Finding Use Def Chains in a Java Program by doing its Dynamic DataFlow Analysis

By

Jyoti Arora
Advisor: Prof. Mark Grechanik

Project Report for fulfilment of
Master's Project requirement for the degree of
Masters of Science in Computer Science

University of Illinois Chicago
April 2016

ACKNOWLEDGEMENTS

First, I would like to thank my project advisor Prof. Mark Grechanik for his guidance, support and patience during the entire course of the project. This project was quite challenging and I learnt a lot of new concepts while working on it. I am really grateful to him for giving me this opportunity to work with him. I would like to make a special mention of my parents, Sh. Surinder Arora and Smt. Parveen Arora and my husband, Anoop for their encouragement and support throughout. Last but not the least, I am thankful to all my friends, old and new, for always being there for me.

Jyoti Arora

University of Illinois Chicago

April 2017

Table Of Contents

Abstract

In integration testing, integrated modules or components are evaluated as a whole to determine if they behave correctly. The larger the project, the more important integration testing is to find bugs efficiently in integrated components of AUT. Skipping integration testing increases the cost of software by approximately from five and up to twenty times if a defect, which is missed during integration testing is found at a later stage.

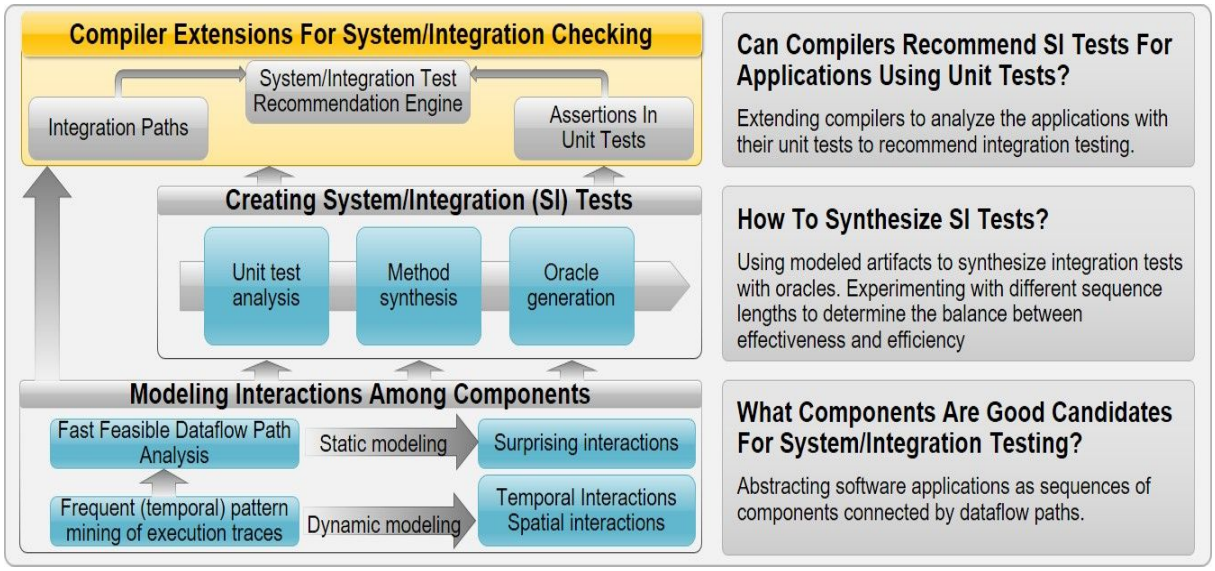
Making integration testing cheaper and more effective is very important and is equally difficult. ASSIST is a framework that aims at automatically generating these integration tests for a given application.

The first and main task of the ASSIST framework is to automatically generate integration tests is to find the interacting components in an application and to identify what components are good for integration testing. This project aims at finding the interacting components in an application by doing its dynamic dataflow analysis and the result of this project will be used to synthesize the integration tests.

Introduction

A main objective for software integration tests is to be effective in finding bugs. An equally important objective is to find bugs in a shorter time period without using computing resources i.e. software integration tests should also be efficient. Currently, there is a lack of understanding of all the factors that influence the effectiveness and efficiency of integration tests.

This master project is a part of the research map that is shown below:



[Courtesy of: “Proposal for ASSIST”, Mark Grechanik et. al]

Each layer of the map roughly corresponds to a summarizing research question on the right side of the map. **At the bottom of the map we position the task of obtaining models of interactions among components in a software application, statically and dynamically**. This is a part where this master project concentrates. With frequent pattern mining, we want to synthesize tests for components that exchange data often. Once we know frequently interacting component, we investigate the components that rarely exchange data as they are most likely to be ignored by the testers. The ultimate goal of ASSIST is to investigate a compiler extension which will produce recommendations for integration testing by analyzing the existing unit tests and the source code of the software application in their entirety.

This project is the bottom layer of the research map that aims at modelling interactions among components. We use already written integration tests and units tests to do the dynamic dataflow analysis of the application to find these interacting components.

Definition of Interacting Components

Interacting components are those which are bound by Use-Definition chains. Hence, our problem statement is to find all the ***Use-Definition chains*** [1]of the application. Use-Definition chains are explained below-

Definition of a Variable

When a variable, v is on the LHS of an assignment statement, such as a(i) then a(i) is the definition of v. Every variable v has atleast one definition by its declaration or initialization.

Use of a Variable

If variable, v, is on the RHS of statement a(i), there is a statement, a(j) with j < i and min(i-j), that it is a definition of v and it has a use at a(i)

For Example-
int i = 1, ----> Definition of a Variable
int j = i; -----> Use of a Variable i, and Definition of variable j

Java is an object oriented language. Class objects interact by modifying or accessing each other's field variables. We find Use-Definition chains among these field variables to find the interacting components. Thes Use-Definition chains that a given testcase covers are obtained from the dynamic analysis of the application by using ***Java Debug Interface*** and ***Eclipse AST Parser libraries.*** An overview of these two libraries is given in the next two sections. The last section explains the overall solution architecture.

