Incremental impact analysis for object-oriented software

by

Luke Bishop

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Suraj Kothari, Major Professor
Simanta Mitra
Yong Guan

Iowa State University
Ames, Iowa
2004

This is to certify that the master’s thesis of

Luke Bishop

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# Table of Contents

LIST OF FIGURES...................................................................................................................................... v

LIST OF TABLES.......................................................................................................................................... vii

ACKNOWLEDGEMENTS ........................................................................................................................... viii

ABSTRACT ................................................................................................................................................. ix

1 INTRODUCTION........................................................................................................................................ 1

2 APPROACH........................................................................................................................................ 4

2.1 DEFINITION OF DEPENDENCY IN CONTEXT OF IMPACT ANALYSIS.................................................. 5

2.2 INCREMENTAL MODEL UPDATE EXAMPLE.................................................................................. 6

2.3 METHOD INPUT/OUTPUT MAPPING .......................................................................................... 9

2.4 PROGRAM DEPENDENCY MODEL ............................................................................................... 12

3 IMPLEMENTATION OF INCREMENTAL IMPACT ANALYZER .................................................... 16

3.1 USE OF THE ECLIPSE IDE ............................................................................................................ 16

3.2 ARCHITECTURE ............................................................................................................................ 17

3.2.1 Incremental Update Manager.................................................................................................... 18

3.2.2 Program Dependency Model Updater...................................................................................... 19

3.2.3 Proposed Change Analyzer ..................................................................................................... 19

3.2.4 Impact Propagator .................................................................................................................. 19

3.3 USER INTERFACE .......................................................................................................................... 20

4 ALGORITHM DESIGN .................................................................................................................. 21

4.1 INCREMENTAL UPDATE MANAGER ALGORITHMS .................................................................... 21

4.2 PROGRAM DEPENDENCY MODEL ALGORITHMS ...................................................................... 23

4.3 PROPOSED CHANGE ANALYZER ALGORITHMS ....................................................................... 29

4.4 IMPACT PROPAGATOR ALGORITHMS ...................................................................................... 32

4.5 METHOD MAPPING (DFA) ALGORITHMS ..................................................................................... 32
4.6 MISCELLANEOUS ALGORITHMS .................................................................................................................. 37

5 RESULTS .......................................................................................................................................................... 38

5.1 PERFORMANCE OF INCREMENTAL APPROACH .................................................................................... 38

5.2 EFFECTIVENESS OF INSTANCE-UNAWARE APPROACH ....................................................................... 40

5.3 CONCLUSIONS & FUTURE WORK .............................................................................................................. 42

6 APPENDIX: ANALYSIS LOG EXAMPLE ......................................................................................................... 44

7 REFERENCES ..................................................................................................................................................... 59
List of Figures

Figure 1 - Method Mapping Code Example ................................................................. 10
Figure 2 - Simple Method Mapping Example ............................................................... 10
Figure 3 - Complex Method Mapping Example ............................................................ 10
Figure 4 - PDM Modify Operation ............................................................................. 13
Figure 5 - PDM Add Operation .................................................................................. 14
Figure 6 - PDM Delete Operation ............................................................................. 15
Figure 7 - Analysis Cycle of Incremental Impact Analyzer ........................................ 18
Figure 8 - Algorithm updateFileModificationStamps ............................................... 22
Figure 9 - Algorithm returnModifiedFileObjects ..................................................... 23
Figure 10 - Algorithm returnMethodCalls (recursive) .............................................. 23
Figure 11 - Algorithm linkMethodDependencyObjects .......................................... 24
Figure 12 - Algorithm orderMethodsForAnalysis ................................................... 25
Figure 13 - Algorithm createDepLink ...................................................................... 25
Figure 14 - Algorithm clearChildLinks ................................................................... 26
Figure 15 - Algorithm mergeDependencyModels .................................................... 26
Figure 16 - Algorithm markValues .......................................................................... 29
Figure 17 - Algorithm calculateImpactOfMarks ....................................................... 30
Figure 18 - Algorithm tracesToMark (recursive) ..................................................... 31
Figure 19 - Algorithm propagateImpact ................................................................... 32
Figure 20 - Algorithm merge ................................................................................ 33
Figure 21 - Algorithm analyzeIfStatement ............................................................... 34
Figure 22 - Algorithm analyzeWhileStatement ...................................................... 34
Figure 23 - Algorithm analyzeDoStatement .......................................................... 35
Figure 24 - Algorithm analyzeForStatement ...................................................... 36
Figure 25 - Algorithm generateUniqueMethodName .......................................... 37
Figure 26 - NumberManipulation.java ............................................................. 45
Figure 27 - Coordinate.java ........................................................................... 45
Figure 28 - Example.java ................................................................................ 46
List of Tables

Table 1 - Large Java Software Attributes ................................................................. 7
Table 2 - Large Program Modification ...................................................................... 8
Table 3 - Dependency Model Merging – Method Addition Cases ............................ 28
Table 4 - Impact Analyzer Implementation Code Metrics ......................................... 39
Table 5 - Impact Analysis Performance Comparison ................................................ 40
Acknowledgements

I would like to thank Dr. Kothari for all of his guidance and inspiration. He has consistently challenged me and as a result I have achieved things beyond my own expectations. Dr. Mitra and Dr. Guan, thank you for helping oversee my research.

Thanks are due to all of my professors during my graduate study that educated me in topics that I am truly interested in. I would also like to thank all members of the KCS lab for many interesting discussions and debates over the past few years. I would especially like to thank Jeremias Saucedas for instilling in me a greater drive to learn. I would like to thank my parents Randy and Sandy for their unconditional love and support. Finally I would like to thank my fiancé Melissa for always loving me and keeping me motivated.
Abstract

Software impact analysis is defined as calculating the set of locations in the software that may be affected given an initial set of proposed changes. Traditionally, there has been a precision vs. computation tradeoff when performing impact analysis. Higher precision methods of calculating impact such as static slicing based techniques are too computationally intensive to be practical for large software. Other techniques such as structural analysis require far less computation, but are less precise. We present an incremental approach with an objective to reduce the number of methods that must be analyzed in order to compute impact. We define a program dependency model as a part of impact analysis to facilitate the incremental approach. We present an implementation in the form of a plug-in for the popular Eclipse IDE. Experimental results show a significant performance gain over traditional static-slicing based techniques.
1 Introduction

Impact analysis has been defined in many ways. Law and Rothermel define impact analysis as determining when a change in one part of a program affects other parts of the program [9]. Impact analysis has also been defined as the activity that identifies the parts of a program to be retested [5]. Ryder and Tip define impact analysis as providing feedback on the semantic impact of a set of program changes [17]. We define software impact analysis as calculating the set of locations in the software that may be affected given an initial set of proposed changes.

Although a program may have been thoroughly tested during its development phases, when it is modified, it must be retested to determine whether changes have been made correctly and to investigate whether those will cause any adverse effect on the program behavior [5]. As software systems become increasingly complex, the interaction between different components becomes harder to predict, and it is no longer feasible to rely on an expert to know what parts should be retested. However, retesting an entire system after making a small number of changes is a waste of resources. To save time and efforts, retesting should be done only on those parts that might be affected by changes, and change impact analysis serves that purpose [5].

The features and advantages gained from software impact analysis should drive the type of analysis performed. These features and advantages should also drive how a developer interacts with the tool and retrieves results. If a tool has excellent analysis results, but is difficult to use or integrate into the development cycle, its value is diminished. If impact analysis is slow it will rarely be used. Impact analysis is a verification and validation technique that must compete with other such techniques for
the maintainer’s time [16]. We propose a tool in the form of a plug-in for the popular Eclipse IDE. The focus of our tool is common sense usability and actionable results. The ease of use of a software impact analysis tool is essential to its success as an aid in the development cycle.

The impact analysis problem has been studied extensively. Many techniques have been devised for calculating impact. Arnold and Bohner [1] provide the following examples of impact analysis techniques:

- Using cross reference listings to see what other parts of a program contain references to a given variable or procedure
- Using program slicing to determine the program subset that can affect the value of a given variable
- Browsing a program by opening and closing related files
- Using traceability relationships to identify changing artifacts
- Using configuration management systems to track and find changes
- Consulting designs and specifications to determine the scope of a change

Another method not mentioned in the above is whole program path-based dynamic impact analysis, which attempts to calculate impact by obtaining dynamic information about system execution and from this information builds a representation of the system’s behavior that it uses to estimate impact [9]. Coupling measurement has also been used as a basis for supporting impact analysis in object-oriented systems. Coupling dimensions that seem to be significantly related to ripple effects are identified and used to rank classes according to their probability of containing ripple effects [2].
There are many types of changes that may impact other parts of a software system. Categories of atomic changes have been defined that all source code edits can be transformed into [17]. How change information is used to calculate impact differs greatly depending on the approach. A comparison of the architecture of impact analysis systems [1] shows that the approaches may differ in the following categories:

- Interface Object Model
- Internal Object Model
- Decomposition
- Impact Model
- Tracing / Impact Approach

Additionally, different impact analysis approaches vary based on the granularity of results.

Many researchers have studied the problem of impact analysis specifically with respect to object-oriented software systems [2] [3] [5] [6] [8] [10] [11] [12] [17]. This reflects the reality that the features provided by the object-oriented paradigm introduce new problems in the maintenance phase, including difficulty of identifying the affected components when changes are made [8]. Some researchers have chosen to study other aspects of impact analysis, such as the motivation for impact analysis [13] [20] or the application of impact analysis to a specific domain [14].

The extent of experimental study of these approaches varies. One simple approach based on coupling between classes has been tested on a relatively large code. It was tested on a C++ code consisting of approximately 90 classes and 40,000 lines of source [2]. Impact analysis based on static slicing has been experimented with to varying
degrees. Static slicing itself has been extensively investigated and experimented with [19]. Much of the research does not provide details about an implementation. Some implementations work on small test codes [15]. Other tools have been tested in the context of actual development of safety-critical avionics software [14], however no performance data has been given. Algorithmic analysis of static slicing based impact analysis also exist [11] [12].

2 Approach

The object-oriented paradigm is based on concepts such as encapsulation, inheritance, polymorphism, and dynamic binding. Although these concepts contribute to the reusability and extensibility of systems, they produce complex dependencies between classes and objects; there are data dependencies, control dependencies and state behavior dependencies that make it difficult to identify the ripple effects of changes [8][15]. A goal of impact analysis is to minimize the unexpected side effects of change [16]. With so many complex interactions in object-oriented software systems, the number of side effects that can be predicted manually is even less than with non-object oriented systems. The global information necessary to calculate these complex interactions implies that an automated process is better suited to perform this task.

Traditionally, an expert software developer familiar with the software system performs this analysis manually. When a change is made, the developer is expected to understand the interactions and know how the change will affect the rest of the system. However, performing this task manually is quite error prone. Lindvall and Sandahl [13] found that many more classes than predicted by an expert are actually changed in the software evolution process. Another key weakness of the manual approach is that project
developers are unaware of their own positive and negative capabilities in predicting change [13]. An automated process for calculating the impact of proposed changes can reduce the error in predicting side effects of new changes.

We present a method of software impact analysis based on maintaining a program dependency model. The model represents all of the dependencies necessary to compute the impact of a change throughout a software system. The program dependency model is incrementally updated as the software evolves. In this way, only changed program elements must be recomputed and updated in the program dependency model.

