1: Union

$C = A \bigcup B = \{x \mid x \text{ is a statement in either A or B}\}$ [Wan96]

C denotes the union of the statement sets in slice A and slice B

The union of two slices is the set of the statements in either A or B. Lucia et al. [Luc03] discussed that in static slicing, the union of two dependence-preserving slices constructed for different criteria can only argment the slice of either. That is, the union of the two slices is still a valid slice except that it might be an unnecessarily large slice of each of

the contributing slicing criteria. Horwitz et al. [Ede94] pointed out that unifying two non-interfering versions of the program slices would safely result in another valid program slice, where two non-interfering versions of the program refer to those slices achieved by graph-based program dependence. Suppose A and B are two program slices of a program P. The statement union C might not be a valid slice since the operations on slice A and slice B might break up the slice properties such as "executable" and "behavior identical". However, in the case of the slicing hierarchy or slicing based coupling measurement derivations(discuss in chapter 3), the union operation is meaningful since it reduces the redundant slicing processes and conservatively gives all the possible statement sets for further analysis.

Definition 2: Intersection

$C = A \cap B = \{x \mid x \text{ is a statement in both A and B}\}$ [Wan96]

C denotes the intersection of the statement sets in slice A and slice B

The intersection of two slices is the set of the common statement in both two slices. The statement set resulting from this intersection operation is not an executable slice in most cases. However, it is useful for obtaining the "crude measure of cohesion'' [Har95] by identifying common statements in the two slices, which has originated from the data token based cohesion measurements in [Ott93]. It also can be used to identify some hot spots for further inspection regarding its potential for reuse.

2.2 Software Measurement

Fenton [Fen97] described software engineering as a "collection of techniques that apply an engineering approach to the construction and support of software products." Software measurement is one of the engineering approaches introduced to support the most critical issues in software development and provide support for planning, predicting, monitoring, controlling, and evaluating the quality of software products [Fen97]. It has been shown that software measurement can provide software engineers and maintainers with guidance in analyzing the quality of their code/design such as maintainability [Bas96, Bri96, Hen96, and Li93]. Software measurements "are used to make numerical measurements of particular aspects of a target software system. Measurements can be applied to support the identification of complex parts of the software that need restructuring. They also reveal tightly coupled parts of the software. Such parts are inflexible for modifications and reuse. They also may represent potential subsystems. Identification of subsystems, in turn, supports program comprehension" [Sys99] and evolution.

2.2.1 Low Coupling and high cohesion

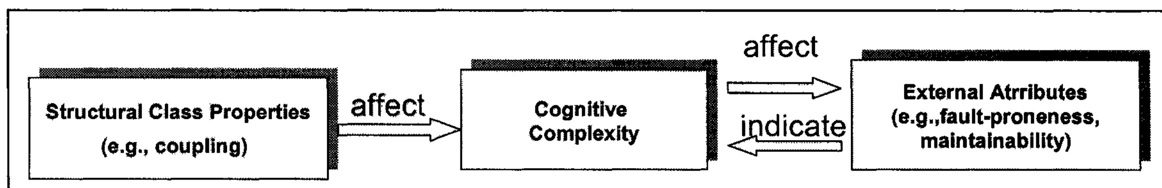


Figure 1 Theoretical basis for the development of Object-Oriented measurements [Ema99]

It is in general accepted that low coupling and high cohesion in a software design lead to better quality products, e.g., in terms of reliability and maintainability [Sys99].

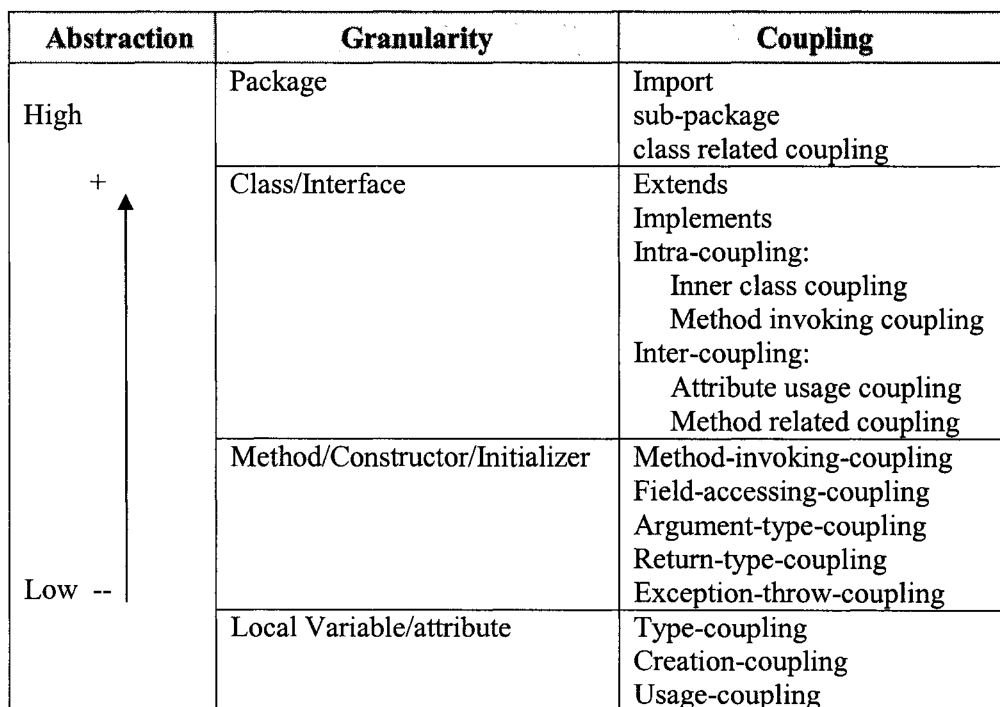
Especially, coupling is recognized for its contribution to software design and its influence on system integration and maintenance cost. Coupling, as defined by Stevens et al. [Ste74], is "the measure of the strength of association established by a connection of one module to another." Constantine and Yourdon [You79] defined coupling based on the relationship of subroutines as measurements for procedural systems. In the context of OO program, minimizes connections between classes or methods also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous "ripple" effects, where changes in one part causes errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc [Ste74]. Strong coupling complicates a system, since a module is harder to understand, change, or correct by itself if it is highly interrelated to other modules. Hence, to reduce the complexity of a system, it is necessary to keep the weakest possible coupling between classes. Page-Jones [Pag80] gave three principle reasons why low coupling between modules is desirable: (1) it reduces the chance that a fault in one module will cause a failure in other modules, (2) it reduces the chance that changes in one module cause problems in other modules, and (3) it reduces programmer time to understand the details of other modules. Figure 1 summarizes the relation between coupling measurements and software quality aspects. In this research, the start point is identifying and deriving different types of coupling based on source code analysis in Java program.

2.2.2 Coupling in OO program

2.2.2.1 OO Hierarchy

In this section, the details of OO concept are not discussed since most software engineers are familiar with the popular OO concept. However, coupling relations in the Java source code are extensively studied in order to compute the coupling measurements based on source code analysis. So some OO concepts, which are closely related to our slicing based coupling measurements discussed later, are briefly presented.

Table 1 OO program Hierarchy



Abstraction	Granularity	Coupling
High	Package	Import sub-package class related coupling
+	Class/Interface	Extends Implements Intra-coupling: Inner class coupling Method invoking coupling Inter-coupling: Attribute usage coupling Method related coupling
Low --	Method/Constructor/Initializer	Method-invoking-coupling Field-accessing-coupling Argument-type-coupling Return-type-coupling Exception-throw-coupling
	Local Variable/attribute	Type-coupling Creation-coupling Usage-coupling

The hierarchy of the Java program and main coupling relations are summarized in the Table 1. The key granularities in Java are: variable, attribute, method / constructor / initialize, class / interface, and package. The major coupling for each granularity is given in Table 1.

In the CONCEPT project, all these dependencies have been identified based on the source code analysis for Java programs. The identified references are classified into three

major categories as declaration coupling, creation coupling, and usage coupling. The details of these identified coupling/references will be discussed in section 6.3.1.

2.2.2.2 OO features affecting coupling

There are many factors in OO program which may complicate the coupling measures. Abstraction, polymorphism, inheritance, and encapsulation are some of the most importance features in OO design, which are advocated to support reuse, improve maintainability, and reduce coupling, etc. However, they also introduce challenges to quantifying the coupling of OO program.

Encapsulation is the inclusion within a program object of all the resources needed by the object to function - basically, the methods and the data. Kung et al [Kun95] pinpointed that "The understanding problem is introduced by the encapsulation and information-hiding features. These features result in the delocalized plan, in which several member functions from possibly several object classes are invoked to achieve an intended functionality. Often, a member function of a class in turn invokes other member functions, resulting in the so-called invocation chain of member functions."

Polymorphism means "having multiple forms", in other words, a variable, a function or an object can have more than one form (implementation). For example, methods in Java can easily be overloaded such that the same method name can be used for several different implementations. The only requirement for method overloading is that each version of the method takes a different set of parameters as arguments (or "different

signature") such as the example below in figure 2 (taken from our current project CONCEPT).

```
...
public class Environment implements Constants
{
        ...
        public ClassDef   getClassDef(String name) throws Exception
        {
                Type type = Type.tType(name);
                return getClassDef(type);
        }
        public ClassDef   getClassDef(Type t) throws Exception
        {
                if(!t.isClass())
                {
                        throw new Exception("Get ClassDef for a none class type!");
                }
                String className = t.getTypeString();
                ClassDef clazz = (ClassDef)classhash.get(className);
                .....
        ...
```

Figure 2 Method overloading in ClassDef

Inheritance is the capability of a class to use the properties of classes above it (super classes). A subclass Y inherits all of the attributes and operations associated with its super class, x. This means that all data structures and algorithms originally designed and implemented for x are immediately available for Y. New attributes and methods can be added to the sub class. A sub class overrides the super class' method only when the derived class needs to be modified to support new features.

Abstraction is a mechanism of removing characteristics from something in order to reduce it to a set of essential characteristics. Abstraction introduces different level of granularities such as package level, class level, method level, etc.

2.2.3 Measuring OO program coupling

A large number of coupling measurements for OO program have been introduced and discussed. The most widely used and criticized measurement suite is from Chidamber and Kemerer [Chi91, Chi94], where they identified 6 Object-Oriented measurements: Weight Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), Lack of Cohesion in Methods (LCOM), which are proposed with the intention to be independent of OO language implementation details. Li and Henry[Li93] assessed Chidamber and Kemerer[Chi91] measurement suite, and introduced message passing coupling(MPC), data abstraction coupling(DAC), and the number of local methods(NOM). Briand et al. describe coupling as the degree of interdependence among the components of a software system and they presented a unified framework for coupling measurement in OO system [Bri97, Bri98]. In our research, coupling between objects (CBO), response for a class (RFC) and message passing coupling (MPC) are selected as the candidate measurements since they are the most widely examined measurements both in theory and in experiment.

- Coupling between object classes (CBO)[Chi94]

Coupling between object classes (CBO) is first proposed in [Chi91], where Coupling between objects for a class is a count of the number of non-inheritance related couples with other classes. CBO relates to the notion that an object is coupled to another object if two objects act upon each other, i.e., methods of one use methods or instance variables of another. In order to improve modularity and promote encapsulation, inter-object couples should be kept to a minimum. However, in 1994, Chidamber and Kemerer refined the

definition of CBO by including the coupling associated to inheritance. The following quotation discussing CBO in details is from [Chi94].

Definition

CBO for a class is a count of the number of other classes to which it is coupled.

Theoretical basis

CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. Since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

Viewpoints

- Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.
- In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.
- A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be. [Chi94]

Discussion

Hitz and Montazeri [Hit 95] studied the Chidamber and Kemerer measurements [Chi91] and presented three critics. The first critic is that it does not differentiate between the strength of couples, assuming that all couples are of equal strength. They differentiated and made a hierarchy of coupling strength. The worst coupling that has been identified is the direct access of foreign instance variables. Also the coupling realized by sending messages to a component of one of the object's components was considered stronger (worse) than sending messages to the object itself. Access to instance variables of foreign classes constitutes stronger coupling than access to instance variables of super-classes. In addition, passing a message with a wide parameter interface yields a stronger coupling compared with a message with a slim interface. The second critic found in [Hit 96] concerns the fact that it is not clear whether messages sent to a part of self - i.e. to an instance variable of class type - contribute to CBO or not. The third deficiency of the CBO metric is that it neglects inheritance related connections which excluded the couples realized by immediate access to instance variables inherited from super classes, a kind of coupling considered to be among the worst types of coupling. [Bar99]

- Response For a Class (RFC)

Response for a class (RFC) measure captures the size of the response set of a class [Fen97]. The response set of a class consists of the set M of methods of the class, and the set of methods invoked by methods in M. In other words, the response set is the set of methods that can potentially be executed in response to a message received by an object of that class [Chi91, Bri00]. RFC measures the communication object among classes

through message passing. A message can cause an object to "behave" in a particular manner by invoking a particular method or set of methods. Methods can be viewed as definitions of responses to possible messages. It is reasonable, therefore, to define a response set for an object in the following manner: Response set of an object = {methods that can be invoked in response to a message to the object}. Note that this set will include methods outside the object as well, since methods within the object may call methods from other objects.

The following quotation is from the original paper of Chidamber and Kemerer[Chi94].

Definition

RFC = | RS | where RS is the response set for the class.

Theoretical basis:

The response set for the class can be expressed as:

$$RS = \{M\} \quad \bigcup_{all\, i} \quad \{Ri\}$$

where { Ri } = set of methods called by method i
and { M } = set of all methods in the class

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. The cardinality of this set is a measure of the attributes of objects in the class. Since it specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and the use of other classes.

Viewpoints

• If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

• The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

• A worst case value for possible responses will assist in appropriate allocation of testing time. [Chi94]

Discussion

By measuring the total communication potential, this measure is not only a measure of the complexity of the class but also related to coupling which looks not independent of the CBO metric [Bar99]. According to the definition of RFC, all coupling with external methods are of equal strength. However, calling a method of a super class cannot be seen as harmful as calling methods of other classes. For example, the default constructor of the super class is called automatically form the constructor of subclasses in Java. Furthermore, overloading a method in a subclass typically contains a call to the overloaded method of the super class. [Sys99]

- MPC

Li and Henry [Li93, Li95] examined the original measurement suite of Chidamber and Kemerer and proposed a new measurement Message Passing Coupling (MPC), which is used to measure the complexity of message passing among classes. Message passing is one of the typical types of communication between the objects in the Object-Oriented paradigm. When an object needs some service that another object provides, messages are sent from one object to the other object. A message is usually composed of the object-ID, the service (method) requested, and the parameter list for that method. MPC is the number of send statements defined in a class, where a send statement is a message sent

out from a method of class A to class B. Although messages are passed among objects, the types of messages passed are defined in classes. Therefore message passing is calculated at the class level instead of the object level [Li93].

Definition

Message Passing Coupling (MPC) is the number of method invocations in a class. [Li93]

MPC = number of send-statements defined in a class

Viewpoints

The number of messages sent out from a class may indicate how dependent the implementation of the local methods is upon the methods in other classes [Bar99].

Discussion

This may not be indicative of the number of messages received by the class. That means that in defining this metric as a coupling measure the authors did not take into account the dependencies of other classes on the class being analyzed. [Bar99]

2.2.4 Measurement tools

Coupling measurements are computed based on source code analysis of software systems. Many enhanced reverse engineering tools have automated the derivation of design measurements to exhibit the inherent design issues and related quality aspects of software systems. In our CONCEPT environment, CBO, RFC, and MPC are automated for quantifying the original source code and the subset of the source in the form of a slice. In what follows, some tools are reviewed to see how these measurements are computed.

- Rigi

Rigi [Won98] is a framework under development at the University of Victoria for program understanding, software analysis, reverse engineering, and programming-in-the-large. One major goal is to extract abstractions from software representations and transfer this information into the minds of software engineers for software evolution purposes. The focus is on summarizing, querying, representing, visualizing, and evaluating the structure of large, evolving software systems. In Rigi, seven measurements are categorized into inheritance measurements, complexity measurements and communication measurements [Sys99]. CBO and RFC are selected as coupling measurements, while MPC is not in the consideration.