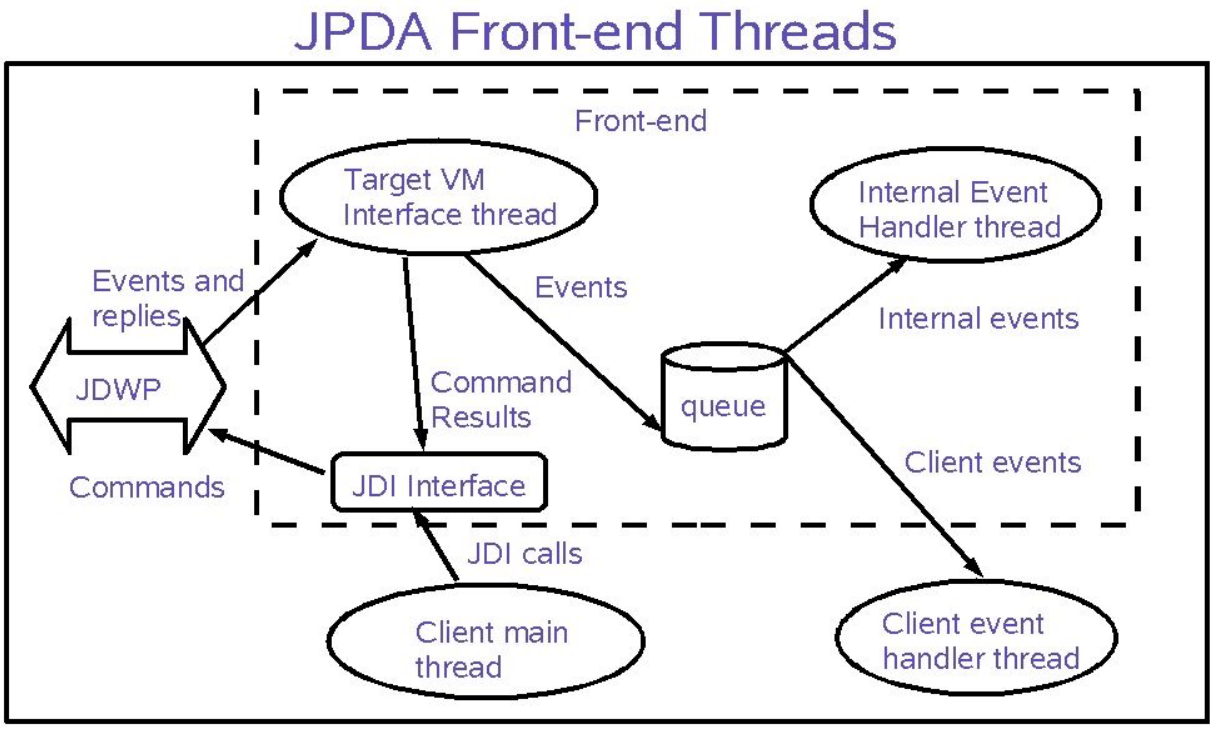
Java Debug Interface

This sections gives a brief overview of the Java Debug Interface:

The Java Debug Interface [2]is a high level Java API providing information providing access to a virtual machine's running state, class, array, fields, and primitive types, and instances of those types. It provides ability to suspend and to resume threads, and to set breakpoints and watchpoints and to set notifications for class loading, thread creation, exceptions. This interface allows developers to write remote debugger application tools.

---

[1] "Use-define chain - Wikipedia." https://en.wikipedia.org/wiki/Use-define_chain. Accessed 16 Apr. 2017.
[2] "Overview (Java Debug Interface) - Oracle Help Center." https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/. Accessed 16 Apr. 2017.



The above diagram explains Java Program Debug Architecture. The application that needs to be analyzed is loaded in the ***Target VM Interface thread*** and the ***client main thread*** monitors the loaded application by registering watchpoints using ***JDI interface.*** Whenever these watchpoints are triggered in the VM Interface thread, the events corresponding to these watchpoints are enqueued in the queue. ***Client event handler thread*** dequeues this queue and call handler function corresponding to each event. Internal Event Handler thread dequeues the queue to take care of internal events not registered by the client main thread.

JDI is the main interface that allows to interrupt the running application. ***EventRequestManager*** is a main class of JDI that manages creation and deletion of Event Requests. We can then define event callbacks corresponding to these registered events in the client event handler thread. The various requests that are used in this project are explained below-

1. *createClassPrepareRequest*

   This request is used to register an event for class loading. Whenever a new class is loaded in the VM, ClassPrepareEvent is triggered and the registered callback corresponding to this event is called.

2. *createModificationWatchpointRequest*

   Creates a new watchpoint which watches modification to a specified field. Whenever a

field variable is modified, ModificationWatchpointEvent is triggered and the registered callback corresponding to this event is called.

3. *createAccessWatchpointRequest*

   Creates a new watchpoint which watches accesses to a specified field. Whenever a field variable is accessed, AccessWatchpointEvent is triggered and the registered callback corresponding to this event is called.

4. *createMethodEntryRequest*

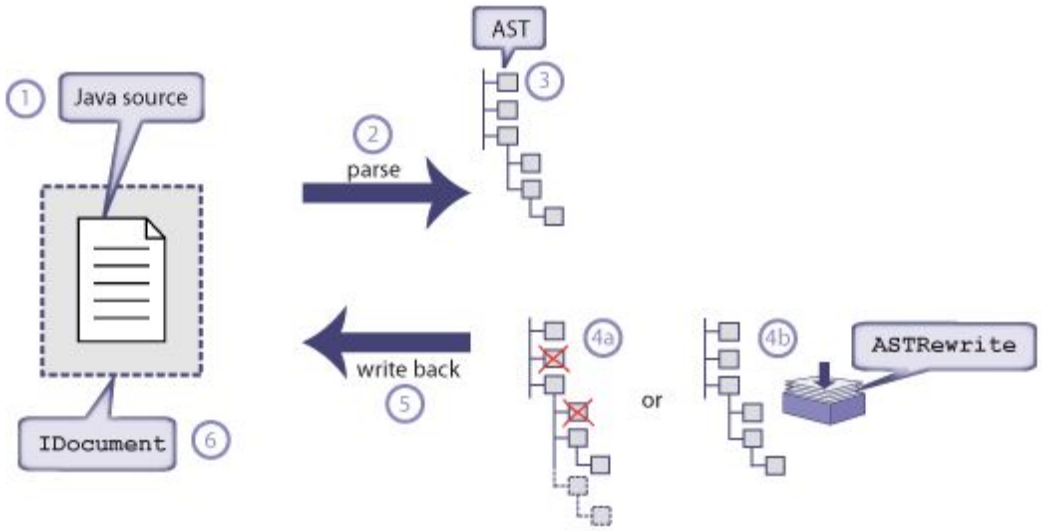
   This request is used to register a watchpoint for all the method entry points. Whenever a method is entered in a vm, MethodEntryEvent is triggered and the registered callback corresponding to this event is called.

5. *createMethodExitRequest*

   This request is used to register a watchpoint for all the method exit points. Whenever a method exits from a vm, MethodExitEvent is triggered and the registered callback corresponding to this event is called.

