Mereological Analysis of Java Programs by Using Existing Mereological Categorisations

Rezart Veliaj

Dissertation 2013

Erasmus Mundus MSc in Dependable Software Systems

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department: Dr Adam Winstanley

Supervisor: Dr James Power

June, 2013

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of the others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:_____ Date:_____

Acknowledgements

First of all, I would like to give my special thanks to my supervisor Dr. James Power for all his assistance and guidance during the project. Special thanks go also to Dr. Rosemary Monahan for all her support and guidance during the entire Academic year at the National University of Ireland, Maynooth.

Abstract

Mereology (from the Greek μερος, 'part') is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole. Java is an object oriented language where every entity could be represented as an object and the presence of part to whole relations and part to part within a whole are present in different ways. The three most common mereological relationships in Java are the binary class relationships of Association, Aggregation and Composition. These binary class relationships are very easy to represent using the modelling languanges such as UML, but hard to detect in Java source code as there is a discontinuity between the source code of a program in Java and the correspoding representation in UML. This project aims to propose an algorithm that will detect the binary class relationships in Java programs and apply the algorithm in some big open source projects to gather statistics on the three mereological relations taken into account for this project. Statistics will be gathered from the application of the algorithm on the big open source projects. The development of the algorithm will be a step forward to bridging the gap between the representation of the binary class relationships in Java programs and their representation in UML. Besides the statistics gathered from the application of the algorithm on some big open source projects will provide useful information on some of the best practices used to implement binary class relationships in Java. Thus the results obtained from the project will be quite useful to writing better dependable software systems.

Contents

Table of Figures

Chapter 1: Introduction

Problem Statement

The most common mereological[1] categorizations in Java are the binary class relationships of Association, Aggregation and Composition.

The association relationship refers to a relationship between two classes where a certain class knows about and holds a reference to another class (Arabestani, 2000). This type of relationship could be bi-directional where each of the classes holds a reference to the other class and could be implemented in Java as follows:

```
class A {
   String stm= "Class A references class B";

   public Stringprocess(B b) {
         // use a B object
         String tmp = b.process(stm);
   }
}

class B {
   public String process(String s) {
         stm= "Class B was referenced by Class A";
          return stm;
   }
}
```

Aggregation is a "has-a" or "whole/part" relationship where the aggregate class contains a reference to another class and has a sort of ownership over the class that it contains. On the other hand the referenced class is "part-of" the aggregate class. In the aggregation relationship there should not be any cyclic references and in case class A references class B and class B on the other hand references class A, then there is a cyclic relationship and no ownership can be determined (Blaha & Rumbaugh, 2004). An example of an aggregation relationship could be the Student-Module relationship. In case Student is a class that contains the information about the students, and there is a Module class that contains the information about particular modules, i.e. Module title, description and the number of credits and a relationship is defined so that a student object should contain at least a module object then the module object is contained in the student object. In this case the module object is part-of the student object and it can be said that a student object *has-a* module object. Therefore the student object is the owner of the module object. This relationship in Java could be implemented as follows:

[1] Mereology (from the Greek μερος, 'part') is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole.

Example of an aggregation relationship between a Student class and a Module Class:

Example:

```
public class Module{

  private String name, description;

   private int credits;

}

 public class Student {

  private Module[] modules = new Module[10];

}
```

The composition relationship is based on the aggregation relationship, but it takes aggregation a step further, by making sure that the owner object is responsible for the lifetime of the objects that it holds (Ramnath & Dathan, 2011). Therefore if the object B is contained within object A, then object A is responsible for the lifetime of the object B. In the case of composition object B cannot exist without object A. An example of a composition relationship is the car and engine.

Example

```
public class Car {
        private Engine engine;

        public Car( Engine engine) {
                this.engine = engine;
        }
}

public class Engine {
        private int capacity;
        private int serialNumber;

        public Engine(int capacity, int serialNumber) {
                        this.capacity = capacity;
                        this.serialNumber = serialNumber;
        }
}
```

A recurrent problem with such binary class relations in Java is their detection in Java programs. Such analysis and detection of the part-whole relations in Java programs is not an easy and straightforward task due to a lack of such tools that provide a thorough detection and analysis of the binary class relationships in Java and the discrepancy that exists between the representation of the Java programs in modelling languages such as UML and the code itself. Therefore, if the code would be reverse engineered to produce UML diagrams from Java code, relations would get lost and we would not be able to see the big picture (Gueheneuc & Albin-Amiot, 2002). This discrepancy hinders the traceability between software implementation and design and consequently slowing down software analysis. As the discrepancy exists between the implementation of the software designs and their implementations makes it difficult to detect the mereological relations in Java programs by solely doing reverse engineering, algorithms are needed to be developed to detect such relations in Java programs.

In addition an application of the algorithm that detects binary class relationships in big open source projects would reveal some of the best practices used to implement such relationships and the usage frequency of such relationships in java source code.

Motivation

Java offers an entirely object-oriented approach to programming as almost every entity could be represented as an object. Therefore the mereological relations in Java objects are present and common in different ways (Arnold, Gosling, & Holmes, 2005). It would be very helpful to detect such relationships in Java programs as reverse engineering could be used to create design models from the Java source code. The most common types of mereological relationships in Java programs are the binary class relationships of Association, Aggregation and Composition and being able to detect such relationships in Java programs would be very helpful to generate the design models from the source code.

There has been research to solve the discrepancy that exists between the programming languages and the design modelling languages such as UML, but there does not yet exist a tool that could do a full representation of Java source code into UML. Besides being able to detect such relationships in big open-source projects would be very beneficial as the programmers who contribute in such projects are usually professionals with a sound knowledge of the language and best practices could be extracted on the patterns that are mostly use to represent mereological relations in Java programs.

The topic is highly relevant to writing dependable software systems as the outcome of the project will be the detection of some of the mereological relationships in Java programs and the application of the detection algorithms in some big open source projects to gather statistics on usage, code patterns, etc that are most commonly used in some big open-source projects, developed in the Java programming language. The results obtained will be very helpful to writing better dependable software systems.

Aims and Objectives

The aim of this project is to detect the binary class relationships of Association, Aggregation and Composition in some big open source projects in Java by using the ASM framework. In order to achieve the aim of the project, the following objectives need to be accomplished:

- Gain an understanding of the binary class relationships of Association, Aggregation and Composition in terms of their properties.
- Gain an understanding of the methods used for Java analysis such as: Control Flow Graph, Call Graph, Data Flow Analysis, Pointer Analysis and bytecode analysis with the ASM framework.
- Get to know the current methods and tools used to detect the binary class relationships in Java.
- Develop an algorithm to detect the binary class relationships of Association, Aggregation and Composition in Java programs by using binary class relationships properties.
- Apply the algorithm on some big open source projects and gather statistics on the results obtained.

Report Structure

The report is organized in the following six chapters:

Chapter 1 contains problem statement, motivation behind this project and the aims and objectives of the project.

Chapter 2 provides the problem context and information on related work on the topic and an overview of the current methods and tools that could be used for the solution of the problem.

Chapter 3 contains the solution of the problem stated in this project. It provides the design and the implementation of the algorithm that detects the binary class relationships in Java programs. In addition, it does also provide an overview of the ASM framework as the algorithms developed to detect the mereological relations were built by using the ASM framework.

Chapter 4 contains an example as a proof of concept where the static and dynamic analysis of the program was conducted by using programs that do the static and dynamic analysis of the program.

Chapter 5 contains statistics gathered from the implementation of the algorithms on some big open source projects and evaluation of the project, what was achieved and the limitations of the algorithm in detecting binary class relationships.

Chapter 6 contains conclusions of the project and what could be done in the future on the project.

Chapter 2: Related Work

Introduction

The project will start with a research in the field of mereology, to get an understanding of the whole-part relationships as described and defined in mereology. The aim of the project is not only to find the different kinds of mereological whole-part relationships, but also how to apply these findings on the programming language of choice, Java in this case. Therefore the literature review for the project will not only encompass a literature review in the field of mereology, but also on how mereological relationships can be specified so that they could be easily detected in Java code. Java code will not be used as purely Java for detecting the mereological relationships, but instead tools that convert Java code into bytecode will be used and an analysis will be performed on the bytecode of Java code. The literature review for the project will comprise a research in the area of mereology and mereological relationships, whole-part relationships in UML diagrams, how the mereological relationships can be defined, the methods used to analyse the programs such as call graphs, control flow graphs, data flow analysis, pointer analysis and the ASM framework that can be used to convert Java code into Java bytecode and offer methods that can be used to analyse the bytecode itself.

Context

UML has a number of ways of describing relationships between classes, associations, dependencies, generalizations, realizations, etc. Association is used to denote the ability of one instance to send messages to another instance. This is a type of structural relationship between the two model elements where one classifier that might be an actor, use case, class, interface, etc can connect to the objects of another classifier (Miles & Hamilton, 2006). Aggregation is used to denote the whole/part relationship. Aggregation is similar to an association, except for the fact that instances cannot have cyclic aggregation relationships, i.e. a part cannot contain its whole, or a classifier cannot be part of itself, but a part of or subordinate of another classifier (Tan, Hao, & Yang, 2003). Composition is quite similar to Aggregation, with the exception that the lifetime of the ‘part’ is controlled by the ‘whole’ (Keet C. M., 2006). The control to the part might be direct or transitive. There is a direct control of the whole on the part when the ‘whole’ takes direct responsibility for creating or destroying the ‘part’ and there is an indirect control when it accepts an already created ‘part’ and passes it to some other ‘part’ that has the complete responsibility for the ‘part’. The composition relationship stands for a time of relationship between the ‘whole’ and the ‘part’ when the lifetime of the ‘part’ classifier is dependent on the lifetime of the ‘whole’ classifier (Miles & Hamilton, 2006). Generalization relationship is used to denote that an entity is based on a general model parent. One entity can be based on more than one parent, and on the other hand from one single entity, many other entities could be derived. In UML diagrams, several classes can constitute a generalization set of another class and many classes could have only one parent class (Miles & Hamilton, 2006).

When forward engineering the code this can be implemented in various ways, depending on what programming language you're using, and what tool you use to generate the code.

However, it is reasonable to assume that the connections between classes in a Java program are much more complex that just these patterns. There's actually a whole subject, called "mereology", that's dedicated to studying and categorising whole-part relationships. The word “mereology” derives from the Greek words: μέρος, root: μερε(σ)-, "part" and the suffix -logy "study, discussion, science") and is centred on the parts and the wholes they form. Mereology emphasizes the relation between the entities and differs from the set theory that is centred on the relation between a set and its elements (Stanford, 2009). Contemporary formulations of mereology grew out of the recent theories, formulated by Leśniewski, Leonard and Goodman (Stanford, 2009) and classical mereology constitutes a formal theory of the part-whole relation (Hovda, 2009).

Problem Explanation

Mereological relations in Java

The investigation of the part-whole relationship, Mereology dates back in the beginning of the 20th century and it was introduced by Lesniewski. Since it was first introduced, it has been greatly expanded with ongoing research from the 1980s with the research and publications of Peter Simons and Achille Varzi as two prominent figures. Before determining mereological relations in Java, a summary of the main basic concepts of mereology will be given. Mereological relations are reflexive, anti-symmetric and transitive.

1. Everything is a part of itself:
   $part\_of(x, x)$

2. Two distinct things cannot be part of each other, or when x is part of y and y happens to be part of x then x and y should be the same thing:
   $(part\_of(x, y) \wedge part\_of(y, x)) \Rightarrow x = y$

3. If x is part of y and y is part of z, then x is part of z:
   $(part\_of(x, y) \wedge part\_of(y, z)) \Rightarrow part\_of(x, z)$

UML Representation vs. Implementation

There has been on-going research to solve the discrepancy and discontinuity that exists between the implementation of the binary class relationships in Java source code and their modelling in UML. Such solutions are centred on the definition of the binary class relationships with respect to their properties and the detection of the different binary class relationships based on such properties.

There has been research of how to bridge the gap between the design of a program in UML and the counterpart development in Java. One such research is the one presented by Yann-Ga ël Guéhéneuc and Hervé Albin-Amiot in the paper “Recovering Binary Class Relationships: Putting Icing on the UML Cake” (Gueheneuc & Albin-Amiot, 2002). The research focuses

on the definitions of the binary class relationships at design level, definition of the properties in binary class relationships at implementation level and redefinition of the binary class relationships in terms of the properties in binary class relationships at implementation level. At design and implementation level there are three binary class relationships: association, aggregation and composition.

At design level, in a UML diagram, an association relationship is presented as a link between two classes and there is no limit on the number of the instances that can be involved in an association relationship. At the implementation level, an association relationship involves instances of two classes where one of the classes is the origin class and the other class is the target class (Kollmann & Gogolla, 2001). Therefore, an association relationship between two classes A and B is the ability of an instance of class A to send a message to an instance of class B. The instances of A and B could be linked together via other binary relationships i.e. association, composition and aggregation (Gueheneuc & Albin-Amiot, 2002).

An aggregation relationship is a relationship of whole and part where the instances of one of the classes are part of the instances of the other class. At implementation level, there is an aggregation relationship between two classes when the definition of one class contains instances of the other class (Noble & Grundy, 1995). As such, the whole needs to define a field of the type of its part that might be simple, array or collection (Gueheneuc & Albin-Amiot, 2002).

A composition relationship at design level is an aggregation relationship with the difference that all its parts are destroyed when the whole is destroyed. A composition relationship at implementation level is defined as an aggregation relationship with a constraint between the lifetimes of the whole and its parts (Marcos, Vela, & Caceres, 2001). Composition relationship allows only one association between the whole and the part to make sure that the lifetime property is preserved (Gueheneuc & Albin-Amiot, 2002).

