Dynamic Analysis

Determine Whole-part relationships

Shen
Yapeng

Final Year Project - 2012
International student in
Computer Science and Software Engineering

NUI MAYNOOTH
Ollscoil na hÉireann Má Nuad

Department of Computer Science
National University of Ireland, Maynooth
Co. Kildare
Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Single/Double Honours in Computer Science/Computer Science and Software Engineering.
Supervisor: Dr. James
Power

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of B.Sc. Single/Double Honours in Computer Science/Computer Science and Software Engineering, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Yapeng Shen Date: 24/2/2012

Abstract

Whole-Part relationship is one of the most important inter-class relationships in OOP. Analysis for Whole-Part relationship is useful for developers and maintainers to have a clear look at a program. In most case, analyzing tools in this area are based on UML which divide Whole-Part relationship into two parts: aggregation and composition. But the two types are not enough to describe all Whole-Part relationships.

This project is aimed to discover some detailed properties existing in Whole-Part relationships from Java program. An instrumenting program is built to trace behaviors of instances in Java program and produce tracing files. It can modify Java bytecode while certain class is loading into JVM. Relationship analysis is based on tracing files. In analysis, three normal properties of Whole-Part relationship are introduced. They are lifetime, exclusivity and multiplicity. Results of analysis are demonstrated through some pie charts and line charts, as well as an analysis_result.txt. The program, tomcat, is tested.

Table of Contents

1. Introduction

This chapter is instruction of the whole project. The first section states the goal of the project, followed by the reasons for doing such research. Then I describe approach used to achieve the goal. In the end, the technical area and the report overview is summarized.

1.1 Goals

The goal of this project is to use dynamic analysis to discover detailed properties of whole-part relationships in a Java program.

1.2 Motivation

There exist many different relationships between objects as well as classes in an Object Oriented Program. The Whole-Part relationship has often been treated as a first-class modeling construct in object-oriented analysis. Analyzing the Whole-Part relationships that exist in program can help software developers and maintainers to have a clear look at the program. It can also contribute to comparing the structure of two different programs.

Nowadays, many different tools can analyze the whole-part relationship in programs. But in most cases, these tools are based on UML (Unified Modeling Language) and the definition of UML is not enough to describe relationships in the whole running program yet. This project builds a program that can discover more detailed properties of relationships in a Java program.

1.3 Method

In this project, I use dynamic analysis to trace Java programs, listing out different featured whole-part relationships between classes.

In order to analyze the relationships in Java programs, I need to trace the behaviors in running programs. ASM is a Java bytecode manipulation and analysis framework. I use this third party library to instrument the bytecode of test programs to trace certain behaviors, such as assignment to instances, invoke of a non-static method and so on. All these behaviors are stored in trace files, waiting for following analysis.

Then analysis program treats trace files as input, restoring the running condition of a

program, and building certain data structures to analyze the whole-part relationships.

Finally, I use JFreeChart(a third party open source Java chart library) to make pie charts and line charts to show the result of the analysis. This visualization can lead to a better understanding of whole-part relationships in a program.

1.4 Report Overview

The rest of this document is organized as follows: Section 2 describes the problem domain and state-of-art. It is about how the whole-part relationships are clarified and the details I should take into consideration. Section 3 is about management of this project. It describes the approach I used to manage this project and the initial plan in the beginning. But it is not so smooth through the whole project, as some changes are made to the plan. A final version of the time table is also listed in this section. In section 4, all design works are stated. The ASM library is used to instrument Java bytecode. Though it is a convenient framework to use, the problem of modeling and the algorithm for instrument needs to be designed carefully. The JVM specification should be taken into consideration. Section 5 demonstrates the software implementation. JVM specification is far more rigorous than Java programming. When it comes to software implementation, many problems appeared that I haven't thought over before. It takes a lot of time to find out these bugs, as well as fixing all of them. Section 6 shows experiments and analysis of results. After finishing the program, trace files and result charts can be produced. I test some programs from Dacapo Benchmark Suite. The Dacapo Benchmark Suite is intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. Summaries and future work are listed in the last section.

2. Background and Problem Statement

This section is focused on the background and problem domain. It shows the concept of whole-part relationship and the points which have been taken into consideration in this project, as well as the problems I attempt to solve.

2.1 Introduction

Binary object relationship is a basic inter-object relationship in OOP (Object-Oriented

Programming) Programs. In some sense, the framework of software is based on these inter-object relationships. In this project, I focused on whole-part relationship, an important kind of binary object relationship in Java programs.

Before we discuss the Whole-part relationship, some concepts should be declared first. Research work has provided many characteristics from the Whole-Part relationships, for example, resultant property, encapsulation, transitivity, etc. [1]. It is not so straightforward to catch these properties. I pick out some important concepts from them:

Lifetime : It is the time recorded for an active object through the whole running program. It starts from the time of object construction, ending when object is collected by GC (Garbage Collection is the process of automatically freeing objects that are no longer referenced by the program) or no longer used later. Actually the death time of an object is much more complicated, I will show more analysis in later section.

Ownership : When one instance is assigned to a field of another instance, there exists an ownership between these two instances. The instance which holds the field is the owner or parent object. The instance assigned to the field is called the child object. This is a basic component for Whole-part relationship.

Exclusivity : Sometimes an instance is only owned by one parent object ever in the whole running time of program and such a relationship satisfies strict exclusivity. There are four kinds of exclusivity I talked about, including the transferal, in next section. Details will show in later sections.

Transferal : An instance is only owned by one instance at a given time, but it can be owned by many instance from the same parent class type in different period, just like the child instance can transfer the ownership to another instance from the same parent class. Such a property is called transferal exclusivity here.

2.2 Literature Review

Anning and Jean-Michel [1] have done some deep research on revising the Whole-Part relationship in UML 1.4. It declared that UML 1.4 does not cover the full spectrum of Whole-Part theory. In UML 1.4, aggregation and composition cannot satisfy all of the requirements of object modeling. They said "the semantics of aggregation and composition has resulted in UML being inevitably incomplete and also erroneous". They list a table for primary and secondary characteristics of Whole-Part. It is a necessary and sufficient set of features to implement Whole-Part in UML. In their theory framework, primary characteristics are criteria

for a new Whole-Part meta-type, while specific kinds of the Whole-Part relationships, such as aggregation and composition, are identified by secondary characteristics. OCL (the Object Constraint Language) expressions are used by them to analyze some of the characteristics.

Table 1 Primary and Secondary Characteristics of Whole-Part

primary characteristics	emergent property, resultant property, asymmetry at instance level, antisymmetry at type level.
secondary characteristic	encapsulation, overlapping, lifetimes (9 cases), transitivity, shareability, configurationality, separability, mutability, existential dependency.

In [2], Yann-Gael and Herve show a more detailed way to recover binary class relationships from a program. They propose definitions of the binary class relationships at implementation level to bridge the gap which exists between object-oriented modeling and programming languages. Four minimal properties are taken into consideration when designing the detecting algorithm: exclusivity, invocation site, lifetime, and multiplicity.

In [3], Brian and James show a strategy to visualize dynamic object relationships in Java programs.

2.3 Problem Statement

Designing (or modeling) and implementation are two indispensable aspects in software engineering. In theory, Implementation is based on designing. Unlucky, design models are often changed during implementation for reasons such as time pressure or changing requirements in many projects. When analyzing one program, the design diagram may be not so useful. There is a need to analyze the program structure at implementation level, instead of the design level. My work focused on implementation-level analysis of Java programs, the most used Object-Oriented programming platform in the world.

Binary relationships are basic structures in an OOP program, and the Whole-Part relationship is an important kind of binary class relationship. Nowadays, aggregation and composition are often used to describe certain relationships at design-level. But they are not always clear enough to show the features between classes [1]. What's worse, when getting design structure using reverse engineering tools, for example Rational Rose, it may not reflect the implementation relationships correctly. [2] In detail, some industrial and open-source tools such as Rational Rose and ARGOUML do not clearly define binary class relationships. Their reverse engineering algorithm may produce erroneous or inconsistent relationships.

There are many characteristics show in [1], comparing to the four properties in [2], but they are too complex to declaring a Whole-Part relationship. In this project, I've taken lifetime, exclusivity and multiplicity into consideration. I followed the method proposed by Brian and James in [3]. In addition, I extended this work by including collections and arrays into Whole-Part relationship analysis.

3. Project Management

In this section, I show you the approach used to manage this project in section 3.1. The initial project plan is depicted here using a Gantt chart. When I meet with some problems, the original schedule cannot be implemented exactly. Such problems and changes are show in Section 3.3. A final version of project record is demonstrated in the end of this section.

3.1 Approach

I used smartsheet.com (an online document editor) to manage project. When meeting with some problems, the Gantt chart in smartsheet.com is easily to be modified to another version, which contributes a lot to the schedule.

3.2 Initial Project Plan

At the beginning of the project, as I had no knowledge of Java bytecode, I spent some time learning about it by reading "Programming for the Java Virtual Machine", as well as the documentation for the ASM 3.0 library. The initial project plan is show in Table 2 as well as Figure 1. Since I cannot make detailed estimate about the plan, this initial version is very different from the final version.

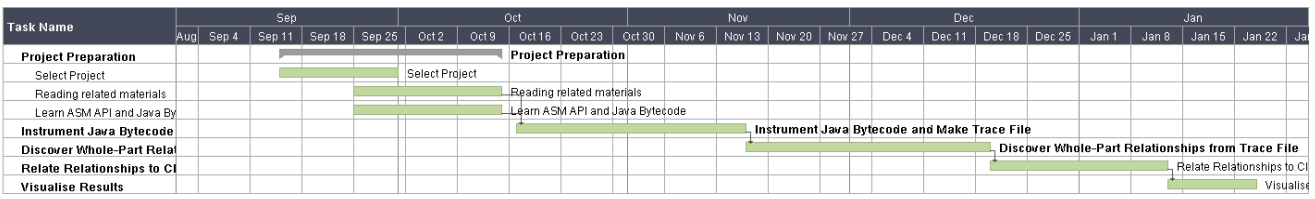

Figure 1 Initial Gantt Chart

Table 2 Initial Schedule

Task Name	Start Date	End Date
Project Preparation	**09/15/11**	**10/14/11**
Select Project	09/15/11	09/30/11
Reading related materials	09/25/11	10/14/11
Learn ASM API and Java Bytecode	09/25/11	10/14/11
Instrument Java Bytecode and Make Trace File	**10/15/11**	**11/15/11**
Discover Whole-Part Relationships from Trace File	**11/15/11**	**12/15/11**
Relate Relationships to Classical Mereology	**01/10/12**	**01/30/12**
Visualize Results	**02/01/12**	**02/10/12**

3.3 Problems and Changes to the Plan

I met with some problems when instrumenting with Java collections. At first, I tried to trace all of methods of collections in the implementation. It took me a long time to solve, though eventually such a function has not be used in analysis.