In Rigi, CBO measures coupling between classes that are not related through inheritance. For calculating CBO, both constructors and methods are taken into account. The following relationships between two classes that are not in a super class-subclass relation are considered to cause coupling: method calls, constructor calls, instance variable assignments, or other kind of instance variable access (usage). Static blocks are not examined.

In Rigi, RFC looks at the combination of the complexity of a class through the number of the methods and the amount of communication with other classes. When calculating RFC, calls between methods, constructors, and static blocks are taken into account. For a class C, let Mi be a set of all methods, constructors and static blocks in C. Let Mo be a set

of methods, constructors, and static blocks belonging to any other classes that are called by the members of Mi. The RFC for class C is calculated as the size of a set Mi U Mo.

- FaMoos

FaMoos is a European project investigating the Object-Oriented Reengineering techniques for dealing with Object-Oriented legacy systems [Bar99]. The key techniques evolved in FaMoos project are software measurement, program visualization, source code abstracting and refactoring. In FaMoos, CBO, RFC, MPC and some other that have proven to be particularly useful measurements have been intensively discussed.

However, in FaMoos, only RFC was implemented as part of their system, and CBO and MPC are only theoretically discussed. RFC here measures complexity and coupling properties of a class by evaluating the size of the response set of the class, i.e. how many methods (local to the class and methods from other classes) can be potentially invoked by invoking methods from the class. RFC for a class C is defined as RFC = |RS|, where the response set RS is given by

$$RS = M \,|\cup \bigcup_{m \in M} Rm$$

M is the set of methods defined in C and Rm is the set of methods called by method $m \in M$.

- From Briand et al.

Briand et al performed an empirical study of design measurement by using a development project which was performed at the University of Maryland. In their study,

they tried to do a comprehensive empirical validation of all the Object-Oriented design measures found in the literature.

They computed CBO, RFC and MPC as well as many other Object-Oriented measurements. In their study, two kind of CBO are derived such as CBO and CBO'. A class is coupled to another, if methods of one class use methods or attributes of the other, or vice versa. CBO for a class is then defined as the number of other classes to which it is coupled. This includes inheritance based coupling (coupling between classes related via inheritance). CBO' for a class is defined similar to CBO except that inheritance-based coupling is not counted.

Similarly to CBO and CBO', RFC_{∞} and RFC_1 are also distinguished in their study. The response set for a class consists of the set M of methods of the class, and the set of methods directly or indirectly invoked by methods in M. In other words, the response set is the set of methods that can potentially be executed in response to a message received by an object or that class. $RFC_{\infty}$is the number of methods in the response set of the class. RFC_1 is same as RFC except that methods indirectly invoked by methods in M are not included in the response set this time. For example, if a class C has one method c, c calls another class B' method b, and b also calls other Class A's method a. In this case, RFC_{∞} according to the definition provided in [Chi91] is 3, whereas RFC_1 in this case is 2 because only direct method invocations are considered.

As to MPC, they compute the number of method invocations in a class.

- CONCEPT

Based on the above reviews of the computation of MPC, RFC and CBO, we derive MPC, RFC and CBO according to the following definitions. **MPC** corresponds to all the direct method invocations (including constructor invocations) to other classes with respect to a particular class. **RFC** is the direct and indirect method invocation couplings to other classes and the entire member functions in a particular class: RFC = the number of directly and indirectly invoked methods + number of the class methods (including constructor). **CBO** provides the number of classes to which a given class is coupled by method invocations and attribute usages. CBO = MPC + Attribute Usage Coupling. In fact, inspecting the definition of CBO reveals that the CBO for a class is a count of the number of other classes to which it is coupled. Therefore, CBO should include all types of causes that lead to the interactions among classes. However, the further explanation is limited to the causes in method invocations and attributes access. Seeing this contradict, we inspect Rigi and FaMoos and other studies, and discovered all these experiments are computing the CBO according to the later explanation. Therefore, we adopt in our CONCEPT project the following CBO measure: CBO is computed based on method invocations and attribute access, and take methods, constructors and initializers into account.

Since our goal is to measure design and code qualities of the subject system, we do not take Java System classes such as in java.lang.*, java.util.* libraries into account. If we were to include these Java System classes, it would skew our results towards the quality

of the Java System classes rather than focusing on the classes of the project under investigation.

2.3 CONCEPT framework

The CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) is a lightweight reverse engineering environment, which we utilize to investigate novel program comprehension techniques and approaches to assist programmers during the creation of mental models while comprehending software systems. Within our CONCEPT project we are exploring new program slicing algorithms and investigate their application in different software engineering sub-domain, e.g. software measurement, design pattern recovery, software visualization, feature analysis and architectural recovery, etc. Figure 3 shows an architectural overview of the CONCEPT project.

The CONCEPT framework is built as a layered architecture with our CONCEPT repository meta-model being the integrating part of the framework. The meta-model stores both static and dynamic source code information derived from the parsing and monitoring layer. The database API layer decouples the analysis (slicing, measurement and feature extraction) layer from the repository. The visualization and application layer are created on top of the analysis layer.

The measurement framework introduced in this paper was designed as a separate plug-in into the analysis layer, making these measurements easily accessible to be integrated in

other parts of the framework (e.g. clustering techniques for software visualization, testing and design evaluation).

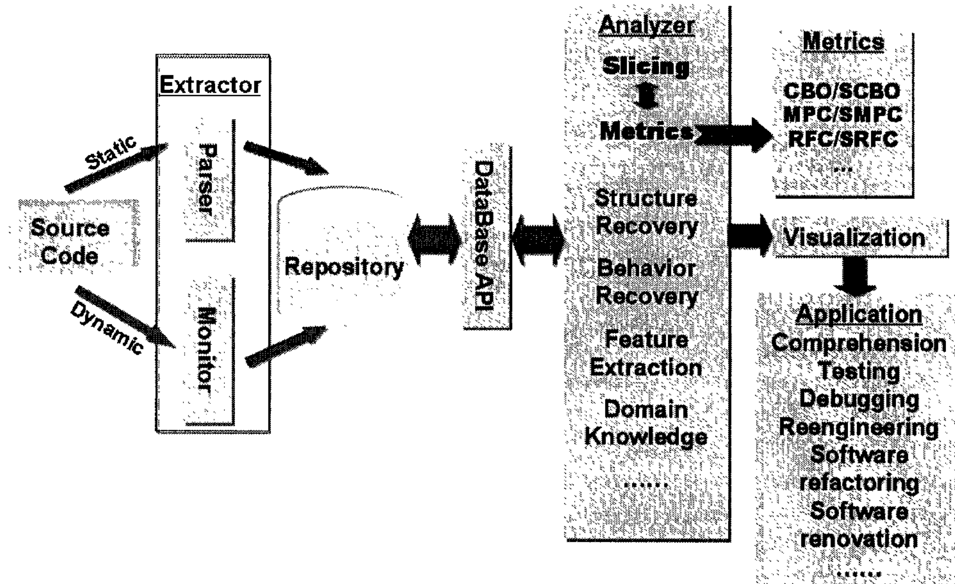


Figure 3 CONCEPT framework

3 SLICING BASED COUPLING MEASUREMENT

In this chapter, we introduce our slicing based measurement framework by combining traditional slicing approaches with traditional coupling measurements to utilize the strengths of both. The framework provides users with an "as needed" approach in analyzing and evaluating Object-Oriented program design. The measurements can be applied at different levels of abstraction, depending on the particular needs and the context of a comprehension task.

3.1 Slicing based measurement literature

Most of the existing research in the area of slicing based measurement focuses on identifying and analyzing class or functional cohesion. Weiser presented in [Wei81] several slicing based measurements, such as coverage, clustering, parallelism, and tightness. These slicing based measurements are mainly concerned with the cohesiveness of the program with respect to the slice. Longworth [Lon85] hypothesized that a coverage measurement could be used to differentiate between different types of cohesion. Thuss [Ott93] introduced a metric slice that was further refined by Ott and Bieman, called the metric data slice. Their study showed that there exists an association between cohesion and the metric slice, and that measures the function cohesion. Harman et al. introduced in [Har97] several new measurements for evaluating the complexity of an expression and applied them on some standard measurements introduced earlier by Ott [Ott93]. Harman et al. [Har97] also introduced an expression metric to calculate the significance of the

intersection of program slices. As a result of their study, they raised concerns about the computability of a cohesion metric based on program slicing. Ott and Bieman extended their work in [Ott98] to provide a more generic approach to their slicing based metric. So far, most of the research in slicing based design measurement is limited to cohesion measurement. To our knowledge, the only research conducted in the area of program slicing based coupling measures is by Harman et al [Har97] and Li [Li01]. Harman and et al., are applying static program slicing as code-level based measurements for assessing coupling, and they outline the use of the measurements as components of predication systems. Their slice-based information flow measurements use both static backward and forward slicing techniques to measure function coupling. Binxin Li [Li01] introduced a slice-based framework for OO coupling measurement. The measurements discussed are based on both Harman and Ott's earlier work by identifying many OO slicing based coupling definitions at different abstraction level, and by extending the information-flow coupling to OO program structures.

3.2 Motivation for the measurement framework

The motivation for our slicing based measurement framework is to take the strengths of both existing slicing algorithms and promising coupling measurements. Program slicing is a program reduction technique, which has been put in the applications of testing, debugging, measurement, and maintenance and comprehension [Har01]. Moreover, program measurement is used for quantifying some aspects of the source code with the purpose to aid understanding, controlling and monitoring software products [Fen97]. Hence, slicing and coupling measurement can be combined to provide an aid in reducing

cognitive burdens, predicating maintenance efforts and prioritizing maintenance tasks to ensure software evolution.

3.2.1 Identify and focus

Typically, a program performs a large set of functions/outputs. Program slicing allows a programmer to decompose a program based on different slicing criteria. Rather than trying to comprehend, measure and evaluate all of a system's functionalities, programmers tend to focus on some selected function and those parts of a program that are directly related to that function. The application of slicing based measurements at different abstraction levels provides additional focus on the application context in which these measurements are applied. This focused approach for applying software measurement provides the ability to evaluate different slices with each other, based on the applied software metric.

3.2.2 Direct versus indirect coupling measurement

The traditional CBO, MPC and RFC for a class are derived by only considering all direct coupling interactions and neglecting other indirect coupled connections. The limitation of these measurements is based on the fact that without further data and control dependency analysis, the source code within a class can only provide the direct coupling information. However, indirect coupling has to be carefully treated during maintenance because it is responsible for ripple effects and change propagation to other parts of the program.

Figure 4 shows the direct and indirect coupling in the message chain that originated from the method a.

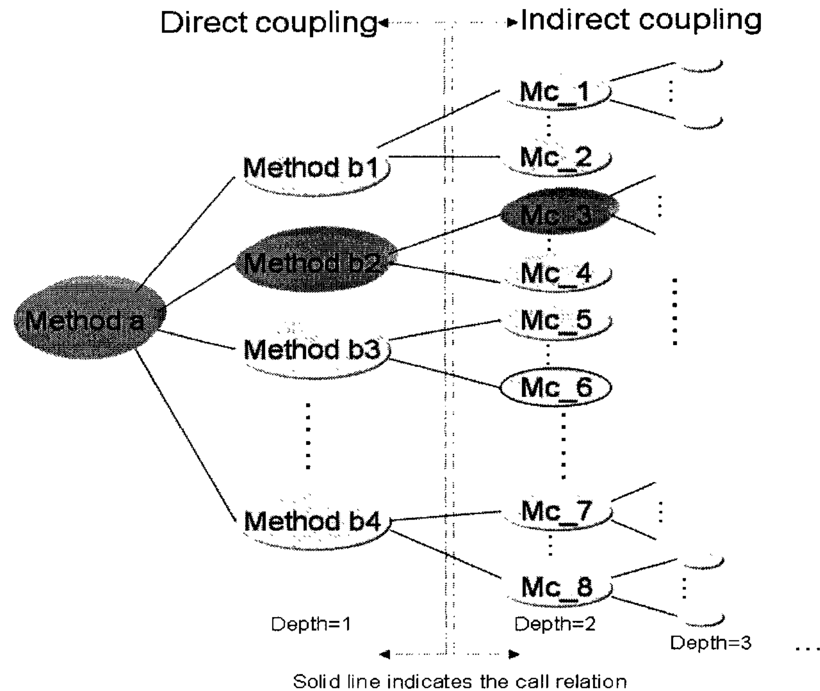


Figure 4 Direct and indirect coupling in a message chain

Indirect coupling can be described as all these relationships that are affected by an existing direct coupling relationship. In order to identify indirect coupling, one has to trace the "invoke" methods and/or classes in the "uses" chain.

As the example in Figure 5 illustrates, two message chains that can be identified are Elevator prompt() → Panel pressButton() → Button press() and Elevator prompt() → Panel howmany() → Button pressed( ). Class Elevator and class Panel have two direct coupling occurrences (indicated by the solid lines), and class Elevator and class Button are coupled through two indirect coupling relationships (dashed lines). If the coupling depth is large, e.g. method a→ method b1→ mc_3→ ... (see Figure 4), the identification

of the involved methods in a complete message chain becomes more complex because the involved methods are encapsulated in different classes and each involved method might invoke a set of other methods. However, within the context of program slicing, identifying the indirect coupling information is very straightforward by identifying the relevant control flow in the slice. For the generic slicing based coupling measurements, both direct and indirect coupling for the class/method are considered and measured.

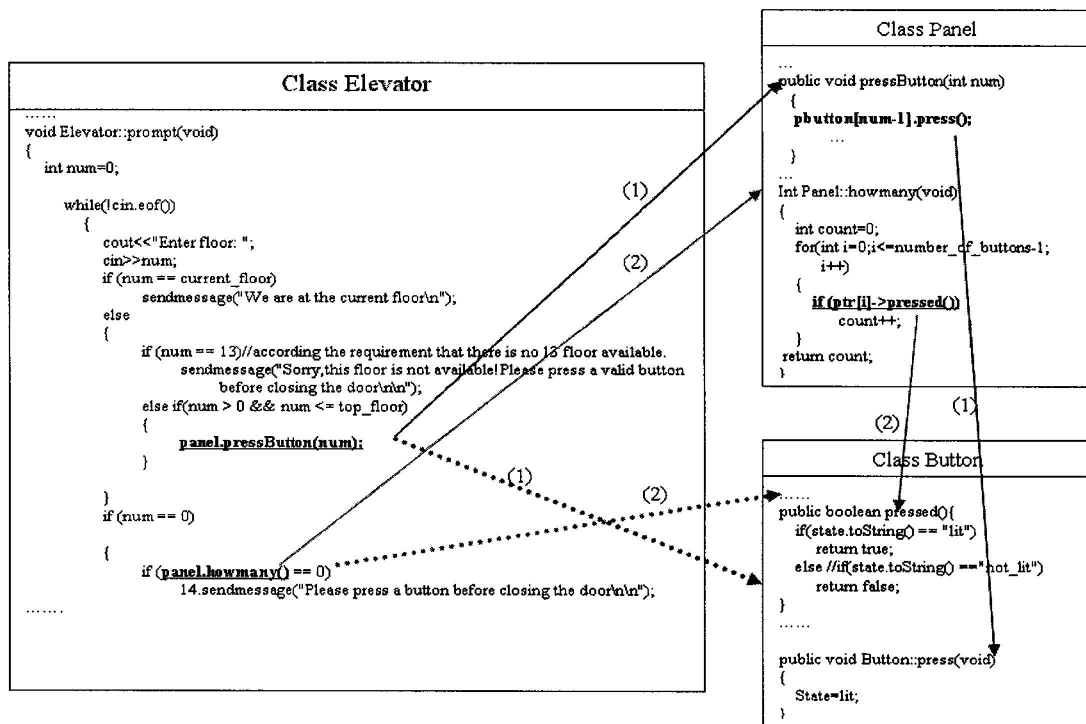


Figure 5 Example for indirect coupling

3.2.3 Measuring export coupling

Import and export coupling are used to distinguish the direction of coupling by assigning client and server roles to a coupling interaction. These coupling types capture the impact of changes performed in external granularity on a given granularity, or the impact on

external granularity when changes are performed in a given granularity. At "locus of impact", a client granularity uses services (in other words, imports some codes or changes from servers), whereas a server provides services (in other words, exports codes or changes to clients) [Bri96]. For example, with respect to a class C, invoking other class's methods introduces import coupling whereas one method in class C invoked by other classes corresponds to export coupling.