2.1 Definition of Dependency in Context of Impact Analysis

The goal of impact analysis is to identify, for a proposed change, the program artifacts that will be directly or indirectly affected by the change. The goal of the program dependency model is to contain precisely the information necessary for computing the impact. The program dependency model must also facilitate incremental updates, in order to cope with a constantly evolving software system. A proposed change to a program element will impact everything that depends on it. The definition of depends in this context is as follows:

*Given two program artifacts, A and B:*

\[ A \text{ depends on } B \text{ iff a change to } B \text{ has potential to propagate and impact } A \]

The definition of change differs depending on the type of program artifact. The simplest example is a dependency relationship between two variables. Given two variables \( q \) and \( r \), if changing the value of \( q \) has potential to change the value of \( r \), then \( r \) depends on \( q \). Another type of dependency relationship is between a variable and a method. For example, if changing the value of some variable \( x \) will possibly alter the
return value or execution of some method $f()$, then the return value of $f()$ depends on $x$.

The symbol $\rightarrow$ will be used to denote a dependency, with the element on the right hand side depending on the element on the left hand side of the symbol. The following types of dependency relationships in object-oriented code have been targeted, and are represented in the program dependency model:

- Variable $\rightarrow$ Variable
- Variable $\rightarrow$ Method
- Method $\rightarrow$ Variable
- Method $\rightarrow$ Method

In the above relationships, all variables and methods are all assumed to be members of a class (as in the Java programming language). Therefore, dependencies between classes are implicit, and based on the variable and function dependencies. The dependency model has one property in particular that is very important to note:

Particular instances of a class are not differentiated. Therefore, the impact of a change that impacts the members of a particular instance of a class will mark the impact in the class definition itself, and not relate the impact to any particular instance. This design decision is necessary to support the incremental model, the core of which is the function dependency mapping. Impact analysis results obtained from using this instance ignorant approach are still meaningful and useful. This approach may produce false positives in some cases, but never false negatives.

### 2.2 Incremental Model Update Example

There are two phases to impact analysis. The first phase is computing the dependency information of a program. For example, with slicing-based analysis, this
involves building abstract syntax trees and performing data flow analysis. For a
structural analysis, it might involve building the abstract syntax trees and analyzing the
class information and building some type of dependency graph. The second phase is the
propagation of the ripple effect of a proposed change. Phase two must be performed
every time a proposed change is to be analyzed, since the proposed change will most
likely be different every time. However, by maintaining the result of the first phase
incrementally, the time to perform the overall analysis can be greatly reduced.

We will now discuss an example of typical slicing-based software impact
analysis, and how our incremental approach compares in terms of computational effort.
We assume that the incremental model has been initialized and only incremental updates
are necessary to maintain the model at this point. Consider a Java program with the
following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Files</td>
<td>1000</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>1200</td>
</tr>
<tr>
<td>Number of Lines of Code</td>
<td>200,000</td>
</tr>
</tbody>
</table>

Now assume the following code is from the specified software system, and the change
shown is performed.
Let us assume that the class `SystemMonitor` is a very high level class and an instance of it is one of the top level objects in this system. Assume the `TempMonitor`, `HumidityMonitor`, and `VelocityMonitor` classes are very complex as well. Now assume the only change made by the user is that an field of type `AccelerationMonitor` is added to the class and the constructor is modified to call the constructor of this new field. At this point, let us say the developer wishes to calculate the impact of a proposed change to the software system. With a non-incremental approach, the first phase of impact analysis must be performed from scratch. This involves loading all 1000 files, inspecting all 1200 classes, and in the case of a slicing-based approach, generating and analyzing (data flow analysis) ASTs for all 200,000 lines of code. With our incremental approach, the model only needs to be updated for the class which has been changed (`SystemMonitor`) to reflect the added field and method call. This is assuming that the class `AccelerationMonitor` already existed in the system. Therefore, only one class needed to be reanalyzed during the first phase with the incremental approach, compared to all 1200 classes with the traditional approach. If the `AccelerationMonitor` class was also newly added, the only
analysis the incremental approach would have to perform would be to analyze the newly added class and then reanalyze SystemMonitor. The propagation phase proceeds in a similar fashion for both approaches. The time that is saved is during the first phase.

2.3 Method Input/Output Mapping

The method input/output mapping is the key to building the program dependency model. It can be thought of as a black box that given a set of inputs, some of which may be marked as impacted, will return the set of affected outputs. By creating a method dependency mapping for each method, the only time a method must be reanalyzed is when the method itself (or something it depends on) is changed. Otherwise, the mappings can be used to propagate the impacts. A method mapping contains all information necessary to propagate impact and to generate other method mappings. Inputs of a method mapping include formal parameters and class fields. These elements qualify as inputs because they are external sources that information can flow into a method. Outputs of a method mapping include class fields and the return value. These are all of the sources that information can flow out of a method. With Java, there is also the concept of a method throwing an exception. In our implementation, this behavior is not modeled, although it is certainly possible and is similar to the behavior of a method return. An example of a method input/output mapping is given in Figure 1.
class Coordinate{
    private String name;
    private int x, y;

    private getName(){
        return name;
    }

    public String setCoord(String nameVal, int xVal, int yVal){
        name = nameVal;
        int xTemp = xVal;
        int yTemp = yVal;
        x = xTemp;
        y = yTemp;
        return getName();
    }
}

Figure 1 - Method Mapping Code Example

Figure 2 - Simple Method Mapping Example

The input output method mapping for the getName method is very simple. The field name of class Coordinate is the input that affects the return value of this method. We now look at the other method, setCoord.

Figure 3 - Complex Method Mapping Example
We denote the mapping of the method *setCoord* as ‘complex’ because it depends on other method mappings, namely the mapping of the *getName* method. We see the following lines of code in the *setCoord* method:

```java
int xTemp = xVal;
int yTemp = yVal;
x = xTemp;
y = yTemp;
```

The values of *Coordinate.x* and *Coordinate.y* depend on the inputs *xVal* and *yVal* indirectly through the variables *xTemp* and *yTemp*. Since *xTemp* and *yTemp* are not externally visible data elements of the method, they do not appear in the method mapping. They are important however, as they are used during analysis to propagate the ripple effect of the parameters *xVal* and *yVal*. In order to create a method mapping, conservative static data flow analysis is performed within a method. At every possible exit point of a method (uses of the *return* keyword or at the end of a void method), the current set of live definitions as computed by the data flow module are saved. After analysis of the entire method is complete, each exit point is analyzed separately. Each of the live definitions is inspected, and only externally visible definitions and definitions of read variables of the return statement are used in the next step. Each of these definitions is traced backwards using backwards slicing. The slicing progresses backwards through the data flow graph, and continues until all leaves of the slice are discovered. These leaves represent the initial inputs used within the method that impact the final value of the variable at the current exit point. Each of the inputs from the leaves of the backwards slice is checked to see if it is an externally visible input (a class field or a formal parameter). If it is an externally visible input, a mapping is created between the externally visible input and the externally visible output. In this example, the only leaf
produced in the backwards slice of the externally visible output field \textit{Coordinate.x} is the parameter \textit{xVal}.

Another aspect of this example that needs discussion is the use of another method mapping in order to build the current method mapping. The method mapping of \textit{getName()} is used to calculate the impact at the point the method is invoked, instead of reanalyzing the method itself. The mapping for \textit{getName()} tells us that the field \textit{Coordinate.name} affects the return value of the method, which in turn affects the return value of the \textit{setCoord()} method. However, the input parameter \textit{nameVal} to the method \textit{setCoord()} first impacts the field \textit{Coordinate.name}. Therefore, when the backwards tracing algorithm is performed from the exit point of the \textit{setCoord()} function, the only leaf that will be traced will be the input parameter \textit{nameVal}.

\section*{2.4 Program Dependency Model}

The program dependency model (PDM) facilitates incremental updates for impact analysis. It is a forest structure, with each node representing a single method and edges denoting call order information. Each node is annotated with an analysis flag, denoting whether analysis is necessary. Each node contains a link to the method mapping for the given node. Method to file dependencies are also recorded.

When a user requests analysis of a proposed change, the model must first be updated. Given an existing PDM \textit{P0}, the incrementally updated PDM \textit{P3} is computed as follows:

- Build second PDM \textit{P1} from files modified since last incremental update, marking all methods in this PDM as needing analysis.
• Merge $P1$ with $P0$ resulting in $P2$. The only methods currently marked for analysis are the methods merged in from $P1$.

• Propagate the analysis flag throughout $P1$. If a method calls a method needing analysis, it too needs to be reanalyzed. This step generates $P3$.

• Recompute method mappings for the methods marked as needing analysis.

Three examples of this process are given in Figures 6-8. The following notation is used in these figures:

• A method name such as $m1$ enclosed in a circle denotes a method that is defined in the current scope of analysis and does not need analysis.

• A method name enclosed in a square denotes a method that is defined in the current scope of analysis, but needs analysis.

• A method name without a circle or a square denotes a method that is called but not defined in the current scope of analysis.

• At the bottom of each state the files involved in the current analysis context are shown, along with the method dependency information.

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td></td>
<td>m2</td>
<td>m1</td>
<td>m1</td>
</tr>
<tr>
<td>m2</td>
<td>m3</td>
<td>m4</td>
<td>m2</td>
<td>m2</td>
</tr>
<tr>
<td>m4</td>
<td></td>
<td></td>
<td>m4</td>
<td></td>
</tr>
</tbody>
</table>

$F1$: m1, m3, m4  
$F2$: m2  

$F1$: m1, m3, m4  
$F2$: m2  

$F1$: m1, m3, m4  
$F2$: m2  

**Figure 4 - PDM Modify Operation**
We first review the PDM modify example. $P0$ shows the initial state of the PDM, with all methods defined and analyzed. In this example, file $F2$ has been updated since the last PDE was built. $P1$ shows the newly built PDM based on file $F2$. We see that method $m2$ is the only method defined, and it contains a method call to $m4$, which is not defined in file $F2$. Since $m2$ comes from a modified file, it is marked as needing analysis. $P2$ shows the state after the initial merge has taken place. The method $m2$ has replaced the previous copy in the model. Finally, in state $P3$, the analysis flag has been propagated to method $m1$, since it calls $m2$, which needs analysis.

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Diagram of PDM Modify Example" /></td>
<td><img src="" alt="Diagram of PDM Modify Example" /></td>
<td><img src="" alt="Diagram of PDM Modify Example" /></td>
<td><img src="" alt="Diagram of PDM Modify Example" /></td>
</tr>
</tbody>
</table>

| F1: m1, m3, m4 | F1: m1, m3, m4 | F1: m1, m3, m4 | F1: m1, m3, m4 |
| F2: m2 | F2: m2, m5 | F2: m2, m5 |

**Figure 5 - PDM Add Operation**

The PDM add example is similar to the modification example, except that in addition to the presence of a definition of $m2$ and a call to $m4$, there is a call to a newly defined method $m5$. We see that after the merge has taken place ($P2$) the method $m5$ is now included in the dependency model. The method to file dependency has also been updated to reflect that $m5$ is included in $F2$. Similarly to the previous example, $P3$ shows the analysis flag after propagation.
For the delete operation, the method to file dependencies are very useful. At state $P1$, we see that method $m2$ has been removed from file $F2$ and replaced with the definition of method $m5$ that contains a call to $m4$. When merging $P0$ and $P1$, the file to method dependencies are compared. Method $m2$ is no longer in the file dependency list for $F2$; therefore it is removed from the model. In state $P2$ we see that $m5$ has been added to the model and linked to the defined method $m4$. Now that $m2$ has been removed, the single tree has been split in two. $P3$ is the same as $P2$ since there is nowhere for the analysis flag to propagate.
3 Implementation of Incremental Impact Analyzer

An implementation of the proposed impact analysis method has been developed. The purpose of this implementation is to provide a proof of concept, and to provide some answers to the following questions:

- Does this incremental approach provide a speed benefit over a non-incremental approach in practice?
- How does the use of non-instance aware propagation algorithms affect the results of impact analysis?
- What is the level of effort required to implement this type of analysis compared to other types of impact analysis?
- Does this analysis work on production level software, and how well?