Properties of the Binary Class Relationships

Binary class relationships have been analysed in terms of the four following properties: Exclusivity property, Invocation-site property, lifetime property and multiplicity property.

Exclusivity property

Exclusivity property is related with the number of instances of class B that can be involved at the same time with the instances of class A. Exclusivity property holds if an instance of class B can be involved with only one instance of class A at the same time. In case more than one instance of Class B is involved with more than one instance of class A, then the exclusivity property does not hold (Gueheneuc & Albin-Amiot, 2002). Exclusivity property either holds or it does not hold and could be expressed via the following notation:

$EX: Class \times Class \rightarrow B$ where given two classes A and B, $EX\ (A, B) \in \{true, false\}$.

EX (A, B) is true if only one instance of B can be involved with only one instance of A at the same time and *EX (A, B)* is false if one instance of class B is involved with more than one instance of class A at the same time. An example would be the car and wheels. A wheel can be part of only car at the same time, therefore the exclusivity property holds. Another example would be a person and his hobbies. A person can have more than one hobby at the same time, therefore the exclusivity property does not hold between a person and a hobby.

Invocation Site property

Invocation Site property is concerned with the messages that instances of class B send to the instances of class A. Instances of class B could send messages to instances of class A via different invocation sites, i.e. field, parameter and local variable (Gueheneuc & Albin-Amiot, 2002). The invocation site property can be denoted as:

$IS: Class \times Class \subseteq any$

When given two classes A and B the IS (A, B) describes those invocation sites for messages sent from the instances of class A to the instances of Class B. In case there is no message sent from the instances of class A to the instances of class B then *IS* (A, B) = Ø or the messages sent from instances of class A could be sent to the instances of class B via a field that might be either a parameter or a local variable of type B, an array field or a field of type collection (Gueheneuc & Albin-Amiot, 2002).

Lifetime Property

Lifetime property is related with the lifetime of the instances of class B with regard to the lifetime of the instances of class A. In case the lifetime property holds then the lifetime of the instances of class B depends on the lifetime of the instances of class A. In programming languages where there is a garbage collector such as Java, the lifetime of the instances is till when the instances are ready to be collected for garbage. Lifetime property could be described with the following property:

LT (A, B) =+ if all the instances of class B are destroyed before the instances of the class A and *LT (A, B)* =- if the time of destruction of the instances of class B is unrelated to the destruction of the instances of class A (Gueheneuc & Albin-Amiot, 2002).

Multiplicity Property

Multiplicity property is used to specify the number of the instances of class B that are allowed to be in a relationship with a certain instance of class A. This property can be denoted by the notation:

$MU: Class \times Class \subset \mathrm{N} \cup \{+\infty\}$ where given two classes A and B: MU (A, B) $\subset \mathrm{N} \cup \{+\infty\}$.

An interval with the minimum and the maximum number is used to show the number of the instances of class B that can be in a relationship with the instances of class A.

Redefinition of the Binary Class Relationships

Binary class relationships will be redefined in terms of the four properties: Exclusivity property, Invocation site property, Lifetime property and Multiplicity property. In the case of

association relationship, exclusivity property does not hold between the instances of class A and the instances of class B and the invocation site property could be any, but there is no invocation site present in the instances of class B. In addition there is no lifetime restriction of the instances of class B when the instances of class A are deleted or collected by the garbage collector and the multiplicity between the instances of class A and B could be from 0 to infinity. All these properties are listed below with notations as they were defined in the previous paragraphs (Gueheneuc & Albin-Amiot, 2002).

AS (A, B) =

EX (A, B) $\in$ B
IS (A, B) = *any*
LT (A, B) $\in^{-}$
MU (A, B) = [0, $+\infty$]

In the case of the aggregation relationship exclusivity property does not hold between the instances of class B and class A and the lifetime of the instances of class B does not depend on the lifetime of the instances of class A. On the other hand the invocation site property holds between the instances of class B with respect to the instances of class A and it could be {field, array field, and collection field}. Aggregation relationship between the instances of class A and B with respect to the instances of class B related to the instances of class A could be described via the following notation (Gueheneuc & Albin-Amiot, 2002):

AG (A, B) =

EX (A, B) $\in$ B
IS (A, B) $\subseteq$ {field, array field, collection field}
LT (A, B) $\in^{-}$
MU (A, B) = [0, $+\infty$]

In the case of the composition relationship exclusivity holds between the instances of class B with respect to the instances of class A as they are part exclusively of the instances of class A. Also the lifetime property holds true as the lifetime of the instances of class B depends on the lifetime of the instances of class A, so that when the instances of class A are destroyed or collected by the garbage collector, they are collected before the instances of class A. Composition relationship between the instances of class A and B with respect to the instances of class B related to the instances of class A could be described via the following notation (Gueheneuc & Albin-Amiot, 2002):

CO (A, B) =

EX (A, B) = *true*)
IS (A, B) $\subseteq$ {field, array field, collection field})
LT (A, B) = +
MU (A, B) = [1, $+\infty$]

Explanation of Program Analysis Techniques

Some of the techniques used to analyse Java programs are: control flow graphs, data flow analysis, call graphs, pointer analysis and frameworks such as ASM 4.0.

Control Flow Graphs

A control flow graph is used to represent all the paths that might be traversed in a program during its execution by using graph notation.

The graph is composed of nodes and edges where each node represents a basic block, which is a piece of code without any jumps or jump targets. The jump targets are used to start the block and the jumps are used to end the block (Hubicka, 2003). Jumps are represented in the control flow graphs with directed edges. In the control flow graphs, there are two special blocks; the entry block and the exit block. Control enters through the entry block in the flow graph and it leaves the graph through the exit block. Control flow graphs are essential and widely used for compiler optimizations and static analysis tools.

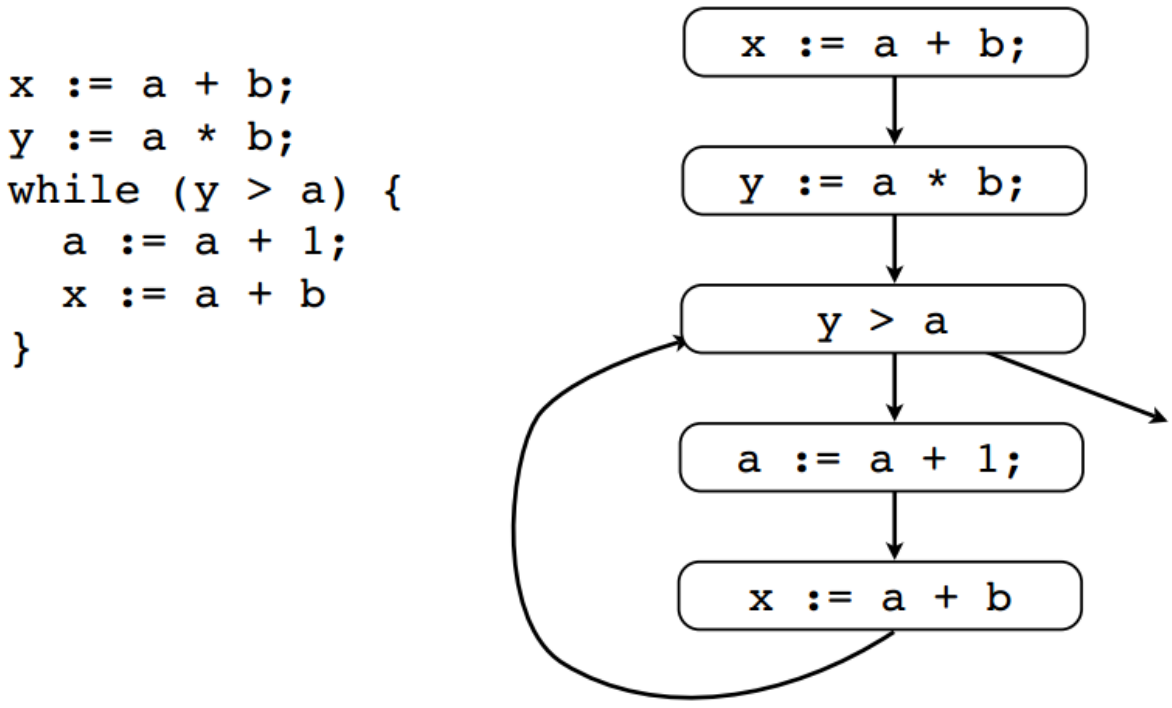


Figure 1 - Example of a Control Flow Graph (Foster, 2011)

Another example of a Control Flow Graph is given below:

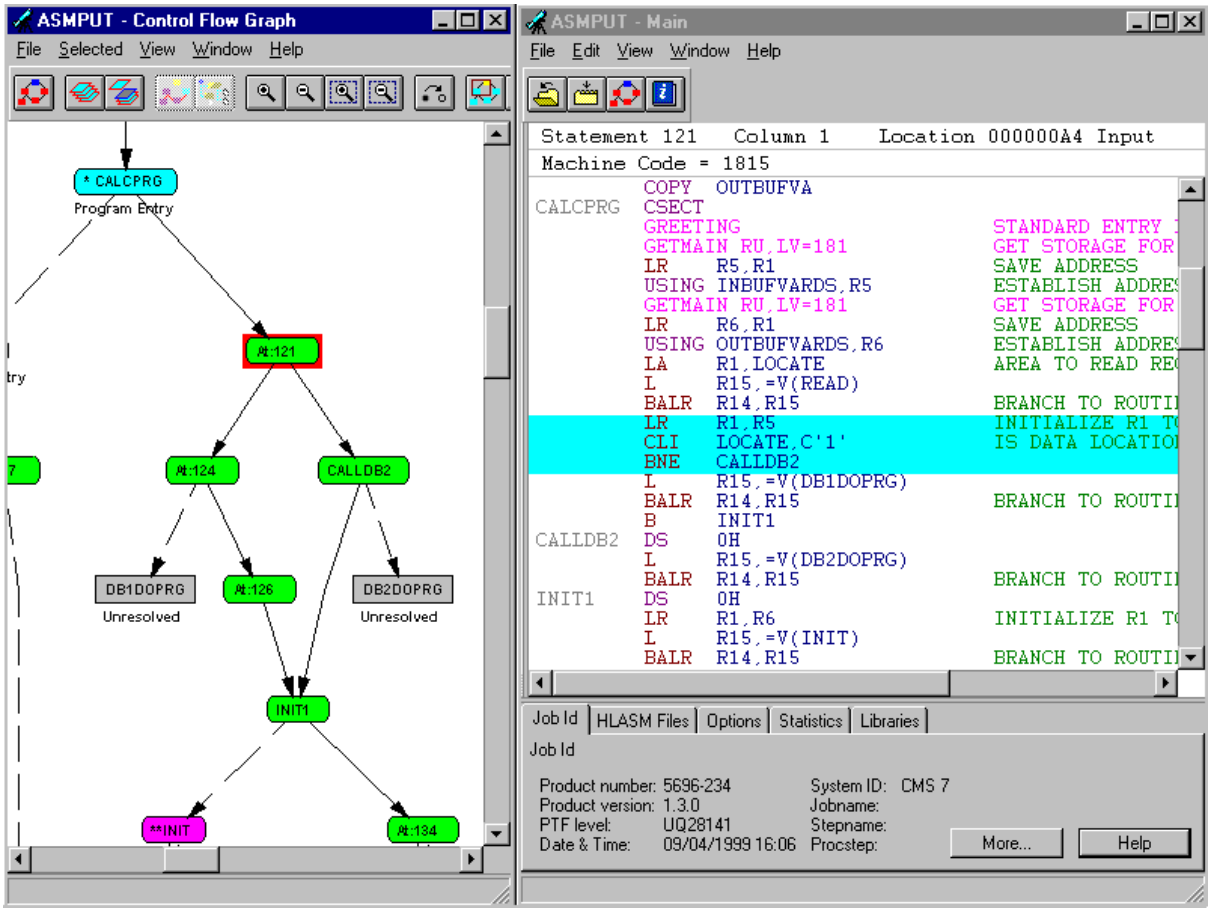


Figure 2 - Control Flow Graph (Corporation, 2010)

Data flow analysis

Data flow analysis is a technique used for gathering information about the possible set of values calculated at various points in a computer program (Foster, 2011). Data-flow analysis is used to derive information about the dynamic behaviour of a program, solely by investigating and analysing the static code. The direction of data-flow analysis could be forward or backward. Program analysis is used to discover information about the programs and represent their dynamic behaviour without executing the program. In order to represent the dynamic behaviour, it needs to represent all the execution instances of the program. A picture is given below to illustrate the difference between static and dynamic analysis.

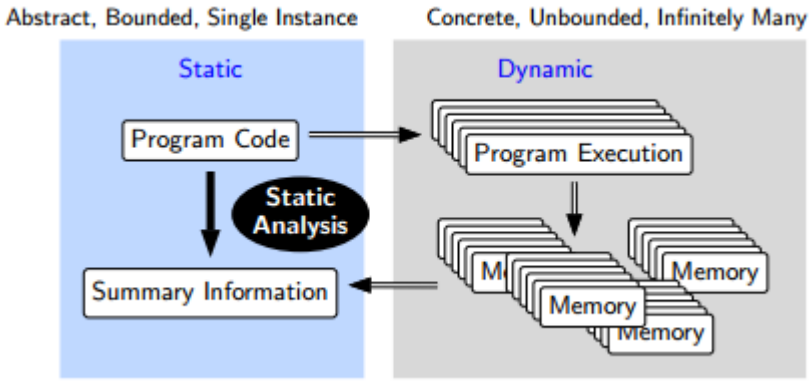


Figure 3 - Static vs dynamic program analysis

Data flow analysis is very useful in code optimization, verification and validation, software engineering and reverse engineering (Khedker, Sanyal, & Karkare, 2009). The information gathered from data flow analysis could be used to improve the time of running the program which results in power efficiency as well. The information obtained from data-flow analysis might as well be used for program verification and validation, i.e. guarantee that the program will never divide a number by 0, will never dereference a null pointer, will close all the open files, etc. In addition, the information acquired from data-flow analysis is very useful in software engineering, i.e. maintenance, bug fixes, enhancements and migration. Another useful use of data-flow analysis is in reverse engineering that is used to understand the program (Khedker, Sanyal, & Karkare, 2009).