Import coupling to a class can easily be determined by analyzing the source code of a class. However, for the identification of export coupling, some source code analysis has to be applied to identify the different files or packages that might be involved. Similar to the direct and indirect coupling, program slicing can aid in identifying the existing export coupling within the slice. This property of program slices allows us to include import and export coupling measurements within our measurement framework.

3.3 Generic slicing based coupling measurement

As part of our slicing-based measurement framework, we adopt traditional source code measurements for program slices. We refer to these measurements as generic slicing based coupling measurements, because it is based on traditional measurements that are re-applied in the more focused context of program slicing. The presented measurements are not limited to a specific slicing algorithm and can be combined with most traditional slicing algorithms, such as dynamic slicing, conditional slicing [Luc01], and predicate slicing [Ril02]. Each of these slicing algorithms has its own properties and application domain, but they share the common goal of identifying all statements relevant to the

behavior of a particular slicing criterion. The already well accepted and validated coupling measurements [Gla00, Ema01, Bri96] such as CBO, RFC and MPC are selected as the start point for deriving generic slicing based coupling measurements. These measurements emphasize measuring coupling characteristics of classes and their interactions. The basic idea of these slicing based generic measurement is to consider only the parts of the source code that are included in the slice, which can provide programmers with the ability to rank, compare, and evaluate the design that implements a particular slicing criterion.

In the following section, we use the notation "#" and "|" which is consistent with [Har97], where "|" denotes a constraint condition and "#" corresponds to a set cardinality. Set cardinality "#" refers to the number of entities in a particular set. S is the entire set of statements in the slice with respect to a given slicing criterion. C corresponds to a class and M to a method scope, where C or M refers to the set of statements in the slice which have direct or indirect control dependence on the particular C or M.

3.3.1 Slicing based coupling measurement

One of the commonalities of MPC, RFC and CBO is that they all measure the coupling at the class level. Therefore, a natural extension to these traditional SMPC, SRFC and SCBO measurement is to apply them on a class. The main difference between MPC, RFC and CBO and their slice equivalents (denoted by the S preceding respectively) SMPC, SRFC and SCBO are:

- The slicing based version of these coupling measures only consider those parts of a class that are included in a computed slice rather than the entire class.
- The slicing based coupling measures capture both direct and indirect coupling within the system rather than only direct coupling of traditional coupling measurements.
- The slicing based coupling can compute both import and export coupling within the slice rather than only import coupling in a class

Based on these differences, we present the notion of slicing based SMPC, SRFC and SCBO as follows:

Slicing based message passing coupling (SMPC) calculates the number of send statements that are included in a class, where a send statement corresponds to a message send from one method in a class to a method in another class in the slice. SMPC measures therefore the complexity of message passing among classes and the import coupling. The larger the SMPC for a particular class the more likely it will depend on other classes.

$$\text{SMPC} \mid C = \#\left(\bigcup_{ST(M) \in S} (ST(M) \in C \wedge M \notin C) \right)$$

ST (M) is the send statement in class C and M is the invoked method of another class

Slicing based Response for a Class (SRFC) computes the number of methods in the slice that potentially can be executed in response to a message received by an object of that class. SRFC includes both member functions (included in the slice) within the class, as well as member functions external to the class that can be invoked by the member functions of a selected class. If a large number of methods can be invoked as a response to a message,

indicating a higher comprehension complexity and therefore complicating also the testing and debugging of that class.

$$SRFC \mid C = \#\left(\bigcup_{M' \in S} M' \in C \right) + \#\left(\bigcup_{ST(M) \in S} (ST(M) \in C \wedge M \notin C) \right)$$

ST(M) is the sent statement in class C; M is the invoked method of another class, M' is the member function in C

Slicing based Coupling between Objects (SCBO) computes the number of classes to which a given class is coupled by using methods or instant variables of other classes in the slice. Excessive coupling between classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.

$$SCBO \mid C = SMPC \mid C + SIVC \mid C$$
$$= SMPC \mid C + \#\left(\bigcup_{IV(V) \in S} IV(V) \in C \wedge V \notin C \right)$$

IV(V) is the statement including instance variable in class C, where V is the instance variable of another class type

SMPC, SRFC, and SCBO characterize the coupling relations of the classes within the slice. The following table 1 summarizes the properties of the above refined slicing based coupling measurements.

Table 2 Properties of the proposed measurements

Measure	Type of coupling	Strength	Import /export coupling	Direct/indirect coupling
SMPC	Method invocation	#method invocations	Import	Direct & indirect
SRFC	Method invocation	#methods invoked	Import	Direct & indirect
SCBO	-Method invocation -Attribute reference	#coupled classes	Import	Direct & indirect

3.3.2 Views on the slice based measurement

The slicing based coupling measurements introduced above provide a class level view of the coupling within a given slice. In fact, a more fine level view of the slice coupling can be obtained on the method level, and a more coarse view of the slice coupling can be obtained by applying these measurements on the entire slice. Besides the previous class level coupling measurements, programmers have the flexibility to look into the interior coupling of the slice at three granularity levels: method, class and entire slice. Thus, they can selectively choose a particular granularity coupling analysis as needed for future maintenance. The following gives the refined definitions for measuring the fine method level coupling and coarse slice level coupling.

SMPC | M (SMPC for a particular method) computes all send statements within a given method, which invoke methods of other classes. SMPC | M provides a more fine level view of the total number of method invocation statements of the given method in the slice, allowing for a message passing ranking of the different methods with respect to the slicing criterion.

$$SMPC \mid M = \#\left(\bigcup_{ST(M') \in S} (ST(M') \in M \wedge M' \in C \wedge M \notin C \right)$$

ST (M') is the sent statement in method M, where M' is the invoked method of another class

SMPC | S (SMPC for the complete slice) is the sum of all the SMPC | C. SMPC | S provides a coarse view of the total number of method invocation statements in the slice. It can be used to compute the overall coupling complexity of different slices.

$$SMPC \mid S = \sum_{C \in S} SMPC \mid C$$

SRFC | M (Slice based response set for a particular method) counts the number of methods that can be potentially executed within the context of the given method in response to a message received by an object of that class (containing the given method) in the slice. SRFC|M is the 1 plus sent statement to other classes (SMPC|M), where 1 is the method itself. SMPC|M are the number of the methods might be invoked to acknowledge an incoming message.

$$SRFC \mid M = 1 + SMPC \mid M$$

SRFC | S (SRFC for the complete slice) is the sum of all the SRFC|C in the given slice. SRFC|S provides a coarse view of the total number of potential methods that can be invoked in respond to some incoming message from other classes. This measurement provides the option to compare different slices with each other or to generally evaluate the slice with respect to its internal interaction.

$$SRFC \mid S = \sum_{C \in S} SRFC \mid C$$

SCBO | M (Slicing based coupling between objects for a given method) computes the number of send statements to invoke the methods of other classes and the number of using instance variables of other classes.

$$SCBO \mid M = SMPC \mid M + SIVC \mid M$$
$$= SMPC \mid M + \# \left(\bigcup_{IV(V) \in S} IV(V) \in M \wedge V \in C \wedge M \notin C \right)$$

SCBO | S (SCBO for the complete slice) corresponds to the sum of all the SCBO|C in the particular slice. SCBO|S provides a coarse view of the total number of message passing coupling and instance variables usage coupling in the given slice. It allows for a direct comparison of the different SCBO among program slices.

$$SCBO \mid S = \sum_{C \in S} SMPC \mid C + SIVC \mid C$$

3.3.3 Measure export coupling

Another important factor that should be considered during the computation of coupling measurement is their coupling direction. The previously presented slice based coupling measurements are based only on import coupling, capturing how the client class uses other parts of the system or import some codes or changes from server classes. Export coupling, on the other hand, captures the impact on other parts. In other words, server class export source codes or changes or provide service to client classes. Within this given context a class might export the change to other classes by method invoked or instance variables being accessed by exterior classes. The overall export coupling within a slice is the sum of all instance variable type coupling and method invoked statements in the slice. Based on these observations, we extend the previous definition of SMPC and SCBO at the different view levels to include the export coupling. The SRFC measures already regard the response set of potential methods that are invoked and consider the calling message as an import coupling to the object and at the same time it is an export coupling to the classes who receive the calling messages.

ESMPC | M (Slicing based message passing export coupling at the method level) computes the number of send statements belonging to other classes who invoke the given method in the slice.

$$ESMPC \mid M = \#\left(\bigcup_{ST(M) \in S} ST(M) \notin C \wedge M \in C \in S \right)$$

ESMPC | C (Slicing based message passing coupling for export direction at class level) corresponds to the number of send statements of other classes who invoke the methods of a given class in the slice.

$$ESMPC \mid C = \sum_{M \in C} ESMPC \mid M$$

ESMPC | S (Slicing based message passing coupling for export direction at slice level) is the sum of all the ESMPC|C in the slice, allowing for a ranking and comparison of the total ESMPC among different slices.

$$ESMPC \mid S = \sum_{C \in S} ESMPC \mid C$$

ESCBO | M (Slicing based coupling between Objects for export direction at the method level) counts the number of send statements of an exterior class that uses the given method. This method level coupling is the same as ESMPC|M. Since methods are not a data type, it is not necessary to consider the instance variables coupling.

$$ESCBO \mid M = ESMPC \mid M$$

ESCBO | C (Slicing based coupling between Objects for export direction at the method level) counts the number of send statements of other classes who use the methods of a given class and use the instance variables whose type is the given class

$$ESCBO \mid C = EMPC \mid C + EIVC \mid C$$

$$= EMPC \mid C + \bigcup_{IV(V) \in S} \left(IV(V) \in C' \wedge V \in C \wedge V \notin C' \right)$$

C' corresponds to an external class

ESCBO | S (Slicing based coupling between Objects for export direction at the method level) is the sum of all ESCBO|C in the slice. It provides an overview of the slice interior classes coupling and therefore allows for a comparison of slices based on their overall export coupling

$$ESCBO \mid S = \sum_{C \in S} ESCBO \mid C$$

3.4 Validation of the measurements

Validating a software evaluation measure is a process of ensuring that it is a proper numerical characterization of the claimed attribute [Fen97]. There are two types of software measurement validation: theoretical and empirical. The theoretical validation is a process of ensuring that the fundamental measure is satisfying the representation condition of measurement theory [Org02]. The empirical validation is a process of establishing the accuracy of the software measurement by empirical means [Org02]. In the context of assessment and comparison, the empirical validation identifies the extent to which a measure characterizes a stated attribute by some test cases against reality.

The presented CBO, MPC and RFC measurements are based on existing design measurements introduced by Chidamber and Kemerer [Chi91, Chi94] and Li and Henry [Li93]. These measurements have already been evaluated both in theory [Bri96, Chi94, Hit96] and by experiment [Bas95, Bas96, Bri96], showing their usefulness during typical testing and maintenance activities for Object-Oriented systems. In our approach, CBO, RFC and MPC belong to ordinal scale, which preserve the ordering or ranking of the classes. High number of CBO, RFC or MPC indicates the high coupling complexity of the class, which in turn indicates the cognitive complexity and maintainability. The basis for the empirical relation systems in CBO and RFC is the set of "ontological principles" proposed by Bunge [Fen97], which was validated by Chidamber and Kemerer [Chi94]

and Hitz et al[Hit96]. The empirical study of Li and Henry[li93] concluded that there "is a strong relationship between measurements and maintenance effort in Object-Oriented systems" and that "maintenance effort can be predicted from combinations of measurements collected from source code". The results from Victor R. Basili et al [Bas96] and El-Eman [Ema99] also claimed that coupling measurements are associated with fault-proneness.

In our research, we applied program slicing, a program reduction technique which has its application roots in software maintenance and software design evaluation [Har97, Wei81], to refine the original CBO, RFC, and MPC measurements. Program slicing, as defined by Weiser [Wei81], is a source code to source code transformation that guarantees the same behavior of the slice (with respect to a slicing criterion) as the original program. We extended the CBO, RFC and MPC measurements to be applied on a program slice, rather than the whole program. Therefore, our proposed coupling measures can be seen as a refinement that contains a subset of the original program design that reflects the same behavior as the original design. The basic advantages of the traditional measurements and their application domains still hold for our measurements. Currently, we are in short of real slicing information, so some interesting empirical study can not be performed. However, the validation of how the slicing based coupling measurements benefit our design elevation and maintenance will be the next focus of our research.

4 ABSTRACTION DRIVEN SLICING BASED MEASUREMENTS

Harman et al. proposed in [Har02] an algorithm that unifies concept assignment and slicing for program comprehension. An executable concept slice is obtained by first identifying a set of related statements by the plausible reasoning system and then slicing all these statements to gain an executable slice for the concept of interest. Li [Li01] computed statement-level slice, method-variable slice, class-level slice and module-level slice to compose a slicing framework to represent the abstraction levels by both utilizing forward and backward slicing.

4.1 Slicing hierarchy

Table 3 Slicing hierarchy

Granularity slicing	Slicing criterion	Definition
Statement level	<M, s, V>	A Statement-level slice consists of all statements and control predicates affecting (or affected by) the variable sets V in statement s
Method level	<C,m,V>	A Method-level slice consists of all statements and control predicates affecting (or affected by) the variable sets V in method m
Class level	<P,fc,V>	Class-level slice consists of all statements and control predicates affecting (or affected by) the variable sets V in class c
Package level	<S,p,V>	Package-level slice consists of all statements and control predicates affecting (or affected by) the variable sets V in package p
Concept level	<S,concept,V>	A concept-level slice consists of all statements and control predicates affecting (or affected by) the variable sets V in the concept

Based on the above work, we propose a slicing hierarchy (see Table 3) that can extend our previously defined slicing based measurements. In essence, the slicing hierarchy only reapplies existing slicing algorithms to compute slices at different abstraction levels. A stepwise approach is proposed for deriving the whole slice abstraction hierarchy. The initial abstraction is at statement level slice that corresponds to a slice at a particular statement. The method level slicing corresponds to a union of all statement level slices within that method, and similarly, the same union principles apply for class/package/concept level slicing. The resulting slices can then be used as the source for our slicing based coupling measurements. The limitation of abstraction based slicing is the cost associated with computing slices within the abstraction hierarchy. The number of required slice computations grows rapidly with the abstraction level. However, these slice computations can be performed as part of a batch processing, with the resulting slices being stored in the repository. The goal of deriving these different abstraction levels is to provide programmers with some additional insights during future comprehension and maintenance tasks. The slice based measurements at the different level will provide users with the ability to gain detailed insights of the structural relationships and existing problems, by evaluating the system from specific view points (slicing criteria).

In chapter 3, we introduced the framework of generic SCBO, SMPC and SRFC measurements and their refinement on different view levels as well as by including export coupling. All these slicing based coupling measurements can be re-applied on our slicing abstraction hierarchy. Programmers can selectively choose the metric abstraction level that provides the required additional insights and guidance to complete the task.

4.2 Abstraction oriented slicing base coupling measurement

Hypothesis 1: Different levels of granularity and detail can be added to coupling measurements to refine these measurements for specific maintenance tasks.

It is possible to derive coupling measures for different levels of granularity in OO program. Coupling measures on the package or concept support a more general analysis and restructuring of the system, compared to the more fine grained measures on the method/statement/class level slice that focuses on more class specific design aspects.