Eclipse AST Parser

It is a Java language parser for creating abstract syntax trees



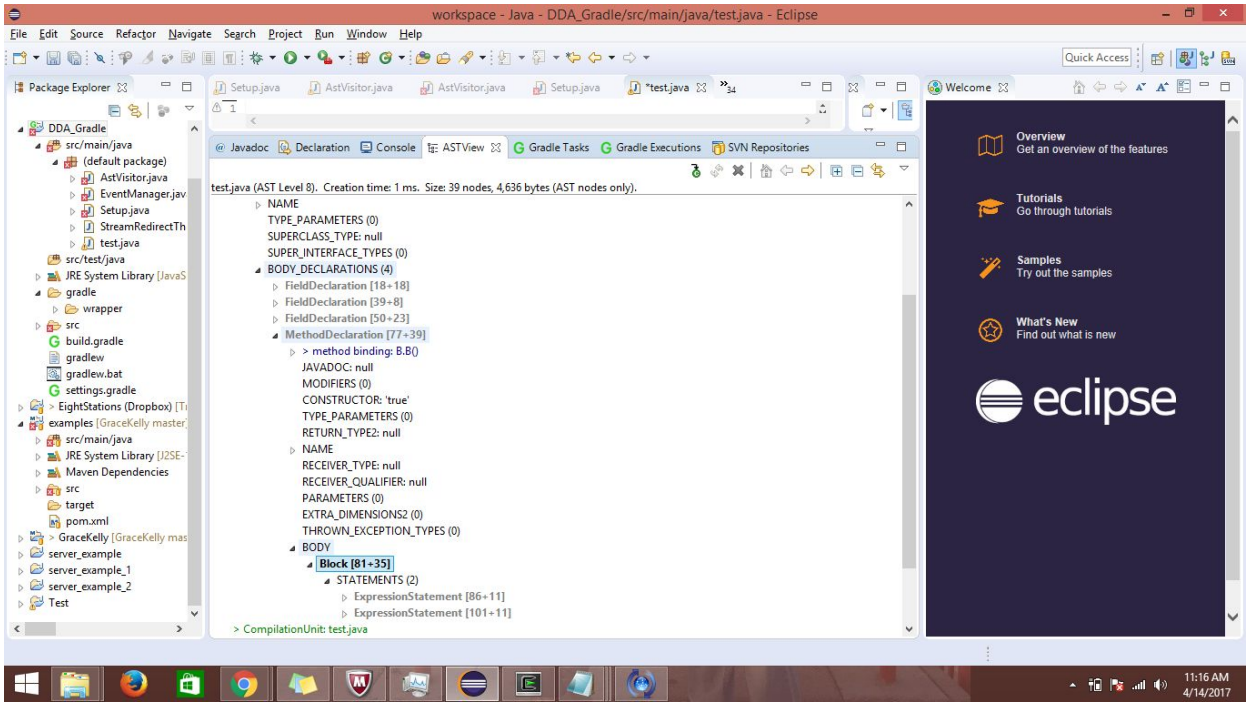
3

[3] "ASTParser (Eclipse JDT API Specification) - Eclipse Help." http://help.eclipse.org/mars/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html. Accessed 16 Apr. 2017.

The above diagram explains how to use Eclipse AST Parser. It reads the Java Source code and populate AST parse tree in a compilation unit. A visitor is used to visit all the nodes of the parse tree. We can modify these nodes while visiting them and record the modifications in a compilation unit. At the end, all the recorded modifications are written back to the text files using ASTRewrite.

```
class B {
        public
        int k =0;
        int l=7;
        int arr[] = new int[2];
        B() {
                arr[0] = 1;
                arr[1] = 2;
        }
}
```

For the code above, Eclipse AST Parser creates parse tree as shown below:



FieldDeclaration nodes are created for fields k, l & arr. Constructor B() is represented by MethodDeclaration node.and the body of the constructor has two expression statements corresponding to “arr[0] = 1” & “arr[1] = 2” respectively.

Project Implementation Details

This section covers the project requirements, the solution architecture and the implementation design that we came up with and how we handled the corner scenarios.

Requirement

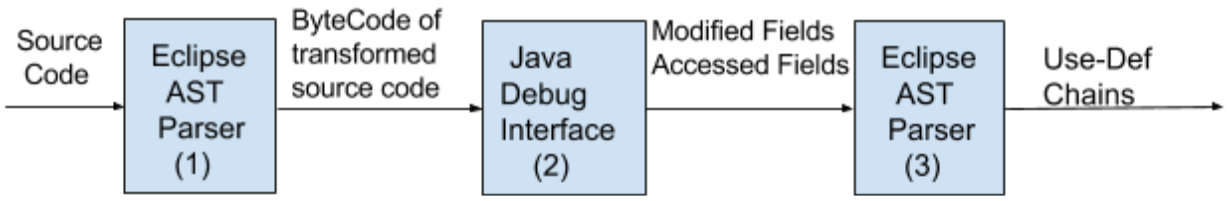
Given a testcase, find all the Use-Definition chains that a testcase covers in a given Java application

```
 Class B {
      int m;
      B(int m_){ m=m_;}
 }

Class A {
   int b;
   void func(){
         B b_ = new B();
         b = b_.m;
   }
}
```

In the above application, A and B are the interacting components as A::b is modified by B::m. We define such relations as use-def relations. We need to populate all such relationships that a given testcase covers.

Solution Architecture



Detailed Design Flow is explained below-

1. We compile the application & start a virtual machine that runs the application.

2. We use JDI to add modified watchpoint events and access watchpoint events for all the field variables in the application. JDI triggers an event whenever a field variable is modified or accessed but does not give one-to-one relation between the modified variable and the variables that modify it. **This is represented by block 2 of the**

architecture diagram.

```
class A {

        int j;
        void func() {
            B b = new B;// B is another class with field variable m
            int l = b.m;
            j = l;
        }
```

For the code above, we add watchpoints on field j of class A and field m of class B. JDI gives us the output that A::j is modified and B::m is accessed.

3. Some field variables interact with each other by exchanging data through some local variables. For example-

```
class A {

        int j;
        void func() {
            B b = new B;// B is another class with field variable m
            int l = b.m;
            j = l;
        }
```

To obtain relationship between A::j and B::m, we need to have relationship between A:: j and l & l and B::m. **JDI does not allow us to add watchpoints for local variables** so we can never obtain these relations.

We therefore use the eclipse AST parser to modify the input source code where we create a field variable corresponding to every local variable and also replace the references of the local variable with the newly created field variable in the entire source code. So, the transformed code for the above example looks like-

```
class A {
        int j;
        int l_0; // Field variable introduced for local variable l
        void func() {
            B b = new B;// B is another class with field variable m
            int l = b.m;
            l_0 = b.m; //l_0 initialized with initializer of b.m
            j = l_0;// l is replaced by l_0
```

}

The above transformation allows us to use JDI for local variables as now we can add watchpoint for l_0. ModificationWatchpoint gives us the information that l_0 is modified by B::m and AccessWatchpoint gives us the information that j is modified by l_0. Combining these two relations we obtain the relation between A::j and B::m. **This transformation needs to be done before block 2 executes, hence we add block 1 for it in the architecture diagram.**

4. After block1 and block2, we obtain a complete list of modified field variables and a complete list of accessed field variables. We still do not have one-to-one mapping between these two i.e. we do not have the relation between the modified variable and the variables that modify it.