The approach of this analysis has been described in the previous sections. However, to help answer the questions above a tool implementing incremental impact analysis is best suited. In the following sections we discuss the choice of using the Eclipse IDE as a target for developing this tool, the architecture of the tool, and the user interface provided by the tool.

3.1 Use of the Eclipse IDE

The Eclipse Integrated Development Environment (IDE) is a kind of universal tool platform – an open extensible environment for language development. It is primarily geared towards Java development, although it provides facilities for any language if support were written for it in the form of a plug-in. The plug-in architecture of Eclipse is very robust, allowing almost all of the functionality of the IDE to be controlled by a
loaded plug-in. Eclipse has many of the convenience features of an IDE such as auto-completion, type checking, etc. built in. Eclipse also provides support for many powerful refactoring operations, including rename, move, change method signature, use supertype where possible, etc. These convenience features and refactoring operations require more than just syntactic analysis to perform, instead relying on information extracted from abstract syntax trees that are built by the IDE on the fly. These ASTs capture semantic information that is important for refactoring and program analysis. We leverage the fact that these trees can be constructed by Eclipse and use them as the base for our analysis.

Other attractive features of targeting our tool towards the eclipse environment are the large user base. Eclipse is a free, open-source tool that has a strong following in academia and industry. The user interface is familiar to many developers, and therefore a tool based on this interface will have a smaller learning curve. The simple fact that the tool is integrated into the environment means that it can be more easily streamlined into the development process. Other facilities of the IDE, such as software project management, can be leveraged as well. Finally, as mentioned before, Eclipse is open-source software. The low-level details of implementation are available for all to see, which is very important, especially when creating a plug-in that is as close to the core of Eclipse as this tool is.

3.2 Architecture

The architecture of this tool can be broken into several components. There is the user interface component, which will be discussed in the following section. The other components of this tool include the incremental update manager, the program
dependency model, the proposed change analyzer, and the impact propagator. A diagram of the analysis cycle is shown in the following figure.

![Analysis Cycle of Incremental Impact Analyzer](image)

**Figure 7 - Analysis Cycle of Incremental Impact Analyzer**

The following subsections describe the components of the analysis cycle of the incremental impact analyzer.

### 3.2.1 Incremental Update Manager

The incremental update manager is responsible for locating the parts of a program that have been changed since last time the program dependency model was updated. In the implementation created for this paper, the update manager simply identifies the changed entities at a file level, although a more advanced update manager could be implemented to identify changes at the class level, or potentially at multiple levels of granularity. Once the incremental update manager has identified the change points, this
information is passed along to the program dependency model for the updates to take place.

3.2.2 Program Dependency Model Updater

The program dependency model must be updated with the incremental changes in order to propagate the proposed changes with respect to the current state of the software system. The incremental update manager provides the change points to the program dependency model. Currently, these change points are at the file level of granularity. Every method element in the program dependency model has a file association. The merging and analysis propagation process described in section 2.4 takes place at this stage.

3.2.3 Proposed Change Analyzer

At this point the program dependency model is up to date, and phase two of the analysis can proceed. Data flow analysis is performed on the file in which the proposed change is marked. The change analyzer inspects the highlighted code provided by the user, and uses this along with the local data flow information to calculate the impacted outputs of the affected local methods. This is done by performing a backwards slice for each of the output elements at every possible exit point of the methods involved. We refer to the set generated at this stage as the initial impact set.

3.2.4 Impact Propagator

The proposed change analyzer has marked the initial impact set. It is now necessary to calculate the ripple effect caused by these changes. The program dependency model is used to propagate the impact throughout the system. As new inputs are marked as impacted, they are added to the impact result set. The program
dependency graph most likely contains cycles, so without some criteria for stopping, propagation will proceed indefinitely. The stopping criteria is that if the propagator has traversed all method mappings a single time without adding a single item to the impact result set, the propagation stage is complete. The results of analysis should be displayed to the user in a meaningful way.

3.3 User Interface

The goal of the user interface for this plug-in is to be extremely easy to use and intuitive. There are two aspects of the user interface: marking proposed changes and viewing the impact analysis results. Marking proposed changes is simply done by highlighting sections of code in the editor window where the proposed change is located, and selecting the “Calculate Impact” menu item. The second purpose of the user interface is to display the results of analysis. It should provide a developer with the results in a way that action can be taken. Typically in object-oriented software development, test cases are applied to classes in the form of unit tests. Since software change impact analysis drives regression testing during software evolution, the developer needs to know what should be retested after making a change. The results are returned as a text view of the Eclipse workspace, and give information about what class fields and method return values may be impacted by the proposed change.
4 Algorithm Design

This section covers the different algorithms used to perform our method of incremental impact analysis. We first describe the notation used for the algorithms before presenting them. An algorithm has some input, and produces either an output or a result. There is a distinction between an output and a result for these algorithms. Outputs are explicitly returned to the caller, whereas results merely are the final result of performing the algorithm. For example, a user may pass a reference to a list to be sorted, and expect that after the algorithm executes, the reference points to a sorted list, although the algorithm does not explicitly return the list. Inputs and outputs to algorithms are defined at the beginning of the algorithm definition. Some other basic data structures are used in the algorithms, such as Set and List. Simple operations on these data types are also used, such as List.add() or Set.remove(). Operations having a bold typeface are algorithms defined in this document. Another notation used is the familiar object-oriented concept of the ‘dot’ operator. This is used to reference a file (or property) of a data item. Each algorithm includes a non-formal description to clear up any ambiguities in the notation.

4.1 Incremental Update Manager Algorithms

The goal of the incremental update manager is to track changes to files in order to support incremental updates to the program dependency model. Our implementation works on the level of tracking updates to files. One will note that the program dependency model works on the level of methods. Therefore our program dependency model has to track a file to method mapping in order to use the results of the incremental update manager. If
one were to implement an incremental update manager that could track updates at the
method level as opposed to the file level, the file related elements could be removed from
the program dependency model. Modifying the incremental update manager to work at
the method level would also most likely lessen the number of methods requiring analysis
for a given update, since only changed methods would be marked as updated instead of
all methods of a changed file.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>updateFileModificationStamps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Set of file dependency model objects</td>
</tr>
<tr>
<td>Output/Result</td>
<td>File dependency model objects from input have modification stamp added</td>
</tr>
<tr>
<td>input Set fileModelObjects</td>
<td></td>
</tr>
<tr>
<td>foreach fileModelObject</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>value = calculateModificationStamp(currentModelObject)</td>
</tr>
<tr>
<td></td>
<td>updateModificationStamp(currentModelObject, value)</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8 - Algorithm updateFileModificationStamps**

We begin with a simple algorithm for updating the modification stamps for files in a
software developer’s project. The input is a set of objects representing the files in the
project, and the result of this algorithm is that each object has been updated with the
current modification stamp. In the implementation of this algorithm, modification stamps
change when a file is changed and then saved.
### Algorithm returnModifiedFileObjects

<table>
<thead>
<tr>
<th>Input</th>
<th>Set of all file dependency model objects contained in the model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output/Result</td>
<td>Set of file dependency model objects</td>
</tr>
</tbody>
</table>

```java
input Set fileModelObjects
output Set modifiedFileObjects

foreach fileModelObject {
    calculateModificationStamp
    if(oldModificationStamp does not exist) {
        modifiedMethods.add(currentFileModelObject)
    }
    else if(oldModificationStamp != newModificationStamp) {
        modifiedMethods.add(currentFileModelObject)
    }
}
return modifiedFileObjects
```

Figure 9 - Algorithm returnModifiedFileObjects

This algorithm is responsible for outputting a set of file model objects that have been updated since the last incremental update to the program dependency model. It simply loops over all file objects, and adds files that are either new (no modification stamp previously exists) or modified (existing and updated modification stamps are different).

The output of this algorithm is consumed by the program dependency model to compute the methods that should be reanalyzed before calculating any impact effects.

### 4.2 Program Dependency Model Algorithms

### Algorithm returnMethodCalls (recursive)

<table>
<thead>
<tr>
<th>Input</th>
<th>An AST node (starting from a MethodDeclaration node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output/Result</td>
<td>A set of method invocations that appear within the given method</td>
</tr>
</tbody>
</table>

```java
input ASTNode currentNode
global output Set methodInvocations

if(currentNode == MethodCall &&
   methodInvocations does not contain MethodCall.name) {
    methodInvocations.add(currentNode)
}
foreach childNode {
    returnMethodCalls(currentChildNode)
}
return methodInvocations
```

Figure 10 - Algorithm returnMethodCalls (recursive)
The `returnMethodCalls` algorithm is recursive in nature. In practice it is implemented using the Visitor software design pattern. The goal of this algorithm is to generate a method call profile for a given method. It walks all children of a method subtree and inspects the current node along the way. If the current node being walked is a method call, it is added to the output set.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th><code>linkMethodDependencyObjects</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Set of unlinked method dependency model objects</td>
</tr>
<tr>
<td>Output/Result</td>
<td>Method dependency model objects have been linked</td>
</tr>
</tbody>
</table>

```java
input methodObjects

foreach methodObject {
    Set methodCalls = \textit{returnMethodCalls}(curMethodObject.astLink)
    foreach methodCall {
        if(methodObjects.contains(methodCall.name)) {
            calledMethod = methodObjects.get(methodCall.name)
            curMethodObject.addChild(calledMethod)
            calledMethod.addParent(curMethodObject)
        }
    }
}
```

**Figure 11 - Algorithm `linkMethodDependencyObjects`**

Dependencies between method model objects are stored as a directed graph representation. The `addChild` and `addParent` operations performed build these graph dependencies. This algorithm loops over all method model objects, and builds parent and child relationships based on the call profiles generated by `returnMethodCalls`.

Calculating the directed call order graph is important because the program dependency model is based on this graph.
**Algorithm orderMethodsForAnalysis**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of linked method dependency model objects</td>
<td>List containing method model objects in order suitable for analysis</td>
</tr>
</tbody>
</table>

```java
input Set methodObjects
output List result

while(methodObjects.size > 0) {
    Set removals
    // check for recursive looping behavior
    foreach methodObject{
        boolean isReady = true
        foreach curMethodObject.children {
            if(methodObjects.contains(curChild)){
                isReady = false
            }
        }
        if(isReady) {
            removals.add(curMethodObject)
        }
    }
    methodObjects.removeAll(removals)
}

return result
```

**Figure 12 - Algorithm orderMethodsForAnalysis**

This algorithm orders the methods for analysis. Method input/output mappings will be generated for each method. When a call to a method is found when calculating the input/output mappings for another method, the I/O mappings for the method being called must already be present. The mapping for a given method is built upon the method mappings of the methods it calls.