Our generic slice based coupling measurements can be re-applied to measure the coupling of the different granularity level within the slice abstraction hierarchy. This easy extension provides the user with the ability to select not only the desired slice granularity, but also the metric that might provide additional insights.

Application of the framework in theory

The ability to reverse engineer Object-Oriented legacy systems has become a vital matter in today's software industry. Hence, maintenance programmers need to comprehend the inner workings of legacy systems and to identify potential design anomalies. Typical application domains for the presented slicing based coupling measurements include re-engineering which is the renovation of the source code as well as the business process. It is performed via a reverse engineering step by first creating a higher level of abstraction, and then a transformation on the design level is applied, followed by a forward engineering step based on the improved design. For the re-engineering, it is essential to identify places of low design quality. In this particular case, our proposed slicing

measurements can be applied at different abstraction levels, and also provide additional insights into the relationship among the software artifacts at each specific level. Other comprehension and maintenance applications for the abstraction hierarchy level based slicing measurements can be found in software transformation or during software renovation. The measurements can guide programmers during these proactive software maintenance tasks by identifying problem areas and allowing for a zoom in/zoom out capability on the measurement level. In the following chapter, we have a closer look at the applicability of our slicing based coupling as part of predicting change efforts during change impact analysis.

5 SLICING BASED COUPLING MEASUREMENT FOR CHANGE IMPACT ANALYSIS

Impact analysis is a long established application domain in software maintenance which attempts to identify the impacts or "ripple effects" of a change in one part of a program on the remaining parts of the program. Change impact analysis is essential to ensure the quality of the software during its evolution [Lee98].Program slicing provides a good foundation for code level change impact analysis since both program slicing and impact analysis are based on low level source code extractions and analysis. Within the context of impact analysis, forward slicing can be used to identify those statements that might be affected by the change in the statement; moreover, backward slicing is also applied to check whether the introduced value of the change is valid in the new context [Wan96]. Forward slicing traces data and control paths originating at the locus of change, while backward slicing traces the paths in the reverse direction.

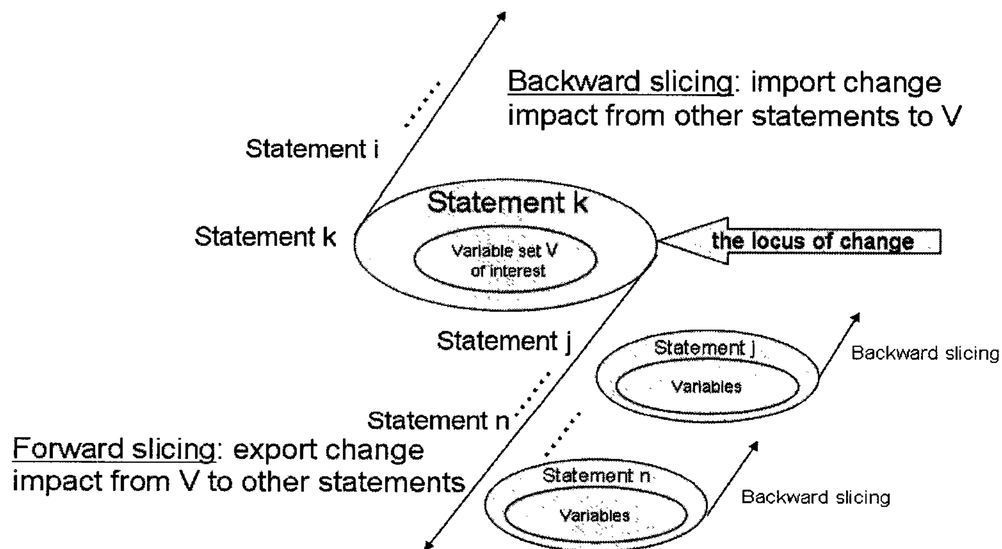


Figure 6 Impact analysis through program slicing

Figure 6 illustrates the combined approach of backward slicing and forward slicing to perform impact analysis for a change at statement K. It has to be noted that a simple unification of forward and backward slices will not result in an "executable" and behavior preserving slice. In order to compute a behavior preserving and executable slice, an additional step is required. A backward slice for every statement in the obtained forward slice has to be computed. The resulting slice however, might be very large and expensive to obtain.

Hypothesis 2: Different slicing algorithms can be added to the generic slice based coupling measurements to refine these measurements for different application domains.

As we addressed earlier, the generic slicing based coupling measures are applicable to different types of slicing algorithms. For the code level change impact analysis, both forward and backward slicing are required to identify the source code impact cause by a change request [Wan96]. All the generic slicing based coupling measures are applicable for change impact associated slices. By applying the generic slicing based coupling measurements on both the forward and backward slicing, the import changes and export changes to that point of the code change can be clearly presented. Thus, the impact slice (both forward and backward slices) based coupling measurements provide a good potential for improving the predication of the change effort. Depending on the abstraction level of change (variable, statement, method, etc.), the user can select the appropriate coupling measurement level.

6 IMPLEMENTATION

6.1 Goal

The CONCEPT project is a reverse engineering environment having a number of capabilities for analyzing source code, including different slicing algorithms, analysis of static and dynamic dependencies, Object-Oriented measurement, design pattern recovery, and software exploration and visualization. As section 2.3 introduced, our software measurement is a separate plug-in into the analysis layer, making the measurements easily accessible to be integrated in other parts of the framework, e.g. clustering techniques for software visualization, testing and design evaluation.

The major goal of design measurement in CONCEPT is to analyze and extract fundamental coupling information (discussed in section 6.3.1) from source code or program slices and then to quantify this information with the goal to compute CBO/SCBO, RFC/SRFC, MPC/SMPC to guide programmers during program comprehending and design evaluation. Our design measurement implementation was performed in two phases: The first phase is the basic data collection and transformation. During this phase, the focus was on source code analysis and data collection. The whole OO hierarchy is built at this phase, including the information of packages, classes, methods (including constructor, initializer), attribute, local variables. Moreover, the basic coupling information is collected based on the identified reference analysis. This information includes client and server class information, coupling location, type etc.

During the second phase, measurements are computed based on the achieved coupling information and OO hierarchy information from the first phase. Traditional CBO, RFC, MPC and slicing based SCBO, SRFC and SMPC are derived and in turn are utilized for program visualization. With the aid of the objective measurement data, some design issues can also be comparatively or statistically derived.

6.2 Basic workflow

The following work flow illustrates the general steps to derive the design measurement (see figure 7).

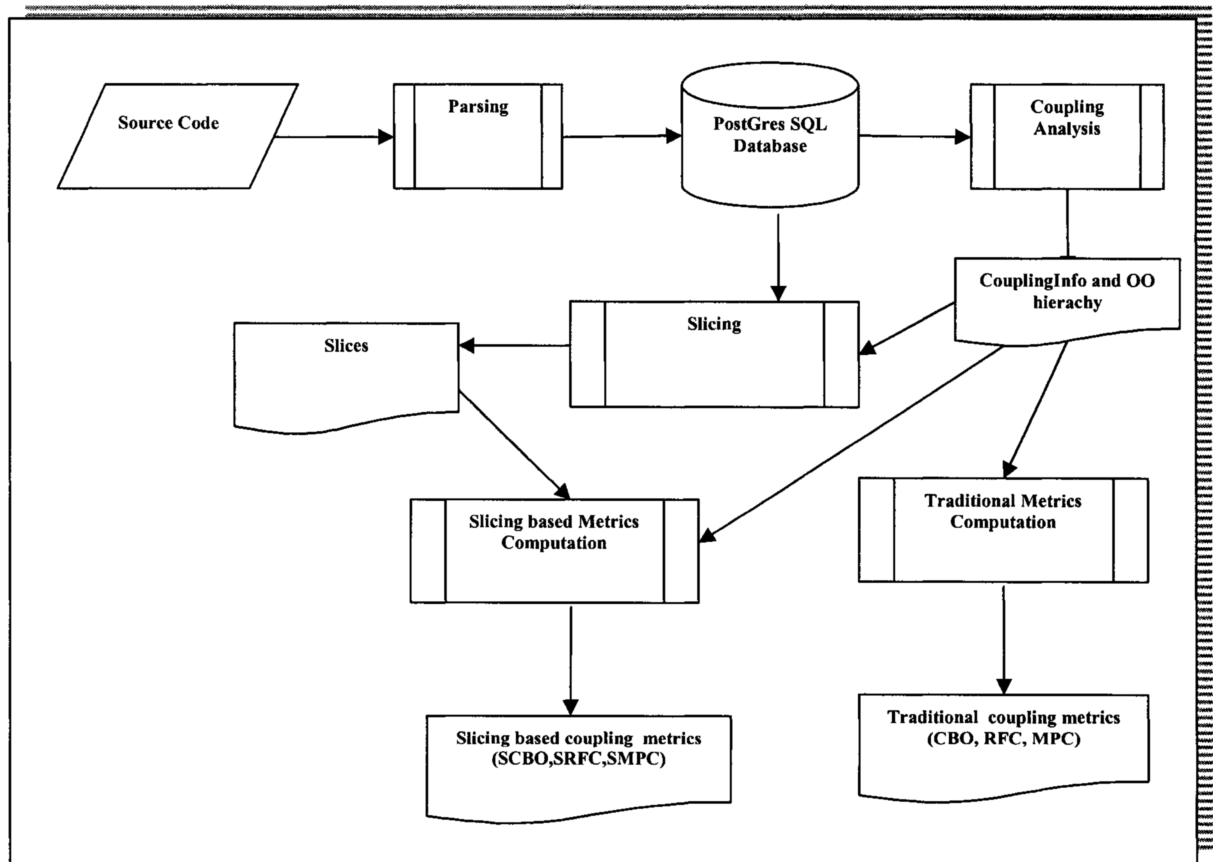


Figure 7 Workflow for measurement derivation

6.2.1 Parsing

The start point of the CONCEPT project is to analyze the source code of the target project. This project uses the JavaC compiler (see Figure 8) to extract the parsed information and represents the information as an AST (Abstract Syntax Tree). An example is given in figure 9 (this work was implemented by Yonggang Zhang [Zha03], another member of the CONCEPT research group, as part of the design pattern recovery implementation).

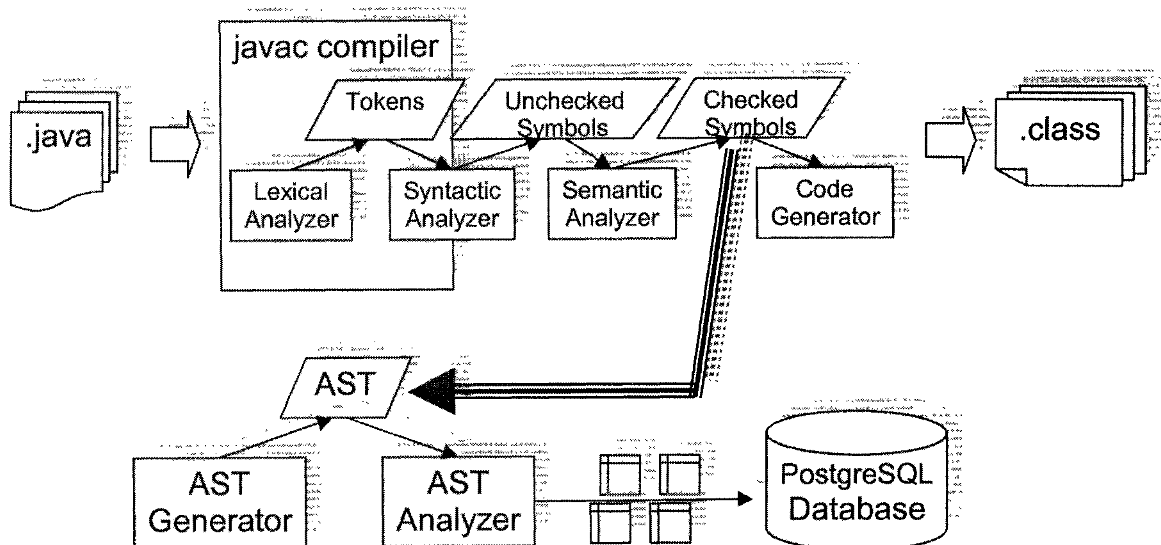


Figure 8 Static source code analysis [Zha03]

6.2.2 PostGreSQL Database

At the beginning of our design, we choose MySQL as the candidate database; however, MySQL does not support multiple queries, which causes many inconvenience when we need some complex queries. Thus, PostGreSQL as a more powerful database becomes

the current data base management system to construct the database of the CONCEPT project. PostgreSQL is an open source data base management system which supports multiple queries and views. Many DBI classes were created by Yonggang Zhang [Zha03], so programmers do not need to know the database design details and can get the interested information by accessing functions in different classes.

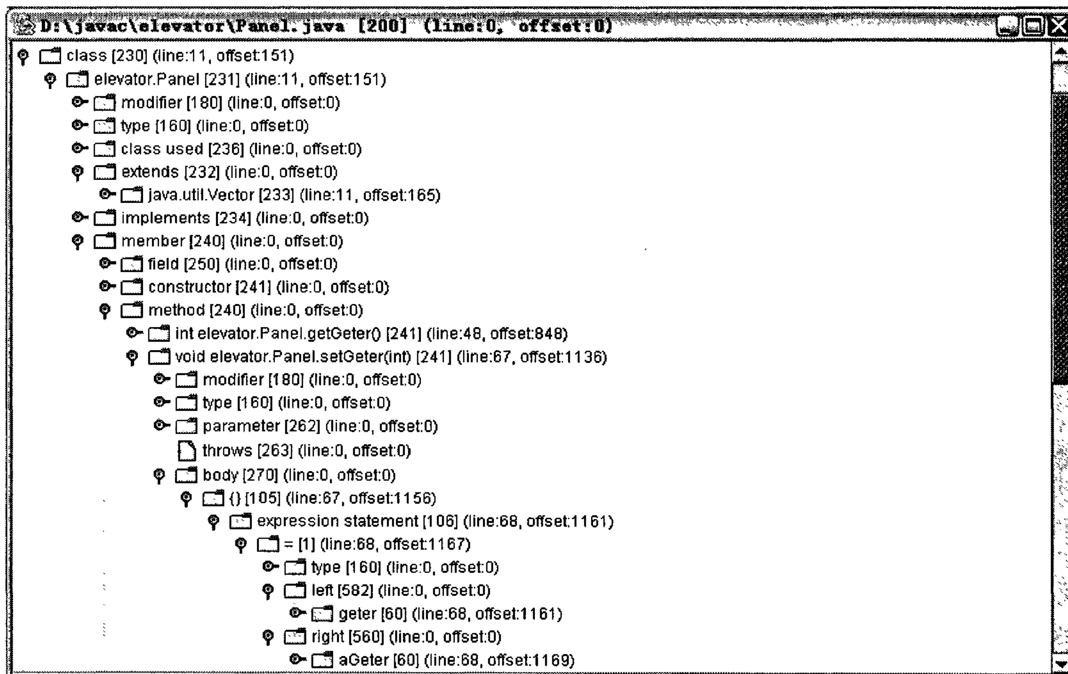


Figure 9 An AST example [Zha03]

In the AST tree, the information above method level are analyzed and stored into the table, and the body of a method, the most complex parts of a java program, is stored into the database as a complex AST node or a sub AST tree. In this way, the analysis result of the method body after parsing will be preserved, which is more flexible for future use since further analysis of the AST tree may lose some information without notice.

6.2.3 Coupling analysis and measurement derivation

Having stored the result of the static source code analysis in the PostGreSQL, it is much straightforward to derive design measurements. The following illustrates the scenario of measurement derivation:

Step 1:

Analyze and collect all references associated information from database

Since the reference information might occur both inside method and outside of the method to a class, we not only need to analyze the information above method level from the database table, we also need to read the AST node object from database and further explore the method body AST to gather the associated reference information.

Step 2:

Build complete coupling information based on the identified references

Coupling information between classes of Java System (such as java.lang.*, java.util.*, etc.) are excluded in our analysis since we focus on the evaluation of the user project. With respect to each identified reference, the source, destination class/package information, and the reference location, reference type, etc. are analyzed for measurement computation.