To obtain this relation, **we add one more step to block 2 and introduce a new block 3 to the architecture diagram**.
Consider the two assignments below where j , m , l , a, b are the field variables of some class:-

```
j = m;
l = a+b;
```

JDI accesses the RHS of an assignment expression before accessing its LHS. JDI events log for the above two assignments is -

1. AccessWatchpointEvent for m is triggered
2. ModificationWatchpointEvent for j is triggered
3. AccessWatchpointEvent for a is triggered
4. AccessWatchpointEvent for b is triggered
5. ModificationWatchpointEvent for l is triggered

From the above event log we can observe that ***between every two modification watchpoints, a list of accessed field variables belong to the later modified field***. We use this heuristic to obtain relationship between modified field and fields that modify it. For the example above, a and b are accessed between j and l and modifies l. This heuristic fails in two scenarios-

Scenario 1

```
j = m
l = a+ func(n) + b;
where func is defined as,
int func(int a){ k = l;}
```

If we have a function on the RHS of an expression, VM will execute function func before in order to execute assignment “l = a+func(n)+b”. The JDI event Log for above scenario is:

1. AccessWatchpointEvent for m is triggered
2. ModificationWatchpointEvent for j is triggered
3. AccessWatchpointEvent for b is triggered
4. AccessWatchpointEvent for n is triggered
5. AccessWatchpointEvent for l is triggered
6. ModificationWatchpointEvent for k is triggered
7. AccessWatchpointEvent for b is triggered
8. ModificationWatchpointEvent for l is triggered

Hence, our above heuristic fails. We modify our heuristic to take care of such scenarios by bringing scope of an assignment into picture. We introduce two new Watchpoints, MethodEntryWatchpoint and MethodExitWatchpoint. ***We maintain a stack of accessed field variables. Whenever the MethodEntryWatchpoint is triggered we add a new set of accessed variables to the stack and whenever MethodExitWatchpoint is triggered, we pop the new set and get back to our previous set of accessed variables.*** For the example above, we now see two set of event logs in two different scopes-

1. AccessWatchpointEvent for m is triggered
2. ModificationWatchpointEvent for j is triggered
3. AccessWatchpointEvent for b is triggered
4. AccessWatchpointEvent for n is triggered
   New Scope func is entered:
   a. AccessWatchpointEvent for l is triggered
   b. ModificationWatchpointEvent for k is triggered
   Scope func is exiting
5. AccessWatchpointEvent for b is triggered
6. ModificationWatchpointEvent for l is triggered

This way we obtain three use-def relationships, j is modified by m, k is modified by func::l and l is modified by a, n and b.

<u>Scenario2</u>

```
l = m;
int i = k;
j = u;
```

where, l,m, k, j , u are the field variables but i is the local variable.
JDI does not allow to add watchpoints for local variables, modified watchpoint event will never be triggered for i but k is a field variable so access watchpoint event gets triggered

for it. Hence, between two modified variables l and j, we have two accessed field variables k and u. We end up generating use-def relation that j is modified by k & u which is not correct. ***To prune k from this relation, we use Eclipse AST Parser and introduce a new block 3.*** Eclipse AST Parser does static analysis of the code and finds out j is modified by u. It takes the relation j modified by k & u from JDI and prune all the variables from accessed fields list which are not found during static analysis of the code and we end up in a correct relation that j is modified by u.

At the end of this phase, we have all the correct use-def relations in the given source code that are executed by a given testcase.

Transformation to handle local variables in case of multithreaded designs

Original source code-

```
class A implements Runnable {
        int j;
        void run() {
            B b = new B;// B is another class with field variable m
            int l = b.m;
            j = l;
        }
}
```

As we explained above, in order to obtain relations propagating through local variables, we added a field variable for every local variable. Field variable is initialized with the initializer of local variable and all the accesses to the local variable is changed into access to newly created field variable

class A implements Runnable {

```
        int j;
        int l_0;
        void run() {
            B b = new B;// B is another class with field variable m
            int l = b.m;
            l_0 = b.m;
            j = l_0;
    }
```

The above transformation will be buggy in case class A is a thread class. Transformation is not semantically equivalent to the original source code. We have created a single field variable l_0 for the local variable l and hence its value gets shared across all the instances of thread Class A which was not the earlier intention.

To handle the above scenario, we came up with the below generic transformation that allows each thread to have its own copy of the local variable created as the field variable.

```
class A implements Runnable {
    /* Mapping between the threadID and local variable value. This map will be
       generated for each local variable which we are converting to the field variable.
       Datatype of key will always be the integer  and the datatype of the value
       will always be the datatype of the local variable */
    ConcurrentHashMap<Integer, Integer> l_0 = new ConcurrentHashMap();
    int j =0;

    public void run(){
            B b = new B();
            int l = b.m;
            l_0.put((int)Thread.currentThread().getId(), b.m);
            j= l_0.get((int)Thread.currentThread().getId());

    }

 }
```

Each thread in a Java application has its unique id and we use this id dynamically to obtain the copy of the local variable corresponding to the running thread. ConcurrentHashMap allows us to modify & access the new field variable across all the thread in a safe manner.

Project Implementation Design

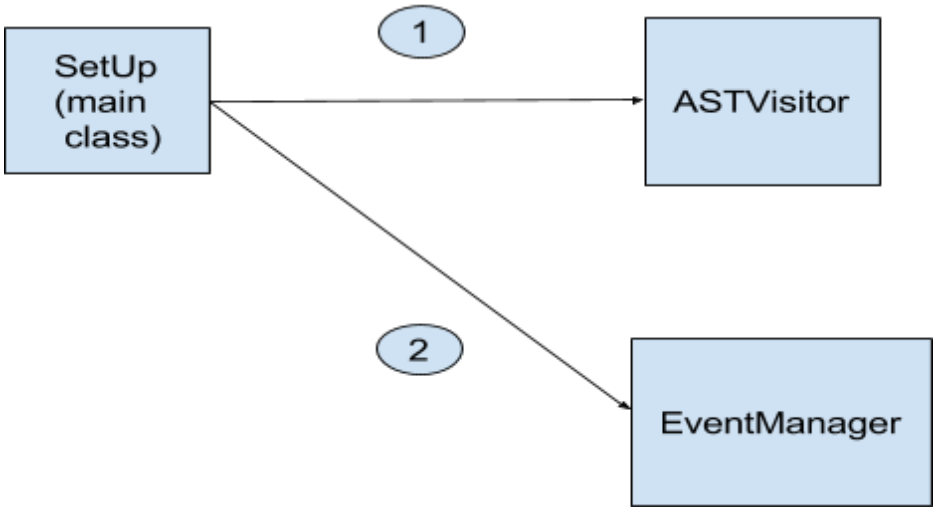
Project Implementation has been divided into three main modules-

1. Setup
   Main class that sets the input required by other modules. It also manages the order in which all the modules needs to be called.

2. EventManager
   This module is responsible to add all the watchPointRequests and also defines the handler when these requests are triggered. This module makes use of Java Debug Interface library.

3. ASTVisitor
   This module is responsible for doing necessary transformations required to handle local variables as explained in the previous section. This module makes use of the Eclipse AST Parser library.

The call diagram of the modules is as shown below. Numbers on the arrows indicate the order in which these modules are called.



Module 1 - Setup

Setup module instantiates an instance of AstVisitor and makes a call to its "***run***" api to do the transformation required to handle the local variables.

Then, it compiles the java application and starts a virtual machine that will run the generated bytecode. It then starts an instance of EventManager and passes the created virtual machine as an argument to it.

Module 2 - EventManager

EventManager is a class that manages registering watchpoint events to the virtual machine and also defining their corresponding handler. Below is a brief overview of various api's that are present in EventManager class.

1. *public void setEventRequests()*

   This api sets all the events on the request manager of the virtual machine that is running the bytecode. Various watchpoints that are enabled by this api are -

   a. createClassPrepareRequest
   b. createMethodEntryRequest
   c. createMethodExitRequest

2. *private void handleEvent(Event event)*

This api takes care to call the appropriate handler whenever an event is triggered.