Memory problems have confused me all the time in this analysis. I've designed a huge data structure for analyzing, and a normal 4 GB main memory in PC is not enough for most of test programs in the Dacapo Benchmark. I've tried to revise the data structure and use data base or serialization approach to avoid such a problem, but that doesn't work. It took me some time to solve this problem.

Since I've wasted a lot of time in instrumenting, there isn't enough time left to relate the Whole-Part relationships to classical mereology. I have to abort this part, making it as future work.

3.4 Final Project Record

Things are going smoothly in the Project Preparation part. I completed the reading on time and turned to design and programming.

Since I haven't got a clear idea about how to analyze the collection cases, a lot of time is wasted in the later period. I have done a lot of work to trace every method invoked by a collection instance, but most of these methods are not useful in analysis.

On the other hand, I spent some time refactoring my code to make it cleaner. I separated the configuration parts from the code into a configuration package to make source code convenient to maintain and modify. This actually works when I change trace sentences in

trace files. The period of debugging lasts long. Bugs often hide deep because of the complicated JVM specification. Analysis starts after most part of instrument code has been accomplished.

Table 3 Final Schedule

Task Name	Start Date	End Date	Predecessors	Duration
Project Preparation	**09/15/11**	**10/14/11**		**22**
Select Project	09/15/11	09/30/11		12
Reading related materials	09/25/11	10/14/11		16
Learn ASM API and Java Bytecode	09/25/11	10/14/11		16
Design System	10/14/11	10/14/11		1
Instrument Java Bytecode and Make Trace File	**10/17/11**	**01/31/12**	**5**	**56**
Build the instrument framework	10/17/11	10/18/11		2
Instrument with normal object	10/19/11	10/25/11	7	5
Instrument with array	10/26/11	11/03/11	8	7
Instrument with collection	11/04/11	11/11/11	9	6
Fixed instrument framework with premain method.	11/14/11	11/15/11	10	2
Test by Dacapo Benchmark and Debug	11/16/11	01/24/12	11	29
Refactor Instrument Code	01/25/12	01/31/12	12	5
Discover Whole-Part Relationships from Trace File	**11/14/11**	**02/01/12**		**37**
Build Framework for Analysis	11/14/11	11/15/11		2
Analyze Alive Object Count	11/16/11	11/18/11		3
Analyze Life Time of Objects	11/21/11	11/23/11		3
interim presentation	12/01/11	12/07/11		5
Analyze Exclusivity of Objects	01/13/12	01/18/12		4
Analyze Multiplicity of Objects	01/19/12	01/23/12		3
Summary	01/24/12	02/01/12		7
Visualise Results	**12/02/11**	**02/08/12**	#REF	30
Draw Line Chart for Object Count	12/02/11	12/02/11		1
Draw Pie Charts for results	02/07/12	02/08/12		2

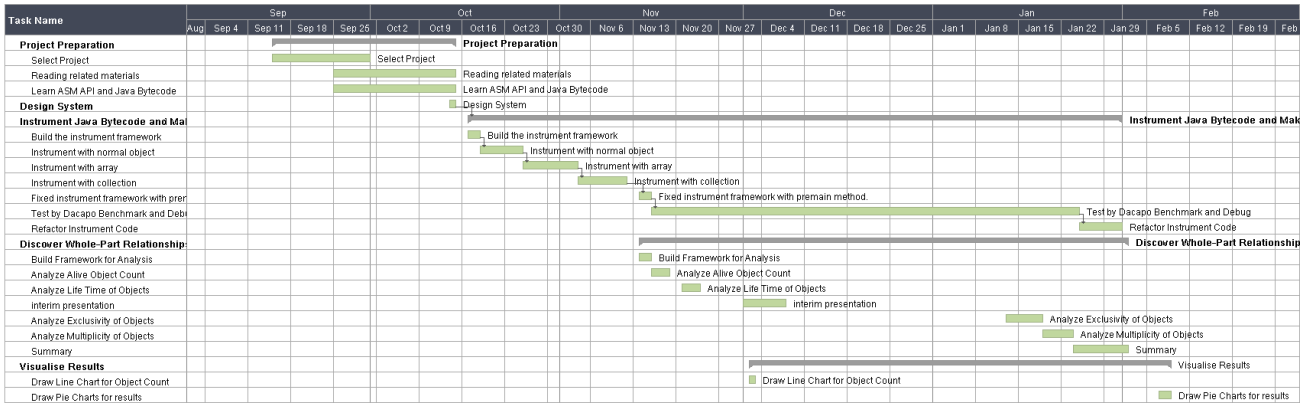

Figure 2 Final Gantt Chart

4. Analysis and Solution

In this section, I talk about the problem modeling, the solution to analyze and why I choose such a solution.

4.1 Problem Modeling

In this project, normal Java programs are inputs. I analyze the Whole-Part relationships in the running program and output the result of analysis. In detail, the whole problem can be broken down into three sub-problems. The first one is tracing proper behaviors of instances in a running Java program. The second is analyzing the result of tracing and discovering certain relationships from it. And the last step is to demonstrate the result.

4.2 Solution

To discover Whole-Part relationships between instances in a running program, we should trace proper behaviors of instances in a given program. There are two ways: One is modifying JVM to detect this information. It is not a good way as it requires a detailed understanding of the JVM specification and lots of work to implement it. The other way is to modify the code to make it possible to trace behaviors. Java is easy for developers to code, but not easily instrumented because of the wide variety of coding idioms. Also the source code of many programs is not accessible. Instead, Java bytecode is much more appropriate. There are lots of mature open-source libraries to instrument it already. The java.lang package provides enough

data structure to make analysis to relationships such as List, Collection, Set etc.. JFreeChart can be used when it comes to demonstration of the results.

5. Software Implementation

This section describes the way to implement instrument and analysis, including ideas in designing period. Section 5.1 states the platform and programming language I chose. After that is designing and coding details.

5.1 Introduction

Ubuntu11.10 (64bit) is used as the operating system. It is convenient to use shell script to control a project on Linux platform.

Java6.0 is chosen as the programming language since it has many mature supportive open-source resources. Eclipse 3.7.0 is the integrated development environment, with ASM Bytecode Outline plugin installed to show a mapping from Java source code to corresponding Java bytecode as well as method that should be invoked in ASM framework to produce such Bytecode.

The most important third party library in my project is ASM 3.0 which can help a lot in instrumenting Java bytecode. It hides bytecode complexity from developers and provides better performance. Programmers do not have to deal with class constant pool and offsets within method bytecode directly.

The JFreeChart library is used to draw certain charts to show results from analysis.

5.2 Design and Coding

Since JVM specification is rigorous and complicated, many unpredictable problems appear when instrumenting Java bytecode. In this section, I focus on how I implemented this system, especially how to instrument Java bytecode using the ASM library.

5.2.1 Instrument Java Bytecode

5.2.1.1 Way to Instrument Bytecode

At first I created a user-defined class loader to instrument Java bytecode. But it is not

generally applicable since in some cases it is not straightforward to find out the entry point of a program.

The **premain()** approach can overcome such a problem. It is a method in java.lang.instrument package, working as entry point before the normal entry point method is executed. I overrode **public byte[] transform(ClassLoader loader, String className,Class<?> classBeingRedefined, ProtectionDomain protectionDomain,byte[] classfileBuffer)** in java.lang.instrument to modify bytecode loading into the class loader dynamically. All modification to Java bytecode that will be loaded by the class loader can be implemented in the **transform**() method. It can work with **public static void premain(String agentArgs, Instrument inst)** while adding to the argument **inst**, acting like a hook.

But we cannot instrument with all classes since some classes are protected by JVM verification to avoid hacker attacks. For example, verification will be applied when classes in java.lang package are loaded. If such class bytecode is modified, a verification error will come out. Luckily, it is not my duty to trace such classes. Analyzing relationships between certain instances specifically from test programs is my goal. To make the analysis more straightforward, some types are ignored such as primitive types, String and wrapped types and some other classes in the Java package. On the other hand, I have to pay attention to array and collection types which is inaccessible. They play an important role in Whole-Part relationship. These points will be described carefully later.

ASM3.0 library is used in the **transform()** method to implement instrumentation. Its framework uses a visitor-like pattern. The core package can be divided into two parts, show in Table 4.

Table 4 Structure in ASM framework

Producer	**ClassReader**
Consumer	Writers such as **ClassWriter**, **FieldWriter**; Adapters such as **ClassAdapter**, **MethodAdapter**; Visitors such as **ClassVisitor** and **MethodVisitor**; Other classes implement above interfaces.