Step 3:

Compute traditional CBO, MPC and RFC

Step 4:

Compute slicing based SCBO, SMPC and SRFC

Step 5:

Compare and analyze the measurement

6.3 Design details

6.3.1 Reference identification

Identifying all reference information from the source code of the target project is essential for design measurement computation. So far, three major reference types have been identified such as declaration reference, creation reference and usage reference with respect to the life of the object creation. **Declaration references** refer to the potential references existing in class declaration, method declaration and variable or attribute declaration statements, which include "declare extends", "declare implements", "declare field", "declare return type", and "declare argument", "declare throw", "declare local variable". **Creation references** refer to the reference associated to some object creation, which include "new object", "new array" and "direct array create". **Usage references** are the key cause for coupling, which mainly includes method invocation references and field access references. Method invocation references include "call own method", "call super method", "call other method", "call outer class method", and field access references include "read/write other field", "read/write own field", "read/write super field", "read/write interface field", "read/write outer class field", "read field array length". In addition to usage references, some other references such as "read/write local", read "local array length" are also identified for slicing application.

6.3.1.1Declaration reference

Table 4 Declaration reference

Reference Name	The Corresponding Constant/Value	Source entities	Destination entities
declare extends	public static final int DECL_REF_EXTEND = 0	Class/ Interface/ Inner class	Class / interface
declare implements	public static final int DECL_REF_IMPLEMENT = 1	Class/ InnerClass	Interface
declare field	public static final int DECL_REF_FIELD = 2	Attribute	Class
declare return type	public static final int DECL_REF_METHOD_RETURN = 3	Method	Class
declare argument	public static final int DECL_REF_METHOD_ARGUMENT = 4	Method / Constructor	Class
declare throws	public static final int DECL_REF_METHOD_THROW = 5	Method/ Constructor	Class
declare local variable	public static final int DECL_REF_LOCALVAR = 6	Local variable	Class

Note: In later class model, Class ClassDef might be class, inner class and interface; Class MemberDef might be method, constructor, initializer, attribute, local variable, and inner class

- declare extends

```
...
public class UsageReference extends Reference
{
    SourceElement src;

    MemberDef des;
...
```

- declare implements

```
...
import java.util.Hashtable;
import java.io.*;
```

```
public final class Type implements Constants
{
    private static final Hashtable typeHash = new Hashtable(231);
    protected int type_id; // this
...
```

- declare field

```
...
public class UsageReference extends Reference
{
    SourceElement src;
    MemberDef des;
...
```

- declare return type

```
...
public ClassDef getClassDef(String name)
{
    return(ClassDef)classhash.get(name);
}
...
```

- declare argument

```
...
public ClassDef getClassDef(String name)
{
    return(ClassDef)classhash.get(name);
}
...
```

- declare local variable

```
...
public static void main(String[] args) throws IOException {
Panel pp = new Panel(3);
pp.AAF();//read local pp ref;call other mehtod elevator.Panel.AAF()
...
```

- declare throws

```
...
protected synchronized void load() throws Exception
{
        Connection con = null;
        java.sql.Statement stmt = null;
        try{
...
```

6.3.1.2 Creation reference

Table 5 Creation reference

Reference Name	The Corresponding Constant/Value	Source entities	Destination entities
New object	public static final int CREATE_REF_NEW_OBJECT = 10	Local variable/ Attribute	Class
New array	public static final int CREATE_REF_NEW_ARRAY = 11	Local array/ Attribute array	Class
Direct array create	public static final int CREATE_REF_ARRAY = 12	Local array/ Attribute array	Class

- new object //new Object();

```
...
public static void main(String[] args) throws IOException {
Panel pp = new Panel(3) ;
pp.AAF();//read local pp ref;call other mehtod elevator.Panel.AAF()
...
```

- new array //Object[] objs = new Object[2];

```
...
number_of_buttons = num;
private Button[] pbutton = new Button[number_of_buttons] ;
for (int i = 0; i < pbutton.length; i++) {
  pbutton[i] = new Button();
```

...

- direct array create

...

```
Object objs[] = {"abc", "def", "ghi"};
```

...

6.3.1.3 Usage reference

- Method invocation

Table 6 Method invocation reference

Reference Name	The Corresponding Constant/Value	Source entities	Destination entities
call other method	public static final int USE_REF_CALL_OTHER = 20	Method/ Constructor/ Initializer	Method/ Constructor
call own method	public static final int USE_REF_CALL_CURRENT = 21	Method/ Constructor/ Initializer	Method/ Constructor
call super method	public static final int USE_REF_CALL_SUPER = 22	Method/ Constructor/ Initializer	Method/ Constructor
call outer class method	public static final int USE_REF_CALL_OUTER = 23	Method/ Constructor/ Initializer	Method/ Constructor

- ❖ call other method: invoked method defined outside the current class
- ❖ call own method: invoked class is defined inside the current class
- ❖ call super method: invoked method is defined in the super class/interface
- ❖ call outer class method: invoked method is defined in outer class

- Field accessing reference

Table 7 Field accessing reference

Reference Name	The Corresponding Constant/Value	Source entities	Destination entities
read other field	public static final int USE_REF_FIELD_READ_OTHER = 30	Local variable/ Attribute	Attribute of another class
write other field	public static final int USE_REF_FIELD_WRITE_OTHER = 31	Local variable/ Attribute	Attribute of another class
read own field	public static final int USE_REF_FIELD_READ_CURRENT = 32	Local variable/ Attribute	Attribute of own class
write own field	public static final int USE_REF_FIELD_WRITE_CURRENT = 33	Local variable/ Attribute	Attribute of own class
read super field	public static final int USE_REF_FIELD_READ_SUPER = 34	Local variable/ Attribute	Attribute of super class
write super field	public static final int USE_REF_FIELD_WRITE_SUPER = 35	Local variable/ Attribute	Attribute of super class
read interface field	public static final int USE_REF_FIELD_READ_SUPER_INTERFACE = 36	Local variable/ Attribute Method	Attribute of interface
write interface field	public static final int USE_REF_FIELD_WRITE_SUPER_INTERFACE = 37	Local variable/ Attribute	Attribute of interface
read outer class filed	public static final int USE_REF_FIELD_READ_OUTER = 38	Local variable/ Attribute	Attribute of outer class
write outer class filed	public static final int USE_REF_FIELD_WRITE_OUTER = 39	Local variable/ Attribute	Attribute of outer class
field array length	public static final int USE_REF_FIELD_ARRAY_LENGTH = 40	Local variable/ Attribute	Attribute array

- Read/write other field: read/write the attribute of another class
- Read/write own field: read/write the own attribute
- read super field: read/write the attribute of super class
- read interface field: read/write the attribute of interface
- read outer class filed: read/write the attribute of out class with respect to an inner class

- ❖ field array length : read the attribute array's length

- Other references

Table 8 Other references

Reference Name	The Corresponding Constant/Value	Source entities	Destination entities
Read local	public static final int USE_REF_LOCALVAR_READ = 42	Local variable/ Attribute	Local variable
Write local	public static final int USE_REF_LOCALVAR_WRITE = 43	Local variable/ Attribute	Local variable
Local array length	public static final int USE_REF_LOCALVAR_ARRAY_LENGTH = 44	Local variable/ Attribute	Local array

- ❖ read/write local: read/write the local variable such as the variable inside constructor, initializer, and constructor of the class
- ❖ local array length: read local array length

This set usage references are mainly for slicing application. In measurement computation, these are not in our consideration. However, the coupling information associated to this reference is collected for future new measurement derivation.

6.3.1.4 Coupling information derivation

The coupling information for measurement computation is analyzed based on the identified references. The following are the key points that should be considered for the identification of coupling information.

- Identify the source entity and destination entity associated to a reference. Analyzing the AST sub tree of a method body is essential to derive all the high level information

such as method level, class level or package level related to the source entity and destination entity.

- Distinguish whether the client and server classes are part of the current project implementation or belong to the Java System respectively
- Consider whether the server is a class or an array when deriving the coupled server class. If the server is an array, the real coupling exists in the component type of the array.
- Distinguish whether the client class or server class belong to a same class
    - inter coupling: coupling occurs between two classes
    - intro coupling: coupling occurs within one class

6.3.2 Class Diagram

The main classes used for deriving design measurements are illustrated in Figure 10. In the class diagram, CouplingDef and MetricsCompute are two of the most important classes for design measurement computation. All the coupling information is analyzed in the CouplingDef class; traditional measurements and slicing based measurements are computed in the MetricsCompute class.

6.3.3 Deriving traditional CBO, RFC and MPC

In what follows, we use a simple elevator program to illustrate implementation issues of the design measurements. The elevator project includes 4 main classes and 4 inner classes as well. Many other features of Java source code are added into the source code for

testing though it might not be relevant to the elevator simulator. The complete source code of the sample program is available in appendix A.

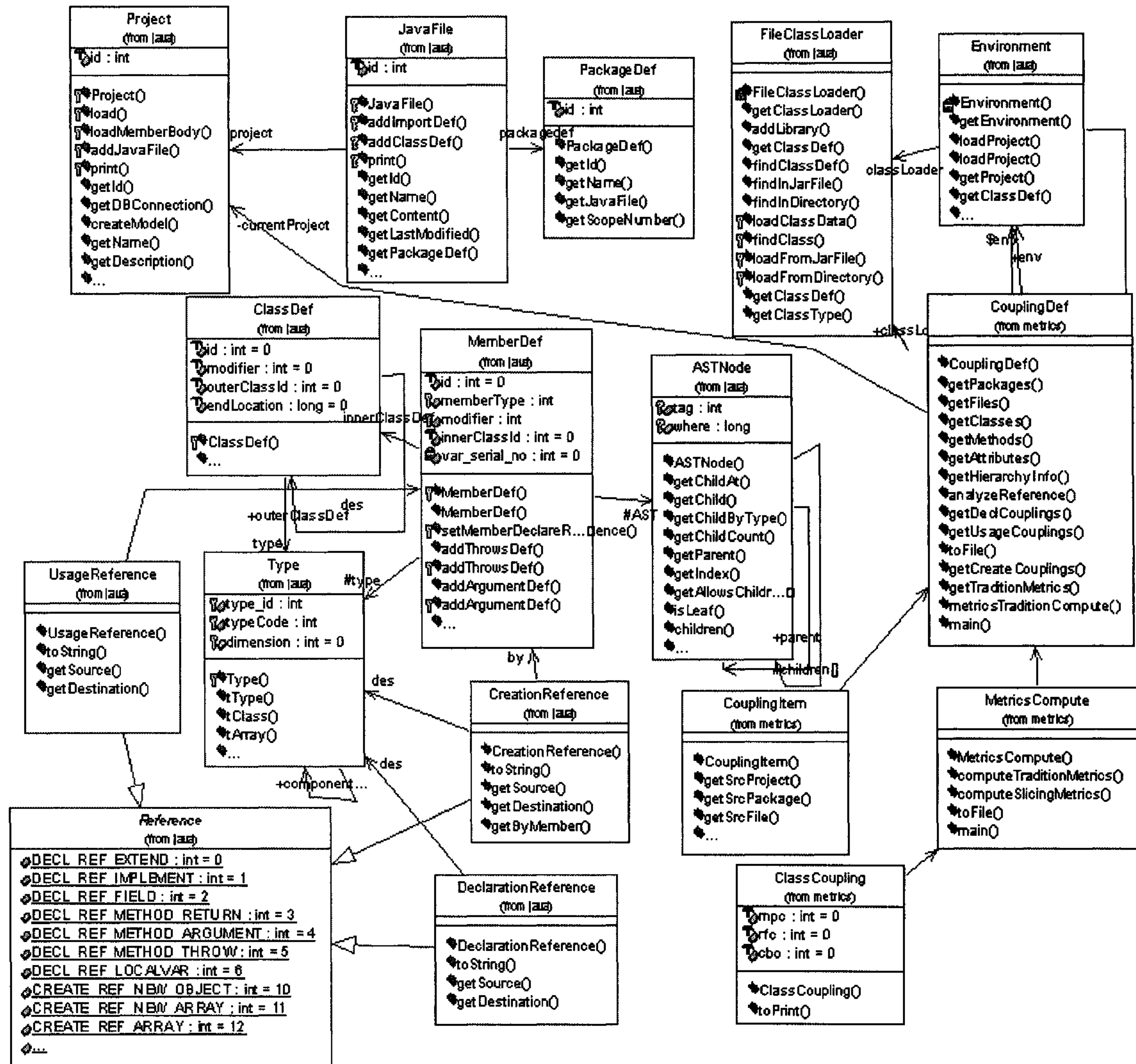


Figure 10 Class Diagram for measurement derivation

To facilitate the validation of the measurements, the coupling details associated to each metric computation are kept. Therefore, the measurement result is identified by the number of the measurements and the associated coupling information, especially the location of the each contributed reference. In this way, the real meaning of measurements are kept, which can help the programmer rapidly locate the interested locus.

- Coupling Between Object Classes(CBO)

CBO relates to the notion that an object is coupled with another object if two objects act upon each other, i.e., methods of one use methods or instance variables of another [Chi94]. Method invocation reference such as "call other/super/out class/ method" and "new object" and "new array" reference are taken into account; moreover, instance variables usage reference such as "read/write other/super/interface/out class field" are taken into account. Table 9 is the result of our MetricsCompute program for CBO computation.

Table 9 CBO from elevator program

Class Name	CBO
elevator.Button	**1**
	elevator.Panel *invoked functions:* int elevator.Panel.howmany(int) at X:\Wenjun\ElevatorComplex\elevator\Button.java L:53,O:1062 void elevator.Panel.<init>(int) at X:\Wenjun\ElevatorComplex\elevator\Button.java L:58,O:1122
elevator.Panel$aInner	**1**
	elevator.Panel$bInner *new array* at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:236,O:4932
elevator.Panel1Local	**0**
elevator.Panel$bInner	**0**
elevator.Panel	**3**
	elevator.Panel1AnotherLocal *invoked functions:* void elevator.Panel1AnotherLocal.<init>() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:167,O:3629

	elevator.Panel$bInner *invoked functions:* void elevator.Panel$bInner.<init>(elevator.Panel) at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:228,O:4798
	elevator.Button *invoked functions*: void elevator.Button.press() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:121,O:2441 boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:96,O:1983 boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:114,O:2287 boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:145,O:3158 void elevator.Button.turnoff() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:75,O:1405 boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:129,O:2689 boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:137,O:2914 void elevator.Button.<init>() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:70,O:1300 *new array* at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:68,O:1191
elevator.Panel1AnotherLocal	**0**
elevator.DoNothing	**0**
elevator.Elevator	**1**

	elevator.Panel *invoked functions:* void elevator.Panel.AAF() at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:114,O:3849 void elevator.Panel.pressButton(int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:90,O:2832 int elevator.Panel.howmany() at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:93,O:2959 boolean elevator.Panel.demandedInThisDirection(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:38,O:954 boolean elevator.Panel.demandedInThisDirection(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:46,O:1207 int elevator.Panel.findMiniumFloor(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:52,O:1394 int elevator.Panel.findMaxiumFloor(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:54,O:1488 void elevator.Panel.buttonOff(int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:59,O:1673 void elevator.Panel.<init>(int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:113,O:3820 void elevator.Panel.<init>(int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:21,O:507

- Message Passing Coupling(MPC)

Message Passing Coupling is the number of sent statements (or method invocations) in the program. "Call other/super/outer class/ method" and "new object/array" are selected for MPC computation. Table 10 is the running result of our MetricsCompute program for MPC computation:

Table 10 MPC from elevator program

Class Name	MPC
elevator.Button	**2**
	int elevator.Panel.howmany(int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Button.java L:53,O:1062 void elevator.Panel.<init>(int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Button.java L:58,O:1122
elevator.Panel$aInner	**1**
	new array of elevator.Panel$bInner at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:236,O:4932
elevator.Panel1Local	**0**
elevator.Panel$bInner	**0**
elevator.Panel	**12**
	void elevator.Button.press() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:121,O:2441 boolean elevator.Button.pressed() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:96,O:1983 boolean elevator.Button.pressed() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:114,O:2287 boolean elevator.Button.pressed() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:145,O:3158 void elevator.Panel1AnotherLocal.bar() of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:168,O:3666 void elevator.Button.turnoff() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:75,O:1405 boolean elevator.Button.pressed() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:129,O:2689 boolean elevator.Button.pressed() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:137,O:2914 void elevator.Button.<init>() of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:70,O:1300 new array of elevator.Button at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:68,O:1191 void elevator.Panel1AnotherLocal.<init>() of elevator.Panel1AnotherLocal at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:167,O:3629 void elevator.Panel$bInner.<init>(elevator.Panel) of elevator.Panel$bInner at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:228,O:4798

elevator.Panel1AnotherLocal	**0**
elevator.DoNothing	**0**
elevator.Elevator	**10**
	void elevator.Panel.AAF() of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:114,O:3849 void elevator.Panel.pressButton(int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:90,O:2832 int elevator.Panel.howmany() of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:93,O:2959 boolean elevator.Panel.demandedInThisDirection(int, int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:38,O:954 boolean elevator.Panel.demandedInThisDirection(int, int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:46,O:1207 int elevator.Panel.findMiniumFloor(int, int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:52,O:1394 int elevator.Panel.findMaxiumFloor(int, int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:54,O:1488 void elevator.Panel.buttonOff(int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:59,O:1673 void elevator.Panel.<init>(int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:113,O:3820 void elevator.Panel.<init>(int) of elevator.Panel at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:21,O:507

Note: As to the "new array and direct create array" reference, only the memory is allocated.