Live and *Reachable* are two notions that should be understood when performing data-flow analysis. An object is considered to be reachable, if it is referred by a root or if not referred by a root, it is referred by a reachable object, so that it can be reached from the roots by following references. In Java programming language, the reference objects are used together with the ordinary references and finalization to generate information used by the garbage collector of what to do when an object is about to die (Ravenbrook, 2000). An object is considered to be live or active if the program will be able to read from it in the future. The term *live* is more usually referred to those objects that are *reachable*. As it is not possible for the garbage collectors to determine if the objects are live, they get this information indirectly by detecting those objects that are provably dead, i.e. those that are not reachable (Ravenbrook, 2000). When an object is unreachable, then its memory could be reclaimed. Although *live* is mostly used to refer to a *reachable* object in programs, *live* and *reachable* object is not exactly the same thing. Determining live objects in a program is very useful as even with the state of the art garbage collection, 24% to 76% of unused memory remains unclaimed (Khedker, Sanyal, & Karkare, 2009)

Some useful examples of data-flow analysis are reaching definitions, live variable, available expressions and very busy expressions. Reaching definitions is used in constant propagation, live variables is used for dead code elimination, available expressions is used to avoid re-computing expressions and very busy expressions is used to save the expressions for later use.

Most of the data-flow analyses have the same structure as they "interpret" the statements in a program and collect the information as they proceed. It would be ideal to perform "perfect interpretation" and be able to collect exact information about the way a program executes, but as the program inputs are not available at compile time and the compiler is not going to run the program to completion during compilation as it just takes too long, it is almost impossible to have a perfect interpretation (Princeton, 2003). Therefore the program is not interpreted exactly, but the behaviour of the program is approximated and is called abstract interpretation. Abstract interpretation is very useful in understanding different program analyses. Analysis via abstract interpretation is performed by the means of a transfer function, a joining operator and a direction. The transfer function f(n) is used to simulate or approximate the execution of the instruction n on its inputs. The joining operator is used to deal with the interpretation of the "if" statements. As it is not known which branch of an "if" statement will be taken into account at compile time, both branches of the "if" statement will be interpreted and the results will be combined. The last remaining part of the abstract interpretation is the direction that could be either forward or backward (Princeton, 2003). In the forward direction, the inputs are obtained from the previous instructions and the in the backward direction, the inputs are obtained from the successor instructions.

Reaching definitions is used to statically compute those definitions that reach a given point in the code and does so by using forward analysis. A definition is considered to reach a program at a point n, if there is a control-flow path from the start node to the point n of the program that does not contain a redefinition of the definition taken into account (Aho, Sethi, & Ullman, 1986). The following examples provide an illustration of the reaching definitions analysis:

Example 1

a1→x: = 2

a2 → y: = x

Example 2

a1→x: = 2

a2 → x: = 3

a3→ y: = x

In the first example a1 reaches a2, but in the second example a1 does not reach a3 as x is redefined at a2.

Live variable analysis or liveness analysis is a data-flow analysis use to compute at a certain point in the program, the variables that may be potentially read before their next write. The set of live variables is determined by using backward analysis (Aho, Sethi, & Ullman, 1986). The following example illustrates live variable analysis.

a1 → a: = 3;

a2 → b: = 5;

a3 → c: = a + b;

The set of the live variables at a3 is {a, b} because these variables are both used in the addition operation, but the set of live variables at a1 is {a}, since the variable b is only defined at a2.

Available expressions analysis is used to determine the set of expressions that do not need to be recomputed in a program. Such expressions are said to be available at such point of the program. The expression should not be modified on any path from the occurrence of the expression to the program point, in order to be available. The available expressions analysis is an example of a forward data-flow analysis. An expression is considered to be available at the start of a basic block in case it is available at the end of the basic block's predecessors (Aho, Sethi, & Ullman, 1986). Available expression analysis could be illustrated as follows: if there is an expression x ⊕ y that occurs in the program, then the expression x ⊕ y is available at point n, if when computed along every path from the entry node to the program point n, neither x nor y have been modified after the last evaluation of x ⊕ y.

Very busy expressions analysis is a variant of available expressions analysis. An expression is considered to be very busy at a point in a program when it is sure that it will be computed at some time in the future. Therefore when starting at the point in question the expression should be reached before its value will change. Very busy expressions analysis constitutes a backward data-flow analysis since is provides information about future evaluations by conducting a backward analysis to earlier computations in the program (Aho, Sethi, & Ullman, 1986).

Call Graphs

A call graph is a graph with nodes and edges where each node represents a function or method defined in the program and an edge represents a call from one method to the other. The call graph is the starting point of the interprocedural analysis. A call graph is a directed graph where for each method defined in the program there is a method and there is an edge from method m1 to method m2 that shows that method m1 is calling method m2 (Microchip Technology Inc, 2012).

Call programs are used as a basic program analysis to get a human understanding of the programs or use the information obtained for further analysis. One such analysis could be the

track of the flow of the values between procedures. Another application of the call graphs would be finding those methods or functions that are never called during the program execution. Call graphs could be either dynamic or static. A dynamic call graph is obtained from the execution the actual program. Therefore a dynamic call graph can be exact, but it only describes only one run or instance of the program. On the other hand, a static call graph is used to describe every possible run of the program. Thus the exact static call graph is un-decidable as call graph algorithms are approximations. In the static call graph, every call relationship that happens is represented and most probably some call relationships that would never occur in the actual runs of the program are presented as well. Call graphs can be constructed to represent various degrees of precision and a more precise call graph approximates more precisely the behaviour of the real program. On the hand generating a more precise call graph requires more time and memory usage.

Explain how to generate call graphs in Java (Java profiling)

Pointer Analysis

Pointer analysis aims to answer the question of "which objects does the pointer p point to?" and the "points-to" is the set of all objects that a pointer can point to (Harvard School of Enginnering and Applied Sciences, 2011). Pointer analysis is very useful during program analysis as the points-to information is quite useful when deleting objects that are not deleted by the development environment as in C++ where the programmer needs to take care or the pointer as there is not garbage collector as in the case of Java. Also pointers are used to represent composition relationships. *Points-to* information in ASM is found in the pts (v; h) where v points to the object h and hpts(h; f; g) where the field f of object h points to object g. In this notation v is a variable name, f is a field name and g and h are line numbers of the allocation sites.

Points-to analysis is one of the most essential static program analysis and it contains all the data that a pointer could reference during the program execution. This type of analysis constitutes the foundation for almost any other program analysis and it is closely related with call graphs (Smaragdakis & Bravenboer, 2009).

There are many dimensions to pointer analysis, i.e. Andersen analysis, Steensgard analysis, One-level flow and Pointer analysis for Java (Harvard, 2011). Andersen and Steensgard analyses are flow insensitive analyses. Andresen style analysis is considered to be the most precise and the slowest analysis, while Steengard's analysis is considered to be fastest but least precise analysis. The analyses are considered to be flow insensitive as they ignore the control-flow graph and assume that the statements in a program could be executed in any order. Therefore, the whole idea behind flow insensitivity is to produce a single solution that holds valid in the entire program, rather than producing a solution for each program point. The general principle behind Andersen's analysis is to view pointer assignments as constraints and then use those constraints to propagate points-to information (Harvard, 2011). An example is given below to illustrate Andersen's pointer analysis.

Flow sensitive pointer analysis

x := &y	x → { y }
z := x	z → { y }
x := &a	x → { a }
b := x	b → { a }

Andersen's flow insensitive analysis

Program	Constraints	Points-to relations
x := &y	x ⊇ { y, a}	x → { y, a}
z := x	z ⊇ x	z → { y, a}
x := &a	b ⊇ x	b → { y, a }
b := x		

Steensgard analysis is a constraint-based analysis as well that uses equality constraints instead of subset constraints and it can be efficiently implemented using UnionFind algorithm (Harvard, 2011).

Datalog

Datalog is a declarative programming language that with respect to syntax is a subset of Prolog. The points-to analysis provides a set of facts of the form pts(v; h) and hpts(h; f ; g) (Smaragdakis & Bravenboer, 2009). The datalog on the other hand contains a number of rules that are used to derive facts regarding the derived relations.

Summary

This chapter contained information about the areas that are related with the implementation of the project. At first, the problem context was given, followed by a literature review on the fields that are related with the development of a solution for the problem given in the context. As the aim of the project is the detection of mereological relations in Java programs, the literature review conducted on areas that provide a solution to the problem was given. Therefore, literature review was conducted on the areas of mereological relations in Java and how they can be detected and the explanation of the program analyses techniques such as: Control Flow Graphs, Data Flow Analyses, Call Graphs, Pointer Analyses and Datalog.

Chapter 3: Solution

Introduction

This chapter will contain the information on the platform used, the ASM 4.1 framework and the design of the algorithm used to analyse the mereological relations in Java programs. At first, the ASM framework and how bytecode analysis is performed by using the ASM framework will be introduced, followed by the design and implementation of the algorithm. The algorithm is designed in a way to detect the properties of Multiplicity, Method Invocation, Lifetime and Exclusivity. The properties are detected by using both a static and dynamic analysis. The properties of multiplicity and method invocation are detected via static analysis and the properties concerning lifetime and exclusivity are detected via dynamic analysis. The algorithm that performs static analysis will be introduced first, followed by the introduction of the dynamic analysis.

Bytecode Analysis with ASM

The word "ASM" itself does not stand for any abbreviations and does not mean anything on its own, but it is just a reference to the __asm__ keyword in C programming language that makes it possible to implement some functions in the assembly programming language (Bruneton, 2011). ASM relies on a new approach that consists in using the "visitor" design pattern. The use of this design pattern gives much better performance than the current existing tools as it does not need to explicitly represent the visited tree objects and is more useful for practical needs. The framework is able to modify existing classes or dynamically generate classes.

Why ASM framework?

There are other frameworks alongside ASM used for analysing, generating and transforming compiled Java classes, but ASM is the most recent and most efficient one. Efficiency over the other existing frameworks made ASM framework the tool of choice for this project. Besides efficiency ASM framework exhibits the other following features that make code analysis easier (Bruneton, 2011):

- It comes with an API that is simple, well designed and modular.
- It provides support for Java 7, the latest Java version.
- It is small, but very fast and robust at the same time.
- It has a large user community behind. The large user community is very helpful as it can help a lot new users.
- ASM is very well documented and has a plug-in for Eclipse.
- Moreover, it has an open source license that gives any end user the freedom to use it any way they want to.

ASM library is designed to read, write, transform and analyze Java classes, but is cannot deal with class loading. Java classes are represented as byte arrays and that is why ASM provides tools to read, write and transform byte arrays by using higher level concepts other than bytes,

such as numeric constants, strings, Java class structure elements, Java identifiers, Java types, etc.

ASM Model and Organization

ASM library comes with two APIs for analysing, generating and transforming Java compiled classes: the core API and the tree API. These two APIs differ as the first API gives an event based representation of the classes, while the tree API gives an object representation of the Java compiled classes (Debasish Ray Chawdhuri, 2012). In the core API that follows the event based model, a class is represented by following a sequence of events, where each of the events represents an element of the compiled Java class, such as the class header, annotation, field or method. Therefore the event based API gives the set of the possible events and the order they should occur. A class parser is present in this API that generates an event for each parsed element of the compiled Java class. The core API provides the opposite as well, generation of a Java compiled class file from a sequence of such events. The opposite is achieved by using a class writer. The object based API is built on top of the event based API, so given the sequence of events, the object based API will build the object based model for the class. A class can be represented via an object based model by representing the class as a tree of objects where each object is used to represent a part of the class that might be a field, a method, etc (Bruneton, Lenglet, & Coupaye). Each object contains references to those objects that represent its constituents. The object based API is a way to convert the sequence of events for a particular compiled Java class to an object tree of that class. The analogy of the object based API is DOM (Document Object Model). Each API has its own advantages and drawbacks and depending on the scope and the purpose they are going to be used for, the end user makes a decision. The most common distinction between the two is the memory that is required by each API, performance and the information provided by each (Bruneton, 2011). The event based API is faster as it requires less memory as there is no need to store and create in memory a tree of objects. On the other hand, as the object based API creates and stores in memory a tree of objects for the entire class, the entire class is in memory and in the case of the event based API only the element corresponding to the current event is in memory.

ASM bytecode framework is implemented in Java programming language. It makes use of a visitor-based approach, used to generate bytecode and make the transformations of the existing classes in a Java project (Kuleshov, 2004). The framework hides the complexity from the developers and is characterised by a better performance compared to other tools such as BCEL, SERP or Javaassist (Kuleshov, 2005). ASM framework is composed of several packages that allow for flexible bundling. The packages are given in the figure below:

Figure 4 - ASM Packages (Kuleshov, 2005)

The packages are given in three layers. Layer 1 contains the core package, Layer 2 contains Tree, Commons, Util and XML packages and Layer 3 contains the Analysis package. The Core package provides the API calls to carry out core operations with the framework such as: read, write, and transform Java bytecode. Therefore the core package is sufficient and good enough to generate bytecode and make the majority of the bytecode transformations.