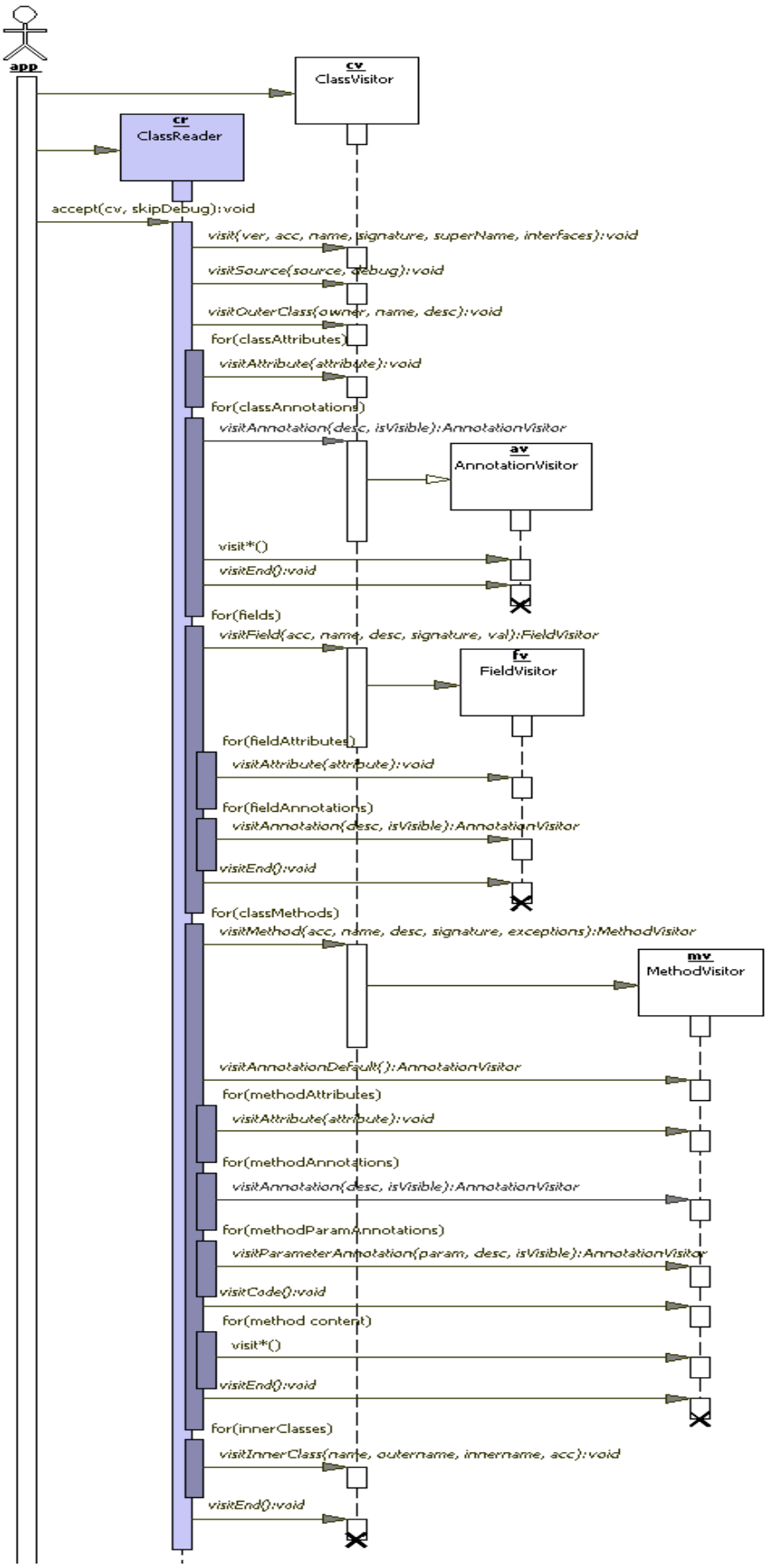

Figure 3 ASM Framework

As Figure 3 shows, the producer, **ClassReader**, can parse the bytecode of a class. By invoking **accept()** method in **ClassReader**, the visit event for different bytecode fragments is fired. **ClassAdapter** is designed as an adapter for **ClassVisitor** which provides a certain way to visit bytecode. The chain of responsibility pattern links all the consumers together. In this project, a simple chain is used by the following style:

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
ASMClassAdapter mca = new ASMClassAdapter(cw);
ClassReader cr = new ClassReader(classfileBuffer);
cr.accept(mca, 0);
retVal = cw.toByteArray();
```

ClassWriter cw is at the end of the chain while **ASMClassAdapter**(inherited from **ClassAdapter**) **mca** is at the beginning of the chain. It is registered to **ClassReader cr** by **accept**() method. **ASMClassAdapter** here can control the delegation to **ClassWriter** to write down bytecode and transfer to the JVM. In this way I can instrument Java bytecode whatever I want.

In **ClassVisitor** (or **ClassAdapeter**, here is **ASMClassAdapter**), **MethodVisitor**() returns a **MethodVisitor** class which is responsible for dealing with bytecode of the methods in detail. There is also an adapter for **MethodVisitor**, called **MethodAdapter**. I've extended many custom **MethodAdapter** for instrumenting, which is show later in this section. The following is a simple example to show how **visitMethod**() controls delegating :

```
public MethodVisitor visitMethod(int access, String name,
                                 String desc, String signature, String[] exceptions){
    MethodVisitor mv ;
    mv = cv.visitMethod(access, name, desc, signature,  exceptions);
    if(name.equals("func")){
       return null;  //Nothing delegated when function name is "func"
    }else{
       return mv;
    }
}
```

In this example, method "func()" in bytecode is removed from instrumented bytecode. Actually I just overrode **visitMethod**() method in **ASMClassAdapter**. **visitField**() which corresponds to field instrument, as well as **visitAnnotation**() are no used in this project.

MethodAdapter can instrument different types of bytecode by **visitInsn**() method, **visitIntInsn**() method etc.. These methods act like a filter for instructions.

```
@Override
public void visitInsn(int opcode) {
   if(opcode == Opcodes.IRETURN || opcode == Opcodes.LRETURN || opcode ==
```

```
                Opcodes.FRETURN || opcode == Opcodes.DRETURN || opcode ==
                Opcodes.ARETURN || opcode == Opcodes.RETURN){
        //put the object into stack as parameter for the following method call
        mv.visitVarInsn(Opcodes.ALOAD, 0);
        mv.visitMethodInsn(Opcodes.INVOKESTATIC, "fyp/instrument/Trace",
                                "makeSureTraceObjectConstructor", "(Ljava/lang/Object;)V");
      }
      super.visitInsn(opcode);
    }
```

Above is a fragment of code from **TraceConstructionMethodAdapter**. In **visitInsn**() method, operation code is checked by "if" statement. If it is **IRETURN** or some other bytecode responsible for returning a method, an invocation of **makeSureTraceObjectConstructor()** method will be inserted in the bytecode file to trace the construction of an object. In this way, **MethodAdapter** can instrument bytecode directly by inserting, removing or modifying instructions.

The **visitMethod()** in **ASMClassAdapter** stands at a high level in the instrumentation strategy. Before that, let's have a look at Trace.java and the Configuration package.

Trace.java is very important. It includes many methods that can be invoked by the instrumented code. Since bytecode is more complicated that Java for developers, putting multi-line instructions into a method in Trace.java is much more convenient instead of inserting into bytecode. On the other hand, it is much easier to maintain code in this way.

fyp.Configuration.TraceFileTranslator is a class responsible for conversion from in-memory data structure such as object id to literal sentences in trace files(trace.txt, field_index.txt and class_index.txt) and reverse. Sentences in trace.txt are show in Table 5. There is one flag in the front of the sentence, indicating behaviors of instances, followed by identifications for objects or classes in order.

In **ASMClassAdapter**, methods are divided into three different ways to instrument by method type: constructor, static method and normal method. These methods can indicate different behaviors of objects. Constructor indicates creation of an object, static method doesn't belong to any objects and normal method can indicate liveness of an object when it is invoked. Table 6 shows corresponding MethodAdapters to these three methods.

The next two sections describe the detailed implementation of the instrumentation.

Table 5 Sentences in Trace.txt(corresponding Java bytecode is based on represents from ASM 3.0 framework)

Sentence in Trace File	**Items Order**	**Corresponding Java bytecode**
[assign]	field, parent, child	PUTFIELD
[assign] [o2a]	child, array(parent)	AASTORE
[assign] [a2o]	field, parent, child	AALOAD
[create]	object, class	<INIT>
[create] [a]	array, size	ANEWARRAY
[create] [c] ...	collection	<INIT> for collection type
[use] ...	object	Add in every method.
[use] [c] [add]	collection, object	INVOKEVIRTUAL for collection type
[use] [c] [remove]	collection, object	INVOKEVIRTUAL for collection type

Table 6 Method Types With Different MethodAdpater

Method Type	**Corresponding MethodAdapter**
Constructor	TraceConstructorMethodAdapter TraceArrayMethodAdapter TraceFieldMethodAdapter
Not Static and Not Constructor	TraceArrayMethodAdapter TraceCollectionMethodAdapter, TraceFieldMethodAdapter TraceMethodMethodAdapter
Static Method	TraceArrayMethodAdapter TraceCollectionMethodAdapter, TraceFieldMethodAdapter

5.2.1.2 Instrument Normal Objects

An instance in a running program should be recorded from its creation to its death. It is straightforward that the time of construction for an object can be marked as the creation time, but it is not so easy to get the end time of an instance without the help of GC program which has not been taken into consideration in this project. An approximate approach has been adopted here: Treating the invocation of any methods of an instance as a mark of alive for this instance, so that the last use of this instance can be regarded as time of death for this instance.

TraceConstructorMethodAdapter is responsible for tracing the construction of instances. It inserts instructions to invoke my **makeSureTraceObjectConstructor()** method in **Trace.java**. This can guarantee the tracing of the construction of an instance. The class type of an object should be traced before the first time of construction of an object. This is done by **traceClassAndFields().** It can write down all fields into fields_index.txt from a certain class type, iterating by inheritance relationships up to the **Object** class type. In this way, fields are identified differently if they are the same field but from parent and child classes in inherited relationships. That can make analysis much more straightforward.

TraceMethodMethodAdapter is responsible for tracing the use of method of instances. It inserts instructions to invoke my **traceMethod()** method in **Trace.java**. This can trace usage of an object. In other words, check if an instance is alive.

When it comes to relationships between classes, I paid attention to assignment instructions, **PUTFIELD** in Java bytecode. This is implemented by **TraceFieldMethodAdapter**. If the construction of an instance has not been traced yet, it will be traced before the **PUTFIELD** by invoking the method **makeSureTraceObjectConstructor()**.

A conflict between **PUTFIELD** and **<INIT>** comes up: An assignment may appear in an constructor or may not. I cannot trace the construct before a **PUTFIELD** or before the **RETURN** instruction in a constructor, since each side may collapse in some cases. A flag should be marked in this situation. This flag exists in a data structure in Trace.java: the **objects** (a hash map identified by unified hash code of an instance) can store all instances created. An invocation of **makeSureTraceObjectConstructor()** method is inserted before every **PUTFIELD** and **RETURN** in constructor. The first invocation for a certain instance is traced while following ones will do nothing.

The synchronizing problem cannot be ignored. If the instrumented program is multi-threaded, it may cause some conflicts when I add one class ID into **classes** (data structure in Trace.java). As a result, **traceObjectConstructor**() in Trace.java is partly decorated by the synchronized key word:

```
synchronized (Trace.class){
    if (!classes.containsKey(obj.getClass().getSimpleName())){
        traceClassAndFields(obj);
    }
}
```

It is noteworthy that inner class (there is a "$" in the class name) case is excluded here because of the unusual hidden outer class reference. Normally, initialization method should be

invoked before its instance members are accessed (in 4.8.2 Structural Constraints [9]). I just ignore this case.

The following code shows the special case for inner-class:

```
public class OutterClass{
   public class InnerClass{
   }
}
```

Let's see the bytecode :

```
public OutterClass$InnerClass(OutterClass);
  Code:
   0: aload_0
   1: aload_1
   2: putfield #10; //Field this$0:LOutterClass;
   5: aload_0
   6: invokespecial      #12; //Method java/lang/Object."<init>":()V
   9: return
}
```

There is one argument in its constructor which does not appear in Java code. And the assignment (**PUTFIELD** instruction) is executed before **<init>**(**INVOKESPECIAL** instruction).

The **GETFIELD** instruction has not been taken into consideration since there is no need. In essence, **GETFIELD** is a preparation instruction to use an instance. It can take certain instance to the operand stack. Instead of tracing instances in **GETFIELD**, usage of an instance can be traced by the following operations such as a method invocation.

5.2.1.3 Instrumenting Containers

As it shows in section 2 of [2], in some tools such as Rational Rose, if class Example2 has a field List of A, when reverse-engineering the source code, the class diagram recovered from source code displays an aggregation relationship between class Example2 and List, not Example2 and A. The initial class diagram is as Figure 4, the source code follows while the class diagram created by reverse-engineering is show in Figure 5.

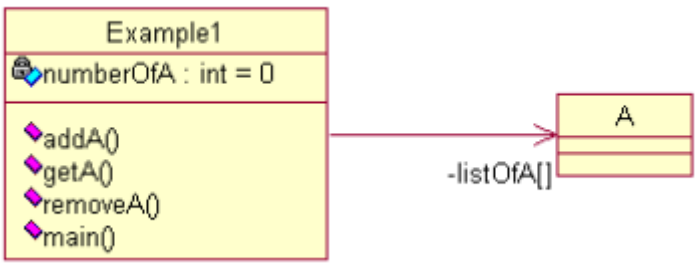

Figure 4 Initial Class Diagram

```
public class Example2 {
    private List listOfAs = new ArrayList();
    public void addA(final A a) {
        this.listOfAs.add(a);
    }
    public A getA(final int index) {
        return (A) this.listOfAs.get(index);
    }
    public void removeA(final A a) {
        this.listOfAs.remove(a);
    }
    public static void main(final String[ ] args) {
        final Example2 example2 = new Example2();
    example2.addA(new A());
    // . . .
    }
}
```

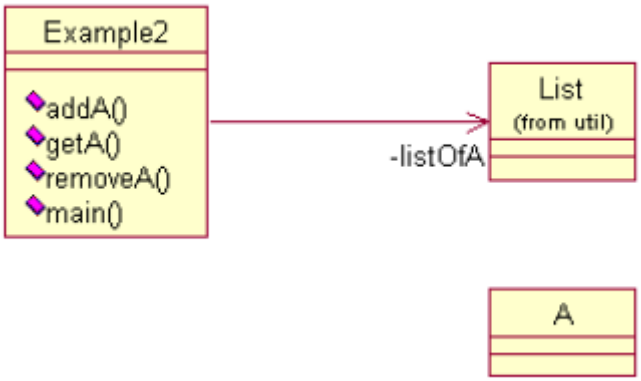

Figure 5 Class Diagram Created by Reverse-Engineering

The collections, as well as arrays, just act as containers in a program, making connections between parent classes and child classes in Whole-Part relationships. I've focused on such a point.

It is simple in array case since bytecode related to array are completely different from others and the array cannot be inherited.

As show in Table 7, there are only three kinds of bytecode that need to be traced in the array case.

Table 7 Important Bytecode of Array

Important Bytecode	Function
ANEWARRAY	Used to create an instance
AALOAD	Used to load a element of an array into operand stack
AASTORE	Used to store a member into an array, as building a connection from the element of the array to the owner of the array.

The collection case is much more complex. Collection operations haven't got unique bytecodes. They are used in a way as other normal objects are. I can just detect the class of method owners, picking collection ones out, and tracing such invocations (in most case it is **INVOKESPECIAL** or **INVOKEVIRTUAL**).

There are up to four arguments for a methods of collection, and the **this** pointer of a collection is always in the first argument position. If there is no or just one parameter in an invocation, I can use **DUP** to trace the **this** pointer of the collection. But if there are two or three or even four arguments in the invoked method, I am not able to trace **this** pointer by just **DUP** the arguments and record it without affecting the original invocation, since the largest size of **DUP** operation is two. The Java reflection mechanism is used in this case. I replaced the original method invocation by invoking a method that can trace the hash code of the collection instance and then invokes the original method. The special method has an object array as its argument to receive any number of arguments. It will invoke a method in Trace.java and the original method will be invoked by this method in a reflected way.

The difficulty I've met was transferring between primitive types and boxed types when they are arguments or return value. This auto-boxing and auto-unboxing process is implemented by compiler automatically in Java, but it is my responsibility in bytecode.

5.2.2 Analyzing on my Trace Files

Trace files (trace.txt and filed_index.txt) are read in by Scanner.java. Scanner will translate trace sentences into certain data structure.

There is a **static int timer** in **Scanner** class, it can be increased by one every time a sentence is read in, acting like a clock.

Firstly, field_index.txt is read in, to build **Analyzer.fields** (ArrayList). Then is the main part, trace.txt. Sentences in trace.txt are divided into three kinds and analyzed by different methods.