- Response For a Class (RFC)

The Response For a Class measure captures the size of the response set of a class [Fen97]. The response set of a class consists of the set M of methods of the class, and the set of methods invoked by methods in M. According to this definition, "call other method/super/outer class/ method" and "new object/array" as well as member methods such as constructor, method and initializers should be the candidate references for the RFC derivation. Especially, in order to derive all potential method invocations, some

indirect method invocations have also to be included in the measurement computation. However, in our design, the components in the response set are unique. That is, if some method of other classes is called several times, we only add the method to the response set one time.

For example,

```
public void move() {
...
if (direction == UP) {
        if (current_floor == top_floor)
           direction = DOWN;
        else
           if (!panel.demandedInThisDirection(top_floor, current_floor))
              direction = DOWN;
     }

     if (direction == DOWN) {
        if (current_floor == 1)
           direction = UP;
        else
           if (!panel.demandedInThisDirection(current_floor, 1))
              direction = UP;
     }
...
}
```

In the above "move()" method of the class "Elevator", method of Panel "panel.demandedInThisDirection(...)" has been called twice; however, we only count one time for the RFC response set computation.

Table 11 is the running result of the MetricsCompute program for RFC metrics computation of the sample "elevator" program.

Table 11 RFC from elevator program

Class Name	RFC
elevator.Panel	**30**
	methods: void elevator.Panel.pressButton(int) void elevator.Panel.setGeter(int) int elevator.Panel.howmany(int) int elevator.Panel.howmany() boolean elevator.Panel.on(int) void elevator.Panel.AAC() boolean elevator.Panel.demandedInThisDirection(int, int) void elevator.Panel.AAA() void elevator.Panel.buttonOff(int) void elevator.Panel.AAD(int, int, int, int, int, int, int, int, int, int, int) int elevator.Panel.findMiniumFloor(int, int) int elevator.Panel.findMaxiumFloor(int, int) int elevator.Panel.getGeter() void elevator.Panel.AAF() void elevator.Panel.AAB() *Constructor:* void elevator.Panel.<init>(int) *Initializer:* void elevator.Panel.<clinit1>() void elevator.Panel.<clinit11>() void elevator.Panel.<clinit4>() void elevator.Panel.<clinit2>() void elevator.Panel.<clinit3>() *Invoked methods:* void elevator.Button.press() of elevator.Button boolean elevator.Button.pressed() of elevator.Button void elevator.Panel1AnotherLocal.bar() of elevator.Panel void elevator.Button.turnoff() of elevator.Button void elevator.Button.<init>() of elevator.Button void elevator.Panel1AnotherLocal.<init>() of elevator.Panel1AnotherLocal void elevator.Panel$bInner.<init>(elevator.Panel) of elevator.Panel$bInner void elevator.Button.press(java.lang.Object) of elevator.Button void elevator.Button.press(int) of elevator.Button elevator.Panel elevator.Button.Test(int) of elevator.Button int elevator.Panel.howmany(int) of elevator.Panel void elevator.Panel.<init>(int) of elevator.Panel
elevator.Panel$aInner	**2**
	methods: void elevator.Panel$aInner.kk() Constructor: void elevator.Panel$aInner.<init>(elevator.Panel)
elevator.Panel1Local	**1**

<table>
<tr><td></td><td>Constructor:void elevator.Panel1Local.<init>()</td></tr>
<tr><td rowspan="2">elevator.Panel$bInner</td><td>1</td></tr>
<tr><td>Constructor:void elevator.Panel$bInner.<init>(elevator.Panel)</td></tr>
<tr><td rowspan="2">elevator.Elevator</td><td>25</td></tr>
<tr><td>Methods:
void elevator.Elevator.main(java.lang.String[])
void elevator.Elevator.prompt()
void elevator.Elevator.openDoor()
void elevator.Elevator.move()
void elevator.Elevator.closedoor()

Constructor:
void elevator.Elevator.<init>(int)

Invoked functions:
void elevator.Panel.AAF() of elevator.Panel
void elevator.Panel.pressButton(int) of elevator.Panel
int elevator.Panel.howmany() of elevator.Panel
boolean elevator.Panel.demandedInThisDirection(int, int) of elevator.Panel
int elevator.Panel.findMiniumFloor(int, int) of elevator.Panel
int elevator.Panel.findMaxiumFloor(int, int) of elevator.Panel
void elevator.Panel.buttonOff(int) of elevator.Panel
void elevator.Panel.<init>(int) of elevator.Panel
void elevator.Panel$bInner.<init>(elevator.Panel) of elevator.Panel$bInner
void elevator.Button.press() of elevator.Button
boolean elevator.Button.pressed() of elevator.Button
void elevator.Panel1AnotherLocal.bar() of elevator.Panel
void elevator.Panel1AnotherLocal.<init>() of elevator.Panel1AnotherLocal
void elevator.Button.turnoff() of elevator.Button
void elevator.Button.<init>() of elevator.Button
void elevator.Button.press(java.lang.Object) of elevator.Button
void elevator.Button.press(int) of elevator.Button
elevator.Panel elevator.Button.Test(int) of elevator.Button
int elevator.Panel.howmany(int) of elevator.Panel</td></tr>
<tr><td rowspan="2">elevator.Button</td><td>9</td></tr>
<tr><td>methods: boolean elevator.Button.pressed()
void elevator.Button.press()
elevator.Panel elevator.Button.Test(int)
void elevator.Button.turnoff()
void elevator.Button.press(int)
void elevator.Button.press(java.lang.Object)
Constructor: void elevator.Button.<init>()
Invoked methods:
int elevator.Panel.howmany(int) at elevator.Panel
void elevator.Panel.<init>(int) at elevator.Panel
void elevator.Button.<init>() at elevator.Button</td></tr>
<tr><td>elevator.Panel1AnotherLocal</td><td>2</td></tr>
</table>

	Method: void elevator.Panel1AnotherLocal.bar() *Constructor:* void elevator.Panel1AnotherLocal.<init>() *Initializer:* void elevator.Panel1AnotherLocal.<clinit1>() void elevator.Panel1AnotherLocal.<clinit2>()
elevator.DoNothing	**1**
	Constructor: void elevator.DoNothing.<init>()

6.3.4 Deriving slicing based SCBO, SRFC, and SMPC

In order to compute SCBO, SRFC and SMPC for a given slice, we will emphasize the following aspects that are different from the traditional CBO, RFC and MPC computation.

- The coupling analysis is limited to source code in the slice. Methods, classes and packages involved in the slice will be identified and considered.
- All direct and indirect coupling will be included in the SCBO, SRFC and SMPC computation.
- Both import and export coupling within the slice will be computed.

The given slice *S<current_floor, Elevator L59>* of our sample elevator project is as follows: {*Elevator L59, Elevator L54, Elevator L53, Elevator L52, Elevator L51, Elevator L24, Elevator L20, Elevator L10, Elevator L9, Elevator L117, Elevator L116, Panel L163, Panel L135, Panel L132, Panel L131, Panel L130, Panel L129, Panel 128, Panel 127, Button L17, Button L16, Button L11, Button L10*}. Table 12, Table 13 and Table 14 are the running result of the SCBO, SRFC and SMPC measurements for the given slice S<current_floor, Elevator L59>.

- Slicing based Coupling between Object Classes (SCBO)

Table 12 SCBO for S<current_floor, Elevator L59>

Classes in the slice	SCBO
elevator.Elevator	**1** **elevator.Panel** void elevator.Panel.buttonOff(int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:59, O:1673 int elevator.Panel.findMaxiumFloor(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:54, O:1488 int elevator.Panel.findMiniumFloor(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:52, O:1394
elevator.Panel	**1** **elevator.Button** boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:137, O:2914 boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:129, O:2689
elevator.Button	**0**

- Slicing based Response for a Class (SRFC)

Table 13 SRFC for S<current_floor, Elevator L59>

Classes in the slice	SRFC
elevator.Elevator	**6** member functions: void elevator.Elevator.move() void elevator.Elevator.<init>(int) void elevator.Elevator.main(java.lang.String[]) invoked methods: void elevator.Panel.buttonOff(int) of elevator.Panel int elevator.Panel.findMaxiumFloor(int, int) of elevator.Panel int elevator.Panel.findMiniumFloor(int, int) of elevator.Panel
elevator.Panel	**3** member functions: int elevator.Panel.findMaxiumFloor(int, int) int elevator.Panel.findMiniumFloor(int, int) invoked method: boolean elevator.Button.pressed() of elevator.Button
elevator.Button	**1** member functions: void elevator.Button.<init>()

- Slicing based Message Passing Coupling (SMPC)

Table 14 SMPC for S<current_floor, Elevator L59>

Classes in the slice	SMPC
elevator.Elevator	**3**
	void elevator.Panel.buttonOff(int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:59,O:1673
	int elevator.Panel.findMaxiumFloor(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:54,O:1488
	int elevator.Panel.findMiniumFloor(int, int) at X:\Wenjun\ElevatorComplex\elevator\Elevator.java L:52,O:1394
elevator.Panel	**2**
	boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:137,O:2914
	boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:129,O:2689
elevator.Button	**0**

6.3.5 Comparison and analysis

In order to validate whether the presented measurements are correctly computed, all the relevant coupling information of CBO, RFC and MPC are kept in vector *vCBO, vRFC and vMPC respectively.* Comparing the derived data in Table 15 with in Table 16, the results of Sun ONE measurement tool are different to the result from our design. For example, CBO of elevator.Panel is 3 in our system, but 9 in Sun ONE measurement tool. There are two reasons that cause the difference in the measure. (1) In Sun ONE, all the Java system classes are taken into account; however, in our system, only the classes in the project are in consideration. (2) Moreover, in Sun ONE, both import coupling and export coupling are taken into account, but in our system, only the import coupling is considered, strictly according to the original definition of CBO [Chi94]. The classes that

elevator.Panel refers to are: ***elevator.Button***, *java.lang.String*, *java.lang.System*, *java.util.Vector*, ***elevator.Panel.1.AnotherLocal***, ***elevator.Panel.bInner*** *and* *java.io.Serializable*, and the classes refer elevator.Panel are *elevator.Elevator* and *elevator.Panel.aInner*. In our system, the classes that elevator.Panel refers to are: ***elevator.Button***, ***elevator.Panel1AnotherLocal***, ***elevator.Panel$bInner***. Excluding the 5 java system classes and 2 export coupling classes in Sun ONE measurement tool, the value 9-5-2=3, which is same as our measurement CBO for elevator.Panel.

Similar as CBO, the difference of RFC of Sun ONE measurement tool and our system are due to (1) as discussed previously, and indirect coupling as well. In Sun ONE, only the direct coupling is computed, without considering the indirect coupling. However, in our approach we consider both direct and indirect coupling according to the original FRC definition in [Chi94]. MPC are also different since they compute MPC for both inter coupling and intra coupling; however, in our system, we only compute the inter coupling.

Table 15 CBO, RFC and MPC in our system

Class Name	CBO	RFC	MPC
elevator.Panel.1.Local	0	1	0
elevator.Panel.aInner	1	2	1
elevator.Panel	3	30	12
elevator.Panel.1.AnotherLocal	0	2	0
elevator.Button	1	9	2
elevator.Elevator	1	25	10
Elevator.Panel.bInner	0	1	0
Elevator.DoNothing	0	1	0

Table 16 CBO, RFC and MPC in Sun ONE Measurement Tool

Metrics

Class Name	CBO	RFC	MPC
elevator.Panel.1.Local	1	2	1
elevator.Panel.aInner	2	3	1
elevator.Panel	9	34	26
elevator.Panel.1.AnotherLocal	2	3	1
elevator.Button	4	11	7
elevator.Elevator	7	25	50
elevator.Panel.bInner	2	2	1
elevator.DoNothing	1	2	1

Within our knowledge, refining existing measurements (CBO, RFC and MPC) through slicing based information are a novel approach to refine and enhance the measurements. We do not have any existing slicing based coupling s tool to compare our result with. At the current stage of the CONCEPT project, due to the lack of slice information, many interesting test and validation can not be performed. However, the ideas to implement the whole slicing based coupling measurements are feasible since all the traditional measurements are correctly computed. Further work is required to use the slicing based information as input to identify all the relevant references with respect to a particular slice.

Table 17 SCBO, SRFC and SMPC for S<current_floor, Elevator L59>

Class Name	SCBO	SRFC	SMPC
elevator.Panel.1.Local	×	×	×
elevator.Panel.aInner	×	×	×
elevator.Panel	1	3	2
elevator.Panel.1.AnotherLocal	×	×	×
elevator.Button	0	1	0
elevator.Elevator	1	6	3
Elevator.Panel.bInner	×	×	×
Elevator.DoNothing	×	×	×

Note: "×" means "not included in the slice"

Table 17 illustrates the results of refining traditional measurements based on a slice S<*current_floor,Elevator L59* >, which is manually constructed by backward slicing the

elevator program. As the results indicate, only three classes are involved in this slice. SCBO for elevator.Panel is 1 while CBO for elevator.Panel is 3. SCBO of elevator.Panel associated coupling is with class elevator.Button by boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:137 and by boolean elevator.Button.pressed() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:129, CBO of elevator.Panel related coupling is with class elevator.Button, elevator.Panel1AnotherLocal, elevator.Panel$bInner. It refers to elevator.Button by 9 references (see table 9), elevator.Panel1AnotherLocal by void elevator.Panel1AnotherLocal.<init>() at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:167, and elevator.Panel$bInner by void elevator.Panel$bInner.<init>(elevator.Panel) at X:\Wenjun\ElevatorComplex\elevator\Panel.java L:228.

6.3.6 Future work

So far, we have finished the implementation of CBO, RFC and MPC, and SCBO, SRFC and SMPC based on the static source code analysis and some manually created slices. The implementation of design measurement is closely related to static source code analysis and program slicing. If source code analysis and program slicing algorithm are not ready, it is impossible to completely derive the measurements of interest. Currently, the source code analysis is very successful since it provides all the details that we need for coupling analysis. However, the slicing algorithms developed such as backward, forward, and hybrid slicing algorithms are just in the process integrating into our PostGreSQL database. By far, we lack slice information to test and validate the slicing

based coupling measurement framework. However, we are confident in applying the measurement framework since traditional coupling measurements have been tested in some real systems such as java.util package and concept.java package 1.0(see appendix B). The java.util package is one of the core package of Java Development Kit, which contains 120 files, 51,993 lines of code, and 23,567 lines of code without comments [Zha03]. The concept.java package contains 196 files, 14,768 lines of code, and 11,658 lines of code without comments [Zha03]. The presented slicing based coupling framework can be seen as a refinement of the traditional CBO, RFC and MPC, which only requires analyzing the slice associated source code. Therefore, once the slicing algorithms can connect to PostGreSQL database and provide real slice information, a meaningful empirical study will be performed, and the usability of the measurements can be validated.

