Soot Introduction

Kong junjun @OS
2008-10-20

Soot, a tool for analyzing and transforming Java bytecode, was developed at McGill University, Canada. Soot is also free software and is licensed under the GNU Lesser General Public License. Its latest version is 2.3.0 (June 3, 2008). Note that Soot requires at least JDK 1.5. The Eclipse plugin requires at least JDK 1.5.

What's soot?

Soot is a Java optimization framework. It provides four intermediate representations for analyzing and transforming Java bytecode:

1) Baf: a streamlined representation of bytecode which is simple to manipulate.
2) Jimple: a typed 3-address intermediate representation suitable for optimization.
3) Shimple: an SSA variation of Jimple.
4) Grimp: an aggregated version of Jimple suitable for decompilation and code inspection.

Soot can be used as a standalone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode.
Initial version was released on March 1, 2000, which was started in 1996-97. This introduction is mainly based on its latest version: 2.3.0.

1. Usage

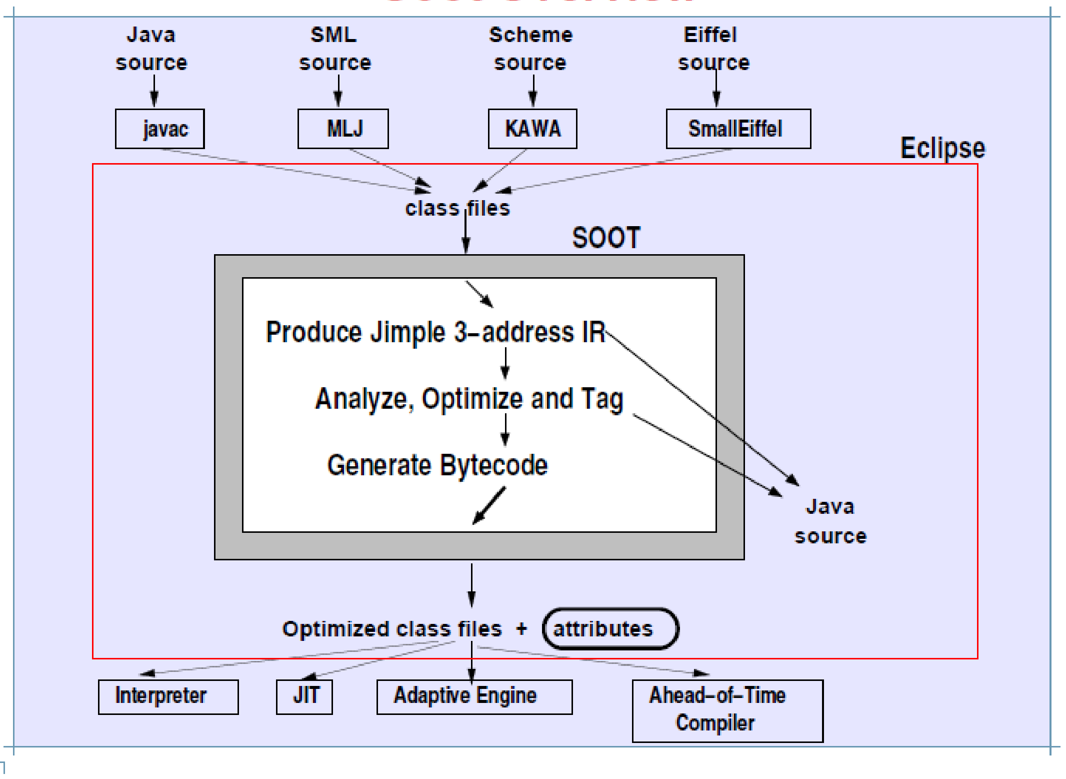
Soot is being used in the following fields:

1) compiler research: for a wide variety of applications
2) graduate course projects: used in several compiler courses