**Algorithm createDepLink**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two dependency objects</td>
<td>Parent and child links are created between the two elements</td>
</tr>
</tbody>
</table>

```java
input MethodObject parent
input MethodObject child

parent.addChild(child)
child.addParent(parent)
```

**Figure 13 - Algorithm createDepLink**
This algorithm simply creates a directed link between two method model objects, and is used in `mergeDependencyModels` while building the overall program dependency model.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>clearChildLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>A dependency object</td>
</tr>
<tr>
<td>Output/Result</td>
<td>All child links are cleared, and each child’s parent link is also removed</td>
</tr>
<tr>
<td>input MethodObject parent</td>
<td></td>
</tr>
</tbody>
</table>
|                  | foreach parent.children {
|                  |   curChild.removeParent(parent)
|                  | } |
|                  | parent.clearChildren() |

**Figure 14 - Algorithm clearChildLinks**

This algorithm removes child links from a given input node, and removes the parent links from the children as well. This algorithm is also used when merging two dependency models.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>mergeDependencyModels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Two dependency models that must be merged. One is the current model, the other is the new changes that must be merged into the current model.</td>
</tr>
<tr>
<td>Output/Result</td>
<td>The dependency model m2 is merged into m1, and model elements needed to be reanalyzed have been flagged</td>
</tr>
<tr>
<td>input DependencyModel m1</td>
<td></td>
</tr>
<tr>
<td>input DependencyModel m2</td>
<td></td>
</tr>
<tr>
<td>Set changePoints</td>
<td></td>
</tr>
<tr>
<td>// check for new files</td>
<td></td>
</tr>
<tr>
<td>foreach m2.files {</td>
<td></td>
</tr>
<tr>
<td>if(!m1.files.contains(currentFile) {</td>
<td></td>
</tr>
<tr>
<td>// merge file-&gt;method mappings for new file</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>// look for deleted methods</td>
<td></td>
</tr>
<tr>
<td>foreach m1.files {</td>
<td></td>
</tr>
<tr>
<td>if(m2.files.contains(currentFile) {</td>
<td></td>
</tr>
<tr>
<td>foreach currentFile.method {</td>
<td></td>
</tr>
<tr>
<td>if(!m2.methods.contains(currentMethod) {</td>
<td></td>
</tr>
<tr>
<td>// delete currentMethod from m1</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 15 - Algorithm mergeDependencyModels**
Figure 15 (Continued) - Algorithm mergeDependencyModels

This algorithm is one of the single most complex algorithms used in the incremental impact analysis process. It takes two program dependency model objects, and merges the
new model into the existing model. Complexities arise when considering the following issues:

- The model being merged is not a complete model, but only a model generated from the incremental analysis.
- The updated model may include new methods, existing methods, or a combination of both.
- The updated model may include links to method calls to methods that are not defined in the existing model or in the updated model (system library calls) that should be removed during the merge.
- Since method input/output mappings are built on the mappings of methods being called, the merge should calculate all model elements that must be reanalyzed after the merge by traversing the graph backwards (with respect to dependency).

When merging an addition the algorithm performs several different functions during the merge to account for different cases. The cases and the actions taken are listed below.

### Table 3 - Dependency Model Merging – Method Addition Cases

<table>
<thead>
<tr>
<th>Case</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>New method with link to existing method</td>
<td>The new method is added to the model, and the link to the updated representation of an existing method will be removed and replaced by the existing representation.</td>
</tr>
<tr>
<td>New method with link to new method</td>
<td>The new method is added to the model, and the new method is added to the new model if it has been defined in the updated model.</td>
</tr>
<tr>
<td>Existing method with link to existing method</td>
<td>Links are updated within the existing representations of the methods in the existing model.</td>
</tr>
<tr>
<td>Existing method with link to new method</td>
<td>New method is added to the existing model if a definition of the method is present in the updated model.</td>
</tr>
</tbody>
</table>
4.3 Proposed Change Analyzer Algorithms

The proposed change analyzer works within a single method body. It takes the highlighted code as user input, and calculates the outputs of the method mapping that should be marked as changed. This task can be split into three algorithms: marking the selected definitions, tracing the definition use chains, and marking the impacted outputs.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>markValues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Information regarding the user text selection</td>
</tr>
<tr>
<td>Output/Result</td>
<td>Definitions are marked, and a set of marked values is returned</td>
</tr>
</tbody>
</table>

```plaintext
input String fileName
input int offset
input int length
output Set markedValues

ast = buildAST(fileName)
dfa = dataFlowAnalysis(ast)
foreach (dfa.definitions) {
  if (curDefinition.position in selection) {
    curDefinition.isMarked = true
  }
  foreach (curDefinition.readVariables) {
    if (curReadVariable.position in selection) {
      curDefinition.isMarked = true
      markedValues.add(curReadVariable)
    }
  }
}
return markedValues
```

Figure 16 - Algorithm markValues
Algorithm | `calculateImpactOfMarks`  
--- | ---  
Input | Set of marked values computed by `markValues`, and DFA information  
Output/Result | Impacted outputs of MethodMapping are returned  
input | Set markedValues  
input | DFA dfainfo  
output | Set markedOutputs  

// mark obvious outputs  
foreach markedValue {  
    if(curMarkedValue.isMappingOutput) {  
        markedOutputs.add(curMarkedValue)  
    }  
}  

// from each return point, trace backwards  
foreach dfainfo.returnPoint {  
    foreach dfainfo.returnPoint.liveDefinitions {  
        if(curLiveDef.isMappingOutput) {  
            if(curLiveDef.tracesToMark()) {  
                markedOutputs.add(curLiveDef.writeVariable)  
            }  
        }  
    }  
}  

return markedOutputs  

**Figure 17 - Algorithm calculateImpactOfMarks**

If an output was explicitly marked by the user, it is obviously an impacted output. The second part of this algorithm is to start from each possible exit point of the method, and trace the exit point live definitions backwards in search of use of a marked definition. If there was any marked definition in the backward trace of this exit point definition, it should be marked also.
<table>
<thead>
<tr>
<th><strong>Algorithm</strong></th>
<th><em>tracesToMark (recursive)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>A definition</td>
</tr>
<tr>
<td><strong>Output/Result</strong></td>
<td>Returns true if a backward trace leads to a marked value, else returns false</td>
</tr>
</tbody>
</table>

```java
input Definition currentDef

if(currentDef.isMarked) {
    return true
}
foreach currentDef.defsThisUses {
    if(currentDefThisUses.tracesToMark()) {
        return true
    }
}
return false
```

**Figure 18 - Algorithm tracesToMark (recursive)**

This recursive function performs the backward tracing through the definition use chains, and is used by the `calculateImpactOfMarks` algorithm to find the impacted output set of a method.
4.4 Impact Propagator Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>propagatelmpact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>The program dependency model with initial impact marks</td>
</tr>
<tr>
<td>Output/Result</td>
<td>The marks have propagated throughout the model</td>
</tr>
</tbody>
</table>

```java
input ProgramDepModel model
input Set initialMarks

Set impactSet
int lastMarkedIndex = 0

while(true)
    foreach model.methodMapping {
        Boolean anyUpdates = false
        foreach(curMapping.inputSet) {
            if(impactSet.contains(curInput)) {
                impactSet.add(curMapping.getOutputMap(curInput))
                anyUpdates = true
            }
        }
        if(anyUpdates) {
            lastMarkedIndex = curMapping.index
        } else if(lastMarkedIndex = curMapping.index) {
            return
        }
    }
```

**Figure 19 - Algorithm propagatelmpact**

The `propagatelmpact` algorithm runs continuously until no changes have been made to the impact set for one full traversal of the method list. This ensures that any additions to the impact list will have a chance to propagate to completion. Once this algorithm has been applied, the impact has been calculated and can be displayed to the user.

4.5 Method Mapping (DFA) Algorithms

The algorithms given in this section are those required to perform conservative analysis of conditional statements whose execution paths cannot be determined statically. These algorithms are implemented as part of the Visitor software design pattern. In the
following algorithms, the `analyze()` method is dispatched to the correct algorithm (method) to analyze the object, depending on its type. Only conservative analysis algorithms are given here, but there are many other algorithms that work on expressions, assignments, etc. that are responsible for performing read/write analysis and creating and maintaining the current set of live definitions. The entire Visitor class for performing data flow analysis is provided in the appendix.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>A set of definitions to be merged into the current live definitions</td>
</tr>
<tr>
<td>Output/Result</td>
<td>The live definitions are merged with the new definitions</td>
</tr>
<tr>
<td>Definitions</td>
<td>newDefinitions</td>
</tr>
<tr>
<td>global Definitions</td>
<td>liveDefinitions</td>
</tr>
</tbody>
</table>

```java
liveDefinitions.addAll(newDefinitions)
```

**Figure 20 - Algorithm merge**

The `merge` algorithm is used by all conditional analysis algorithms to produce a conservative result. A conditional analysis algorithm, such as the algorithm to analyze an `if` statement, must take different execution paths into account. One path may be when the test condition evaluates to true and the `then` portion of the `if` statement is executed. Another case may be when the test condition evaluates to false and instead the `else` portion of the `if` statement is executed. To simulate both scenarios during analysis, the state of live definitions is cached before analyzing either the `then` or `else` block. Then each block is analyzed starting from the state before the `if` statement. After the definitions have been calculated down these mutually exclusive paths, the two sets must be merged. The merge algorithm simply merges all new definitions together in a way that the new set created is a superset of the two separate sets.
### Algorithm analyzeIfStatement

**Input**
An AST node representing an if statement

**Output/Result**
The live definitions are updated with conservative DFA results

```java
input ASTNode ifStatement
global Definitions liveDefinitions

ifStatement.getTest().analyze()
Definitions liveDefsCopy = liveDefinitions
ifStatement.getThen().analyze()
Definitions thenDefsCopy = liveDefinitions
liveDefinitions = liveDefsCopy

if(ifStatement.hasElse()) {
    ifStatement.else().analyze()
}
merge(thenDefsCopy)
```

**Figure 21 - Algorithm analyzeIfStatement**

The algorithm for an `if` statement was used in the discussion of the merge algorithm, so discussion of it here will be brief. An `if` statement can have two forms: `if/then/else` or `if/then`. This algorithm analyzes the test condition first, which is always executed. Then in the case of an `if/then/else` analyzed the then and else blocks and merges the results. In the case of `if/then` the `then` block is merged with the definitions before the `then` block.