In case a developer would need more than the API functionalities provided by the Core package then the packages in the Layer 2 Tree, Commons, Util and XML packages could be used. Tree package provides the in-memory representation of the Java bytecode. It is very useful as it provides a big picture of the Java bytecode, but quite memory intensive though (Kuleshov, 2005). Commons bytecode has been added in ASM frmework since release 2.0 and it offers many commonly used bytecode transformations and adapters that are used to further simplify bytecode generation. The Util package as the name says it is a utility package that offers many helper classes and bytecode verifiers. The helper classes and verifiers can be used in development or testing. The last package from Layer 2 is XML package. This package contains an adapter that is used to convert bytecode structures into XML or vice-versa (Kuleshov, 2005). In addition, it has SAX-compliant adapters that make use of XSLT, used to define bytecode transformation. The Analysis package offers basic data-flow analysis and type-checking algorithms for the Java bytecode methods, stored in the tree-structure of the tree package (Kuleshov, 2005).

Structure of Compiled Java classes

It is very essential to know what is contained in a compiled Java class as ASM will get the information from the compiled Java classes. A compiled Java class is comprised of 3 important parts: the part that provides information for the class, the part that provides information about the class fields and a section that provides information for the class methods (Bruneton, 2011).

The section about the class describes the modifiers as either public or private, the name of the class, its super class, the interfaces and the annotations of the class. There is a section for each field declared in the class and each such section contains information about the modifiers, the name, the type and the annotations of the field. There is a section for each method and the constructor of the class. Each such section contains information regarding the modifiers, the name, the parameter types and the return type of each method. In addition, this section contains the compiled code of the method as a sequence of bytecode instructions (Bruneton, 2011).

The overall structure of a compiled class is given in the figure below:

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

Figure 5 - organization of a compiled class (* stands for zero or more) (Bruneton, 2011)

Java Source files and compiled Java classes do not contain the exact same information and do not have the same structure. A compiled class holds information for one class only, while the source for a class might contain more than a class. An example would be when a source file contains a class and an inner class inside it. In this case there would be two compiled classes for the source class. Anyway, the main class file contains references to the class files of the inner classes and also the inner classes that have been defined inside methods contain references to their enclosing method (Bruneton, 2011) .

Another difference between the Java source class and the Java compiled class lies in the presence of a constant pool section in the Java compiled class. The pool is an array that contains all the numeric, string and type constants that are present in the class. In addition, types are represented differently in Java source classes and Java compiled classes. Other differences include the absence of comments in Java compiled classes and package and import sections, but the presence of attributes and annotations in Java compiled classes is used for that and fully qualified names are used in Java compiled classes (Bruneton, 2011).

Class Visitor of the ASM API

A very important class of the ASM API is the Class Visitor that gathers information about the Java class file. Every method in the Class visitor contains a reference to the Java class file structure as given in figure 1.

Most useful methods of this class are given in the figure below (Bruneton, 2011):

```
public void visitSource(String source, String debug);
public void visitOuterClass(String owner, String name, String desc);
AnnotationVisitor visitAnnotation(String desc, boolean visible);
public void visitAttribute(Attribute attr);
public void visitInnerClass(String name, String outerName, String innerName, int access);
public FieldVisitor visitField(int access, String name, String desc, String signature, Object
value);
public MethodVisitor visitMethod(int access, String name, String desc, String signature,
String[] exceptions);
```

Figure 6 - Methods in Class Visitor

The methods *visitAnnotation*, *visitiField* and *VisitMethod* return objects of type *AnnotationVisitor*, *FieldVisitor* and *MethodVisitor* respectively. The simple sections of the Java class file are visited with one method and the return type on those methods is void, as it is the case for *visitSource*, *visitOuterClass*, *visitAttribute*, but in the case of *visitAnnotation*, *visitField* and *visitMethod* the information returned is much more complex and that is why auxiliary classes are used such as: AnnotationVisitor, FieldVisitor and MethodVisitor (Bruneton, 2011).

The methods of the ClassVisitor need to be called in a specific order given below (Bruneton, 2011):

*visit visitSource? visitOuterClass? (visitAnnotation | visitAttribute)**
*(visitInnerClass | visitField | visitMethod)**
visitEnd

The order given above shows that ClassVisitor can be used by making at most one call to visitSource and visitOuterClass methods followed by zero or more calls to visitAnnotation or visitAttribute methods and then afterwards via calling the methods visitInnerClass, visitField or visitMethod zero or more times till visiting the class ends.

Design and Implementation of the Algorithm using the ASM 4.1 Framework

The algorithm is comprised of two parts: the part that does the static analysis and the part that does the dynamic analysis. The part that does the static analysis will be used to detect multiplicity and method invocation, while the part that does the dynamic analysis will be used to detect lifetime and exclusivity properties.

Design of the Algorithm that does the Static Analysis

At first a diagram will be given that illustrates the application of the algorithms that does the static analysis on the Java programs.

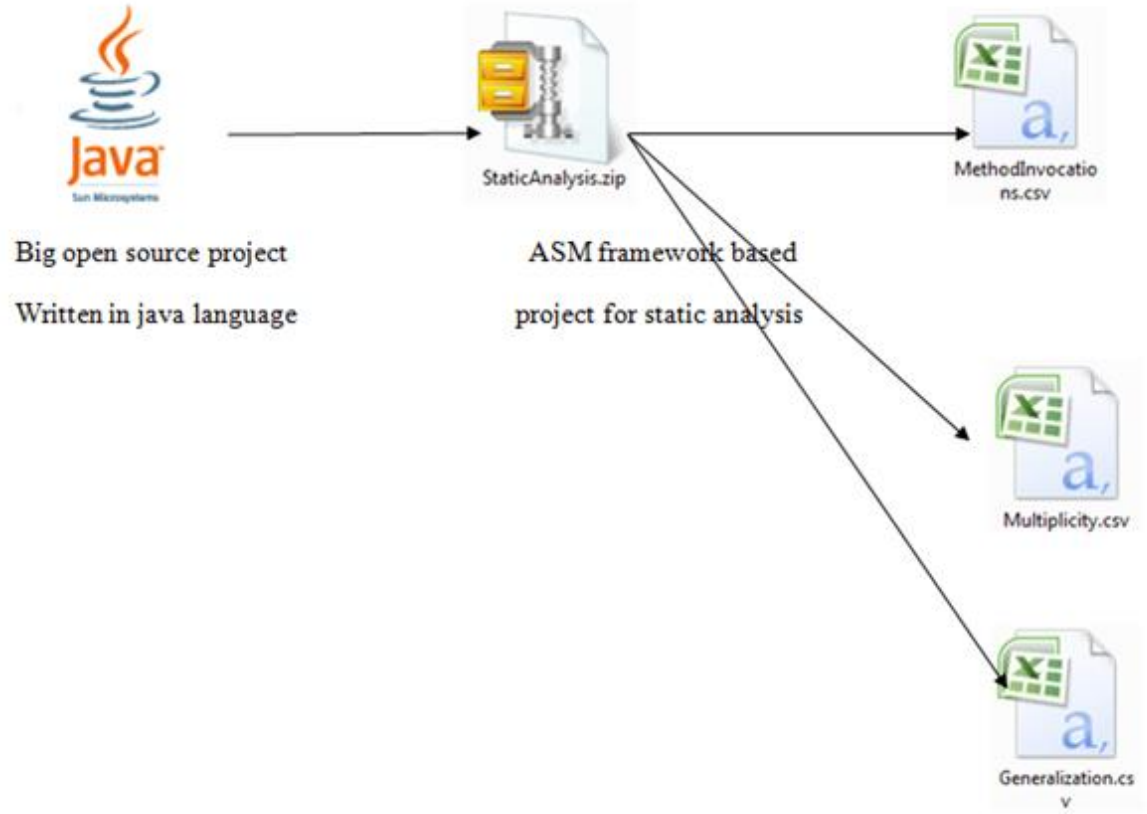


Figure 7 - Static Analysis of Java Programs

In order to proceed with the static analysis a simple tool was built based on the ASM framework. The tool takes as its argument a jar file (given by the full path name) and it produces a csv file with the information regarding multiplicity and invocation site. As well information about all the generalizations available is shown, since it will be easier and better practice to know which of the classes is sub-class of another class, in order to narrow down more the possible cases of bi-directional relationships.

When checking for multiplicity we have to look for all the attributes of each class and as well inside methods if there exists any field of variable of another class. In order to do that, we have to look only for the fields which do not correspond to the primitive built in java variables. Also all the arrays of variables, not belonging to arrays of objects will be discarded since they are not in the interest of the study, since we are looking for relationship between classes in the program.

The following code checks only for variables belonging to other classes (they start with the letter L in bytecode representation) and as well for array of objects or collections of such classes (java/util). Some more detailed work should be done in this regard since java/util library does not contain only enumeration types, but some simple types like (java/util/Date or java/util/calendar).

A known issue stands with untyped collections. This is more evident with the untyped collections which are not used with java generics in the first moment of declaration. This can

make it hard to find what type some of the collections are. For the ones which are used with generics ASM offers a very simple way to find throgh its own API. The code given in Appendix A retrieves all the needed information.

The information obtained in three different .csv files, respectively: MethodInvocations.csv, Multiplicity and Generalization.csv is further processed to detect generalizations, associatons and aggregations.

Detection of generalizations

A diagram is first given to illustrate the detection of the generalizations.

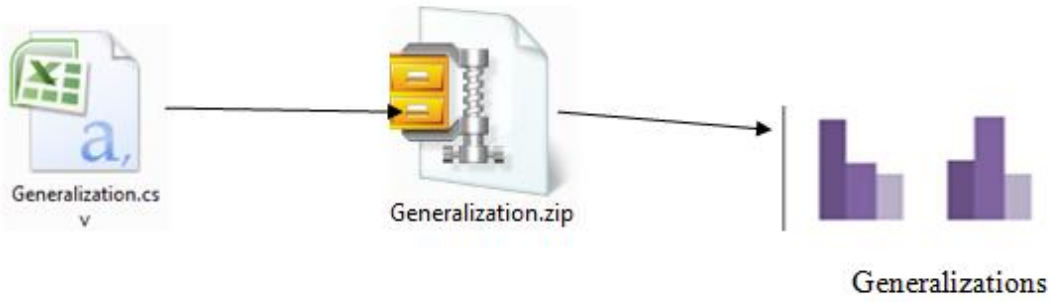


Figure 8 - Detection of Generalizations

In order to detect the generalization it is very simple through the class visitor method to find all the implement interfaces as well the corresponding super class for the given class.

The following code snippet will show that:

```
public void visit(int version, int access, String name,
                  String signature, String superName, String[] interfaces)
    {
       this.currentClassName  = name;

       p1.setParentClass(name);
       if(!superName.contains("java/lang/Object"))
       {
       System.out.println("////***** We are in class"+name +" is a subclass of
"+superName);
       generalizations.add("Class:"+name+",Class:"+superName);
       }
       else
       {
       System.out.println("////***** We are in class"+name +" it doesnt have
superclass");
       generalizations.add("Class:"+name+",No");
       }
       this.methodCounts.clear();
    }
```

As we know all classes in Java, they by default have the same superclass java/lang/Object. In order to distinguish between classes which do have a superclass and the ones which don't have was checked in the code if the superclass is java/lang/Object.

Knowing generalizations, even it is not helpful in the detection of the binary relationships in itself it is very useful, since in the invocation site, all the methods which will be called by another owner through InvokeSpecial will be discarded since they do not provide valuable information in the detection of the bi-directional relationships.

Invocation sites (message sending)

Through the help of AS we can look through the bytecode in order to detect the different Invoke methods that a class X uses to send messages to class Y. It is more useful in this case not to consider the majority of the InvokeVirtual invocations since in itself they are not messages sent by a class X sent to class Y, but just purely they are related to the inheritance, an inheriting class, calling methods of the superclass. The following code snippets shows detection of the different Invocation calls.

```
public void visitMethodInsn(int opcode, String owner,String name, String desc)
      {

            String opcodeName="";
            if(opcode==182)
            {
                  opcodeName="INVOKEVIRTUAL";
            }

            if(opcode==183)
            {
                opcodeName="INVOKESPECIAL";
            }

            if(opcode==184)
            {
                opcodeName = "INVOKESTATIC";
            }

            if(opcode==185)
            {

                  opcodeName = "INVOKEINTERFACE";
            }
            if((!owner.contains("java/io/")) && (!owner.contains("java/util/"))
&& (!owner.contains("java/lang/")) && ((owner!=
ClassComplexityCounter.p1.getParentClass())))
                        {
methodCalls.add(opcodeName+","+name+","+ClassComplexityCounter.p1.getParentClass()
+","+owner);
                        }

}
```

As it was previously mentioned, all the information is stored in a CSV file and will be processed together with the dynamic analysis results. The CSV files obtained from the application of the algorithm were further processed to reveal the information regarding

multiplicity and invocation sites. A diagram is given below to illustrate the generation of the information regarding aggregations and associations from the presence of the invocation sites and multiplicity properties.