Future work will focus on comparing and analyzing the obtained measurements for large systems, and validating and refining the existing measurements to indicate design quality. Moreover, another direction might apply the current slicing based measurements on dynamic execution or dynamic slices. Thus, the real behavior of the program will be analyzed and quantified. In fact, more work can be put on software measurement associated information visualization, since a good visualization technique to reflect these measurements can identify the "hot spots" and speed up the comprehending and maintaining process.

7 CONCLUSION

The presented research has introduced a slicing-based coupling measurement of OO program. These measurements are extensions of already well-known and proven coupling measures, namely CBO, RFC, and MPC. In the context of this research, we refine these measurements by combining them with program slicing which resulted in our slicing based SCBO, SRFC and SMPC measures. These measurements provide more focused views on different slices and additional insights on the coupling. Especially, the inclusion of direct and indirect coupling, as well as import and export coupling, explore a new perspective on coupling. In addition, a slicing hierarchy is introduced that provides different slicing granularities on which these measurements can be applied. Some applications of these measurements in theory were discussed, namely in testing, debugging, preventive maintenance and change impact analysis. The presented measurements are partially implemented within our CONCEPT prototype framework, which is currently being used to conduct a first set of preliminary experimental analysis. Future research efforts will focus on a detailed empirical research to validate the applicability or usability of these measurements. It is anticipated that as part of the empirical studies, some additional useful slicing based measurements can be derived and the existing generic coupling measurements will be refined.

BIBLIOGRAPHY

[Abr95] F. Abreu, M. Goulão, R. Esteves, "Toward the Design Quality Evaluation of Object-Oriented Software Systems", 5th International Conference on Software Quality, Austin, Texas, USA, October 1995.

[Bas95] V. Basili, L. Briand, W. Melo, "Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented systems", Technical Report, University of Maryland, Department of Computer Science, CS-TR-3395, 1995

[Bas96] V.R. Basili, L.C. Briand, W.L. Melow, "A Validation of Object-Oriented Design Metrics as Quality Indicators",IEEE Transactions on Software Engineering, 22 (10), pp.751-761, 1996.

[Bie95] M. Bieman and B. Kang, "Cohesion and reuse in an Object-Oriented paradigm", Proc. ACM Symposium on Software Reusability (SSR-95), pp. 259-262, 1995

[Bri97] L. Briand, P. Devanbu, W. Melo, "An Investigation into Coupling Measures for C++", Technical Report ISERN 96-08, IEEE ICSE '97, Boston, USA, May 1997.

[Bri93] L. Briand, S. Morasca, V. Basili, "Measuring and Assessing Maintainability at the End of High-Level Design", IEEE Conference on Software Maintenance, Montreal, Canada, September 1993.

[Chi91] S.R.Chidamber, and C.F. Kemerer, "Toward a Metric Suite for Object-Oriented Design", Proceedings of 6th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications(OOPLSLA), phoenix, AZ, pp.197-211, 1991

[Chi94] S.R.Chindamber, and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," IEEE Transactions on Software Engineering, 20(6), pp.476-98

[Agr93] H. Agrawal, R. DeMillo, and E. Spafford, "Debugging with dynamic slicing and backtracking", Software – Practice and Experience, 23(6), pp. 589-616, 1993

[Lon85] H.D. Longworth, "Slice-based program metrics", Master's thesis, Michigan Technological University, 1985

[Ede94] J. Eder, G. Kappel, and M. Schrefl, "Coupling and Cohesion in OO Systems", Tech. Report, Univ. of Klagenfurt, 1994

[Lyl86] J. Lyle, and M. Weiser, "Experiments on slicing-based debugging tools", Proceedings of the 1st Conference on Empirical Studies of Programming, pp. 187-197, 1986.

[Fen97] E. Fenton, S. L. Pfleeger, "Software Metrics - A rigorous & Practical Approach", International Thomson Computer Press, London, 1997

[Hart95] J.M. Hart, "Experinece with Logical code analysis in software reuse and reengineering", In AIAA computing in Aeorspace, 10, pp.1243-1262, San Antonio, Tx, 1995

[Har97] M. Harman, M. Okulawon, and B. Sivagurunathan, S. Danicic, "Slice-Based Measurement of Coupling", IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution (PMESSE'97), Boston, Massachusetts, pp.28-32., 1997

[Hen96] B. Henderson-Sellers, “Software Metrics”, Prentice Hall, Hemel Hempstead, 1996

[Hit95] M. Hitz, and B. Montazeri, “Measuring Coupling and Cohesion in Object-Oriented Systems”, Proc. Of the International Symposium of Applied Corporate Computing (SACC 95), 1995

[Hit96] M. Hitz, B. Montazeri, “Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective”, IEEE Transactions on Software Engineering, 22 (4), 276-270, 1996

[Kor88] B. Korel and J. Laski, “Dynamic program slicing”, In. Proc. Letters, 29(3), pp. 155-163, Oct. 1988

[Wei82] M. Weiser, “Programmers use slices when debugging”, Communications of the ACM, 25, pp.446-452, 1982

[Li93] W. Li and S. Henry, “Object-Oriented metrics that predict maintainability”, Journal of systems and software, 23(2), pp. 111-122, 1993

[Tip95] F. Tip, “A survey of program slicing techniques”, Journal of Program Languages, 3(3), pp. 121-189, 9/1995

[Ott89] L. Ott and J. Thuss, “The relationship between slices and module cohesion”, Proc. 11th International Conference on Software Engineering, pp. 192-204, 1989

[Ott93] L. Ott and J. Thuss, “Slice based metrics for estimating cohesion”, Proc. IEEE-CS International Software Metrics Symposium, pp. 71-81, 1993

[Pag80] M. Page-Jones, “Practical Guide to Structured System Design”. Prentice Hall, 1980

[Sys01] T. Systa, K. Koskimies, and H. Müller, "Shimba - An Environment for Reverse Engineering Java Software Systems," Software Practice & Experience, Vol 31, No 4, pp. 371-394, April 2001

[Ril01] J. Rilling, “Maximizing Functional Cohesion of Comprehension Environments by Integrating User and Task Knowledge”, 8th IEEE Working Conference on Reverse Eng., Stuttgart, Germany, pp. 157-165, 2001

[Ste74] W. Stevens, G.J. Myers, and L.L. Constantine, “Structured design”, IBM Systems Journal 13(2), pp.115-- 139, 1974

[You79] E. Yourdon and L.L. Constantine, “Structured Design”, Prentice-Hall, Englewood, New Jersey, 1979

[Wei81] M. Weiser, “Program Slicing”, Proceedings of the 5th international conference on Software engineering, pp.439-449, March 1981

[Ril02] J. Rilling, H. F. Li, and D. Goswami, “Predicate-Based Dynamic slicing of Message Passing Programs”, Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02), Montreal, Canada, 2002

[Luc01] A. D. Lucia, “Program slicing: Methods and applications”, 1st IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, USA, pp. 142-149,2001

[Kor98] B. Korel and J. Rilling, “Program Slicing in Understanding of Large Programs”, IEEE Proceedings of the 6th IWPC ‘98, pp. 145-152, Ischia, Italy, June 1998

[Gla00] D. Glasberg, K. El Emam, W. Melo, and N. Madhavji, "Validating Object-Oriented Design Metrics on a Commercial Java Application", Technical Report, National Research Council of Canada, NRC/ERB-1080, 2000

[Ema01]K. E. Emam, "Object-Oriented Metrics: A Review of Theory and Practice", Technical Report, National Research Council of Canada, NRC/ERB-1085, 2001

[Li01] B. Li, "A Hierarchical Slice-Based Framework for Object-Oriented Coupling Measurement", TUCS Technical Reports, Turku Centre for Computer Science, NO415, 2001

[Ott98] L. Ott and J. Bieman, "Program slices as an abstraction for cohesion measurement", Journal of Information and Software technology, 40(1112), pp.691-699, November 1998

[Bri96] L. Briand, W.Daly John, and J. Wust, "A unified Framework for coupling measurement in Object-Oriented systems", Fraunhofer Institue for Experimental software engineering, Kaiserslautern Germany, Isern-96-14,1996

[Wan96] Y. Wang, W-T. Tsai, X. Chen, S. Rayadurgen, "The Role of Program Slicing in Ripple Effect Analysis", SEKE, pp369-376, 1996

[Mul00] H. A. Muller, J. H. Jahnke, D. B. Smith, M-A. Storey, S. R. Tilley, and K. Wong, "Reverse Engineering: A Roadmap", The Future of Software Engineering, Anthony Finkelstein, ACM Press, 2000

[Har02] M. Harman, N. Gold, R. Hierons and D. Binkley, "Code Extraction Algorithms Which Unify Slicing and Concept Assignment", 9th Working Conference on Reverse Engineering (WCRE'02), 2002

[Gup92] R. Gupta, M. Harrold, and M. Soffa, "An approach to regression testing using slicing", In Proceedings of the Conf. on Software Maintenance, pp. 299-306, 1992

[Gop91] R. Gopal, "Dynamic program slicing based on dependence relations", In Proceedings of the Conference on Software Maintenance, pp. 191-200, 1991

[Kor94] B. Korel and S. Yalamanchili, "Forward Derivation of Dynamic Slices", Proceedings of the Intern. Symposium on Software Testing and Analysis, pp. 66-79, Seattle, 1994

[Har95] M. Harman, S. Danicic, and Y. Sivagurunathan, "Program comprehension assisted by slicing and transformation", Procedding of Ist UK Program Comprehension Workshop, Durham, UK, 1995

[Hor90] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs", ACM Transactions on Progr. Languages and Systems, 12(1), pp.26--60, January 1990

[Luc03] A. D. Lucia, M. Harman, R. Hierons, and J. Krinke, "Unions of Slices are not Slices", the proceedings of the 7th European Conference on Software Maintenance and Reengineering, 2003 in Benevento, Italy, 2003

[Sys99] T. Systa, and P. Yu, "Using OO Metrics and Rigi to Evaluate Java Software", Report of Department of Computer Science University of Tampere, 1999
citeseer.nj.nec.com/391880.html

[Bar99] H. Bar, M. Bauer, O. Ciupke, and S. Demeyer, "The FAMOOS Object-Oriented Reengineering Handbook", 1999
http://www.iam.unibe.ch/_famoos/handbook/

[Won98] K. Wong, "The Rigi User's Manual - Version 5.4.4", June 30, 1998
http://www.rigi.csc.uvic.ca/Pages/publications.html

[Har01] M. Harman, and R. M. Hierons, "An overview of program slicing", Software Focus 2, 3 (2001), pp85-92, 2001

[Lit99]T. Littlefair, "An investigation into the use of software code metrics in the industrial software development environment", PhD thesis, Edith Cowan University, 1999
http://www.fste.ac.cowan.edu.au/~tlittlef/LittlefairPhDThesis.pdf

[IEE93] "IEEE Software Engineering Standards", Standard 610.12-1990, pp.47-48, 1993

[Pre01] R. S. Pressman, "Software Engineering: A Practitioner's Approach", Fifth Edition, McGraw Hill, 2001

[Zha03] Y. Zhang, "Automatic design pattern recovery", Master thesis, Concordia University, 2003

[Mey98] B. Meyer, "The role of Object-Oriented metrics", in Computer (IEEE), vol. 31, no. 11, pages 123-125, November 1998

[Har95] M.Harman, S.Danicic, B.Sivagurunathan, B.Jones and Y.Sivagurunathan, "Cohesion Metrics", 8th International Software Quality Week (QW'95), San Francisco CA, May 30th - June 2nd. 1995, paper 4-T-4. S, 1995

[Ott93] L. M.Ott and J. J. Thuss, "Slice based metrics for estimating cohesion", In Proceedings of the IEEE-CS-International Metrics Symposium, page 71-81, Baltimore, Maryland, USA, May 1993, IEEE Computer Press Society Press, Los Alamitos, California, USA, 1993

[Ema99] K. El Emam, S. Beniarbi, N. Goel, and S. Rai, "A Validation of Object-Oriented Metrics", Technical Report, National Research Council of Canada, NRC/ERB-1063, October 1999

[Lee98] M. L. Lee, "Change Impact Analysis of Object-Oriented Software", PhD thesis, George Mason University, 1998

[Orm02] O. Ormandjieva, "Deriving new measurements for real-time reactive systems", PhD thesis, Concordia University, 2002