### Algorithm analyzeWhileStatement

**Input**
An AST node representing a while loop statement

**Output/Result**
The live definitions are updated with conservative DFA results

```java
input ASTNode whileStatement
global Definitions liveDefinitions

whileStatement.getTest().analyze()
Definitions liveDefsCopy = liveDefinitions
whileStatement.getBody().analyze()
whileStatement.getTest().analyze()
Definitions oneLoopIterationDefsCopy = liveDefinitions
whileStatement.getBody().analyze()
whileStatement.getTest().analyze()

merge(liveDefsCopy)
merge(oneLoopIterationDefsCopy)
```

**Figure 22 - Algorithm analyzeWhileStatement**
At a high level, the algorithm to analyze a while loop is similar to the algorithms to analyze other loop types such as the do while loop and for loop. Different execution paths are merged, including no iterations, one iteration, and two iterations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>analyzeDoStatement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>An AST node representing a do while loop statement</td>
</tr>
<tr>
<td>Output/Result</td>
<td>The live definitions are updated with conservative DFA results</td>
</tr>
</tbody>
</table>

```java
input ASTNode doWhileStatement
global Definitions liveDefinitions

doWhileStatement.getBody().analyze()
doWhileStatement.getTest().analyze()
Definitions liveDefsCopy = liveDefinitions
doWhileStatement.getBody().analyze()
doWhileStatement.getTest().analyze()
merge(liveDefsCopy)
```

Figure 23 - Algorithm analyzeDoStatement

The algorithm analyzeDoStatement analyzes the behavior of a single iteration and two iterations of the loop. Note that with a do while loop, the body is always executed at least once since the test is not checked until after the first iteration.
### Algorithm `analyzeForStatement`

<table>
<thead>
<tr>
<th>Input</th>
<th>An AST node representing a for loop statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output/Result</td>
<td>The live definitions are updated with conservative DFA results</td>
</tr>
</tbody>
</table>

```java
input ASTNode forStatement
global Definitions liveDefinitions

List initializers = forStatement.getInitializers()
List updaters = forStatement.getUpdaters()

foreach initializers {
    initializer.analyze()
}
forStatement.getTest().analyze()
Definitions liveDefsCopy = liveDefinitions
forStatement.getBody().analyze()

foreach updaters {
    updater.analyze()
}
forStatement.getTest.analyze()
Definitions oneLoopIterationDefsCopy = liveDefinitions
forStatement.getBody().analyze()

foreach updaters {
    updater.analyze()
}
forStatement.getTest.analyze()

merge(liveDefsCopy)
merge(oneLoopIterationDefsCopy)
```

**Figure 24 - Algorithm `analyzeForStatement`**

The `analyzeForStatement` algorithm is somewhat more complex than the other two loop analysis algorithms. *For* loops contain an initialization section and an update section in addition to the test expression. Most of the added complexity in this algorithm is to correctly analyze the initializers and updaters in the correct order. This algorithm analyzes execution of zero, one, and two iterations and merges the results.
### 4.6 Miscellaneous Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>generateUniqueMethodName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>An AST node representing a MethodDeclaration or MethodInvocation</td>
</tr>
<tr>
<td>Output/Result</td>
<td>A String with a unique name generated from method properties</td>
</tr>
<tr>
<td>input</td>
<td>ASTNode method</td>
</tr>
<tr>
<td>output</td>
<td>String uniqueName</td>
</tr>
</tbody>
</table>

```
uniqueName += method.fullyQualifiedPackageName
uniqueName += '. ' + method.name
uniqueName += '-' + method.fullyQualifiedReturnType
foreach method.parameter {
    uniqueName += '-' + currentParameter.fullyQualifiedType
}

return uniqueName
```

**Figure 25 - Algorithm generateUniqueMethodName**

This algorithm simply builds a string that will be unique for each unique method. It uses the package name, method name, return type, and parameter types to build the name.

This unique method name is the key that is used to store and look up methods in the program dependency model and all associated analyzers.
5 Results

5.1 Performance of Incremental Approach

The primary advantage of using this incremental approach is the promise of the accuracy of slicing-based impact analysis without the computational requirements typically required of this type of approach. Since the amount of computation required to update the incremental model is very closely tied to the changes made, we cannot state definitively the speed increase gained from this approach. Indeed, diabolical examples exist that involve a single change requiring the entire program dependency model to be updated.

The environment used for testing purposes was configured to be as close to a real world environment as possible. We used Eclipse version 2.1.1 running on Windows XP as the platform our implementation plugged into. The host computer included a Pentium 4 CPU running at 2.4 GHz and 512 megabytes of memory. There were two sets of code (Eclipse projects) used for testing. The first project used for testing consisted of simple test cases for verification of correctness. The second project was a large and complex code used for performance benchmarking. The code for this second project was actually the code for our incremental analyzer. This code provided not only a real world target to perform measurements with, but it also tested the robustness of our analyzer at handling real-world codes. The metrics of the large project are given in Table 4.
Table 4 - Impact Analyzer Implementation Code Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Files</td>
<td>28</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>540</td>
</tr>
<tr>
<td>Number of Lines of Code</td>
<td>8951</td>
</tr>
</tbody>
</table>

There were two methodologies used in the testing process. For assessing the correctness of the tool, the tool analyzed the small test cases, and the results were checked manually. These test cases were also useful during the debugging process while developing the tool. The results of manually checking small test cases showed that the output of the tool is what is expected. Some false positives do exist in the output, but this is to be expected in any static analysis tool.

The second type of test that was done was a performance test. The goal of this test was to measure the performance of our incremental approach compared to a non-incremental slicing approach. Two performance measures were calculated: the number of methods analyzed to calculate impact, and the time required to analyze these methods. In order to take these measurements, our tool was modified to measure the time taken to rebuild the necessary abstract syntax trees, update the dependency model, perform data flow analysis, and propagate the impact. The number of methods reanalyzed in this process was also logged.

Three types of analysis were performed. The first was impact analysis (*Normal Impact Analysis*) without using method mappings or incremental updates. Every time a method is called, data flow analysis must be performed. The analysis time for normal impact analysis was 12338ms. The second analysis (*Method Map Impact Analysis*) uses method mappings, but does not use incremental updates. Data flow analysis is only
performed on each method once with this approach. The analysis time for method map impact analysis was 6885ms. All 540 methods were analyzed using these two approaches. The final type of impact analysis performed was the incremental analysis described in this paper. Table 5 displays the results of the performance testing on the large body of code. Columns two and three show the speedup of an incremental approach with respect to the other two approaches.

Table 5 - Impact Analysis Performance Comparison

<table>
<thead>
<tr>
<th>Sample</th>
<th>Normal / Incremental</th>
<th>Method Map / Incremental</th>
<th>Incremental Analysis Time</th>
<th>Incremental # Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23.957</td>
<td>13.370</td>
<td>515 ms</td>
<td>48</td>
</tr>
<tr>
<td>2</td>
<td>65.979</td>
<td>36.818</td>
<td>187 ms</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>43.907</td>
<td>24.502</td>
<td>281 ms</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>32.901</td>
<td>18.360</td>
<td>375 ms</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>39.419</td>
<td>21.997</td>
<td>313 ms</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>131.255</td>
<td>73.245</td>
<td>94 ms</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>10.130</td>
<td>5.653</td>
<td>1218 ms</td>
<td>69</td>
</tr>
<tr>
<td>8</td>
<td>71.733</td>
<td>40.029</td>
<td>172 ms</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>158.180</td>
<td>88.269</td>
<td>78 ms</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>60.778</td>
<td>33.916</td>
<td>203 ms</td>
<td>76</td>
</tr>
<tr>
<td>Average</td>
<td>63.824</td>
<td>35.616</td>
<td>343.6 ms</td>
<td>45.7</td>
</tr>
</tbody>
</table>

We see that by simply using method mappings, a speedup of approximately 1.8 is achieved compared to normal impact analysis. When the incremental approach is used, a performance gain of an order of magnitude is achieved with respect to both normal and method map based impact analysis.

5.2 Effectiveness of Instance-Unaware Approach

The data flow analysis used to drive analysis and the program dependency model does not track specific instances of classes. While this approach will never produce false
negatives, the potential for producing more false positives than an instance-aware approach exists. Just as the number of false positives produced by conservative analysis depends on the code that is being analyzed, the increase in false positives from an instance-aware to an instance-unaware approach also depends on the code being analyzed. In order to attempt to gauge the effect of using an instance-unaware approach, several coding scenarios will be examined, and the results of an instance-aware approach will be compared to the instance-unaware approach used in this method of analysis.

For this discussion, we will group the use of variables of the same type into three categories:

- Single instance of a variable of a given type
- A small, finite set of instances of a given type
- A large set of instances of a given type, stored within another data structure

With the first usage scenario, only one instance of a given type exists in the context of analysis. Since only one instance exists, an instance-unaware approach performs just as well as an instance-aware approach. There is only one possible variable of the given type, and knowing the instance is not necessary. This is the simplest scenario of the three, however in many applications this scenario is very common.

The second scenario involves a small, finite set of instances of a given type. An example of this scenario could be a calendar application that tracks the days Sunday through Saturday using objects of type `WeekDay`. In this example, exactly seven instances of objects of type `WeekDay` will be created. Consider an example where the instance representing `Monday` has a field written to, and then the same field is read from the instance representing `Tuesday`. The instance unaware approach will assume that
impact is propagated through this sequence of actions, while an instance aware approach will know that the objects Monday and Tuesday are not the same instance. This scenario is the most likely to cause an increase in the number of false positives due to an instance unaware approach.

The final scenario is the case of a large, perhaps ever-changing number of instances of a given type, stored in a data structure. Examples of data structures typically used are arrays, lists, sets, trees, etc. The typical approach for static analysis is to treat an array and similar data structures as a single variable, and not to attempt to track the elements being inserted and removed/read from the data structure. Therefore even with an instance aware approach, precision regarding instances is lost when an element is inserted into a data structure of this type. In a similar fashion, this same precision is lost with the instance unaware approach.

From these three common usage scenarios, we see that the most troublesome case for an instance unaware approach is scenario two. However, scenarios one and three are also very common usage patterns. In the end the increase in the number of false positives dependant on instance tracking is related to the code being examined.

5.3 Conclusions & Future Work

This research has shown that an incremental approach to performing impact analysis can provide significant performance gains. With impact analysis methods there has traditionally been a performance to precision tradeoff. This method provides a slicing-based approach with a smaller computational burden. The precision tradeoff takes the form of using an analysis that does not differentiate between different instances of a type that results in a higher number of false positives. The application of the results
of impact analysis must be considered. Often, as mentioned previously, the results of impact analysis are used to drive regression testing. Some number of false positives is acceptable, because if after performing impact analysis and retesting all items marked as impacted, the developer can say that all elements that were actually impacted (plus some that weren’t) were retested. Impact analysis facilitates targeted regression testing saving developers time and effort.

Research into applications of impact analysis results could be done to increase the utility of the results. For example, the results of impact analysis could be used along with information contained in the program dependency model to execute test cases in a meaningful order that tested the dependencies. Finally, work could be done to increase the precision of the approach presented here. While the current form is not aware of specific instances of types, some extensions could possibly be made to facilitate some of this analysis. While the use of a program dependency model constrains the amount of instance tracking that can be done, an approach using multiple levels of granularity could be devised to provide more accurate results where possible.