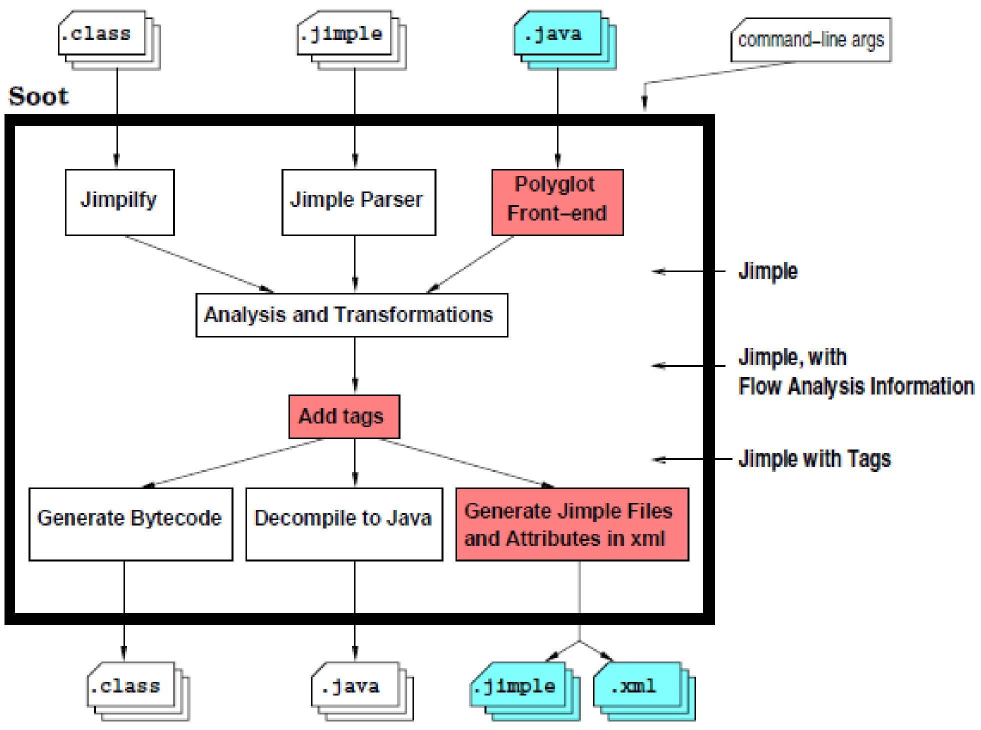
For example, uwartloo uses Soot in Advanced Compiler Design (Spring 2008). (http://plg.uwaterloo.ca/~olhotak/cs744/) Besides, BRICS in Denmark uses Soot for research. (http://www.brics.dk/SootGuide/)

2. Overview of the entire system

Soot Overview



From: http://www.sable.mcgill.ca/soot/tutorial/pldi03/tutorial.pdf



From: http://plg.uwaterloo.ca/~olhotak/cs744/4soot.pdf

3. Intermediate representations

One of the main benefits of Soot is that it provides four different Intermediate Representations (IRs) for analysis purposes. Each of the IRs has different levels of abstraction that give different benefits when analyzing, they are: Baf, Grimp, Jimple and Shimple.

1) Baf: is a compact rep. of Bytecode (stack-based)
2) Jimple: is Java's simple, typed, 3-addr (stackless) representation
3) Shimple: is a SSA-version of Jimple
4) Grimp: is like Jimple, but with expressions agGRegated

Jimple

Three address representation (Jimple) provides a good foundation for program analysis. Jimple is:

- principal Soot Intermediate Representation
- 3-address code in a control-flow graph
- a typed intermediate representation
- stackless

Here we give an example.

```
public class Test {
        public static void main(String[] args)
        {
                int i=0;
                String cstr="Hello ";
                for(;i<10;i++)
                {//output   a string "Hello ?"
                        String str;
                        str=cstr+i;
                        System.out.println(str);
                }
        }
}
```

1 Java Source Code

The above public class Test will be represented by Jimple as following:

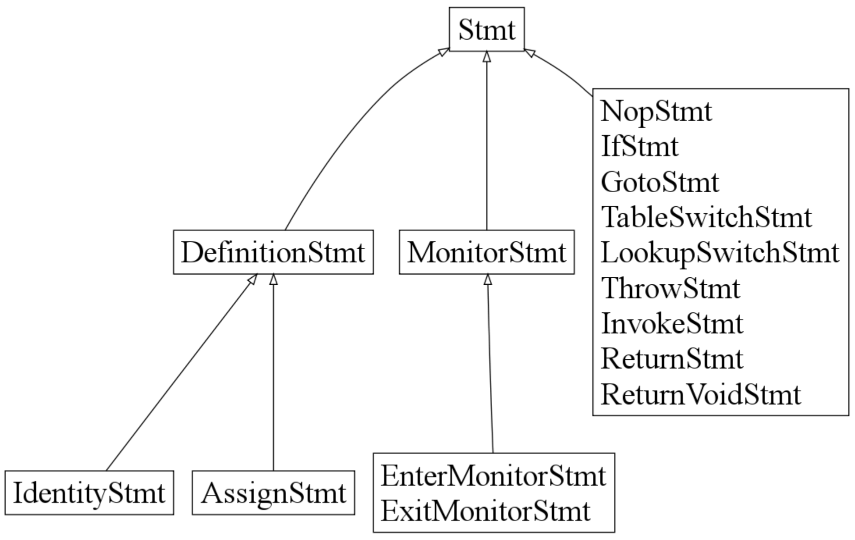
```
public class Test extends java.lang.Object
{

      public static void main(java.lang.String[])
      {
            java.lang.String[] args;
            int i, temp$3, temp$4;
            java.lang.String cstr, str, temp$1;
            java.lang.StringBuffer temp$0;
            java.io.PrintStream temp$2;

            args := @parameter0: java.lang.String[];
            i = 0;
            cstr = "Hello ";

        label0:
            nop;
            if i < 10 goto label1;

            goto label2;

        label1:
            nop;
            temp$0 = new java.lang.StringBuffer;
            specialinvoke temp$0.<java.lang.StringBuffer: void <init>()>();
            virtualinvoke temp$0.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.Object)>(cstr);
            virtualinvoke temp$0.<java.lang.StringBuffer: java.lang.StringBuffer append(int)>(i);
            temp$1 = virtualinvoke temp$0.<java.lang.StringBuffer: java.lang.String toString()>();
            str = temp$1;
            temp$2 = <java.lang.System: java.io.PrintStream out>;
            virtualinvoke temp$2.<java.io.PrintStream: void println(java.lang.String)>(str);
            nop;
            temp$3 = i;
            temp$4 = temp$3 + 1;
            i = temp$4;
            goto label0;

        label2:
            nop;
            return;
       }
  public void <init>()
       {
            Test this;

            this := @this: Test;
            specialinvoke this.<java.lang.Object: void <init>()>();
            return;
       }
}
```