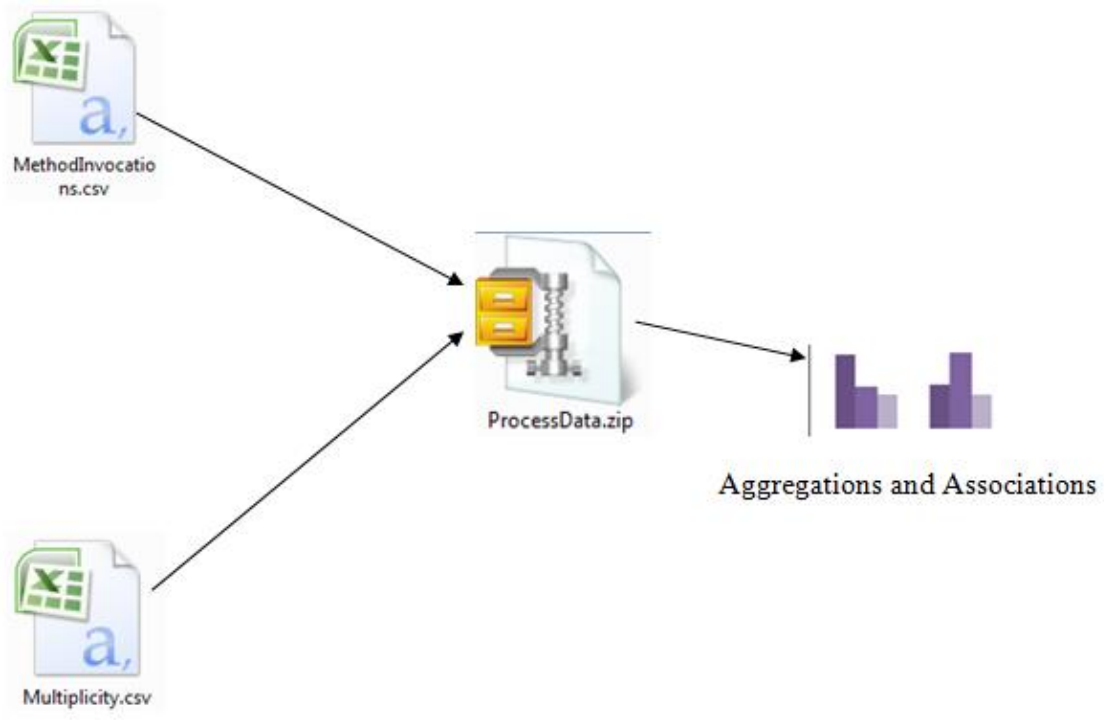


Figure 9 - Detection of Aggregations and Associations

Dynamic analysis

Through the help of the ASM framework will be conducted the code instrumentation. In order to have more meaningful analysis big open source projects like Apache Ant should be considered for such analysis since they come with a big test suite available as well. Knowing the fact that it is crucial to have a large data set for an effective dynamic analysis in order to gather more information on the runtime of objects and to be able to differ more precisely between aggregation and composition, as it is mentioned composition is more strict than aggregation and when the owner of the object dies, the part as well does not have any opportunity to be called from other methods. A diagram is given below to illustrate the algorithm that is used for dynamic analysis of the Java programs.

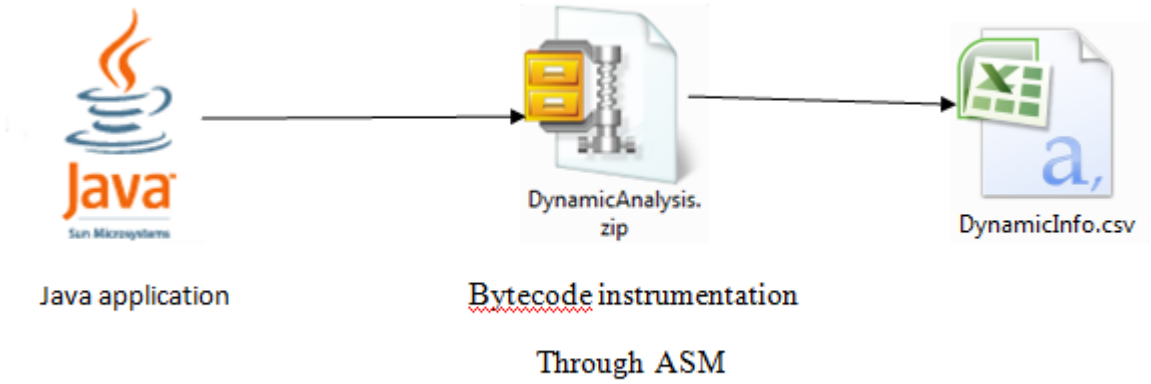


Figure 10 - Dynamic Analysis of the Java Programs

For the dynamic analysis the exclusivity and lifetime properties need to be detected if they hold true or false for a relation between two classes. Lifetime property will be checked if it holds true or false by visiting the constructor of the class and detecting inside the constructor the objects of the other classes created. This means that when the object is destroyed the objects that were created at the constructor of the object, will get destroyed before the object taken into account gets destroyed. In this case the exclusivity property holds as well. An example is given below to illustrate the idea behind checking exclusivity property if it holds true or false.

```
public class Student extends Person {
        // Instance variables
        private Grade grade; // grade for the corresponding course code
        // Constructor
        public Student(String name, String address) {
                super(name, address);
                grade = new Grade(“Mathematics”, 90);
        }
}
```

The constructor of the class student will be visited on enter and exit and the objects created with the name of the corresponding classes will be recorded. The relationship between the first class and classes that have objects instantiated at the constructer of the first class will be classified as composition. In the example above there is a composition between class Person and class grade as grade is instantiated in the constructor of class Person.

The exclusivity property will be checked by visiting with the ASM framework a class file with tests on the classes under consideration. The event when a new object is created will be recorded and in case the creation of an object is always accompanied by the creation of

another object then this leads to the idea that these objects are tied together and the exclusivity property holds but when the creation of an object is not followed by the creation of another object then that means that the same object is used for different objects and the exclusivity property does not hold.

A program was built based on the ASM framework to check if the exclusivity and lifetime properties hold true. The program is based on the modification of the .class files of the Java programs. A sequence diagram is given below to illustrate the sequence of events that take place when transforming bytecode.

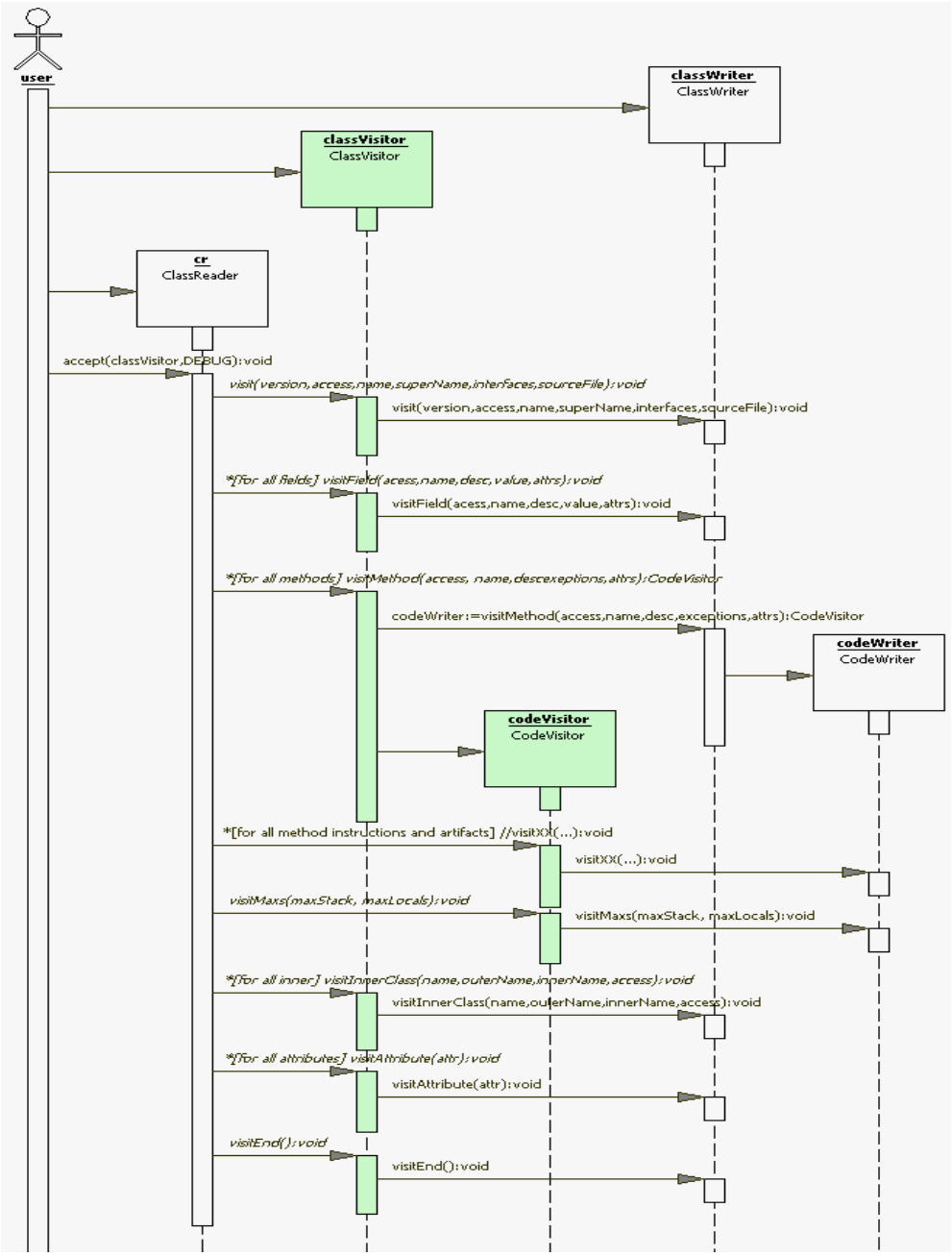
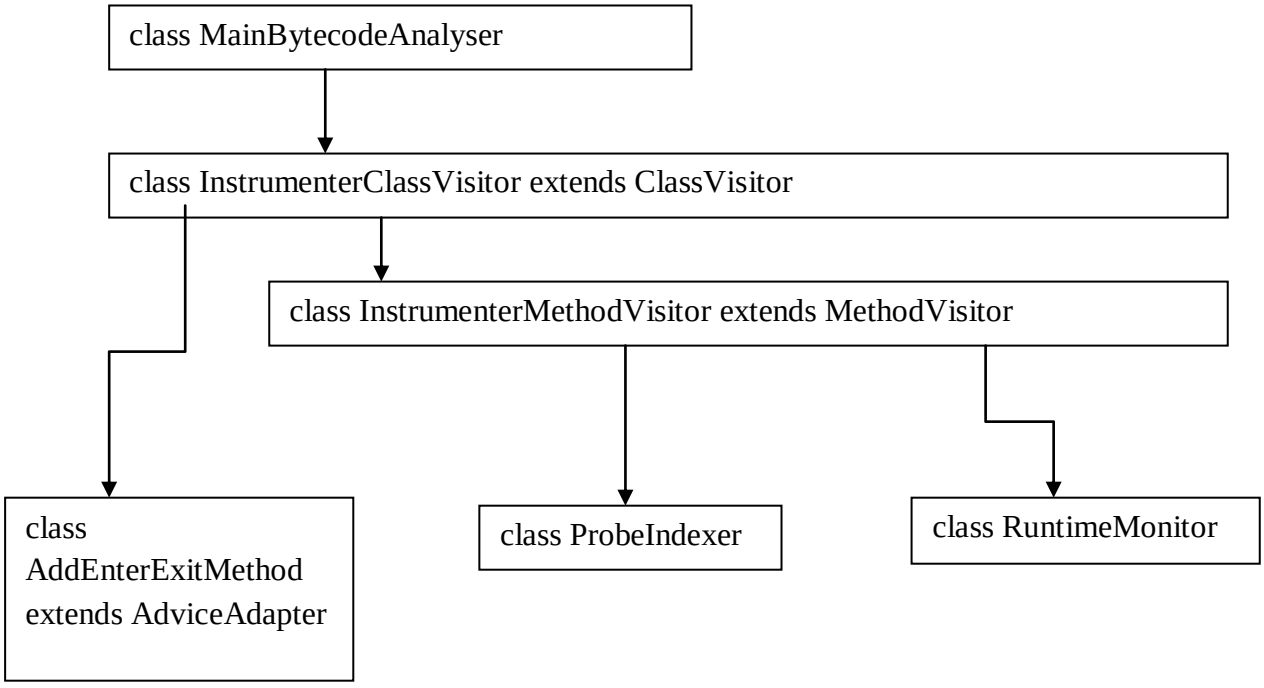


Figure 11 - Bytecde Transformation (Sequence Diagram) (Kuleshov, Using the ASM Toolkit for Bytecode Manipulation, 2012)

As it can be seen from the diagram above, instances of the classes ClassReader, ClassWriter and ClassVisitor have to be instantiated in order to manipulate the bytecode of the .class Java files. These classes have been instantiated in the program implemented to instrument the bytecode of the given .class files in the following way:

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
              ClassVisitor tracer = cw;
              if (noisy)
                    tracer = new TraceClassVisitor(cw, new
PrintWriter(System.out));
              CheckClassAdapter checker = new CheckClassAdapter(tracer);
              InstrumenterClassVisitor instrumenter = new
InstrumenterClassVisitor(
                           probeIndex, checker, className);
              ClassReader cr = makeReaderFrom(classfilename);
              // Perform the visit:
              cr.accept(instrumenter, ClassReader.SKIP_FRAMES);
              // Dump the instrumented class to a file:
              this.writeNewClassFile(classfilename, cw.toByteArray());
```

The InstrumenterClassVisitor extends the class ClassVisitor. The sequence of events that happens is: The files with the extension .class are visited and checked on each method enter, exit and object instantiation. In case such events are encountered, then new bytecode is generated to mark such events and after the file .class has been fully visited, then previous contents of the file are re-written with new bytecode of the file acquired from the bytecode instrumentation of the file. The program consists of 6 classes that are called in the following sequence:



Every method in the .class is visited from the ClassVisitor and for every method an instance of the classes InstrumenterMethodVisitor and AddEnterExitMethod is created that check for each method respectively if a new instance is create or detect there is Method enter event or method exit. All the events of object creation, method enter and exit is recorded in a .csv file. The class ProbeIndexer is used to capture all the events that will be written in a .csv file.

All the methods are visited in the ClassVisitor file by using the function visitMethod. The function visitMethod returns information regarding the access flags, name, signature and method attributes.