```
if(TraceFileTranslator.isCreateSentence(str)){
   processCreateSentence(str);
}
if(TraceFileTranslator.isUseSentence(str)){
   try {
      processUseSentence(str);
   } catch (Exception e) {
      e.printStackTrace(System.out);
   }
}
if(TraceFileTranslator.isAssignFromArraySentence(str)
      ||TraceFileTranslator.isAssignObjectSentence(str)
      ||TraceFileTranslator.isAssignToArraySentence(str)){
   processAssignSentence(str);
}
```

These processing methods can parse trace sentences into certain types and invoke certain methods in **Analyzer.java**.

Before describing the functions of these three methods, the data structures in Analyzer.java should be described. There are five important data structures:

a) **public static List<Integer> objectCount = new ArrayList<Integer>();**

It is an ArrayList of Integer, storing object counts at every time interval.

b) **public static List<LinkedList<FieldObjectOwnedByList>> fields**

= new ArrayList<LinkedList<FieldObjectOwnedByList>>();

As Figure 6 shows, it is a list (list1) storing all fields while every field maintains a list (list2) of objects which are child objects of this field, and every assignment from these objects to other objects are recorded in a list (list3). Every assignment should be traced in **fields**.

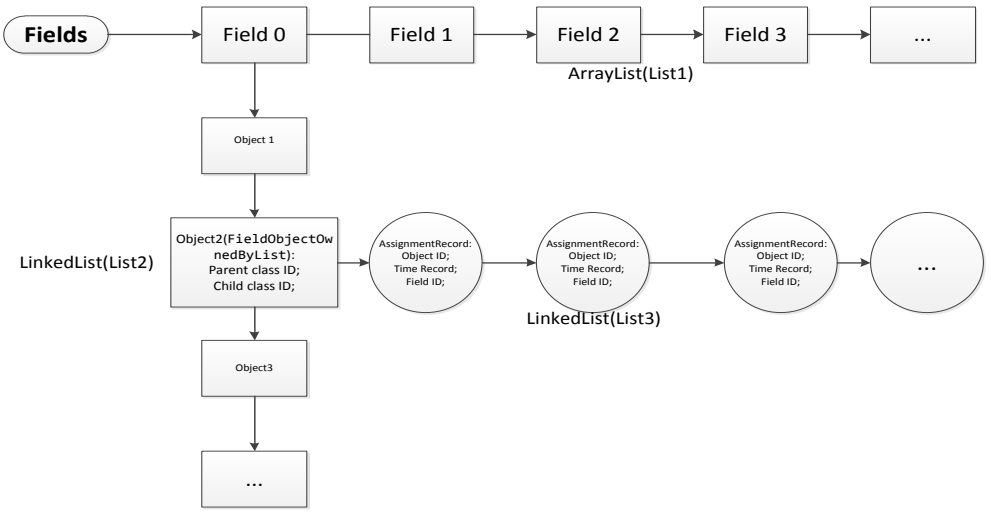

Figure 6 Fields Data Structure

c) **public static Map<Integer, ContainerRecord> containers = new HashMap<Integer, ContainerRecord>();**

The data structure is demonstrated in Figure 7. It is a hash map for containers (array and collection). The key is the identity hash code of the object, and the value is **ContainerRecord**. The **ContainerRecord** contains the parent object ID and field ID for this container, making it possible to make connections between parent object and members in containers. Every array and collection created will be added into this data structure.

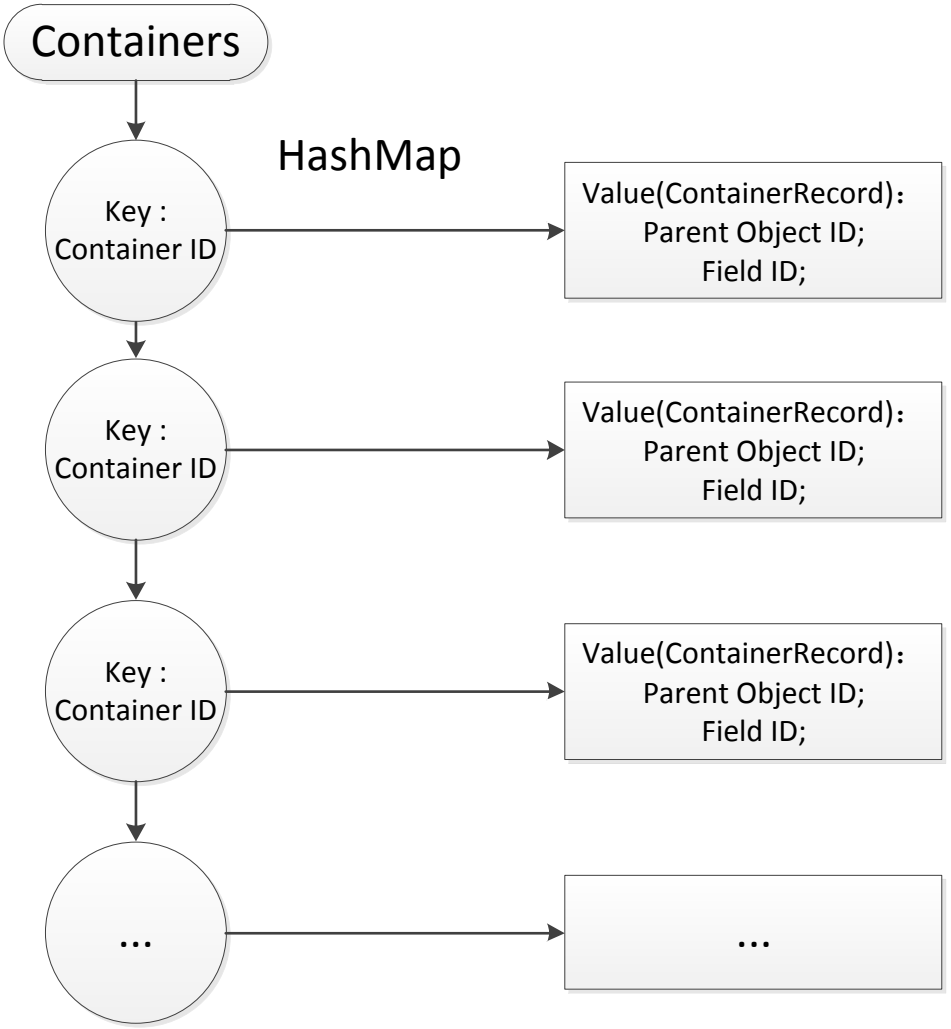

Figure 7 Containers Data Structure

d) **public static Map<Integer, ObjectRecord> objects = new HashMap<Integer, ObjectRecord>();**

It is a hash map for all objects created in program. The key is the identity hash code of the object and the value is an **ObjectRecord**. Information such as the life time and

class ID of this object is in the **ObjectRecord**. All normal objects will be added into this data structure. Every object also maintains two lists : an owning objects list (when the object is assigned with other objects, these assignments will be traced in owning objects list) and an owned by objects list (when the object is assigned to other objects, these assignments will be traced in owning objects list). Though it is redundant to keep assignments information in both **fields** and **objects**, it is essential to keep these two lists so as to decrease the time for finding out ownerships in both parent to child and child to parent directions.

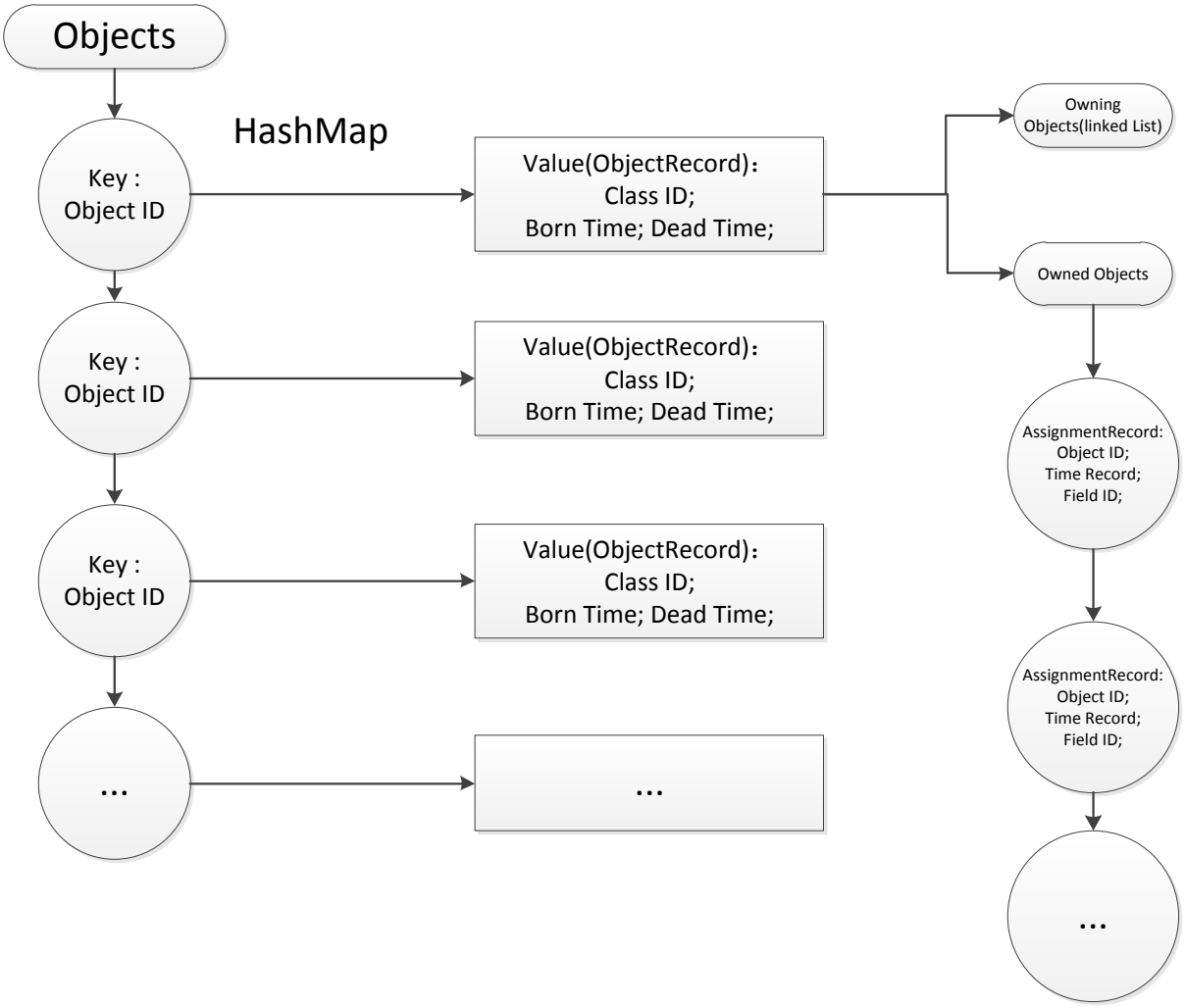

Figure 8 Objects Data Structure

e) **public static Map<ClassRelation,ClassRelationRecord> classesRelations = new HashMap<ClassRelation,ClassRelationRecord>();**

This is a hash map for all relationships in program, identified by ClassRelation(parent class, field ID and child class). This is the core data structure which holds the result of Whole-Part relationship analysis. In initial design, class inheritance has

been taken into consideration, so field ID is not the only identification for ClassRelation. There may be different parent class type and child type in a same field. But in the implementation, I trace and index fields iteratively, that is to say, even a same field is differently indexed in parent class and child class. That can make the result in a lack of accuracy.

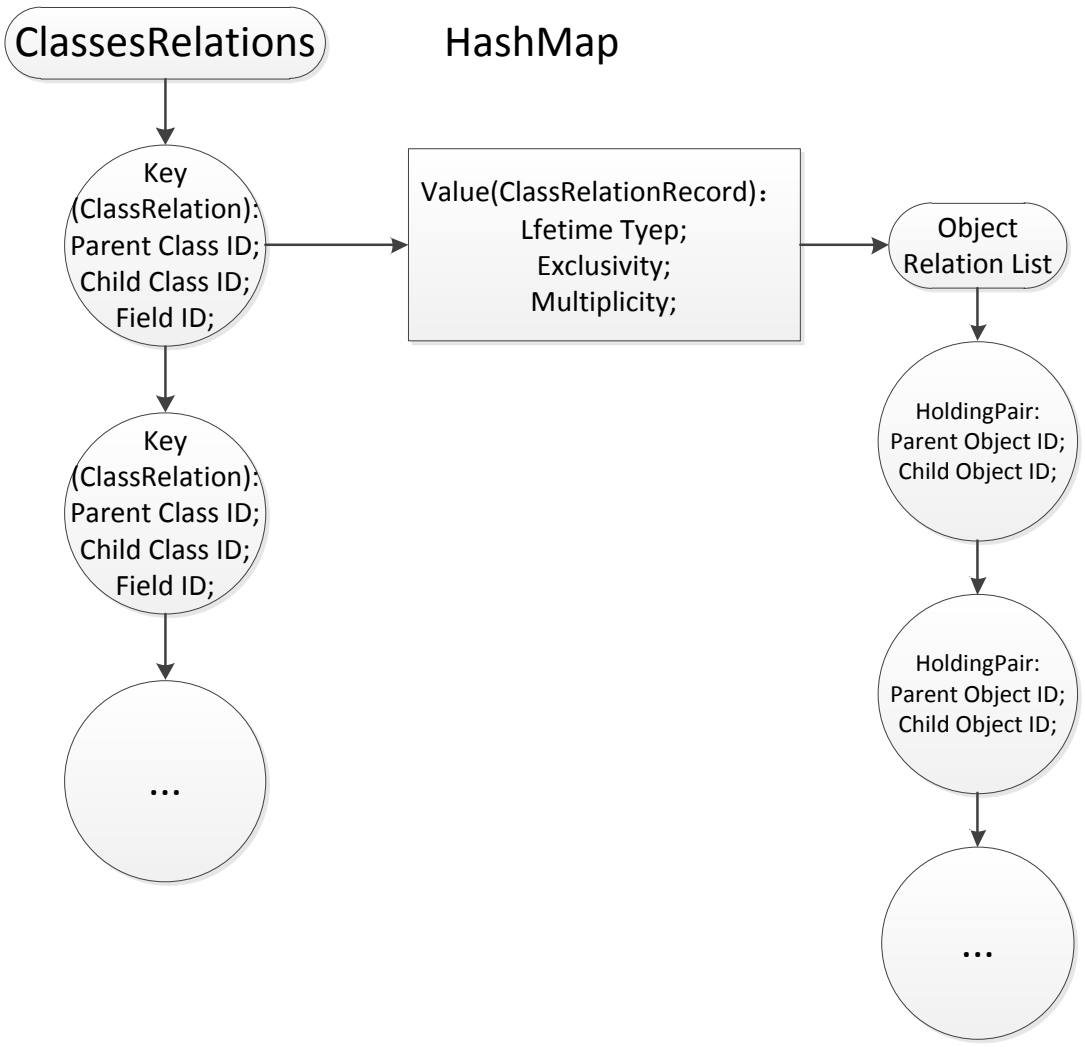

Figure 9 ClassesRelations Data Structure

In **processCreateSentence()**, if it is a normal object (excluding array and collection), a data entry will be added in objects. The identity hash code of this object will be the key, **ObjectRecord** can be created to maintain information of this object such the start time. Time of death will also be set to the same time, updated later when the object is used. In container case, a data entry will be added into containers, but life time is not recorded as it makse no sense in analysis.

In **processUseSentence**(), if it is a normal object, **updateObjectDeadTime(objectId,**

timer) will be invoked to update **deadTime** and **objectCounts**: dead time of this object will be set to current **timer**, and all members of **objectCounts** between the last time of death and the new dead time of this object will be increased by one. If it is an add method of a collection, the parent object of the collection will be found in **containers** and the child object will be treated as directly assigned to the parent object. It is the same for **isAssignToArraySentence** case in **processAssignSentence()**.

In **processAssignSentence()**, there are three cases : First, if it is a normal object assignment, this assignment will be recorded in both fields and objects. Second, if it is an assign to array case, the parent object of the array will be found, and an assignment will be recorded between parent object and child object. Third, if it is an assignment from an array or from a collection, the parent object id can be set in **containers**.

The last step of analysis is processing all relationships in **classesRelations** based on three properties(lifetime ,exclusivity, multiplicity):

Lifetime types are divided into four types. As show in Figure 10, the black line stands for lifetime period of object. The owner object (or parent object) is above the child object. For example, in type 1, the build time of the parent object is earlier than the child object and death time of the parent object is later than the child object.

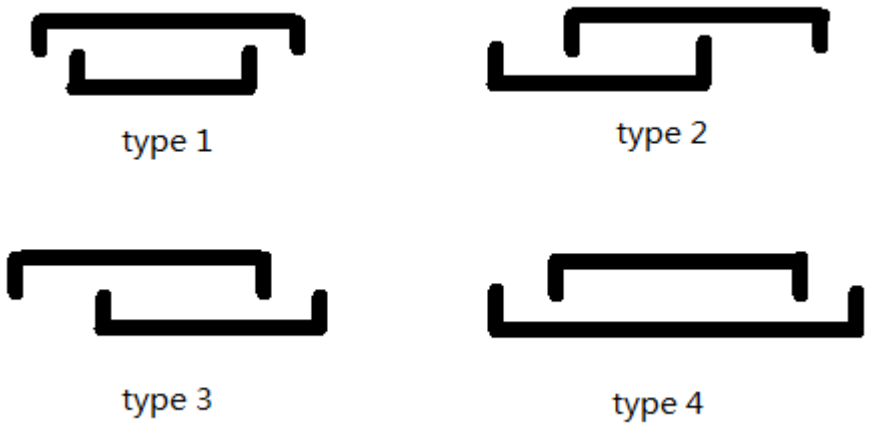

Figure 10 Lifetime Types

In **AnalyzeLifeTimeRelationships()** (method in Analyzer.java), all entry in **Analyzer.classesRelations** will be processed. Lifetime of the parent and child object will be got by object ID from objects. An array, **int lifeTimeType[4]**, will maintain the value by increasing by one when certain type of lifetime in a relationship is found out. This array can