3. *private void classPrepareEvent(ClassPrepareEvent event)*

   This is the handler for ClassPrepareRequest watchpoint added in setEventRequests. It adds *ModificationWatchpointRequest* and *AccessWatchpointRequest* for all the field variables of the class.

4. *private void fieldWatchEvent(AccessWatchpointEvent event)*

   Handler for AccessWatchpointRequest set by classPrepareEvent.

5. *private void fieldWatchEvent(ModificationWatchpointEvent event)*

   Handler for Modification WatchpointRequest set by classPrepareEvent.

6. *private void methodEntryEvent(MethodEntryEvent event)*

   Handler for MethodEntryWatchpointRequest set by setEventRequests.

7. *private void methodExitEvent(MethodExitEvent event)*

   Handler for MethodExitWatchpointRequest set by setEventRequests.

Module 3 - AstVisitor

This module defines the visitor for eclipse AST Parser. The transformation to the source code to handle local variables is done using this module. ASVisitor visits the parse tree and *record all the modifications in the compilation unit*. At the end, all the recorded modifications are written back to the document using *rewrite on the compilation unit*. Below is a brief overview of the various api's that are present in the ASTVisitor class.

1. *public void run()*

   This is the main method that does everything. It parses the source code and defines a visitor for the parse tree that was created. After all the modifications are recorded in compilation unit by the visitor, this method makes a call to rewrite on the compilation unit.

2. *private FieldDeclaration createNewField(AST ast, String name, int modifiers)*

   This method creates a new field variable for each local variable. It is called when the visitor visits the *VariableDeclarationStatement* node for the local variables. It creates a

ConcurrentHashmap corresponding to the ast node that is passed to this module.

3. *public void endVisit(VariableDeclarationStatement node)*

   For each VariableDeclarationStatement node that is visited for a local variable, it creates MethodInvocation node whose argument is an expression as mentioned below-

   int l = b.m; --------> *VariableDeclarationStatement*

   l_0.put((int)Thread.currentThread().getId(), b.m); -------> *l_0 is a new field variable that has been created corresponding to l. Insert a new MethodInvocation node with two arguments, first arguments is always "(int)Thread.currentThread().getId()" and second argument is "initializer of the VariableDeclarationStatement"*

4. *public boolean visit(SimpleName node)*

   Eclipse AST creates a SimpleName node for a variable name. If a SimpleName nod belongs to the local variable corresponding to which a new field variable has been created, all such nodes are replaced by a MethodInvocation node in this method as explained below-

   Old Expression:
   j = l , where l is some local variable

   New Expression:
   j= l_0.get((int)Thread.currentThread().getId()); -------> *l_0 is a new field variable that has been created corresponding to l. Insert a new MethodInvocation node with argument as "(int)Thread.currentThread().getId()"*

Conclusion

An application that finds use-def chains covered by a testcase for a given application is successfully developed. An attempt has been made to model interactions among components by using concepts of dynamic dataflow analysis in tandem with static analysis Successful tests were conducted with Java applications and there is a scope to use this module as a part of other software engineering problems.

Source Code

Setup.java

```
import com.sun.jdi.VirtualMachine;
import java.io.File;
```

```
import com.sun.jdi.Bootstrap;
import com.sun.jdi.connect.*;

import java.util.Map;
import java.io.IOException;
import java.util.List;

import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Setup {

                private static void runProcess(String command) throws Exception {
                    Process pro = Runtime.getRuntime().exec(command);
                    pro.waitFor();
                   // System.out.println(command + " exitValue() " + pro.exitValue());
                }
                // TODO take care of global variables


                public static void main(String[] args) throws IOException {
                AstVisitor visitor = new AstVisitor(null, true);
            visitor.run();
            String output_filepath = visitor.get_output_filepath();
            String compile_cmd = "javac ";
            compile_cmd += output_filepath;
            try {
                runProcess(compile_cmd);
            } catch (Exception e) {
                e.printStackTrace();
            }




        LaunchingConnector conn = null;
                // Connector is used to establish connection between a debugger
                // and a target VM
                List<Connector> connectors =
       Bootstrap.virtualMachineManager().allConnectors();
```

```
for (Connector connector : connectors) {
    // Find a launching connector as it launches a VM before connecting to it
    if (connector.name().equals("com.sun.jdi.CommandLineLaunch")) {
        conn = (LaunchingConnector)connector;
    }
}
if(conn == null)
    throw new Error("No launching connector");

Map<String, Connector.Argument> arguments = conn.defaultArguments();
// Name of the main function that needs to be loaded
Connector.Argument mainArg = (Connector.Argument)arguments.get("main");
mainArg.setValue("test");
// Add the classpath option to the launched vm as it does not where to
// search for the main class added in the mainArg
Connector.Argument options =(Connector.Argument)arguments.get("options");
File f = null;
String currentDir = System.getProperty("user.dir");
System.out.println(currentDir);
String option_val;
option_val = "-cp " + currentDir + "\\src";
options.setValue(option_val);
try {
    VirtualMachine vm = conn.launch(arguments);
    EventManager mgr = new EventManager(vm);
    mgr.setEventRequests();
    mgr.start();




  // If a target VM is launched through this function, its output and
    // error streams must be read as it executes. These streams are available
    // through the Process object returned by VirtualMachine.process().
    // If the streams are not periodically read, the target VM will stop executing
    // when the buffers for these streams are filled.
```

```
        Process process = vm.process();

            // Copy target's output and error to our output and error.
            StreamRedirectThread errThread = new StreamRedirectThread("error reader",
                                        process.getErrorStream(),
                                        System.err);
            StreamRedirectThread outThread = new StreamRedirectThread("output reader",
                                        process.getInputStream(),
                                        System.out);
            errThread.start();
            outThread.start();
            vm.resume();
        } catch (IOException exc) {
            throw new Error("Unable to launch target VM: " + exc);
        } catch (IllegalConnectorArgumentsException exc) {
            throw new Error("Internal error: " + exc);
        } catch (VMStartException exc) {
            throw new Error("Target VM failed to initialize: " +
                        exc.getMessage());
        }
    }
}
```