2 Jimple Representation for class Test

At the heart of the Soot bytecode optimization framework is the Jimple Internal Representation. We have already seen Soot's ability to load Java .class files. Soot can output the Jimple representation of classes in a textual format (.jimple files) and reread this textual format back into Soot.

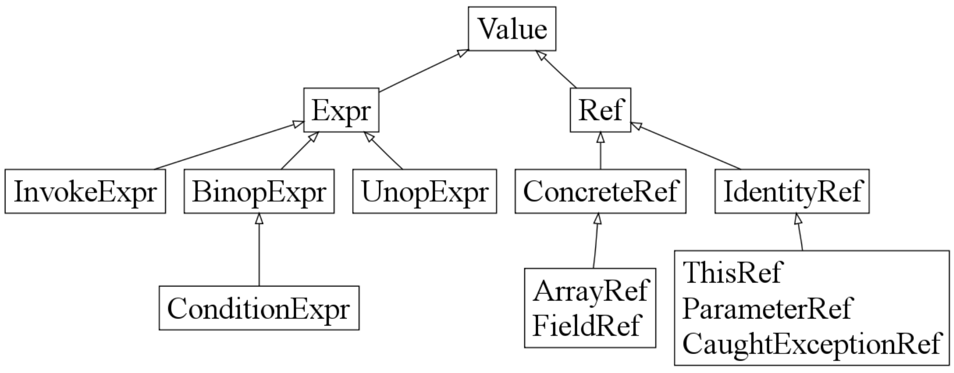
The ability to parsing textual .jimple files is quite handy. For example, a user could output classfiles as .jimple files, tweak these by hand, and then use Soot to parse them and output tweaked bytecode. Soot thus provides a high-level facility for editing class files. Another potential application of the .jimple files is to cache optimized classfiles for future analysis. Soot can, at a later time, read these files back in, and will not need to reoptimize the original classfile. This leads to faster compilation times. Finally, a user, or a software tool, can write a class in the Jimple textual format, use Soot to parse the Jimple, and thus create bytecode.

The Jimple format is self-explanatory. The SableCC grammar for textual Jimple can be found in the source code distribution of Soot; it is called jimple.scc.



3-1 Jimple Statements

From: http://plg.uwaterloo.ca/~olhotak/cs744/4soot.pdf



3-2 Jimple Expressions

From: http://plg.uwaterloo.ca/~olhotak/cs744/4soot.pdf

4. Tags

Tags are objects that can be attached to Hosts, Hosts are objects that can hold Tags.

```
package soot.tagkit;
public interface Tag {
        public String getName ();
        public String toString();
}

package soot.tagkit;
public interface Host {
public void addTag (Tag t);
public Tag getTag (String aName);
public List getTags ();
public void removeTag (String name);
public boolean hasTag (String aName);
}
```

Soot can display the results of a data-flow analysis using three different kinds of tags:

- StringTag – shows a string when you hover over the tag with your mouse
- ColorTag – highlights the tagged Stmt or Value with a color
- LinkTag – lets you jump from the tagged Stmt or Value to the line of another Stmt by clicking on the source Stmt or Value

Soot displays these tags right in line with the generated code and with the source code.

5. Soot - Eclipse Plugin

Introduction

The Soot - Eclipse Plugin integrates Soot, a Java optimization framework, with Eclipse, allowing the user to:

1) optimize class files automatically and use more advanced optimization options
2) import class files and decompile them using the Dava Decompiler
3) use a Jimple editor with syntax highlighting
4) view attribute information at source and IR levels
5) develop analyses with an interactive control flow graph

This plugin works with the Eclipse official release 3.1 and should run in any of the 3.x releases. Since version 2.0, Soot includes a plugin that makes it possible to use Soot from Eclipse. You can also to develop Soot within the Eclipse environment.