show an accurate count for all objects in a given class relationship. The final step is to summarize from this count array to get a right lifetime type for the class relationship. As show in Table 8, count arrays can always fit in one of the rows, and the type corresponding to bold font in that row is just the lifetime type of the class relation.

Table 8 Lifetime Decision Mechanism

Type 1	Type 2	Type 3	Type 4
0	0	0	**Non-zero**
0	0	**Non-zero**	0
0	0	Non-zero	**Non-zero**
0	**Non-zero**	0	0
0	Non-zero	0	**Non-zero**
0	Non-zero	Non-zero	**0**
0	Non-zero	Non-zero	**Non-zero**
Non-zero	0	0	0
Non-zero	0	0	**Non-zero**
Non-zero	0	**Non-zero**	0
Non-zero	0	Non-zero	**Non-zero**
Non-zero	**Non-zero**	0	0
Non-zero	Non-zero	**0**	Non-zero
Non-zero	Non-zero	**Non-zero**	0
Non-zero	Non-zero	**Non-zero**	Non-zero

Exclusivity is much more complicated. There are four kinds of exclusivity, showed in Table 9:

Table 9 Exclusivity Clarification

Exclusivity	**Explanation**
EXCLUSIVITY_NO	It doesn't satisfy all other three types.
EXCLUSIVITY_TRANSFER	One child object can be assigned to many parent objects. But the child object cannot be owned by more than one parent object at the same time.
EXCLUSIVITY_GLOBAL	The child object is owned by only one object ever in the whole program.
EXCLUSIVITY_LOCAL	The child object can owned by many object, but it cannot be owned by more than one object from certain field. In

	other words, an object is owned by only one object in certain field and can be owned by object from other field.

In **AnalyzeExclusivityAndTransferal()**(method in Analyzer.java), **Analyzer.fields** will be looped by field and by every object in field (In figure 6, list 1 and list 2 are looped). The local exclusivity of this class-field-class can be figured out in **Analyzer.fields** since the assignment is organized by field and child object. The global exclusivity can be figured out in **Analyzer.objects** since the assignments which certain object act as child object is maintained in it. When it comes to transferal exclusivity, **Analyzer.objects** is also used to provide the assignment from certain object while certain object acting as parent object. If an object in **Analyzer.fields** is transferal, it cannot be assigned to another object before the former parent object is assigned by another object in this field. As the lifetime case, count of all objects can be showed in the array **int exclusivity[4]**. And exclusivity for the relationship will be the first non-zero one in count array, **exlucisivity**[4].

The last property, multiplicity is determined by if the field is an array or collection or not. Array and collection are object sets, in those cases, multiplicity is true. Or it is false.

5.2.3 Showing Results

All properties of each class-field-class relationship are listed in analysis_result.txt while the percentage of clarification for each property is showed in pie charts. Object counts through whole lifetime of the program can also be demonstrated in a line chart. JFreeChart library is used to draw these charts. All these code is in fyp.analysis.result package.

5.3 Verification

For convenient, smoke test is used in instrument phase of this project.

The testing code is in test.java, root package of this project. When a function(for example tracing **PUTFIELD**) is added, the corresponding test code which implements such behaviors(in this case, assignment to field of object) will be in test.java. It is not difficult to design the test code since all function is based on obvious behavior of objects, such as creating an array, assignment to a collection, a method of an object is invoked etc.. Then, a script file (smoke_test in FYP/pro directory), will be executed in terminal. In many cases, some problems such as verify error from JVM will be listed in console. After resolving these bugs,

output of instrumenting phase(result.txt) will be checked to see if it has traced the test program in the right way.

Testing in the analysis and results phase is the same as the instrumentation phase.

All code has passed smoke test.

5.4 Validation

The Dacapo Benmark Suite (dacapo-9.12-bach.jar) is used to test the validation. In this period, many hidden bugs that I haven't noticed appeared such as inner-class problem and synchronization problems show in section 5.2.1.2.

Six benmarks can be tested throughly by instrument program. When it came to analysis, a memory problem appeared. As some huge data structures need lots of memory, 4 GB is not enough for analysis phase.

Some problems still exist in other benmarks. They are listed in Table 8.

Table 10 Bugs Still Exist

Benmark Program	**Bugs**
avrora	Digest validation failed for stderr.log, expecting 0xda39a3ee5e6b4b0d3255bfef95601890afd80709 found 0x38f57ad0634f3d3925185e7f0de471fe1b7b9f66
batik	org.apache.batik.transcoder.TranscoderException: null Enclosed Exception: java.lang.Object cannot be cast to java.lang.String
eclipse	java.lang.reflect.InvocationTargetException
fop	java.lang.reflect.InvocationTargetException
h2	java.io.IOException: Stream closed
jython	SystemError: Failed to load the builtin codecs: Cannot import _codecs, missing class org.python.modules._codecs *** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message transform method call failed at ../../../src/share/instrument/JPLISAgent.c line: 824
sunflow	Exception in thread "Thread-6" java.lang.OutOfMemoryError: GC overhead limit exceeded

6. Experiments and Results

The experiments here are based on the Dacapo Benchmark Suite. Six test programs in this benchmark can be successfully instrumented, but most of them met with memory problems in analysis. In section 6.2, I will make a comparison of analysis results from two different

programs in the Dacapo Benchmark Suite.

6.1 The Experiments

6.1.1 Instrument

Putting the program to be traced into FYP/api package, making FYP/api as the working directory and executing the following instruction:

```
# java -javaagent:ASMInstrument.jar XXX
```

The **-javaagent** option can lead the program to be instrument.

The output is three tracing files in FYP/result directory. Introduction of the tracing files are listed in Table 8.

Table 11 Introduction of Tracing Fiels

File Name	Function
class_index.txt	It records simple name of classes, indexed by class id in increasing order.
field_index.txt	It records field names, indexed by field id in increasing order. The name of owner class is also recorded.
trace.txt	Record all related instructions executed. Class and field name used in instruction are replaced by corresponding id.

6.1.2 Analysis and Showing Result

When analyzing, trace.txt is the main input. Through configuration file fyp.properties in FYP/pro, the program can find out the right input and make analysis. Making FYP/api as the working direcotry and execute:

```
#java -classpath ./:ASMInstrument.jar:jfreechart-1.0.14.jar:jcommon-1.0.16.jar
fyp.analysis.Scanner
```

Output of this step includes many charts (lifetime pie chart, exclusivity pie chart, multiplicity pie chart and object count line chart) and an analysis_result.txt file.

6.2 Results

Six programs from Dacapo Benmark Suite can be successfully instrumented, they are **luindex**, **lusearch**, **pmd**, **tomcat** and **xalan**. Since lack of memory resource, only **tomcat** can be completely analyzed till now.

To make a comparison, I've picked 200MB data from trace.txt of **pmd**. Following is

result charts for **tomcat** and part of **pmd**. If not declared, tomcat is in the left hand side while pmd is in the right.

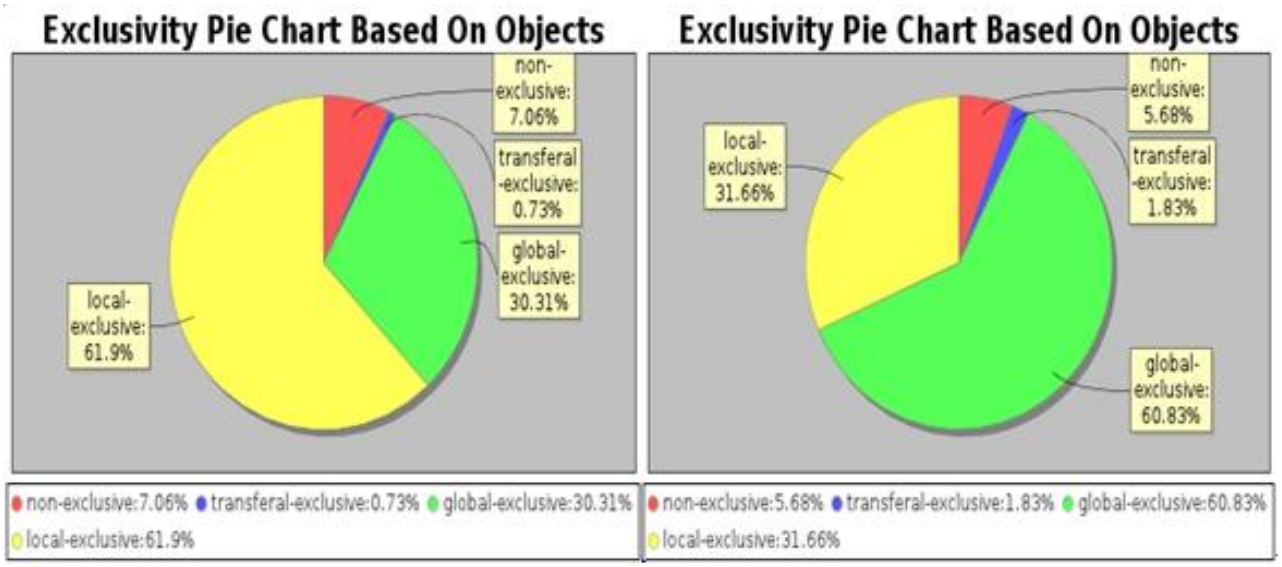

Figure 11 Exclusivity Pie Chart Based on Objects

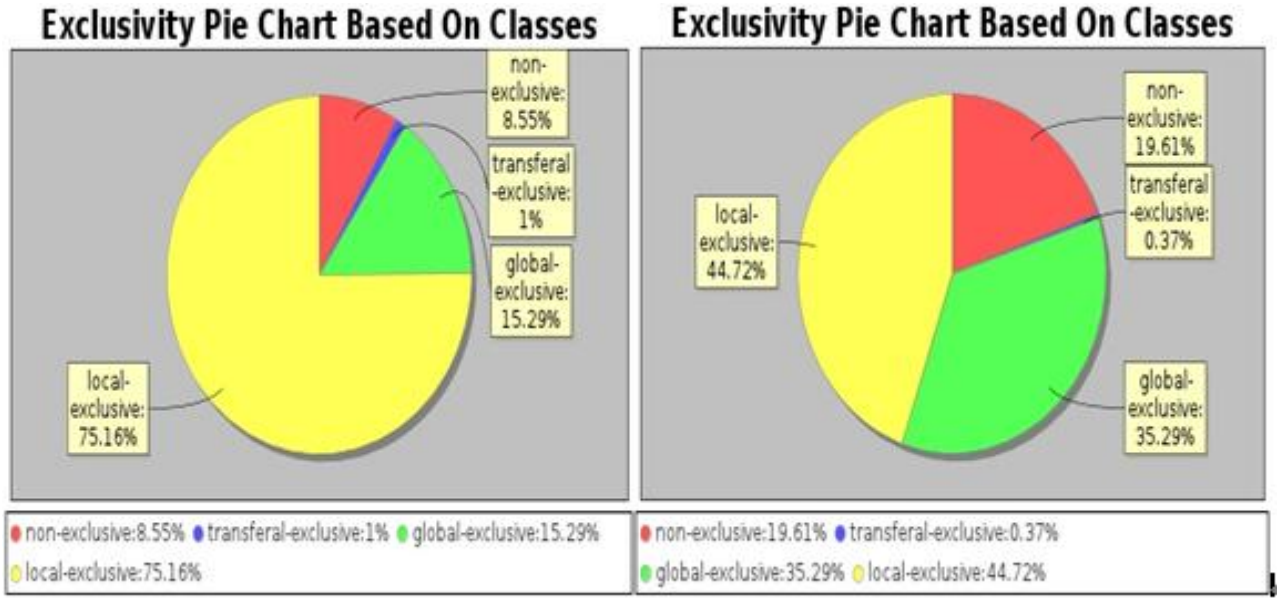

Figure 12 Exclusivity Pie Chart Based on Classes

Figure 11 and 12 shows the exclusivity percentage. In tomcat, there are more local-exclusive relationships. Transferal-exclusive relationships are little in both cases. Percentage of Global-exclusive and non-exclusive relationships in pmd is more than in tomcat.

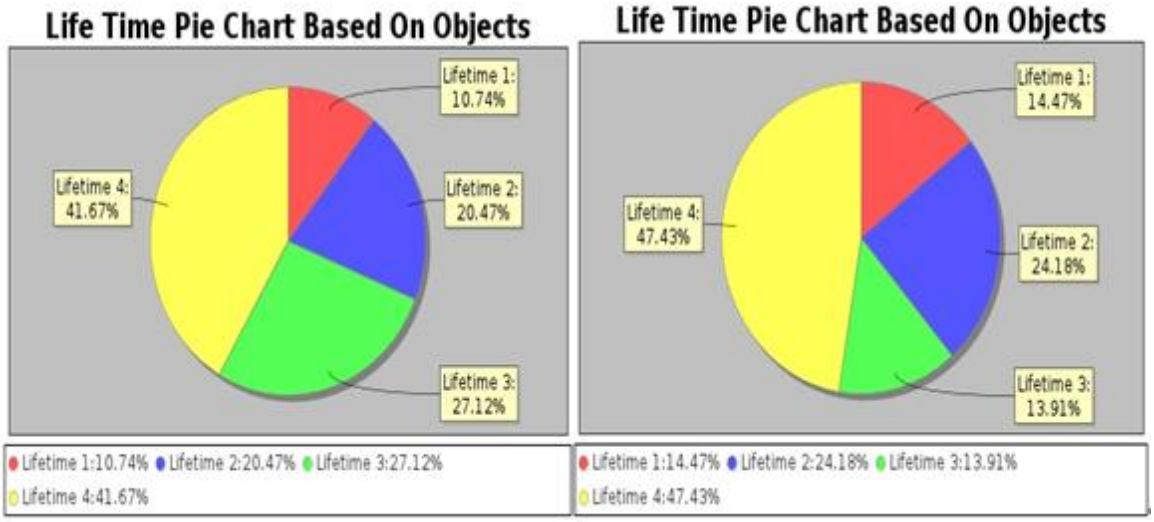

Figure 13 Life Time Pie Chart Based On Objects

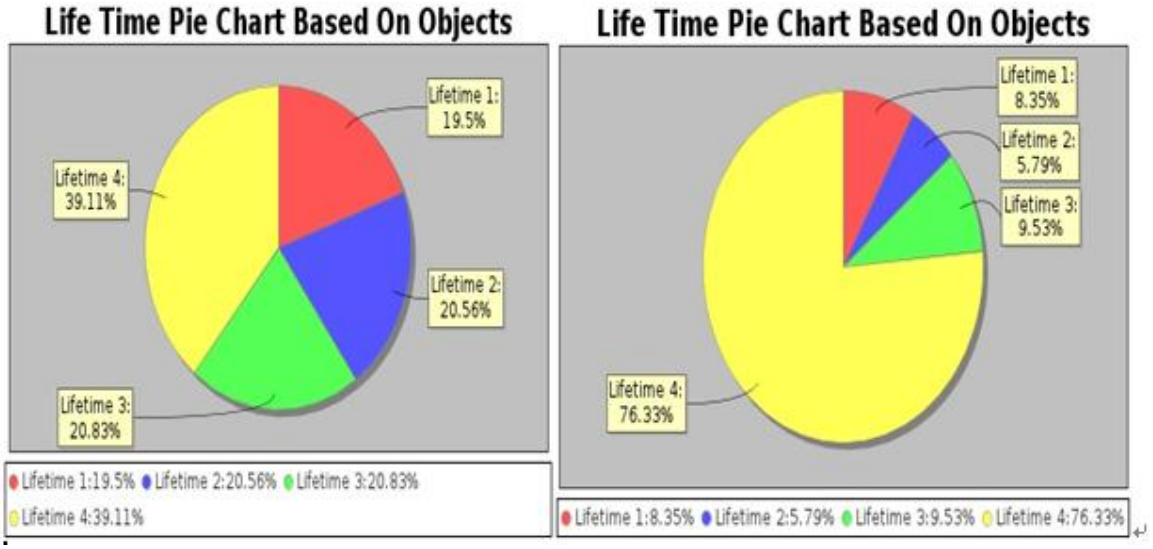

Figure 14 Lifetime Pie Chat Based On Classes

Figure 13 and 14 shows percentage of lifetime types (Figure 10). In tomcat case, percentage of type 1 in objects based chart is less than that in classes based chart. In pmd case, percentage of type 4 in classes based chart is much more than that in objects based chart. Most of relationships here, about 76.26%, is in type 4. Counts of relationships in other three types are nearly the same.

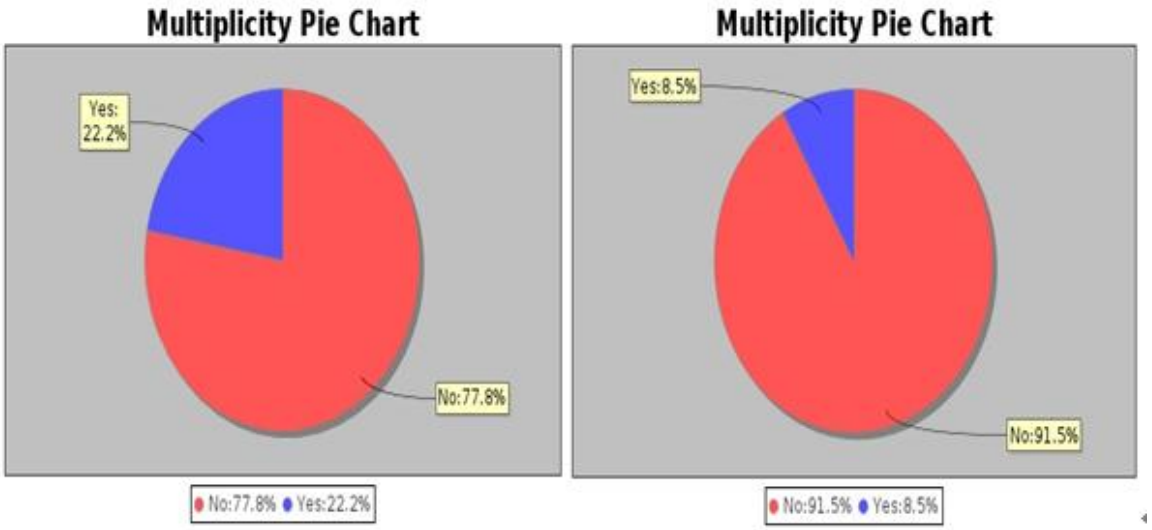

Figure 15 Multiplicity Pie Chart

Figure 12 shows the multiplicity. Only 22.2% and 8.5% is multiple ones, as array or collection in field.

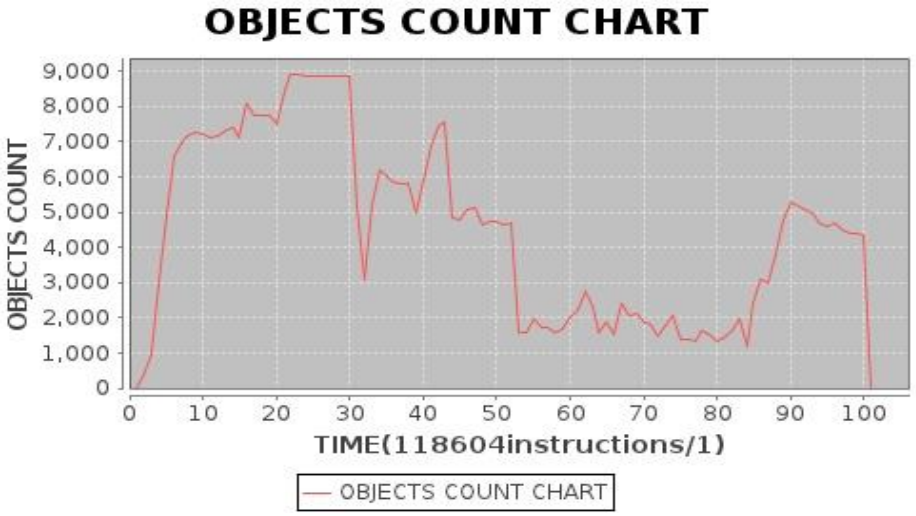

Figure 16 pmd Objects Count Chart

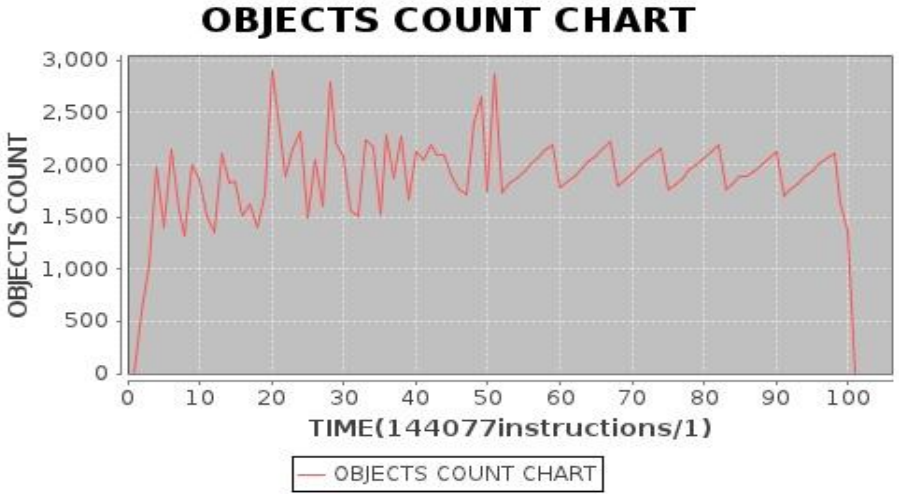

Figure 17 Tomcat Objects Count Chart

Figure 16 and 17 show object counts through life time of program. It is a little strange in pmd case since the analysis_result.txt data is a subset of original one to avoid memory problem. We can see a cycling in the latter half of tomcat case.

6.3 Validity

I've made analysis to lifetime, exclusivity and multiplicity in binary relationships in Java program but have not related them into mereology, the classical Whole-Part relationships. And Existing tools are focused on some normal relationships such as aggregation and composition. It is hard to get validity of the results since it is different from other research. What I can do now is to improve algorithm to make analysis as accurate as possible.

7. Conclusions

In this section, I summarized what I have done till now in this project, as well as what I've learned and experiences I've gained from it.

7.1 Solution Review

This project can be divided into three sub-problems: instrumenting with Java bytecode, analyzing trace files and showing results.

The first step is to instrument with Java bytecode, inserting the proper instructions in the right place to trace the behavior of test programs. This is accomplished with the help of the ASM 3.0 framework. Since some classes are protected by the class loader mechanism in JVM, classes in Java program can be divided into two kinds: classes whose bytecode can be modified and classes whose bytecode cannot be modified. I've instrumented with the former one by modifying methods of such classes. When it came to the latter type, I have to ignore most of them. In a special case, some types such as collection and array should be traced. I can detect invocation of them in methods of former classes. It is hard to cover all Java idioms which have a great impact on Java bytecode instrumenting. Even I've spent a lot of time on it, there are also some problems left here. For example, my instrumenting program will ignore behaviors of inner classes and multiple arrays; many useful methods from collection types have not been analyzed (I've just used **add(Object)** and **remove(Object)** in analysis). Program after instrumenting runs much slower than it used to be as reflection and IO are used often in