Incremental impact analysis provides a method of reducing the computational burden associated with traditional slicing-based impact analysis. Method mappings enable the analyzer to only compute the data flow information for a method once as opposed to every time a method is invoked. Method mappings also serve as a mechanism to propagate impact through only the relevant program elements, completely removing slicing through methods from this process. The incremental approach ensures that only the modified methods will be analyzed between two analysis periods.
6 Appendix: Analysis Log Example

This appendix is very useful for understanding the full analysis process. There are three files involved in the analysis log shown. The contents of the three files are below. The log shows the process of analysis starting from an empty program dependency model; therefore all of the methods are analyzed in the first pass. The highlighted area in the file NumberManipulation.java is the initial impact being calculated. First, the files needing to be rebuilt are calculated. Next, the order of analysis for the methods is determined. Then each method is analyzed in order and a method mapping is created. After each mapping is created, it is associated with a model element in the program dependency model. After all of the analysis is complete and the program dependency model is current, the local impact of the highlighted code is calculated. Finally, this local impact is propagated throughout the model and the final impact is calculated.
package edu.iastate.kcs;

public class NumberManipulation {
    int a=0, b=1, c=2, x=3, y=4, z=5;
    NumberStorage storage = new NumberStorage();
    String s = new String("Hello, world!");

    public void manipulateFields()
    {
        String s2 = new String(s);
        b = s2.hashCode();
        storage.setI(a);
        x = storage.getI();
    }
}

class NumberStorage{
    private int i=0;

    public void setI(int value){
        i = value;
    }
    public int getI(){
        return i;
    }
}

Figure 26 - NumberManipulation.java

package edu.iastate.kcs;

public class Coordinate {
    private String name;
    private double x, y;

    public Coordinate(double xArg, double yArg){
        name = new String();
        x = xArg;
        y = yArg;
    }
    public Coordinate(String nameArg, double xArg, double yArg){
        name = nameArg;
        x = xArg;
        y = yArg;
    }
    public double getX(){ return x; }
    public double getY(){ return y; }
    public String getName() { return name; }
    public String factorCoords(double factor){
        x = x * factor;
        y = y * factor;
        return name;
    }
}

Figure 27 - Coordinate.java
package edu.iastate.kcs;

public class Example {
    Coordinate origin;
    public static void main(String args[]) {
        if (args.length != 2) {
            System.err.println("usage: java Example xCoord yCoord");
            return;
        }
        Example ex = new Example();
        Coordinate user = new Coordinate("user", Double.parseDouble(args[0]),
                Double.parseDouble(args[1]));
        double result = ex.calcDistanceToOrigin(user);
        System.out.println("Distance to origin: " + result);
    }
    public Example(){
        origin = new Coordinate("origin", 0, 0);
    }
    public double calcDistanceToOrigin(Coordinate coord){
        return calcDistanceBetweenCoords(origin, coord);
    }
    public double calcDistanceBetweenCoords(Coordinate c1, Coordinate c2){
        double c1x = c1.getX();
        double c2x = c2.getX();
        double c1y = c1.getY();
        double c2y = c2.getY();
        double xDelta = Math.abs(c1x - c2x);
        double yDelta = Math.abs(c1y - c2y);
        return Math.sqrt(Math.pow(xDelta, 2) + Math.pow(yDelta, 2));
    }
}

Figure 28 - Example.java

ANALYSIS LOG:

SELECTED INFO: NumberManipulation.java
Selection Offset: 147
Selection Length: 39

SEARCHING FOR MODIFIED FILES:

File: Coordinate.java
File: Example.java
File: NumberManipulation.java

REBUILDING MODIFIED FILES:

File: Coordinate.java
ENG: New AST Built. 8848 Bytes
File: Example.java
ENG: New AST Built. 15506 Bytes
File: NumberManipulation.java
ENG: New AST Built. 8262 Bytes

CREATING METHOD ANALYSIS ORDER...
3 Compilation Units to Inspect
SEARCHING FOR METHODS...
Method Found: calcDistanceToOrigin
Method Found: manipulateFields
Method Found: Example
Method Found: Coordinate
Method Found: factorCoords
Method Found: Coordinate
Method Found: getName
Method Found: main
Method Found: getY
Method Found: getI
Method Found: setI

DEP: void edu.iastate.kcs.NumberManipulation.manipulateFields() Marked for Analysis
DEP: java.lang.String edu.iastate.kcs.Coordinate.factorCoords(double) Marked for Analysis
DEP: double edu.iastate.kcs.Example.calcDistanceToOrigin(edu.iastate.kcs.Coordinate) Marked for Analysis
DEP: java.lang.String edu.iastate.kcs.Coordinate.getName() Marked for Analysis
DEP: void edu.iastate.kcs.Coordinate.Coordinate(java.lang.String, double, double) Marked for Analysis
DEP: void edu.iastate.kcs.NumberStorage.setI(int) Marked for Analysis
DEP: void edu.iastate.kcs.Coordinate.Coordinate(double, double) Marked for Analysis
DEP: void edu.iastate.kcs.Example.main(java.lang.String[]) Marked for Analysis
DEP: double edu.iastate.kcs.Example.calcDistanceBetweenCoords(edu.iastate.kcs.Coordinate, edu.iastate.kcs.Coordinate) Marked for Analysis
DEP: void int edu.iastate.kcs.NumberStorage.getI() Marked for Analysis
DEP: double edu.iastate.kcs.Example.calcDistanceBetweenCoords(edu.iastate.kcs.Coordinate, edu.iastate.kcs.Coordinate) Marked for Analysis
DEP: void edu.iastate.kcs.Example.Example() Marked for Analysis
DEP: double edu.iastate.kcs.Example.calcDistanceBetweenCoords(edu.iastate.kcs.Coordinate, edu.iastate.kcs.Coordinate) Marked for Analysis

ENG: ANALYSIS ORDER:

manipulateFields
factorCoords
calcDistanceToOrigin
getName
Coordinate
setI
Coordinate
main
getY
calcDistanceBetweenCoords
getI
Example

-----> DFA: Analyzing MethodDeclaration manipulateFields
-----> DFA: Clearing Analysis Information
-----> DFA: --> Initializing Class Field Set
VAL: Creating ValueElement: a=0:NumberManipulation.java:73
VAL: Creating ValueElement: b=1:NumberManipulation.java:77
VAL: Creating ValueElement: c=2:NumberManipulation.java:81
VAL: Creating ValueElement: x=3:NumberManipulation.java:85
VAL: Creating ValueElement: y=4:NumberManipulation.java:89
VAL: Creating ValueElement: z=5:NumberManipulation.java:93
VAL: Creating ValueElement: storage=new NumberStorage():NumberManipulation.java:114
VAL: Creating ValueElement: s=new String("Hello, world!"):NumberManipulation.java:154
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: a:NumberManipulation.java:73
VAL: Creating ValueElement: 0
DEF: Creating Definition: a:NumberManipulation.java:73
DEF: Adding Read Variable: a
VAL: Creating ValueElement: b:NumberManipulation.java:77
VAL: Creating ValueElement: 1
DEF: Creating Definition: b:NumberManipulation.java:77
DEF: Adding Read Variable: b
VAL: Creating ValueElement: c:NumberManipulation.java:81
VAL: Creating ValueElement: 2
DEF: Creating Definition: c:NumberManipulation.java:81
DEF: Adding Read Variable: c
VAL: Creating ValueElement: x:NumberManipulation.java:85
VAL: Creating ValueElement: 3
DEF: Creating Definition: x:NumberManipulation.java:85
DEF: Adding Read Variable: x
VAL: Creating ValueElement: y:NumberManipulation.java:89
VAL: Creating ValueElement: 4
DEF: Creating Definition: y:NumberManipulation.java:89
DEF: Adding Read Variable: y
VAL: Creating ValueElement: z:NumberManipulation.java:93
VAL: Creating ValueElement: 5
DEF: Creating Definition: z:NumberManipulation.java:93
DEF: Adding Read Variable: z
VAL: Creating ValueElement: storage:NumberManipulation.java:114
-----> DFA: Analyzing ClassInstanceCreation: new NumberStorage()
VAL: Creating ValueElement: storage:NumberManipulation.java:114
DEF: Creating Definition: storage:NumberManipulation.java:114
DEF: Adding Read Variable: storage
VAL: Creating ValueElement: s:NumberManipulation.java:154
-----> DFA: --> Creating Dummy Definitions For Parameters
VAL: Creating ValueElement: s2:NumberManipulation.java:234
-----> DFA: Analyzing ClassInstanceCreation: new String("Hello, world!")
VAL: Creating ValueElement: s2:NumberManipulation.java:234
DEF: Creating Definition: s2:NumberManipulation.java:234
DEF: Adding Read Variable: s
DEF: Chaining: s2:234 -> s:154
-----> DFA: Analyzing Assignment: b;s2.hashCode()
VAL: Creating ValueElement: b:NumberManipulation.java:257
-----> DFA: Analyzing MethodInvocation: s2.hashCode()
VAL: Creating ValueElement: b:NumberManipulation.java:257
DEF: Creating Definition: b:NumberManipulation.java:257
DEF: Adding Read Variable: s2
DEF: Chaining: b:257 -> s2:234
-----> DFA: Analyzing MethodInvocation: storage.setI(a)
VAL: Creating ValueElement: storage:NumberManipulation.java:279
VAL: Creating ValueElement: a:NumberManipulation.java:292
-----> DFA: Analyzing Assignment: x=storage.getI()
VAL: Creating ValueElement: storage:NumberManipulation.java:279
DEF: Creating Definition: x:NumberManipulation.java:299
DEF: Adding Read Variable: storage
DEF: Chaining: x:299 -> storage:114
-----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of void edu.iastate.kcs.NumberManipulation.manipulateFields():
MAP: Input: [edu.iastate.kcs.NumberManipulation.storage] Impacts: 
[edu.iastate.kcs.NumberManipulation.x]
MAP: Input: [edu.iastate.kcs.NumberManipulation.s] Impacts: 
[edu.iastate.kcs.NumberManipulation.b]

-----> DFA: Analyzing MethodDeclaration factorCoords
-----> DFA: --> Initializing Class Field Set
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: y:Coordinate.java:100
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: name:Coordinate.java:74
DEF: Creating Definition: name:Coordinate.java:74
DEF: Adding Read Variable: name
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: x:Coordinate.java:97
DEF: Creating Definition: x:Coordinate.java:97
DEF: Adding Read Variable: x
VAL: Creating ValueElement: y:Coordinate.java:100
VAL: Creating ValueElement: y:Coordinate.java:100
DEF: Creating Definition: y:Coordinate.java:100
DEF: Adding Read Variable: y
Creating Dummy Definitions For Parameters

Creating ValueElement: double factor:Coordinate.java:492

Creating Definition: factor:Coordinate.java:492

Adding Read Variable: factor

Analyzing Assignment: x=x*factor

Creating ValueElement: x:Coordinate.java:514

Analyzing InfixExpression: x*factor

Creating ValueElement: x:Coordinate.java:518

Creating ValueElement: factor:Coordinate.java:522

Creating Definition: x:Coordinate.java:514

Adding Read Variable: x

Chaining: x:514 -> factor:492

Chaining: x:514 -> x:97

Analyzing Assignment: y=y*factor

Creating ValueElement: y:Coordinate.java:536

Analyzing InfixExpression: y*factor

Creating ValueElement: y:Coordinate.java:540

Creating ValueElement: factor:Coordinate.java:544

Creating Definition: y:Coordinate.java:536

Adding Read Variable: y

Adding Read Variable: factor

Chaining: y:536 -> y:100

Analyzing ReturnStatement: return name;

Creating ValueElement: name:Coordinate.java:565

Returning Read On Variable: name

Mapping Outputs To Inputs

Current Mapping of java.lang.String edu.iastate.kcs.Coordinate.factorCoords(double):

Input: [factor] Impacts: [edu.iastate.kcs.Coordinate.y]

Input: [factor] Impacts: [edu.iastate.kcs.Coordinate.x]