APPENDIX A ELEVATOR PROGRAM

```
//**************
//Eelevator Class
//**************

package elevator;

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

public class Elevator {
   //data member
   private boolean OPEN = true, CLOSED = false; //Door
   private int current_floor, next_floor;
   final int top_floor;
   Panel panel;
   public String EEEEEEEEEEEEEEEEEEEEEEEEEEEEE;


   private final boolean UP = true, DOWN = false; //Direction
   private boolean door, direction;

   //constructor
   public Elevator(int num) {
      top_floor = num;
      panel = new Panel(num);
      door = OPEN;
      direction = UP;
      current_floor = 1;
   }

   public void closedoor() {
      door = CLOSED;
      System.out.println("Closing the door!");
   }

   public void move() {
      //find Direction
      if (direction == UP) {
         if (current_floor == top_floor)
            direction = DOWN;
         else
            if (!panel.demandedInThisDirection(top_floor, current_floor))
               direction = DOWN;
      }

      if (direction == DOWN) {
         if (current_floor == 1)
            direction = UP;
         else
            if (!panel.demandedInThisDirection(current_floor, 1))
```

```
            direction = UP;
    }

    //find The outgoing floor
    if (direction == UP) {
      current_floor = panel.findMiniumFloor(top_floor, current_floor);
    } else
      current_floor = panel.findMaxiumFloor(current_floor, 1);

    System.out.println("Now moving to floor: " + current_floor);
    System.out.println("Arrived at floor: " + current_floor);

    panel.buttonOff(current_floor);
    openDoor();
  }

  public void openDoor() {
    door = OPEN;
    System.out.println("Door is open!");
    System.out.println();
  }

  //this elevator simulates one use case :
  //only when elevator stops, the pressed buttons are to be responded
  //by equiped button panel.
  public void prompt() throws IOException {
    int num = 0;
    String strnum;

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.println(
      "Enter the floor you want to go: (between 1 and " + top_floor + ")");
    while ((strnum = in.readLine()) != null) {
      num = Integer.parseInt(strnum);

      if (num == current_floor)
        System.out.println("We are at the current floor!");
      else
        if (num == 13) //there is no 13 floor
          System.out.println(
            "Sorry, this floor is not existed. Please press a button again!");
        else
          if (num > 0 && num <= top_floor) {
            panel.pressButton(num);
          } else
            if (num == 0) { //go
              if (panel.howmany() == 0)
                System.out.println(
                  "Nobody demande, evelator is stopping, please press a button before closing the
door!");
              else {
                closedoor();
                move();
              }
            } else
              if (num == -1)
```

```
                    break;
                  else
                    System.out.println("Invalid input, repeat again!");
          System.out.println(
            "Enter the floor you want to go: (between 1 and " + top_floor + ")");
        } //end of while

        System.out.println("Good-bye!");
      }

      public static void main(String[] args) throws IOException {
        Panel pp = new Panel(3);
        pp.AAF();//read local pp ref;call other mehtod elevator.Panel.AAF()

        final int number = 15; //constant
        Elevator Otis = new Elevator(number);

        System.out.println("This is a simulator of an elevator.");
        System.out.println("0: close the door and go -1: terminate the program");
        System.out.println("Start the program now!");
        System.out.println();
        Otis.prompt();
      }
    }

    //**************
    //Panel Class
    //**************

    package elevator;
    import java.util.Vector;
    import java.io.*;

    public class Panel extends Vector implements java.io.Serializable{

      public String PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP;
      String s[][][][] = new String[2][2][2][2];



      static {
        //let us do sth. static
      }
      {
                    String s = "a";
        int i = 0;
        i = 2;
        // what can I do here?
      }
      {
        int i = 4;
        i = 5;

      }
      static {
        //can i have two static init?
```

```
}

private int number_of_buttons;
private Button[] pbutton;
Vector v = new Vector();

private int geter = 0;


        /**
        *
        * @return the int value of geter.
        */
        public int getGeter(){
   String s = null;
   s = null;
   int k = 0;
   return k;
}

        /**
        *
        * @param aGeter - the new value for geter
        */
        public void setGeter(int aGeter){
                geter = aGeter;
   String s = "";
   System.out.println(Integer.toString(geter).toString().toUpperCase());
        }



public Panel(int num) {
   number_of_buttons = num;
   pbutton = new Button[number_of_buttons];
   for (int i = 0; i < pbutton.length; i++) {
      pbutton[i] = new Button();
   }
}

public void buttonOff(int num) {
   pbutton[num - 1].turnoff();
   this.v.size();
   String[] aa = {"1","3","5"};
   String b = null;
   b = "a" + "b";
   b = aa[1];
   final int x, y;
   int k = aa.length;
   k = aa.length;
   PrintStream o = null;
   o = System.out;
   Vector c;
   c = v;
   b = new String("s");
   b = (String)v.elementAt(0);
```

```
    if((String)b instanceof String){};
}

public int howmany() {
    int count = 0;
    for (int i = 0, l = 4; i <= number_of_buttons - 1; i++) {
        if (pbutton[i].pressed()) {
            count++;
        }
    }
    return number_of_buttons;
}

public int howmany(int i){

    final int a;
    int b;
    a = 3;
    b = 0;
        return 2;
}

public boolean on(int num) {
    if (pbutton[num - 1].pressed())
        return true;
    else
        return false;
}

public void    pressButton(int num) {
    pbutton[num - 1].press();

    return;
    //System.out.println("the NO."+num+" is pressed!");
}

public int findMiniumFloor(int upFloor, int downFloor) {
    for (int i = downFloor; i <= upFloor; i++) {
        if (pbutton[i - 1].pressed())
            return i;
    }
    return 0;
}

public int findMaxiumFloor(int upFloor, int downFloor) {
    for (int i = upFloor; i >= downFloor; i--) {
        if (pbutton[i - 1].pressed())
            return i;
    }
    return 0;
}

public boolean demandedInThisDirection(int upFloor, int downFloor) {
    for (int i = downFloor, j = 0; i <= upFloor; i++) {
        if (pbutton[i - 1].pressed()){
         int a = 0;
```

```
        a++;
        {
          a = 0;
          a--;
        }
        return true;
      }
    }
    {
      class AnotherLocal {
        void bar() {
          class Local {}; // ok
        }
        {
          //initializer1
        }
        {
          //initializer2
        }
      }
      AnotherLocal b = new AnotherLocal();;
      b.bar();
    }

    switch(upFloor){
      case 1:
        System.out.println(1);
        break;
      case 2:
        return true;
      default:
        System.out.println("unknow");
    }

    try{
      int k;
      k = 0;
    }catch(java.lang.Exception e){
      e.printStackTrace();
    }
    finally{
      int k = 1;
    }
    return false;
  }

  public void AAA(){
    int i = 0;
    int j = 0;
    super.clear();

  }

  public void AAB(){
    int i = 0;
    int j = 0;
```

```
    }

    public void AAC(){
        int i = 0;
        int k = 1;
        {
            boolean j = true;
        }
        {
            boolean j = true;
            if(j){}
            java.lang.String s = Integer.toString(1);
        }
        boolean j = false;
        if(j){}
        AAB();
    }

    public void AAD(int a, int b, int c, int d, int e, int f, int g, int h, int i, int k, int l){

    }
    public void AAF(){
        bInner mm = new bInner();
    }

    class aInner{
        public aInner(){
        }
        public void kk(){
            bInner[] b = new bInner[2];
        }
    }
    class bInner{
        public bInner(){
        }
    }
    static {
    }
}

//**************
//Button Class
//**************
package elevator;

public class Button{

    private final boolean NOT_LIT=false;
    private final boolean LIT=true;
    private boolean state;
    public  java.lang.Integer i;
    private String BBBBBBBBBBBBBBBBBBBBBBBBB;

    //methods
    public Button() {
        state = NOT_LIT;
```

```
  }

  public void press() {
     state = LIT;
     i = new Integer(3);
     press(i);
  }
  public void press(int i){
     int a = 1;
     boolean b = true;
     byte c = 1;
     long d = 1;
     float e = 1.0f;
     double f = 1.0d;
     String g = "1";
     short h = 1;
     char j = '1';
     String kg = null;
  }
  public void press(java.lang.Object o){

  }
  public boolean pressed() {
     if (state == LIT)
        return true;
     else //if(state.toString() =="not_lit")
        return false;
  }

  public void turnoff() {
     state = NOT_LIT;
     press(3);
     int i = 0;
        Test(1).howmany(i);

  }

  public Panel Test(int i){
        return new Panel(i);
  }
}

//**************
//DoNothing Class
//**************
package elevator;
public class DoNothing {
   public DoNothing() {
   }

}
```

APPENDIX B MEASUREMENT RESULT

CBO, RFC and MPC for **concept.java package 1.0**

file number=196 Package number = 9 Class number = 201			
Class Name	CBO	RFC	MPC
concept.java.tree.VarDeclarationStatement	2	4	1
concept.java.tree.LengthExpression	2	5	1
concept.java.tree.UnaryExpression	1	5	1
concept.javad.attr.codeAttr	4	19	12
concept.javad.attr.codeAttr$exceptInfo	2	4	6
concept.javad.jconst.constUtf8	2	13	6
concept.java.tree.AndExpression	2	2	1
concept.java.tree.Statement	1	3	1
concept.java.tree.LabelStatement	2	4	1
concept.javad.attr.innerClassAttr	2	7	4
concept.javad.attr.innerClassAttr$innerClassInfo	3	9	9
concept.java.tree.BodyElement	1	2	1
concept.idp.CompositePattern	2	12	8
concept.java.tree.BreakStatement	2	4	1
concept.java.ThrowsDef	4	10	8
concept.java.tree.TryStatement	2	5	3
concept.java.tree.AssignAddExpression	2	8	1
concept.java.tree.ForStatement	2	4	1
concept.java.tree.BinaryBitExpression	1	6	1
concept.java.tree.StringExpression	3	9	1
concept.idp.SingletonPattern	2	14	13
concept.javad.util.errorMessage	0	3	0
concept.idp.ClassAdapterPattern	4	12	12
concept.idp.Pattern	0	5	0
concept.java.tree.SynchronizedStatement	2	4	1
concept.javad.methodInfo	10	33	35
concept.javad.methodInfo$methodTypes	0	1	0
concept.java.tree.CharExpression	3	7	1
concept.java.tree.MultiplyExpression	2	7	1
concept.java.tree.ContinueStatement	2	4	1
concept.java.tree.CaseStatement	2	4	1
concept.java.tree.DivideExpression	2	8	1
concept.java.tree.DivRemExpression	1	7	1
concept.java.ASTNode	2	32	3
concept.java.tree.NegativeExpression	2	5	1
concept.javad.attr.synthAttr	1	3	1
concept.java.Strings	0	2	0
concept.java.tree.UnsignedShiftRightExpression	2	7	1
concept.java.tree.LessOrEqualExpression	2	7	1
concept.java.tree.AssignMultiplyExpression	2	8	1
concept.idp.Relation	1	11	6
concept.javad.classFile	13	88	49
concept.java.tree.IncDecExpression	1	5	1

concept.java.tree.LongExpression	3	8	1
concept.java.tree.BinaryCompareExpression	1	6	1
concept.java.ClassPath	4	27	32
concept.java.tree.InlineNewInstanceExpression	2	4	1
concept.java.tree.EqualExpression	2	7	1
concept.java.UsageReference	3	12	4
concept.java.tree.BinaryAssignExpression	1	6	1
concept.java.tree.BooleanExpression	3	8	1
concept.java.tree.ShiftRightExpression	2	7	1
concept.javad.attr.localVarTabAttr$localVarEnt	4	10	10
concept.javad.attr.localVarTabAttr	2	14	5
concept.java.ExtendsDef	4	10	8
concept.java.Modifier	1	21	0
concept.java.tree.BinaryEqualityExpression	1	6	1
concept.java.tree.LessExpression	2	7	1
concept.java.tree.AssignUnsignedShiftRightExpression	2	8	1
concept.java.tree.ReturnStatement	2	5	1
concept.java.tree.NaryExpression	2	6	3
concept.java.tree.ConditionalExpression	2	6	1
concept.idp.IdpMethod	1	4	3
concept.util.Command	0	5	0
concept.java.tree.IfStatement	2	4	1
concept.idp.ObjectAdapterPattern	4	12	12
concept.java.tree.FloatExpression	3	8	1
concept.idp.IdpHierarchy	1	7	2
concept.java.tree.SwitchStatement	2	5	3
concept.javad.attr.srcFileAttr	4	12	6
concept.java.tree.PostDecExpression	2	6	1
concept.java.tree.IntegerExpression	1	8	1
concept.java.tree.NullExpression	3	9	1
concept.java.tree.LocalVariable	6	30	5
concept.java.tree.AssignBitXorExpression	2	8	1
concept.javad.util.accString	1	3	0
concept.java.tree.NewArrayExpression	2	6	1
concept.java.tree.ArrayAccessExpression	2	5	1
concept.javad.jconst.constDouble	1	12	2
concept.javad.Main	3	69	10
concept.java.tree.BitNotExpression	2	5	1
concept.javad.fieldInfo	10	25	26
concept.java.tree.DoStatement	2	4	1
concept.javad.attr.attrInfo	1	4	1
concept.java.tree.AssignExpression	2	7	1
concept.javad.attr.constValueAttr	2	10	5
concept.java.tree.CompoundStatement	2	5	3
concept.javad.classFileHeader	1	6	4
concept.javad.attr.deprecAttr	1	3	1
concept.javad.util.accData	1	14	0
concept.java.tree.FieldExpression	2	5	1
concept.java.tree.PositiveExpression	2	5	1
concept.idp.demos.Bridge.Implementor	0	2	0
concept.java.tree.ConstantExpression	2	7	1
concept.java.tree.ASTParser	103	256	654
concept.java.tree.MethodExpression	2	6	1

concept.javad.util.typeDesc	1	5	1
concept.idp.IdpClan	1	7	2
concept.java.tree.AssignOpExpression	1	7	1
concept.java.tree.PreDecExpression	2	6	1
concept.java.tree.ConvertExpression	2	5	1
concept.java.tree.BinaryShiftExpression	1	6	1
concept.java.Reference	0	4	0
concept.javad.classFieldSec	2	24	5
concept.idp.FactoryMethodPattern	1	6	1
concept.javad.jconst.constLongConvert	2	7	3
concept.idp.Model	14	111	165
concept.java.tree.GreaterOrEqualExpression	2	7	1
concept.java.tree.FinallyStatement	2	4	1
concept.javad.util.objNameFormat	0	3	0
concept.java.tree.DeclarationStatement	3	5	3
concept.java.tree.BitXorExpression	2	7	1
concept.java.tree.CommaExpression	2	6	1
concept.idp.demos.Bridge.Abstraction	2	6	3
concept.java.tree.GreaterExpression	2	7	1
concept.idp.BridgePattern	4	12	11
concept.java.tree.AssignShiftLeftExpression	2	8	1
concept.java.PackageDef	3	7	2
concept.idp.Main	1	74	4
concept.javad.jconst.constFloat	2	9	2
concept.java.tree.AddExpression	2	7	1
concept.javad.attr.attrFactory	12	23	15
concept.javad.classDeclSec	5	13	17
concept.java.tree.NotExpression	2	5	1
concept.java.tree.DefaultStatement	2	4	1
concept.java.tree.AssignBitAndExpression	2	8	1
concept.javad.classMethodSec	2	13	7
concept.java.tree.AssignShiftRightExpression	2	8	1
concept.java.ImportDef	3	7	2
concept.java.tree.IntExpression	3	6	1
concept.javad.jconst.constName_and_Type_info	4	14	6
concept.javad.util.access_and_modifier_flags	0	0	0
concept.javad.classAttrSec	3	8	5
concept.java.tree.AssignBitOrExpression	2	8	1
concept.java.tree.AssignRemainderExpression	2	8	1
concept.java.Constants	0	0	0
concept.idp.IdpClass	1	2	1
concept.java.tree.NotEqualExpression	2	7	1
concept.idp.demos.Bridge.ConcreteImplementorA	0	3	0
concept.java.DeclarationReference	3	12	5
concept.java.tree.CastExpression	2	6	1
concept.java.Type	2	14	19
concept.java.tree.IdentifierExpression	2	5	1
concept.idp.Console	16	150	117
concept.java.tree.ThrowStatement	2	4	1
concept.idp.IdpAttribute	1	2	1
concept.javad.attr.exceptAttr	5	7	7
concept.java.CreationReference	4	13	5
concept.java.Project	7	59	46

concept.java.Project1relation	0	2	0
concept.java.MemberDef	12	90	66
concept.javad.jconst.constPoolTags	0	0	0
concept.java.tree.BinaryArithmeticExpression	1	6	1
concept.java.tree.TypeExpression	2	6	1
concept.java.tree.CatchStatement	2	4	1
concept.java.tree.WhileStatement	2	4	1
concept.java.tree.ShiftLeftExpression	2	7	1
concept.java.tree.SuperExpression	2	2	1
concept.javad.util.dataRead	1	6	9
concept.javad.attr.lineNumTabAttr	2	6	4
concept.javad.attr.lineNumTabAttr$lineEntry	1	5	2
concept.javad.jconst.constInt	2	8	2
concept.java.Environment	8	131	33
concept.java.tree.PostIncExpression	2	6	1
concept.java.FileClassLoader	5	71	38
concept.java.JavaFile	6	26	6
concept.java.tree.ExprExpression	2	5	1
concept.java.tree.ByteExpression	3	7	1
concept.java.tree.BitAndExpression	2	7	1
concept.java.tree.Expression	1	9	1
concept.javad.jconst.constRef	4	13	6
concept.java.tree.BinaryExpression	1	6	1
concept.java.tree.BitOrExpression	2	7	1
concept.java.ImplementsDef	4	10	8
concept.java.tree.ShortExpression	3	7	1
concept.java.tree.ExpressionStatement	2	4	1
concept.java.tree.SubtractExpression	2	7	1
concept.javad.jconst.constBase	1	8	1
concept.java.tree.AssignSubtractExpression	2	8	1
concept.javad.jconst.constPool	10	23	15
concept.java.SourceElement	1	10	0
concept.java.tree.RemainderExpression	2	8	1
concept.javad.jconst.constClass_or_String	4	17	8
concept.java.tree.AssignDivideExpression	2	8	1
concept.java.tree.ArrayExpression	2	6	1
concept.java.tree.InstanceOfExpression	2	6	1
concept.java.tree.CreationalReference	1	2	1
concept.java.ClassDef	9	75	62
concept.java.tree.NewInstanceExpression	2	6	1
concept.idp.DecoratorPattern	5	15	15
concept.idp.IdpProgram	0	1	0
concept.javad.jconst.constLong	1	11	2
concept.java.tree.OrExpression	2	2	1
concept.idp.Entity	0	6	0
concept.java.tree.InlineMethodExpression	2	4	1
concept.idp.demos.Bridge.ConcreteImplementorB	0	3	0
concept.java.tree.PreIncExpression	2	6	1
concept.java.tree.BinaryLogicalExpression	1	6	1
concept.java.tree.DoubleExpression	3	8	1
concept.java.tree.ThisExpression	2	6	2
concept.java.tree.InlineReturnStatement	2	4	1