EventManager.java

```
import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;

import java.util.*;
import java.io.IOException;
import java.io.PrintWriter;


public class EventManager extends Thread {

        // java library files which we want to exclude for generating any events
```

```
private String[] excludes = {"java.*", "javax.*", "sun.*",
                    "com.sun.*"};



private final VirtualMachine vm;   // Running VM
private boolean connected = true;
Stack<Set<String>> names = new Stack<Set<String>>();
HashMap<String, Set<String>> expression_map = new HashMap<String,
Set<String>>();
List<Field> watched_fields = new LinkedList<Field>();

EventManager(VirtualMachine vm) {
        this.vm = vm;
}

private void decompile_expr_map() {
Iterator it = expression_map.entrySet().iterator();
   while (it.hasNext()) {
        Map.Entry pair = (Map.Entry)it.next();
        System.out.println("Modified Field");
        System.out.println(pair.getKey());
        System.out.println("Accessed Field");
        Set<String> value = (Set<String>)pair.getValue();
        for(String s : value) {
                System.out.println(s);
        }
        System.out.println("#####################################");

   }
}

public void run() {
          EventQueue queue = vm.eventQueue();
     while (connected) {
          try {
                  EventSet eventSet = queue.remove();
          EventIterator it = eventSet.eventIterator();
          while (it.hasNext()) {
                  handleEvent(it.nextEvent());
          }
          eventSet.resume();
        } catch (InterruptedException exc) {
```

```
                // Ignore
            } catch (VMDisconnectedException discExc) {
                //handleDisconnectedException();
                break;
            }
        }
        AstVisitor visitor = new AstVisitor(expression_map, false);
             visitor.run();
        decompile_expr_map();

    }



    public void setEventRequests() {
            EventRequestManager mgr = vm.eventRequestManager();
            //if(ev == Events.E_WATCH_ACCESS_FIELDS || ev ==
                        Events.E_WATCH_MODIFY_FIELDS) {
            ClassPrepareRequest cpr = mgr.createClassPrepareRequest();
            for (int i=0; i<excludes.length; ++i) {
                    cpr.addClassExclusionFilter(excludes[i]);
       }




cpr.setSuspendPolicy(EventRequest.SUSPEND_ALL);
cpr.enable();
MethodEntryRequest mtr = mgr.createMethodEntryRequest();
            for (int i=0; i<excludes.length; ++i) {
                    mtr.addClassExclusionFilter(excludes[i]);
       }
mtr.setSuspendPolicy(EventRequest.SUSPEND_ALL);
mtr.enable();
MethodExitRequest mer = mgr.createMethodExitRequest();
            for (int i=0; i<excludes.length; ++i) {
                    mer.addClassExclusionFilter(excludes[i]);
            }
mer.setSuspendPolicy(EventRequest.SUSPEND_ALL);
mer.enable();
            //}
    }
```

```
        private void handleEvent(Event event) {
    if (event instanceof AccessWatchpointEvent) {
       fieldWatchEvent((AccessWatchpointEvent)event);
    } else if (event instanceof ModificationWatchpointEvent) {
        fieldWatchEvent((ModificationWatchpointEvent)event);
    } else if (event instanceof ClassPrepareEvent) {
        classPrepareEvent((ClassPrepareEvent)event);
    } else if (event instanceof MethodEntryEvent) {
        methodEntryEvent((MethodEntryEvent)event);
    } else if (event instanceof MethodExitEvent) {
            methodExitEvent((MethodExitEvent)event);
    } else if (event instanceof VMDeathEvent) {
      //  vmDeathEvent((VMDeathEvent)event);
    } else if (event instanceof VMStartEvent) {
        //  vmDeathEvent((VMDeathEvent)event);
    } else if (event instanceof VMDisconnectEvent) {
        //vmDisconnectEvent((VMDisconnectEvent)event);
    } else {
        throw new Error("Unexpected event type");
    }
}

        private void fieldWatchEvent(AccessWatchpointEvent event)  {



   Field field = event.field();
    Set<String> names_list = names.peek();
    String accessField = field.name() + "." +  event.location().lineNumber();

    names_list.add(accessField);
    //System.out.println("Access Watch Event 1");
                //System.out.println(accessField);
}

        private void fieldWatchEvent(ModificationWatchpointEvent event) {

                Field field = event.field();
                Set<String> names_list = names.peek();
                Set<String> names_list_new = new HashSet();
                for(String s : names_list) {
                        names_list_new.add(s);
                }
```

```
            String modField = field.name() + "." + event.location().lineNumber();
            expression_map.put(modField, names_list_new);
            names_list.clear();
    }

    private void addFieldWatcher(Field field) {
            EventRequestManager mgr = vm.eventRequestManager();
            ModificationWatchpointRequest req_1 =
              mgr.createModificationWatchpointRequest(field);
        for (int i=0; i<excludes.length; ++i) {
            req_1.addClassExclusionFilter(excludes[i]);
        }
        req_1.setSuspendPolicy(EventRequest.SUSPEND_NONE);
        req_1.enable();
            AccessWatchpointRequest req_2 =
        mgr.createAccessWatchpointRequest(field);
for (int i=0; i<excludes.length; ++i) {
   req_2.addClassExclusionFilter(excludes[i]);
}
req_2.setSuspendPolicy(EventRequest.SUSPEND_NONE);
req_2.enable();

    }

    // This event is triggered because we had created
    // ClassPrepareRequest which means a new class has been
    // loaded. Now we gave to add watchpoint request for its fields
    private void classPrepareEvent(ClassPrepareEvent event) {
List<Field> fields = event.referenceType().visibleFields();
for (Field field : fields) {
    Iterator iter = watched_fields.iterator();
    boolean watched = false;
    while(iter.hasNext()) {
            Field val = (Field)iter.next();
            if(val.equals(field)) {
                    watched = true;
            }
    }
    if(!watched) {
            //System.out.println(field.name());
            addFieldWatcher(field);
            watched_fields.add(field);
    }
```

```
    }
      }

      private void methodEntryEvent(MethodEntryEvent event)  {
            ThreadReference thread = event.thread();
            Method method = event.method();
            try {
                  List<LocalVariable> args = method.variables();
                  ListIterator<LocalVariable> it = args.listIterator();
                  while(it.hasNext()) {
                        LocalVariable lv = it.next();
                        System.out.println(lv.name());

                  }

            } catch(AbsentInformationException ex) {

            }
            String method_name = method.name() + ".";
            try {
                  List<StackFrame> frames = thread.frames();
                  // This step is needed because we need the method call location not the
method declaration location
                  if(frames.size() > 1) {
                    StackFrame frame = frames.get(1);
                    method_name += frame.location().lineNumber();
                  }
            } catch(IncompatibleThreadStateException ex) {

            }
            if(!names.empty()) {
                  Set<String> names_list = names.peek();
                  names_list.add(method_name);
            }
            Set<String> names_list_new = new HashSet();
            names.push(names_list_new);
  }

      private void methodExitEvent(MethodExitEvent event)  {
            names.pop();
  }
```

}

ASTVisitor.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.resources.IWorkspaceRoot;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.jdt.core.IJavaElement;
import org.eclipse.jdt.core.IJavaProject;
import org.eclipse.jdt.core.ISourceManipulation;
import org.eclipse.jdt.core.ITypeRoot;
import org.eclipse.jdt.core.JavaCore;
import org.eclipse.jdt.core.dom.AST;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.Expression;
import org.eclipse.text.edits.MalformedTreeException;
import org.eclipse.text.edits.TextEdit;
import org.eclipse.text.edits.UndoEdit;
import org.eclipse.jdt.core.dom.ASTParser;
import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.AnonymousClassDeclaration;
import org.eclipse.jdt.core.dom.Assignment;
import org.eclipse.jdt.core.dom.BodyDeclaration;
import org.eclipse.jdt.core.dom.CastExpression;
import org.eclipse.jdt.core.dom.ClassInstanceCreation;
import org.eclipse.jdt.core.dom.CompilationUnit;
import org.eclipse.jdt.core.dom.ExpressionStatement;
import org.eclipse.jdt.core.dom.FieldDeclaration;
import org.eclipse.jdt.core.dom.IVariableBinding;
import org.eclipse.jdt.core.dom.ImportDeclaration;
import org.eclipse.jdt.core.dom.InfixExpression;
```