Analyzing MethodDeclaration calcDistanceToOrigin

Clearing Analysis Information

Initializing Class Field Set

Creating Dummy Definitions For Class Fields

Creating Definition: origin:Example.java:67

Creating ValueElement: origin:Example.java:67

Creating ValueElement: origin:Example.java:67

Creating Definition: origin:Example.java:67

Adding Read Variable: origin

Creating Dummy Definitions For Parameters

Creating ValueElement: Coordinate coord:Example.java:598

Creating Definition: coord:Example.java:598

Adding Read Variable: coord

Analyzing ReturnStatement: return calcDistanceBetweenCoords(origin,coord);

Analyzing MethodInvocation: calcDistanceBetweenCoords(origin,coord)

Creating ValueElement: example:java:654

Creating ValueElement: coord:Example.java:662

Returning Read On Variable: origin

Returning Read On Variable: coord

Mapping Outputs To Inputs

Current Mapping of double edu.iastate.kcs.Example.calcDistanceToOrigin(edu.iastate.kcs.Coordinate): 

Input: [coord] Impacts: [Return Value of double edu.iastate.kcs.Example.calcDistanceToOrigin(edu.iastate.kcs.Coordinate)]

Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: y:Coordinate.java:100
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: name:Coordinate.java:74
DEF: Creating Definition: name:Coordinate.java:74
DEF: Adding Read Variable: name
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: x:Coordinate.java:97
DEF: Creating Definition: x:Coordinate.java:97
DEF: Adding Read Variable: x
VAL: Creating ValueElement: y:Coordinate.java:100
VAL: Creating ValueElement: y:Coordinate.java:100
DEF: Creating Definition: y:Coordinate.java:100
DEF: Adding Read Variable: y
-----> DFA: --> Creating Dummy Definitions For Parameters
-----> DFA: Analyzing ReturnStatement: return name;
VAL: Creating ValueElement: name:Coordinate.java:448
-----> DFA: Returning Read On Variable: name
-----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of java.lang.String edu.iastate.kcs.Coordinate.getName():
MAP: Input: [edu.iastate.kcs.Coordinate.name] Impacts: [ReturnValue of java.lang.String
edu.iastate.kcs.Coordinate.getName()]

        ----> DFA: Analyzing MethodDeclaration Coordinate
        ----> DFA: Clearing Analysis Information
        ----> DFA: --> Initializing Class Field Set
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: y:Coordinate.java:100
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: y:Coordinate.java:100
DEF: Creating Definition: name:Coordinate.java:74
DEF: Adding Read Variable: name
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: x:Coordinate.java:97
DEF: Creating Definition: x:Coordinate.java:97
DEF: Adding Read Variable: x
VAL: Creating ValueElement: y:Coordinate.java:100
VAL: Creating ValueElement: y:Coordinate.java:100
DEF: Creating Definition: y:Coordinate.java:100
DEF: Adding Read Variable: y
-----> DFA: --> Creating Dummy Definitions For Parameters
VAL: Creating ValueElement: String nameArg:Coordinate.java:230
DEF: Creating Definition: nameArg:Coordinate.java:230
DEF: Adding Read Variable: nameArg
VAL: Creating ValueElement: double xArg:Coordinate.java:246
DEF: Creating Definition: xArg:Coordinate.java:246
DEF: Adding Read Variable: xArg
VAL: Creating ValueElement: double yArg:Coordinate.java:259
DEF: Creating Definition: yArg:Coordinate.java:259
DEF: Adding Read Variable: yArg
-----> DFA: Analyzing Assignment: name=nameArg
VAL: Creating ValueElement: name:Coordinate.java:276
VAL: Creating ValueElement: nameArg:Coordinate.java:283
DEF: Creating Definition: name:Coordinate.java:276
DEF: Adding Read Variable: nameArg
DEF: Chaining: name:276 --> nameArg:230
-----> DFA: Analyzing Assignment: x=xArg
VAL: Creating ValueElement: x:Coordinate.java:295
VAL: Creating ValueElement: xArg:Coordinate.java:299
DEF: Creating Definition: x:Coordinate.java:295
DEF: Adding Read Variable: xArg
DEF: Chaining: x:295 --> xArg:246
-----> DFA: Analyzing Assignment: y=yArg
VAL: Creating ValueElement: y:Coordinate.java:308
VAL: Creating ValueElement: yArg:Coordinate.java:312
DEF: Creating Definition: y:Coordinate.java:308
DEF: Adding Read Variable: yArg
DEF: Chaining: y:308 \rightarrow yArg:259
-----> DFA: Mapping Outputs To Inputs

MAP: Input: [yArg] Impacts: [Return Value of void edu.iastate.kcs.Coordinate.Coordinate(java.lang.String, double, double)]
MAP: Input: [yArg] Impacts: [edu.iastate.kcs.Coordinate.y]
MAP: Input: [nameArg] Impacts: [Return Value of void edu.iastate.kcs.Coordinate.Coordinate(java.lang.String, double, double)]
MAP: Input: [nameArg] Impacts: [edu.iastate.kcs.Coordinate.name]
MAP: Input: [xArg] Impacts: [Return Value of void edu.iastate.kcs.Coordinate.Coordinate(java.lang.String, double, double)]
MAP: Input: [xArg] Impacts: [edu.iastate.kcs.Coordinate.x]

-----> DFA: Analyzing MethodDeclaration setI
-----> DFA: Clearing Analysis Information
-----> DFA: --> Initializing Class Field Set
VAL: Creating ValueElement: i:0:NumberManipulation.java:366
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: i:NumberManipulation.java:366
VAL: Creating ValueElement: 0
DEF: Creating Definition: i:NumberManipulation.java:366
DEF: Adding Read Variable: i
-----> DFA: --> Creating Dummy Definitions For Parameters
VAL: Creating ValueElement: int value:NumberManipulation.java:393
DEF: Creating Definition: value:NumberManipulation.java:393
DEF: Adding Read Variable: value
-----> DFA: Analyzing Assignment: i=value
VAL: Creating ValueElement: i:NumberManipulation.java:408
VAL: Creating ValueElement: value:NumberManipulation.java:412
DEF: Creating Definition: i:NumberManipulation.java:408
DEF: Adding Read Variable: value
DEF: Chaining: i:408 \rightarrow value:393
-----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of void edu.iastate.kcs.NumberStorage.setI(int):
MAP: Input: [value] Impacts: [edu.iastate.kcs.NumberStorage.i]

-----> DFA: Analyzing MethodDeclaration Coordinate
-----> DFA: Clearing Analysis Information
-----> DFA: --> Initializing Class Field Set
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: y:Coordinate.java:100
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: name:Coordinate.java:74
DEF: Creating Definition: name:Coordinate.java:74
DEF: Adding Read Variable: name
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: x:Coordinate.java:97
DEF: Creating Definition: x:Coordinate.java:97
DEF: Adding Read Variable: x
VAL: Creating ValueElement: y:Coordinate.java:100
VAL: Creating ValueElement: y:Coordinate.java:100
DEF: Creating Definition: y:Coordinate.java:100
DEF: Adding Read Variable: y
-----> DFA: --> Creating Dummy Definitions For Parameters
VAL: Creating ValueElement: double xArg:Coordinate.java:126
DEF: Creating Definition: xArg:Coordinate.java:126
DEF: Adding Read Variable: xArg
VAL: Creating ValueElement: double yArg:Coordinate.java:139
DEF: Creating Definition: yArg:Coordinate.java:139
DEF: Adding Read Variable: yArg
DEF: Adding Read Variable: c1
VAL: Creating ValueElement: Coordinate c2:Example.java:733
DEF: Creating Definition: c2:Example.java:733
DEF: Adding Read Variable: c2
VAL: Creating ValueElement: cl:Example.java:760
-----> DFA: Analyzing MethodInvocation: c1.getX()
VAL: Creating ValueElement: cl:Example.java:766
DEF: Creating Definition: cl:Example.java:760
DEF: Adding Read Variable: c1
DEF: Adding Read Variable: x
DEF: Chaining: cl:760 -> c1:718
DEF: Chaining: cl:760 -> x:97
VAL: Creating ValueElement: c2x:Example.java:788
-----> DFA: Analyzing MethodInvocation: c2.getX()
VAL: Creating ValueElement: c2:Example.java:794
DEF: Creating Definition: c2x:Example.java:788
DEF: Adding Read Variable: c2
DEF: Adding Read Variable: x
DEF: Chaining: c2x:788 -> c2:733
DEF: Chaining: c2x:788 -> x:97
VAL: Creating ValueElement: cly:Example.java:816
-----> DFA: Analyzing MethodInvocation: c1.getY()
VAL: Creating ValueElement: c1:Example.java:822
DEF: Creating Definition: cly:Example.java:816
DEF: Adding Read Variable: c1
DEF: Adding Read Variable: c2
DEF: Chaining: c2y:844 -> c2:733
VAL: Creating ValueElement: xDelta:Example.java:872
-----> DFA: Analyzing MethodInvocation: Math.abs(c1x-c2x)
-----> DFA: Analyzing InfixExpression: clx-c2x
VAL: Creating ValueElement: clx:Example.java:890
VAL: Creating ValueElement: c2x:Example.java:896
DEF: Creating Definition: xDelta:Example.java:872
DEF: Adding Read Variable: c2x
DEF: Adding Read Variable: clx
DEF: Chaining: xDelta:872 -> c2x:788
DEF: Chaining: xDelta:872 -> clx:760
VAL: Creating ValueElement: yDelta:Example.java:913
-----> DFA: Analyzing MethodInvocation: Math.abs(c1y-c2y)
-----> DFA: Analyzing InfixExpression: cly-c2y
VAL: Creating ValueElement: cly:Example.java:931
VAL: Creating ValueElement: c2y:Example.java:937
DEF: Creating Definition: yDelta:Example.java:913
DEF: Adding Read Variable: cly
DEF: Adding Read Variable: c2y
DEF: Chaining: yDelta:913 -> cly:816
DEF: Chaining: yDelta:913 -> c2y:844
-----> DFA: Analyzing ReturnStatement: return
Math.sqrt(Math.pow(xDelta,2)+Math.pow(yDelta,2));
-----> DFA: Analyzing MethodInvocation: Math.sqrt(Math.pow(xDelta,2)+Math.pow(yDelta,2))
-----> DFA: Analyzing InfixExpression: Math.pow(xDelta,2)+Math.pow(yDelta,2)
-----> DFA: Analyzing MethodInvocation: Math.pow(xDelta,2)
VAL: Creating ValueElement: xDelta:Example.java:975
VAL: Creating ValueElement: 2
-----> DFA: Analyzing MethodInvocation: Math.pow(yDelta,2)
VAL: Creating ValueElement: yDelta:Example.java:997
VAL: Creating ValueElement: 2
-----> DFA: Returning Read On Variable: yDelta
-----> DFA: Returning Read On Variable: xDelta
-----> DFA: Mapping Outputs To Inputs

MAP: Input: [Return Value of double edu.iastate.kcs.Coordinate.getX()] Impacts: [Return Value of double edu.iastate.kcs.Example.calcDistanceBetweenCoords(edu.iastate.kcs.Coordinate, edu.iastate.kcs.Coordinate)]

MAP: Input: [c2] Impacts: [Return Value of double edu.iastate.kcs.Example.calcDistanceBetweenCoords(edu.iastate.kcs.Coordinate, edu.iastate.kcs.Coordinate)]

MAP: Input: [c1] Impacts: [Return Value of double edu.iastate.kcs.Example.calcDistanceBetweenCoords(edu.iastate.kcs.Coordinate, edu.iastate.kcs.Coordinate)]