```
public MethodVisitor visitMethod(int access, String methodName,
                    String methodDesc, String signature, String[] exceptions) {

       MethodVisitor mv = super.cv.visitMethod(access, methodName, methodDesc,
                           signature, exceptions);

              if (mv != null) {
                    mv = new InstrumenterMethodVisitor(probeIndex, mv, methodName,
                                 this.name, methodDesc);
                    mv = new AddEnterExitMethod(probeIndex, access, methodName,
                                 methodDesc, mv, this.name);
              }
              return mv;
```

The program takes as an argument the list of the all .class files and a test class that is used to test the application and visits all the methods on Enter and Exit. Any objects created on each Enter and Exit of the methods is recorded with the corresponding information of the class it belongs to and the hash code.

The part of the code that captures entering and exiting a method is implemented by using the class AddEnterExitMethod that is inherited from the class AdviceAdapter of the ASM framework. The code that captures information on method Enter is give below:

```
protected void onMethodEnter() {
              super.mv.visitFieldInsn(GETSTATIC, "java/lang/System", "err",
                           "Ljava/io/PrintStream;");
              super.mv.visitLdcInsn("Entering " + className + "." +
name.toString()
                           + "()");
              super.mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
                           "println", "(Ljava/lang/String;)V");

              // register in file
              this.probeIndexer.registerMethodEnter(className + "." +
name.toString()
                           + "()");
       }
```

The part of the code that captures that captures on method Exit is given below:

```
protected void onMethodExit(int opcode) {
              if (opcode != ATHROW) {
```

```
                        onFinally(opcode);
                }
        }

private void onFinally(int opcode) {
                super.mv.visitFieldInsn(GETSTATIC, "java/lang/System", "err",
                                "Ljava/io/PrintStream;");
                super.mv.visitLdcInsn("Exiting " + className + "." + name.toString()
                                + "()");
                super.mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
                                "println", "(Ljava/lang/String;)V");

                // register in file
                this.probeIndexer.registerMethodExit(className + "." +
name.toString()
                                + "()");
        }
```

The program also detects when new objects are created and the part of the code that detects the instantiation of new objects is given below:

```
@Override
public void visitTypeInsn(int opcode, String type) {
                if (opcode == Opcodes.NEW) {
                        super.mv.visitCode();
                        try {
                                super.mv.visitFieldInsn(Opcodes.GETSTATIC,
"java/lang/System",
                                                "err", "Ljava/io/PrintStream;");
                                super.mv.visitLdcInsn(type.toString());
                                super.mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
                                                "java/io/PrintStream", "print",
"(Ljava/lang/String;)V");
                                // record data in CSV
                                Object o = new Object();
                                int index =
probeIndexer.registerObjectCreate(type.toString(),
                                                o);
                        } catch (Exception e) {