tracing operations.

After tracing, analysis is made to trace files. The analysis program builds some data structures to restore instances of a running program and assignments between instances. The assignments are most important part in analysis since they build ownerships up. As showed in Section 8, lifetime, exclusivity and multiplicity of class relationships will be discovered in analyzing. Some problems exist here: Two relationships cannot be looked as the same if owner class is inherited from the other owner class, as well as the child classes. The way how instrumenting program traced fields of a class cause such a problem. There are lots of Whole-Part relationships in a program, making analyzing time-consuming as well as memory-consuming.

Finally, JFreeChart is used to make charts for analysis results.

7.2 Project Review

It is my first time to deal with Java bytecode. I started to instrument Java bytecode on demand after about a half month reading. Many problems appeared such as Verify Errors. Dr. James helped me a lot by giving useful ideas when I was stumped by these problems. I've made some mistakes on schedule. It took too long to instrumenting with bytecode and there is no time left to accomplish the last step, relating relationships to classical mereology. Project management skills should be improved.

7.3 Key Skills

I've learnt a lot from this project. I've learned how the Java code is compiled into Java bytecode, as well as basic mechanism in JVM. I've learned how to use the ASM library to instrument Java bytecode, as well as some important design pattern in this framework such as adapter pattern, decorator pattern and visitor pattern. I use ASM library to modify Java bytecode to trace behaviors of tested program. This is the main part of my project. Reflection mechanism in Java is also used when tracing some special methods. I've used Collections in Java a lot to make analysis. When doing a similar project, all these skills can contribute a lot as they are fundamentals of relationships analysis in OOP.

7.4 Future Work

Future work includes:

- To relate my current results of analysis to classical mereology.
- To take inner classes into consideration which have been ignored in this project.
- To implement tracing other important methods of Collections such as **addAll(Collection) or remove(int).**
- To implement tracing multiple array behaviors.
- To optimize data structure in analyzing program, as well as JVM configurations, as to save memory and time.
- To take inheritance into consideration.
- To get a more accurate lifetime of instances.
- To run programs in a server with a larger memory.
- To solve bugs left when testing with Dacapo Benmark Suit.

7.5 Conclusion

In this project, I've used dynamic analysis to discover some properties from Whole-Part relationships in Java program. ASM library contributes to instrument with Java bytecode and JfreeChart library helps to demonstrate results in charts. Lifetime, exclusivity and multiplicity are proposed to show properties of Whole-Part relationships. To make it straightforward, multiple array and some methods of Collection types, as well as inheritance relationships, are ignored in analysis.

Detection algorithm is tested by Dacapo Benchmark Suit. Till now, six programs from Dacapo Benchmark Suite can be successfully run through with instrumenting program. Since lack of memory, only **tomcat** can be completely analyzed. Instead, I picked out first 200MB of trace files of other tested programs for analyzing. Results of analysis may be less accurate in this way. However, the results of analysis can show pictures of how three properties (lifetime, exclusivity and multiplicity) exist in a program.

8.Acknowledge

I gratefully thank my supervisor James Power who gave me lots of useful advices in this project and Joseph Timoney who assisted me in daily life.

9.Reference

[1] F.Barbier, B.Henderson-Sellers, A.Le Parc-Lacayrelle, J.-M.Bruel: Formalization of the Whole-Part Relationship in the Unified Modeling Language, IEEE Transactions on Software Engineering, 29(5), IEEE Computer Society Press, pp. 459-470, 2003

[2] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot. "Recovering Binary Class Relationships: Putting Icing on the UML Cake". OOPSLA '04 Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 301-314

[3] Brian A. Malloy and James F. Power, "Using a Molecular Metaphor to Facilitate Comprehension of 3D Object Diagrams". IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA, September 20-24, 2005.

[4] Bill Venners. Inside the Java2 Virtual Machine. McGraw-Hill Companies, January 6, 2000, pp. 1-703

[5] Joshua Engle. Programming for the Java Virtual machine. Massachusetts :Addison-Wesley Longman Inc, 1999, pp. 1-512

[6] Eugene Kuleshov "Introduction to the ASM 2.0 Bytecode Framework." Internet: http://asm.ow2.org/doc/tutorial-asm-2.0.html, Oct 17th ,2005 [Nov. 10th 2011].

[7] Tim Lindholm Frank Yellin . "The JavaTM Virtual Machine Specification Second Edition The class File Format." Internet: http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html, Jun 12th ,2005 [Nov. 20th, 2011].