----> DFA: Analyzing MethodDeclaration getY
----> DFA: Clearing Analysis Information

----> DFA: ---> Initializing Class Field Set
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: y:Coordinate.java:100

----> DFA: ---> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: name:Coordinate.java:74
VAL: Creating ValueElement: name:Coordinate.java:74
DEF: Creating Definition: name:Coordinate.java:74
DEF: Adding Read Variable: name
VAL: Creating ValueElement: x:Coordinate.java:97
VAL: Creating ValueElement: x:Coordinate.java:97
DEF: Creating Definition: x:Coordinate.java:97
DEF: Adding Read Variable: x
VAL: Creating ValueElement: y:Coordinate.java:100
VAL: Creating ValueElement: y:Coordinate.java:100
DEF: Creating Definition: y:Coordinate.java:100
DEF: Adding Read Variable: y

----> DFA: ---> Creating Dummy Definitions For Parameters
----> DFA: Analyzing ReturnStatement: return y;
VAL: Creating ValueElement: y:Coordinate.java:400

----> DFA: Returning Read On Variable: y

----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of double edu.iastate.kcs.Coordinate.getY():

----> DFA: Analyzing MethodDeclaration getI
----> DFA: Clearing Analysis Information

----> DFA: ---> Initializing Class Field Set
VAL: Creating ValueElement: i=0:NumberManipulation.java:366

----> DFA: ---> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: i:NumberManipulation.java:366
VAL: Creating ValueElement: 0
DEF: Creating Definition: i:NumberManipulation.java:366
DEF: Adding Read Variable: i

----> DFA: ---> Creating Dummy Definitions For Parameters
----> DFA: Analyzing ReturnStatement: return i;
VAL: Creating ValueElement: i:NumberManipulation.java:455

----> DFA: Returning Read On Variable: i

----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of int edu.iastate.kcs.NumberStorage.getI():
MAP: Input: [edu.iastate.kcs.NumberStorage.i] Impacts: [Return Value of int edu.iastate.kcs.NumberStorage.getI()]

----> DFA: Analyzing MethodDeclaration Example
----> DFA: Clearing Analysis Information

----> DFA: ---> Initializing Class Field Set
VAL: Creating ValueElement: origin:Example.java:67

----> DFA: ---> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: origin:Example.java:67
VAL: Creating ValueElement: origin:Example.java:67

MAP: Current Mapping of double edu.iastate.kcs.Coordinate.getY():

----> DFA: Analyzing MethodDeclaration getI
----> DFA: Clearing Analysis Information

----> DFA: ---> Initializing Class Field Set
VAL: Creating ValueElement: i=0:NumberManipulation.java:366

----> DFA: ---> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: i:NumberManipulation.java:366
VAL: Creating ValueElement: 0
DEF: Creating Definition: i:NumberManipulation.java:366
DEF: Adding Read Variable: i

----> DFA: ---> Creating Dummy Definitions For Parameters
----> DFA: Analyzing ReturnStatement: return i;
VAL: Creating ValueElement: i:NumberManipulation.java:455

----> DFA: Returning Read On Variable: i

----> DFA: Mapping Outputs To Inputs
DEF: Creating Definition: origin:Example.java:67
DEF: Adding Read Variable: origin
-----> DFA: --> Creating Dummy Definitions For Parameters
-----> DFA: Analyzing Assignment: origin=new Coordinate("origin",0,0)
VAL: Creating ValueElement: origin:Example.java:513
-----> DFA: Analyzing ClassInstanceCreation: new Coordinate("origin",0,0)
VAL: Creating ValueElement: 0
VAL: Creating ValueElement: 0
DEF: Creating Definition: y:Coordinate.java:100
DEF: Adding Read Variable: y
DEF: Creating Definition: name:Coordinate.java:74
DEF: Creating Definition: x:Coordinate.java:97
DEF: Adding Read Variable: x
DEF: Creating Definition: origin:Example.java:513
-----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of void edu.iastate.kcs.Example.Example():

REBUILDING FILES FOR LOCAL ANALYSIS:
File: NumberManipulation.java
ENG: New AST Built. 8262 Bytes
CREATING METHOD ANALYSIS ORDER...
1 Compilation Units to Inspect
SEARCHING FOR METHODS...
Method Found: getI
Method Found: setI
Method Found: manipulateFields
DEF: void edu.iastate.kcs.NumberStorage.setI(int)Marked for Analysis
DEF: void edu.iastate.kcs.NumberManipulation.manipulateFields()Marked for Analysis
DEF: int edu.iastate.kcs.NumberStorage.getI()Marked for Analysis

ENG: ANALYSIS ORDER:
setI
manipulateFields
getI

-----> DFA: Analyzing MethodDeclaration setI
-----> DFA: Clearing Analysis Information
-----> DFA: --> Initializing Class Field Set
VAL: Creating ValueElement: i=0:NumberManipulation.java:366
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: i:0:NumberManipulation.java:366
VAL: Creating ValueElement: 0
DEF: Creating Definition: i:NumberManipulation.java:366
DEF: Adding Read Variable: i
-----> DFA: --> Creating Dummy Definitions For Parameters
VAL: Creating ValueElement: int value:NumberManipulation.java:393
DEF: Creating Definition: value:NumberManipulation.java:393
DEF: Adding Read Variable: value
-----> DFA: Analyzing Assignment: i=value
VAL: Creating ValueElement: i:NumberManipulation.java:408
VAL: Creating ValueElement: value:NumberManipulation.java:412
DEF: Creating Definition: i:NumberManipulation.java:408
DEF: Adding Read Variable: value
DEF: Chaining: i:408 -> value:393
-----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of void edu.iastate.kcs.NumberStorage.setI(int):
MAP: Input: [value] Impacts: [edu.iastate.kcs.NumberStorage.i]

-----> DFA: Analyzing MethodDeclaration manipulateFields
-----> DFA: Clearing Analysis Information

-----> DFA: --> Initializing Class Field Set

VAL: Creating ValueElement: a=0:NumberManipulation.java:73
VAL: Creating ValueElement: b=1:NumberManipulation.java:77
VAL: Creating ValueElement: c=2:NumberManipulation.java:81
VAL: Creating ValueElement: x=3:NumberManipulation.java:85
VAL: Creating ValueElement: y=4:NumberManipulation.java:89
VAL: Creating ValueElement: z=5:NumberManipulation.java:93
VAL: Creating ValueElement: storage=new NumberStorage():NumberManipulation.java:114
VAL: Creating ValueElement: s=new String("Hello, world!"):NumberManipulation.java:154

-----> DFA: --> Creating Dummy Definitions For Class Fields

VAL: Creating ValueElement: a:NumberManipulation.java:73
VAL: Creating ValueElement: b:NumberManipulation.java:77
VAL: Creating ValueElement: c:NumberManipulation.java:81
VAL: Creating ValueElement: x:NumberManipulation.java:85
VAL: Creating ValueElement: y:NumberManipulation.java:89
VAL: Creating ValueElement: z:NumberManipulation.java:93
VAL: Creating ValueElement: storage:NumberManipulation.java:114

-----> DFA: Analyzing ClassInstanceCreation: new NumberStorage()

VAL: Creating ValueElement: s:NumberManipulation.java:250

-----> DFA: Creating Dummy Definitions For Parameters

VAL: Adding Read Variable: a
VAL: Adding Read Variable: b
VAL: Adding Read Variable: c
VAL: Adding Read Variable: x
VAL: Adding Read Variable: y
VAL: Adding Read Variable: z

-----> DFA: Analyzing ClassInstanceCreation: new String(s)

VAL: Creating ValueElement: s:NumberManipulation.java:154

-----> DFA: Analyzing Assignment: b=s2.hashCode()

VAL: Creating ValueElement: b:NumberManipulation.java:257

-----> DFA: Analyzing MethodInvocation: s2.hashCode()

VAL: Creating ValueElement: s2:NumberManipulation.java:261

-----> DFA: Analyzing Assignment: x=storage.getI()

VAL: Creating ValueElement: x:NumberManipulation.java:299

-----> DFA: Analyzing MethodInvocation: storage.getI()

VAL: Creating ValueElement: x:NumberManipulation.java:303

-----> DFA: Analyzing Assignment: x=storage.getI()

VAL: Creating ValueElement: x:NumberManipulation.java:299

-----> DFA: Analyzing MethodInvocation: storage.getI()

VAL: Creating ValueElement: x:NumberManipulation.java:303

-----> DFA: Analyzing Assignment: x=storage.getI()

VAL: Creating ValueElement: x:NumberManipulation.java:299

-----> DFA: Analyzing MethodInvocation: storage.getI()

VAL: Creating ValueElement: x:NumberManipulation.java:303
DEF: Adding Read Variable: i
DEF: Chaining: x:299 -> storage:114
DEF: Chaining: x:299 -> i:366
-----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of void edu.iastate.kcs.NumberManipulation.manipulateFields():
MAP: Input: [edu.iastate.kcs.NumberManipulation.storage] Impacts:
[edu.iastate.kcs.NumberManipulation.x]
MAP: Input: [edu.iastate.kcs.NumberManipulation.s] Impacts:
[edu.iastate.kcs.NumberManipulation.b]
MAP: Input: [edu.iastate.kcs.NumberManipulation.a] Impacts:
[edu.iastate.kcs.NumberManipulation.x]
MAP: Input: [edu.iastate.kcs.NumberManipulation.a] Impacts:
[edu.iastate.kcs.NumberManipulation.i]
MAP: Input: [Return Value of int edu.iastate.kcs.NumberStorage.getI()] Impacts:
[edu.iastate.kcs.NumberManipulation.x]

-----> DFA: Analyzing MethodDeclaration getI
-----> DFA: Clearing Analysis Information
-----> DFA: --> Initializing Class Field Set
VAL: Creating ValueElement: i=0:NumberManipulation.java:366
-----> DFA: --> Creating Dummy Definitions For Class Fields
VAL: Creating ValueElement: i:NumberManipulation.java:366
VAL: Creating ValueElement: 0
DEF: Creating Definition: i:NumberManipulation.java:366
DEF: Adding Read Variable: i
-----> DFA: --> Creating Dummy Definitions For Parameters
-----> DFA: Analyzing ReturnStatement: return i;
VAL: Creating ValueElement: i:NumberManipulation.java:455
-----> DFA: Returning Read On Variable: i
-----> DFA: Mapping Outputs To Inputs

MAP: Current Mapping of int edu.iastate.kcs.NumberStorage.getI():
MAP: Input: [edu.iastate.kcs.NumberStorage.i] Impacts: [Return Value of int edu.iastate.kcs.NumberStorage.getI()]

-----> DFA: MARKING DEFINITIONS & VALUE ELEMENTS...
-----> DFA: FILE: NumberManipulation.java RANGE: 147-186
-----> DFA: Marking ValueElement: edu.iastate.kcs.NumberManipulation.s:154

CALCULATING LOCAL IMPACT OF MARKS:
INITIAL IMPACT SET:
edu.iastate.kcs.NumberManipulation.b
edu.iastate.kcs.NumberManipulation.s

FINAL IMPACT SET:
edu.iastate.kcs.NumberManipulation.b
edu.iastate.kcs.NumberManipulation.s

FULL ANALYSIS TIME: 375
INCREMENTAL ANALYSIS TIME: 375
NUMBER OF FILES MODIFIED: 3
NUMBER OF METHODS ANALYZED: 13
7 References