                        }

                        // add method to the existing class
                        try {
                                super.mv.visitVarInsn(Opcodes.ALOAD, 0);
                                super.mv.visitMethodInsn(Opcodes.INVOKESTATIC,
                                                "RuntimeMonitor", "hitObjectCreate",
                                                "(Ljava/lang/Object;)V");
                                super.mv.visitTypeInsn(opcode, type);
                        } catch (Exception e) {
                        }
                }
        }
```

Detection of the event when new objects are created is implemented by overriding the method **public void visitTypeInsn(int opcode, String type)** and modifying the corresponding class file where the new object was detected with info that new object has

been created by adding methods that print information regarding the class that the newly object created belongs to and the hash code of the objects. After the application of the algorithms that do the static and the dynamic analysis of the programs, final results can be obtained about the presence of the associations, aggregations and compositions in java programs. A diagram is given below that illustrates the generation of the final results.

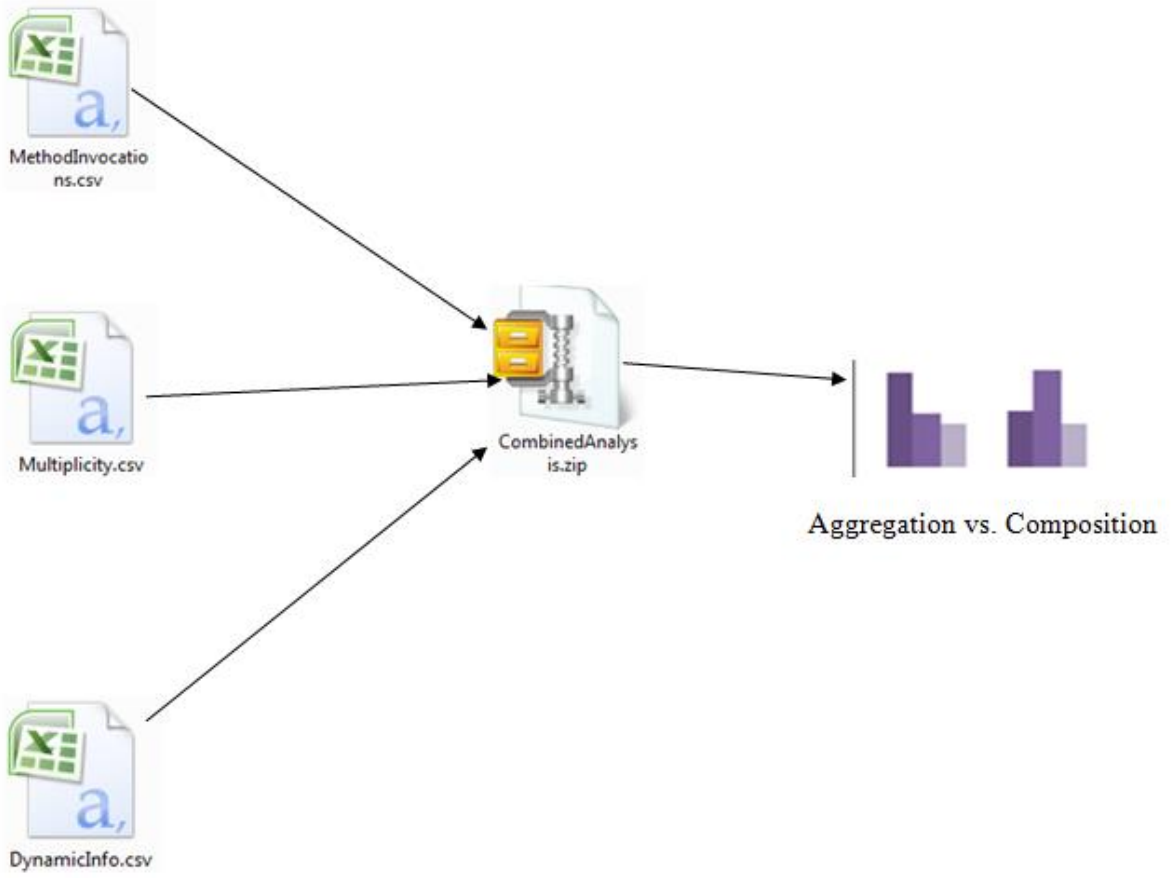


Figure 12 - The separation of Aggregations from Compositions after the application of the dynamic analysis

Summary

This chapter presented the ASM framework and the design and implementation of the algorithms that detect mereological relations in Java programs, respectively: associations, aggregations and compositions. ASM framework was first introduced as the algorithms that instrument the bytecode to detect the relations between the classes are implemented by using the ASM framework. After an overview of the ASM framework, the algorithms that do static and dynamic analysis of the java programs were explained.

Chapter 4: Example (Proof of Concept)

Introduction

This chapter contains the application of the algorithms that do the static and the dynamic analysis on a small program. The program is provided as a proof of concept to demonstrate how the algorithms are applied on the programs, how the results are obtained from the application of the algorithm and how the analysis is conducted to derive the right conclusions regarding the presence of mereological relations in Java programs. At first the mereological relations that exist in the program are given via a UML diagram and afterwards the results obtained from the application of algorithms are given and an analysis is done on the results obtained to check if the results obtained from the application of the algorithms correspond with the mereological relations presented in the class diagram.

Class Diagram of the Application

A small application consisting of 6 classes with the relationships as shown in the class diagram below was analyzed by using the static and dynamic analysis programs.

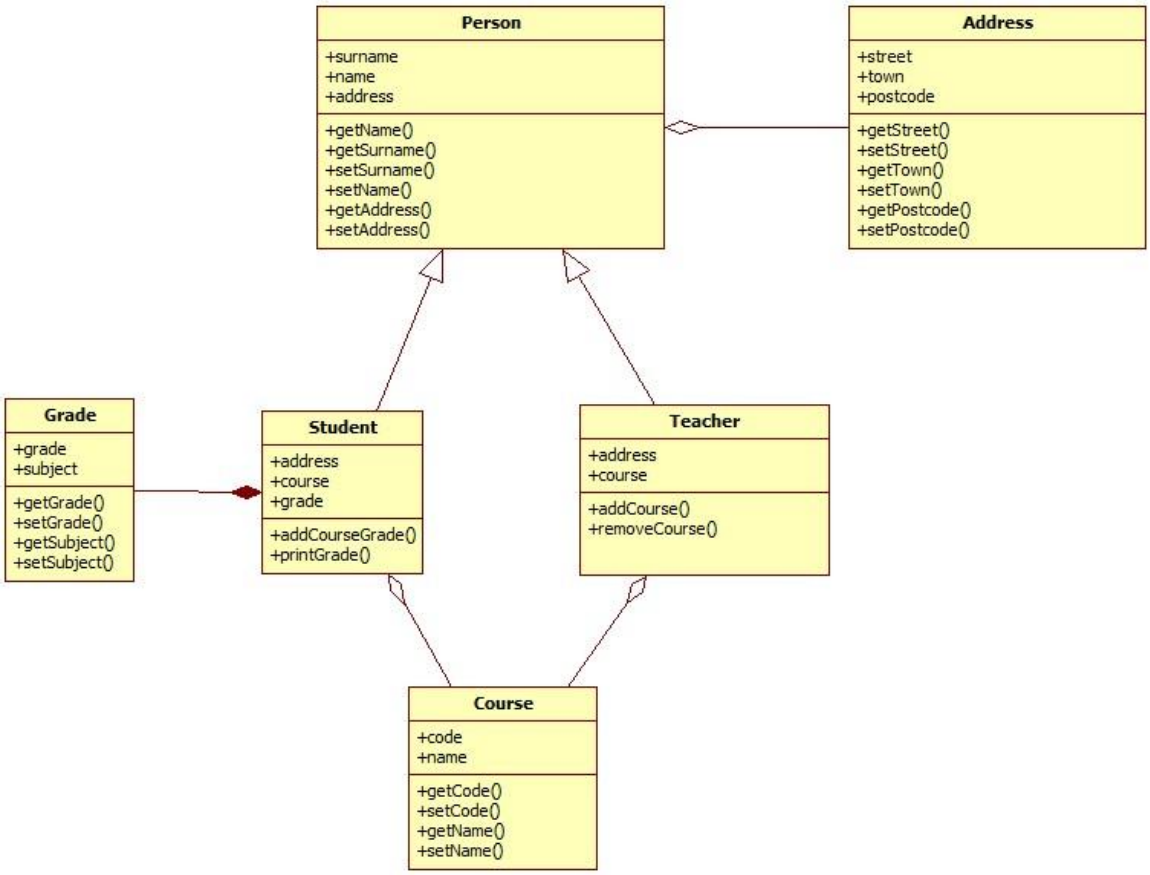


Figure 13 - Class Digram of course management system

Application of the Algorithms that do Static and Dynamic Analysis

The classes given above will be checked by using the applications for static and dynamic analysis of the Java programs and checked if the information obtained complies with the actual mereological relations between the classes. After performing a static analysis on the small application of course management system, the following results were revealed.

The results obtained from the static analysis analysis comprise data regarding generalizations, method invocations and multiplicity. The generaliations detected were the following:

Table 1 - Detection of Generalizations

Class	**Superclass**
Course	No
Grade	No
Teacher	Person
Address	No
Person	No
Student	Person

The information provided in the table above reveals that there is a generalization relationship between the classes Teacher and Person and Student and Person as both classes Teacher and Student are derived from the class Person.

The information obtained regarding method invocation is given in the below:

INVOKESPECIAL,<init>,Teacher,Person

INVOKESPECIAL,<init>,Teacher,Course

INVOKESPECIAL,<init>,Teacher,Course

INVOKESPECIAL,<init>,Student,Person

INVOKESPECIAL,<init>,Student,Grade

INVOKEVIRTUAL,getName,Student,Course

INVOKEVIRTUAL,getGrade,Student,Grade

From the information above, there is message sending from Teacher to Person, Teacher to Course, Student to Person, Student to Grade, Student to Course and Student to Grade. There is inheritance between the classes (Teacher, Person) and (Student, Person). Because there is inheritance between Person and Teacher and Person and Student, these two relations are crossed out, therefore the relationships that remain to be further analyzed are (Teacher, Course), (Student, Course) and (Student, Grade).

The information obtained regarding multiplicity is given in the table below:

Table 2 - Detection of Multiplicity

Current Class	**Accessability**	**varName**	**Type**	**Generics**	**Multiplicity**
Teacher	private	course	Course	Not using generics	0..1
Person	private	address	Address	Not using generics	0..1
Student	private	course	Course	Not using generics	0..1
Student	private	grade	Grade	Not using generics	0..1

The static analysis provided information regarding the invocation site and multiplicity properties and dynamic analysis will give insight regarding the exclucivity and lifetime properties.

The following information was obtained after applying the dynamic algorithm to the small application of course management system:

Table 3 - Detection of Lifetime Property

Event	**Class**	**Hash code** (in case Object Create)
Entering,Student.<init>()	Student	
Object Create Grade	Student	3288014
Exiting,Student.<init>()	Student	

The information above shows that for every object of class Student created there will be an object of class Grade created. This reveals that the lifetime of the instances of the class Student is longer than the lifetime of the instances of the class Grade and when the instances of the class Student get destroyed, the instances of the class Grade will get destroyed before.

From the application of the algorithm to the file Test.class that is used to test the classes of the application taken into consideration, the following information was revealed:

Table 4 – Detection of Exclusivity

	Event Type	Class	Hash code
1	Object Create	Student	3325285
2	Object Create	Course	29525730

3	Object Create	Grade	29089096
4	Object Create	Student	5912867
5	Object Create	Grade	27766975
6	Object Create	Student	30866355
7	Object Create	Grade	2102960
8	Object Create	Teacher	12582949
9	Object Create	Teacher	30587319
10	Object Create	Address	18615648

The classes Student, Course and Grade are marked in different colours. Three instances of class Student have been created that have been accompanied by the creation of one class class course and 3 instances of the class Grade. The proportional number of the instances of the class Grade to the number of the instances of the class Student shows that the exclusivity property holds between the instances of the instances of class Grade and the instances of class Student, but the same cannot be said for the relationship between Student and Course as when three instances of class Student were created, there was only one instance of the class Course created, therefore exclusivity property does not hold between the instances of the class Student and the instances of the class Course. Also there have been two objects of the Class Teacher created and only one Course, therefore the exclusivity property does not hold between the instances of the class Course and the instances of the class Teacher. In the end there has been only one instance of the class Address created and from this information it can be derived that the exclusivity property does not hold true between the instances of the classes Teacher or Student and Address.

After the analysis conducted from the application of the Static and the Dynamic Analysis the following information was revealed regarding the mereological relationships between the classes:

There is aggregation between the Classes (Person, Address), (Teacher, Course) and (Student, Course) and there is a composition between the classes Student and Grade. Also there is a generalization relationship between the classes (Person, Teacher) and (Person, Student). The results acquired from the application of the static and the dynamic algorithms complies with the mereological relationships that actually exist among the classes.

Summary

This chapter presented the application of the algorithms on a small program. The data obtained from the application of the algorithms that conducted static and dynamic analysis of the program revealed those mereological relations that exist in the program and that were

presented at first via a Class diagram. The proof of concept showed that the algorithms detect correctly the various mereological relations in Java programs.

Chapter 5: Evaluation

Introduction

This chapter contains an evaluation of the algorithms implemented to detect various mereological relations in Java programs by applying the algorithms on big open source projects. The static algorithm was applied on 5 big open source projects and the results obtained were analyzed to detect the presence of the associations and aggregations in the java projects taken into account. The algorithm was applied only on five big open source projects due to the time constraints and if more time was at the disposal of the project, more projects would have been analyzed regarding the presence of the various mereological relations.

Statistics

For the static part 5 big open source projects were taken under investigation in order to gather important data on the possible relationships between the classes in the project. The Java projects that were chosen, were the following:

-Apache Maven 3.1.1

-Junit 4.11

-Apache Ant 1.9.1

-JHotDraw 6 beta version

-ApacheIvy 2.3.0

Static analysis can provide information only on the multiplicity and message sending properties. Based on the 2 properties that were mentioned previously, Multiplicity and Message Sending are the only two properties that information can be gathered for from the static analysis only and as such the data collected will be about those mereological relations that are defined by the multiplicity and message sending properties. It is able to differ between associations and aggregations, as well as to have a more general statistical data on the programs chosen to be analyzed. Therefore, this chapter will not only contain information regarding associations and aggregations, but also on the overall number of classes on each project and the number of generalizations. After applying the tool on each big open source project the following data was gathered regarding the number of the classes on each of the projects taken into account.

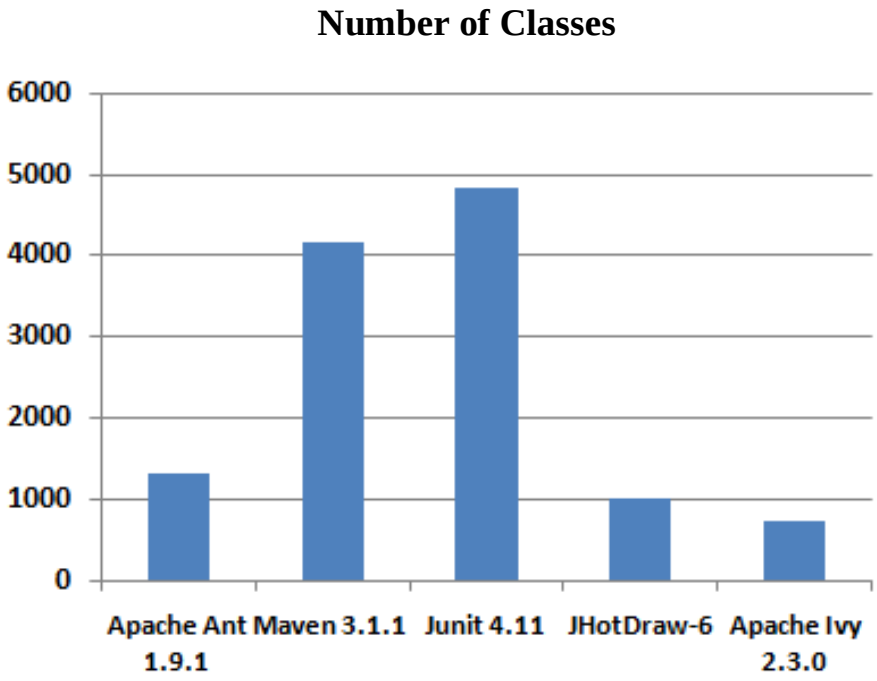


Figure 14 - Number of classes on each project

Since it is important to figure out the generalizations in the code; in order not to miss any relationship which can have an InvokeSpecial call in it, and still not to be a generalization, the information about generalizations was as well collected. In terms of generalizations the following data was collected for each project was acquired:

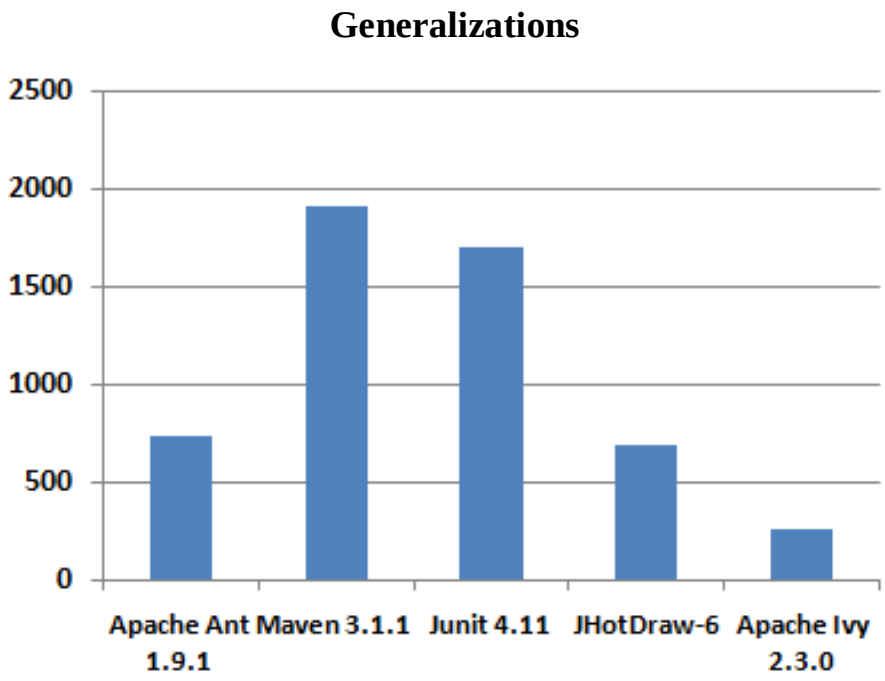


Figure 15 - The number of Generalizations

It was important to figure out which of the other relationships can be bi-directional in terms that if A class send messages to class B, the other way around class B send messages to class A, since from the IMS property for both aggregation and composition the relation has to be only from one side, not from the both sides.

After processing the gathered data from the static analysis the following chart will show the number of bi-directional relationships which based on the definition cannot be association, aggregation or composition. The following table shows the number of such bi-directional relationships for each project:

Bi-directional Relationships

Table 5 – Bi-directional Relationships

Project	Number of Bi-directional relationships
Apache Ant 1.9.1	49
Maven 3.1.1	50
Junit 4.11	11
JHotDraw-6	1
Apache Ivy 2.3.0	17

After finding the number of bi-directional relationships for each project, than we could remove them and try to find the number of relationships which could be association, aggregation or composition. The following table shows the number after removing for each project the possible bi-directional relationships.

Table 6 - Number of asociation+aggregation+composition

Project	Number of association+aggregation+composition
Apache Ant 1.9.1	5605
Maven 3.1.1	15047
Junit 4.11	5952
JHotDraw-6	2564
Apache Ivy 2.3.0	4135

Based on the other IMS definition which states that in the case of the aggregation, it has to be in general one and one field, while for the association more than 1, the following data was gathered for each project based solely on the number of different fields:

Table 7 - Number of Aggregations+Compositions

	Associations	Aggregations+Compositions
Apache Ant 1.9.1	3705	1900
Maven 3.1.1	8263	6784
Junit 4.11	3190	2762
JHotDraw-6	1402	1162
Apache Ivy 2.3.0	2235	1900

In terms of the percentage for each project beteween associations and aggregations with compositions would be expressed in the following pie charts:

Legend

With red color the percentage of aggregation with composition relationships.

With blue color the percentage of association relationships.

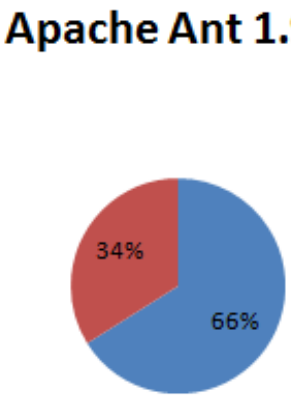


Figure 16 - Results for Apache Ant

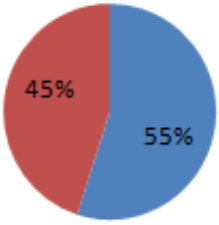


Figure 17 - Results for Maven Ivy

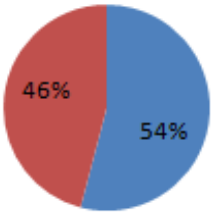


Figure 18 - Results for Apache

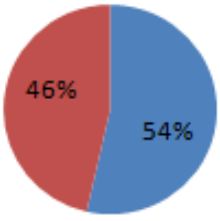


Figure 19 - Results for Junit

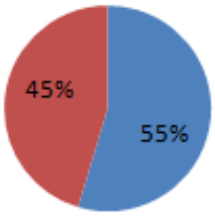


Figure 20 - Results for JHotDraw

The following chart can show in terms of relationships expressed for each project:

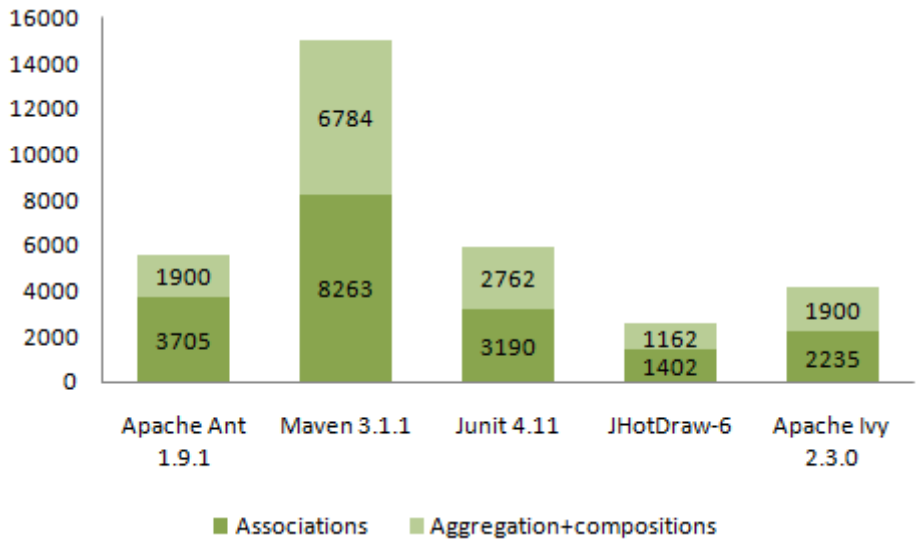


Figure 21 - Associatons vs. Aggregations + Compositions

Detection of the Lifetime and Exclusivity Properties

The detection of the lifetime and exclusivity properties was done as a Proof of Concept. A small application was taken into account with 6 classes and the the results obtained from the application of the dynamic algorithm are given in Chapter 5.

Limitations of the algorithm

The algorithm for detecting the mereological relations in Java programs comprises of two parts: the part that does the static analysis and the part that does the dynamic analysis. The part that does the static analysis and detects the properties of multiplicity and invocation sites, provides a full analysis even on big projects and this was demonstrated by the application of the algorithm on five big open source projects. On the other hand the algorithm that detects the lifetime and exclusivity properties is limited as it is not capable of dealing with the instantiation of new objects contained in an array. The algorithm can detect properly when there is only one instance created and this was demonstrated from the proof of concept, but it does not handle the instantiation of the arrays.

Summary

This chapter presented the aplication of the algorithms on five big open source projects, the results obtained from the application of the algorithms and the limitations of the algorithm. It was revealed that there is no limitation on the part of the algorithm that does the static analysis, but the part that does the dynamic analysis is limited as it does not handle properly the instantiation of the arrays.

Chapter 6: Conclusions

Introduction

This chapter contains a critical analysis of the data collected, threat to validity and future work. At first, an analysis of the data collected from the application of the algorithm on five big open source projects is given. Then threat to validity is given that contains information on what might be a threat to the validity of the information obtained from the application of the algorithms. In the end, what might be carried out as future work is given.

Critical Analyses of the Data Acquired

After analyzing the data collected, expressed via the different charts in the previous chapter the following conclusions can be deducted:

- It seems that there is a connection between the number of classes and the number of generalizations. This means that proportionally every bigger project in terms of classes, contains in itself greater number of generalizations as well. Taking into account the fact that all projects which were analyzed, were all Java applications and as we know that in Java programming language it is not possible to extend more than 1 class, it is clear that the results comply with good practices defined for Java programming language.

- In terms of bi-directional relationships, there is a less visible connection between the number of bi-directional relationships in terms of program size, which in this case we refer to the program size, as the number of classes each program contains. As it can be seen from the graphs, JUnit which is significantly bigger compared to Ant (rougly 3 times bigger in terms of number of classes), the number of bi-directional relationships tends to be nearly 4 times smaller, thus making the number of bi-directional relationships 12 times more in Ant compared to JUnit. As it can be inferred from the results and taking into account that the decision to have bi-directional relationship is purely a design decision and we can infer that in the case of JUnit the decision was more strict to avoid cirular relationships. The same can be inferred for JHotdraw with a single bi-directional relationships.

- In terms of one-directional relationships, it can be seen that there is a higher percentage of associations in relation to aggregation and compositions. Interesting is that from the 5 projects which were chosen to be analyzed, in terms of percentages in Apache Ant there is a higher gap between associations and aggregations with compositions. For Ant is nearly 1:2 the ration of associations to aggregation with composition while for the other 4 projects is nearly 45:55 in terms of percentages.

- The study was proven to be right in the case of a small example, since the relationships were detected accurately by combing both the static and dynamic analysis. Through dynamic analysis and based on the exclusivity and lifetime porperty, it is possible to narrow down the study in order to have a complete classification of all relationships represented in each project.

Threat to Validity

- From the the statistics acquired, mainly two different properties were used in order to differ between association and aggregation with composition, which were multiplicity and message sending between the classes. Still it is important to know how these properties would divide the relationships in respect to what developers were intending these relationships to be in reality. A UML diagram for the whole projects, which for big projects, seems to be impractical because of its size, would have helped in order to measure bettter the accuracy of the study. Even on the other side, a UML diagram is not concise, as it was mentioned earlier that the reverse engineering tools like Rational Rose, they are not totally precise. Developers on the other side, they have different programming habits in converting UML diagrams to code, depending on their own design choices. Claiming that, it is important to deduce that the accuracy of the study lies between a margin of error.

- Another option, in order to quantify the accuracy of the study would be to check manually and to count the number of different relationships in it. This would be error prone as well, since it is likely that human check on the whole code would produce some level of inaccuracy.

- In order to gather all the static information, different tools were used in accordance with each other. Different tools were used for fetching the static information thorugh the ASM framework and different tools for measuring the fetched data were used, so it can happen that the inaccuracy can happen because different tools were combined which leads to some possible mistakes.

- For the study 5 big open source projects were taken into consideration. It is possible that for 5 other projects the results would differ. Even more in the case of projects written in different programming languages the results are likely to change.

Future Work

- In order to make the study more complete and to attain a more detailed information on the different relationships, it would be very useful to apply the dynamic analysis to big projects and quantify the correctness of both algorithms in detecting especially aggregations vs compositions. To be considered is that fast algorithms for processing the files would be required, since the files produced with the dynamic analysis for large test cases, are expected to be very big in size.

- A very interesting topic would be to compare the results from this analysis to results which are made on the same big open sources and version as well chosen in the study, in order to quantify more into details how different would be in results, a different approach to detect the mereological relationships. In this case different frameworks, other than ASM would be considered.

- To go further with the study on the big open source projects, would be more representative to choose at least 10 more projects (preferably bigger), in order to generalize further the study and to go even more into detailed statistics on the software design models chosen by developers.

- It would be very helpful to integrate such studies on merelogical relationships in order to improve the quality of the UML tools and especially to be able to have more specialized tools which deal with reverse enginnering practices.

Summary

This chapter provided an analysis on the data gathered from the application of the algorithm on five big open source projects, a presentation of a possible threat to validity and what remains to be done on this project as future work.

References

4. Aho, Sethi, & Ullman. (1986). *Compilers - Principles, Techniques, and Tools.* Addison-Wesley.

5. Arabestani, S. (2000). *Relations in Object-Oriented Analysis.*

6. Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java Programming Language.* Addison Wesley.

7. Barbier, F., Henderson-Sellers, B., Parc-Lacayrelle, A. L., & Bruel, J.-M. (2003, May). Formalization of the Whole-Part Relationship in the Unified Modeling Language. *IEEE Transactions On Software Engineering* .

8. Blaha, M. R., & Rumbaugh, J. R. (2004). *Object-oriented Modeling and Design with UML*. Prentice Hall.

9. Bruel, Henderson-Sellers, Barbier, Parc, L., & France. (2001). *Improving the UML Metamodel to Rgorously Specify Aggregation and Composition.*

10. Bruneton, E. (2011, October 29). *ASM 4.0 A Java bytecode engineering library*. Retrieved May 2013, 19, from OW2 Consortium: http://download.forge.objectweb.org/asm/asm4-guide.pdf

11. Bruneton, E., Lenglet, R., & Coupaye, T. (n.d.). *ASM: a code manipulation tool to implement adaptable systems*. Retrieved May 19, 2013, from OW2 Consortium: http://asm.ow2.org/current/asm-eng.pdf

12. Chawdhuri, D. R. (2011, October 6). *Manipulating Java Class Files with ASM 4 - Part One : Hello World!* Retrieved May 19, 2013, from Geeky Articles: http://www.geekyarticles.com/2011/10/manipulating-java-class-files-with-asm.html

13. Corporation, I. (2010). *The interaction between source code and the control flow graph*. Retrieved 05 31, 2013, from IBM Corporation: http://publib.boulder.ibm.com/infocenter/zos/v1r12/index.jsp?topic=%2Fcom.ibm.zos.r12.asmk200%2Fputgin.htm

14. Debasish Ray Chawdhuri, D. R. (2012, October 13). *Manipulating Java Class Files with ASM 4 - Part Two: Tree API*. Retrieved May 19, 2013, from Geeky Articles : http://www.geekyarticles.com/2011/10/manipulating-java-class-files-with-asm_13.html

15. Foster, J. (2011). *Data Flow Analysis* . Retrieved 05 31, 2013, from Harvard School of Engineering and Applied Sciences: http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf

16. Gueheneuc, Y.-G., & Albin-Amiot, H. (2002). *Recovering Binary Class Relationships: Putting Icing on the UML Cake.*

17. Gueheneuc, Y.-G., Albin-Amiot, H., Douence, R., & Cointe, P. (2002). Bridging the Gap between Modeling and Programming Languages. France.

18. Harvard. (2011). *Pointer Analysis*. Retrieved 06 09, 2013, from Harvard School of Engineering and Applied Sciences: http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf

19. Harvard School of Enginnering and Applied Sciences. (2011). *Pointer Analysis*. Retrieved 05 31, 2013, from Harvard School of Enginnering and Applied Sciences: http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf

20. Hubicka, J. (2003, 05 05). *Control Flow Graph*. Retrieved 05 31, 2013, from UCW: http://www.ucw.cz/~hubicka/papers/proj/node6.html#SECTION02420000000000000000

21. Keet, C. M. (2006). Part-Whole relations in object-role models. *OTM'06 Proceedings of the 2006 international conference on On the Move to Meaningful Internet Systems: AWeSOMe, CAMS, COMINF, IS, KSinBIT, MIOS-CIAO, MONET - Volume Part II* (pp. 1118-1127). Heidelberg: Springer-Verlag.

22. Keet, M. (2006). *Introduction to part-whole relations: mereology, conceptual modelling and mathematical aspects.* Bolzano.

23. Khedker, U. P., Sanyal, A., & Karkare, B. (2009). *Data Flow Analysis: Theory and Practice.* CRC Press (Taylor and Francis Group).

24. Kollmann, R., & Gogolla, M. (2001). Application of UML Associations and Their Adornments in Design Recovery. *8thWorking Conference on Reverse Engineering* . Los Alamitos: IEEE.

25. Kuleshov, E. (2005, 08 17). *Introduction to the ASM 2.0 Bytecode Framework*. Retrieved 05 31, 2013, from OW2 Consortium : http://asm.ow2.org/doc/tutorial-asm-2.0.html

26. Kuleshov, E. (2004, October 6). *Using the ASM framework to implement common Java bytecode transformation patterns*. Retrieved May 19, 2013, from OW2 Consortium: http://asm.ow2.org/current/asm-transformations.pdf

27. Kuleshov, E. (2012, October 21). *Using the ASM Toolkit for Bytecode Manipulation*. Retrieved June 29, 2013, from OW2 Consortium: http://asm.ow2.org/doc/tutorial.html

28. Le, D. T., & Janicki, R. *On a Parthood Specification Method for Component Software.* Hamilton.

29. Marcos, E., Vela, B. C., & Caceres, P. (2001). Aggregation and composition in object-Relational database design. *proceedings of the 5th east-european conference on Advances in Databases and Information Systems* (pp. 195–209). Springer-Verlag.

30. Microchip Technology Inc. (2012, 05 27). *How To Work With The Call Graph*. Retrieved 05 31, 2013, from Microchip Technology Inc: http://microchip.wikidot.com/mplab:how-to-work-with-the-call-graph#toc0

31. Noble, J., & Grundy, J. (1995). Explicit relationships in object-oriented development. *proceedings of the 18th conference on the Technology of Object-Oriented Languages and Systems* (pp. 211–226). Prentice-Hall.

32. Princeton. (2003). *Iterative Dataflow Analysis*. Retrieved from Princeton University: http://www.cs.princeton.edu/courses/archive/spring03/cs320/notes/analysis2.pdf

33. Ramnath, S., & Dathan, B. (2011). *Object-Oriented Analysis and Design.* Springer.

34. Ravenbrook. (2000, December 15). *The Memory Management Glossary.* Retrieved from The Memory Management Glossary: http://www.memorymanagement.org/glossary/r.html#reachable

35. Richner, T., & Ducasse, S. (1999). Recovering high-level views of object-oriented applications from static and dynamic information. *Proceedings ICSM'99 (International Conference on Software Maintenance)* (pp. 13-22). IEEE.

36. Smaragdakis, Y., & Bravenboer, M. (2009). *Using Datalog for Fast and Easy Program Analysis.* Retrieved 05 31, 2013, from http://cgi.di.uoa.gr/~smaragd/doop-datalog2.0.pdf

Appendix A

Code that is part of the Static Analysis