```
import org.eclipse.jdt.core.dom.Initializer;
import org.eclipse.jdt.core.dom.InstanceofExpression;
import org.eclipse.jdt.core.dom.MethodDeclaration;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.Modifier;
import org.eclipse.jdt.core.dom.Name;
import org.eclipse.jdt.core.dom.ParameterizedType;
import org.eclipse.jdt.core.dom.SimpleName;
import org.eclipse.jdt.core.dom.Statement;
import org.eclipse.jdt.core.dom.StringLiteral;
import org.eclipse.jdt.core.dom.StructuralPropertyDescriptor;
import org.eclipse.jdt.core.dom.SuperFieldAccess;
import org.eclipse.jdt.core.dom.Type;
import org.eclipse.jdt.core.dom.TypeDeclaration;
import org.eclipse.jdt.core.dom.VariableDeclaration;
import org.eclipse.jdt.core.dom.VariableDeclarationFragment;
import org.eclipse.jdt.core.dom.VariableDeclarationStatement;
import org.eclipse.jdt.core.dom.rewrite.ASTRewrite;
import org.eclipse.jdt.internal.compiler.ast.PostfixExpression;
import org.eclipse.jface.text.BadLocationException;
import org.eclipse.jface.text.Document;

import com.sun.jdi.PrimitiveType;


public class AstVisitor {
	String filepath = new String();
	String source_path = new String();

	HashMap<IVariableBinding, String> oldVarVsNewVar = new HashMap();
	// TODO Currently using this index to generate name of new vars
	// Change it to unique string generation code
	static int iVarCnt = 0;
	HashMap<String, Set<String>> expression_map;
	String current_key = null;
	boolean monitor_current_key = false;
	Set<String> rvars = new HashSet();
	AST ast = null;
	boolean modify = false;
	LinkedList<SimpleName> nodesToBeReplaced = new LinkedList();


	AstVisitor(HashMap<String, Set<String>> expression_map_, boolean modify_) {
```

```
                expression_map = expression_map_;
                modify = modify_;
                String currentDir = System.getProperty("user.dir");
                source_path = currentDir + "\\src";
                filepath = source_path + "\\test.java";
        }

        String get_output_filepath() {
                return filepath;
        }

        private FieldDeclaration createNewField(AST ast, String name, int modifiers) {
                VariableDeclarationFragment fragment= ast.newVariableDeclarationFragment();

                fragment.setName(ast.newSimpleName(name));
                ClassInstanceCreation cInst = ast.newClassInstanceCreation();
                // Initializing the map here

cInst.setType(ast.newSimpleType(ast.newSimpleName("ConcurrentHashMap")));
                fragment.setInitializer(cInst);
                FieldDeclaration declaration= ast.newFieldDeclaration(fragment);
                //Creating ConcurrentHashMap for local fields
                //TODO look at the second argument type
                ParameterizedType newType =
ast.newParameterizedType(ast.newSimpleType(ast.newSimpleName("ConcurrentHashMap")));

newType.typeArguments().add(ast.newSimpleType(ast.newSimpleName("Integer")));

newType.typeArguments().add(ast.newSimpleType(ast.newSimpleName("Integer")));
      declaration.setType(newType);
      declaration.modifiers().addAll(ast.newModifiers(modifiers));
                return declaration;
        }

        public ASTNode getOuterClass(ASTNode node) {
      String str = "no";
            do {
               node= node.getParent();
            } while (node != null && node.getNodeType() != ASTNode.TYPE_DECLARATION);

             return node;
        }
```

```
        public ASTNode getOuterMethod(ASTNode node) {
      String str = "no";
            do {
               node= node.getParent();
            } while (node != null && node.getNodeType() !=
ASTNode.METHOD_DECLARATION);

             return node;
        }

        private String readFileToString(String filePath) throws IOException {
                StringBuilder fileData = new StringBuilder(1000);
                BufferedReader reader = new BufferedReader(new FileReader(filePath));

                char[] buf = new char[10];
                int numRead = 0;
                while ((numRead = reader.read(buf)) != -1) {
                        //System.out.println(numRead);
                        String readData = String.valueOf(buf, 0, numRead);
                        fileData.append(readData);
                        buf = new char[1024];
                }

                reader.close();

                return  fileData.toString();
        }

        public void run() {
                ASTParser parser = ASTParser.newParser(AST.JLS4);
                try {
                        parser.setKind(ASTParser.K_COMPILATION_UNIT);
                        parser.setResolveBindings(true);
                        Map options = JavaCore.getOptions();
                        parser.setCompilerOptions(options);
                        String unitName = "test.java";
                        parser.setUnitName(unitName);
                        String[] sources = { source_path };
                        String[] classpath = {};
                   parser.setEnvironment(classpath, sources, new String[] { "UTF-8"}, true);
                        parser.setSource(readFileToString(filepath).toCharArray());

            } catch(IOException ex) {
```

```
		}


		final CompilationUnit cu = (CompilationUnit) parser.createAST(null);
		ASTRewrite rewriter = ASTRewrite.create(cu.getAST());
		Document document = null;
		try {
			document = new Document(readFileToString(filepath).toString());
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		TextEdit edits;

		if(modify) {
			cu.recordModifications();
			ast = cu.getAST();
			ImportDeclaration decl = cu.getAST().newImportDeclaration();

decl.setName(cu.getAST().newQualifiedName(cu.getAST().newName("java"),
					cu.getAST().newSimpleName("util")));

decl.setName(cu.getAST().newQualifiedName(cu.getAST().newName(decl.getName().toString()
),
					cu.getAST().newSimpleName("concurrent")));

decl.setName(cu.getAST().newQualifiedName(cu.getAST().newName(decl.getName().toString()
),
				cu.getAST().newSimpleName("ConcurrentHashMap")));
			cu.imports().add(decl);
		}

		cu.accept(new ASTVisitor() {
			Set names = new HashSet();
			public boolean visit(Assignment node) {
				if(!modify) {
					monitor_current_key = true;
				}
				return true;
			}

			public void endVisit(Assignment node) {
```

```
                        if(!modify) {
                                if( expression_map.containsKey(current_key)) {
                                        Set<String> rvars_orig =
expression_map.get(current_key);
                                        Iterator<String> it = rvars_orig.iterator();
                                        while (it.hasNext()) {
                                                String var = (String)it.next();
                                                if(!rvars.contains(var)) {
                                                        it.remove();
                                                }
                                        }
                                }
                                monitor_current_key = false;
                                current_key = null;
                                rvars.clear();
                        }
                        return;
                }


                public void endVisit(VariableDeclarationStatement node) {
                        List fragments = node.fragments();
                        Iterator iter = fragments.iterator();
                        while(iter.hasNext()) {
                                VariableDeclarationFragment fragment =
(VariableDeclarationFragment) iter.next();
                                SimpleName name = fragment.getName();

                                if(!modify) {

                                        this.names.add(name.getIdentifier());
                                }
                                Type type = node.getType();
                                // IVariablebinding for checking if its a field and use
node.getType to check
                                // if it is a primitive field
                                IVariableBinding binding = fragment.resolveBinding();
                                if(binding != null && !binding.isField() &&
node.getType().isPrimitiveType() && modify) {
                                        MethodDeclaration method = (MethodDeclaration)
getOuterMethod(fragment);
                                        TypeDeclaration parent_class = (TypeDeclaration)
getOuterClass(method);
```

```
                                        String new_name = name.getFullyQualifiedName()
+ "__" + iVarCnt;
                                        iVarCnt++;
                                        BodyDeclaration declaration = createNewField(ast,
new_name,Modifier.STATIC);
                                        oldVarVsNewVar.put(binding, new_name);
                                        parent_class.bodyDeclarations().add(0,
declaration);
                                        MethodInvocation new_field_initialization =
ast.newMethodInvocation();

new_field_initialization.setName(ast.newSimpleName("put"));

new_field_initialization.setExpression(ast.newSimpleName(new_name));

                                    CastExpression cast_expr = ast.newCastExpression();
                                    MethodInvocation invoc1 = ast.newMethodInvocation();
                                    invoc1.setName(ast.newSimpleName("currentThread"));
                                    invoc1.setExpression(ast.newSimpleName("Thread"));
                                    MethodInvocation invoc2 = ast.newMethodInvocation();
                                    invoc2.setName(ast.newSimpleName("getId"));
                                    invoc2.setExpression(invoc1);
                                    cast_expr.setExpression(invoc2);
                                    new_field_initialization.arguments().add(cast_expr);
                                    InstanceofExpression second_expr =
ast.newInstanceofExpression();

new_field_initialization.arguments().add(ASTNode.copySubtree(second_expr.getAST(),
fragment.getInitializer()));
                                        ExpressionStatement expr =
ast.newExpressionStatement(new_field_initialization);
                                        List stmts = method.getBody().statements();
                                        int curr_index = 0;
                                        Iterator iter1 = stmts.iterator();
                                        // Finding index where to insert the initializer
                                        // expression
                                    // Finding index where to insert the initializer expression
                                    while(iter1.hasNext()) {
                                        Object statement = iter1.next();
                                        if(statement.equals(node)) {
                                                stmts.add(++curr_index, expr);
                                        }
                                        curr_index++;
```

```
                                    }

                                }

                            }
                            return;
                        }


                        public boolean visit(SimpleName node) {
                            String var = node.toString() + "." +
cu.getLineNumber(node.getStartPosition());
                            if(oldVarVsNewVar.containsKey(node.resolveBinding()) &&
modify) {
                                //TODO replace this node with the new expression node
                                //try {
                                MethodInvocation replacement =
ast.newMethodInvocation();
                                //ASTNode parent = node.getParent();

                                replacement.setName(ast.newSimpleName("get"));

replacement.setExpression(ast.newSimpleName(oldVarVsNewVar.get(node.resolveBinding())));

                                CastExpression cast_expr = ast.newCastExpression();
                                MethodInvocation invoc1 = ast.newMethodInvocation();
                                invoc1.setName(ast.newSimpleName("currentThread"));
                                invoc1.setExpression(ast.newSimpleName("Thread"));
                                MethodInvocation invoc2 = ast.newMethodInvocation();
                                invoc2.setName(ast.newSimpleName("getId"));
                                invoc2.setExpression(invoc1);
                                cast_expr.setExpression(invoc2);
                                replacement.arguments().add(cast_expr);
                                final ASTNode parent = node.getParent();
                                final StructuralPropertyDescriptor descriptor =
node.getLocationInParent();
                                if (descriptor != null) {
                                    if (descriptor.isChildProperty()) {
                                        parent.setStructuralProperty(descriptor, replacement);
                                        node.delete();
                                    } else if (descriptor.isChildListProperty()) {
                                        @SuppressWarnings("unchecked")
```

```
                                        final List<ASTNode> children = (List<ASTNode>)
parent.getStructuralProperty(descriptor);
                                        children.set(children.indexOf(node), replacement);
                                        node.delete();
                                    }
                                }
                                        return true;
                            }
                                if(monitor_current_key && current_key == null) {

                                        current_key = var;
                                } else if(monitor_current_key) {
                                        rvars.add(var);
                                }
                                return true;
                        }

                });
                if(modify) {
                        try {
                                edits = cu.rewrite(document, null);
                                edits.apply(document);
                                PrintWriter writer = new PrintWriter(filepath, "UTF-8");
                                writer.print(document.get());
                                writer.close();

                        } catch(IOException ex){
                        } catch(MalformedTreeException ex) {
                        } catch(BadLocationException ex) {
                        }
                }

        }
}
```

StreamRedirectThread.java

```
import java.io.*;

class StreamRedirectThread extends Thread {
```

```
    private final Reader in;
    private final Writer out;

    private static final int BUFFER_SIZE = 2048;

    /**
     * @param name  Name of the thread
     * @param in    Stream to copy from
     * @param out   Stream to copy to
     */
    StreamRedirectThread(String name, InputStream in, OutputStream out) {
        super(name);
        this.in = new InputStreamReader(in);
        this.out = new OutputStreamWriter(out);
        setPriority(Thread.MAX_PRIORITY-1);
    }


    public void run() {
        try {
            char[] cbuf = new char[BUFFER_SIZE];
            int count;
            while ((count = in.read(cbuf, 0, BUFFER_SIZE)) >= 0) {
                out.write(cbuf, 0, count);
            }
            out.flush();
        } catch(IOException exc) {
            System.err.println("Child I/O Transfer - " + exc);
        }
    }
}
```

Jars Required

Below is a list of jars required by the project:

1. commons-collections-3.2.1
2. commons-configuration-1.6
3. commons-io-1.4
4. commons-lang-2.5
5. commons-logging-1.1.1
6. org.eclipse.core.contenttype_3.4.1.R35x_v20090826-0451
7. org.eclipse.core.jobs_3.4.100.v20090429-1800

8. org.eclipse.core.resources_3.5.2.R35x_v20091203-1235
9. org.eclipse.core.resources-3.8.100.v20130521-2026
10. org.eclipse.core.runtime_3.5.0.v20090525
11. org.eclipse.core.runtime-3.7.0
12. org.eclipse.equinox.common_3.5.1.R35x_v20090807-1100
13. org.eclipse.equinox.common-3.6.0.v20100503
14. org.eclipse.equinox.preferences_3.2.301.R35x_v20091117
15. org.eclipse.jdt.astview_1.1.8.201108081127
16. org.eclipse.jdt.astview_1.1.9.201406161921
18. org.eclipse.jdt.core-3.8.0.v_c18
19. org.eclipse.osgi_3.5.2.R35x_v20100126
20. org.eclipse.osgi-3.7.1
21. org.eclipse.text_3.5.0
22. tools

References

[1] https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/

[2]http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTParser.html

[3] https://en.wikipedia.org/wiki/Use-define_chain

[4] https://en.wikipedia.org/wiki/Data-flow_analysis

[5] https://en.wikipedia.org/wiki/Gradle