```
if(desc.startsWith("L") && !name.equals("this") && (!(desc.contains("java/lang/"))
&& (!(desc.contains("java/io/")))))
      {
            System.out.println("field//////"+ " " + access+" " +
currentClassName.substring(m+1)+ " "+"  " + name+ "  "+desc+"  "+signature+ "  ");
            if(access == 2 && !(signature == null))
            {
                  if(desc.contains("java/util") || desc.startsWith("["))
                  {
attReferenced.add(currentClassName+","+"private"+","+name+","+desc +"," +
signature+","+"1..M");
                  } else
                  {

attReferenced.add(currentClassName+","+"private"+","+name+","+desc +"," +
signature+"," +"0..1");
                  }
            }
            if(access == 2 && signature == null )
            {

                  if(desc.contains("java/util") || desc.startsWith("["))
                  {

attReferenced.add(currentClassName+","+"private"+","+name+","+desc +"," + "Not
using generics"+","+"1..M");
                  } else
                  {

attReferenced.add(currentClassName+","+"private"+","+name+","+desc +"," + "Not
using generics"+","+"0..1");
                  }
            }
            if(access == 0  && !(signature == null)  )
            {
                  if(desc.contains("java/util") || desc.startsWith("[") )
                  {

      attReferenced.add(currentClassName+","+"public"+","+name+","+desc +"," +
signature+","+"1..M");
                  } else
                  {

      attReferenced.add(currentClassName+","+"public"+","+name+","+desc +"," +
signature+","+"0..1");
                  }
            }
            if(access == 0  && signature == null )
            {
                  if(desc.contains("java/util") || desc.startsWith("["))
                  {

      attReferenced.add(currentClassName+","+"public"+","+name+","+desc +"," +
"Not using generics"+","+"1..M");
                  } else
```

```
                    {

      attReferenced.add(currentClassName+","+"public"+","+name+","+desc +"," +
"Not using generics"+","+"0..1");
                    }
              }
              if(access == 1 && !(signature == null) )
              {
                    if(desc.contains("java/util") || desc.startsWith("["))
                    {

      attReferenced.add(currentClassName+","+"protected"+","+name+","+desc +"," +
signature+","+"1..M");
                    } else
                    {

      attReferenced.add(currentClassName+","+"protected"+","+name+","+desc +"," +
signature+","+"0..1");
                    }
              }
              if(access == 1 && signature == null )
              {
                    if(desc.contains("java/util") || desc.startsWith("["))
                    {
    attReferenced.add(currentClassName+","+"protected"+","+name+","+desc +","
+"Not using generics"+","+"1..M");
                    } else
                    {
  attReferenced.add(currentClassName+","+"protected"+","+name+","+desc +"," +"Not
using generics"+","+"0..1");
                    }
              }

      }

      return null;
    }
```