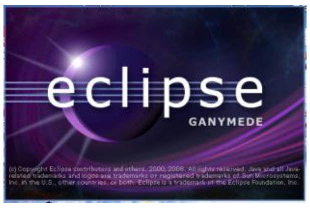
Setting up the Soot Eclipse plugin

For instruction on how to set up the Eclipse plugin, if your Eclipse version is 3.3.x, then refer to http://www.sable.mcgill.ca/soot/eclipse/updates/.. If you are using Eclipse 3.4.0, do as follows.

Step 1: set up Eclipse 3.4.0 first

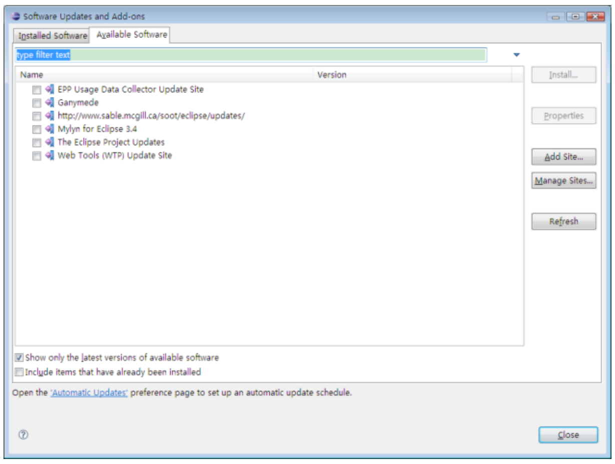
Goto website: http://www.eclipse.org/, and download Eclipse 3.4.0 freely

Step 2: Start Eclipse



Step 3: left click "Help" within Eclipse then choose "Software Updates…"

Now you get the window:



Software Updates and Add-ons

Installed Software | Available Software

type filter text

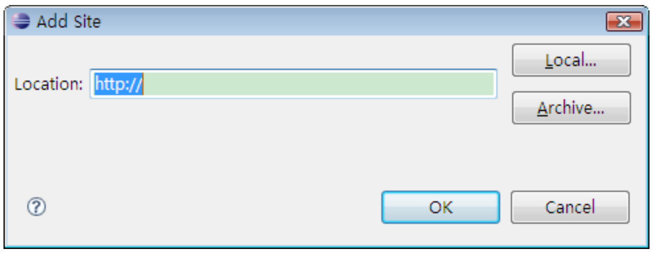
Name	Version
EPP Usage Data Collector Update Site	
Ganymede	
http://www.sable.mcgill.ca/soot/eclipse/updates/	
Mylyn for Eclipse 3.4	
The Eclipse Project Updates	
Web Tools (WTP) Update Site	

[✓] Show only the latest versions of available software
[ ] Include items that have already been installed

Open the 'Automatic Updates' preference page to set up an automatic update schedule.

- Select the tab "Available Software"
- Left click the button "Add site…"

Now a small dialog window pops up:



Add Site

Location: http://

- Use this site for the URL: http://www.sable.mcgill.ca/soot/eclipse/updates/
- Click ok
- Select the row named http://www.sable.mcgill.ca/soot/eclipse/updates/
- Expand the item and select soot
- Click "Install…" on the upper right

The plugin will be installed and you will be asked to restart Eclipse. Press: Yes as this is usually necessary. Once Eclipse has restarted the Soot - Eclipse plugin will be available for use within Eclipse.

Developing With Soot in Eclipse

Control-flow graph is called UnitGraph in Soot. A unit graph contains statements as nodes, and there is an edge between two nodes if control can flow from the statement represented by the source node to the statement represented by the target node.

A data-flow analysis associates two elements with each node in the unit graph, usually two so-called flow sets: one in-set and one out-set. These sets are (1) initialized, then (2) propagated through the unit graph along statement nodes until (3) a fixed point is reached.

In the end, all you do is inspect the flow set before and after each statement. By the design of the analysis, your flow sets should tell you exactly the information you need.

There exist three different kind of FlowAnalysis implementations in Soot:

- ForwardFlowAnalysis – this analysis starts at the entry statement of a UnitGraph and propagates forward from there,
- BackwardsFlowAnalysis – this analysis starts at the exit node(s) of a Unit Graph and propagates back from there (as an alternative you can produce the InverseGraph of your UnitGraph and use a ForwardFlowAnalysis; it does not matter); and
- ForwardBranchedFlowAnalysis – this is essentially a forward analysis but it allows you to propagate different flow sets into different branches. For instance, if you process a statement like if(p!=null) then you may propagate the into “p is not null” into the “then” branch and “p is null” into the ”else” branch.

What direction you want to use depends entirely on your analysis problem.

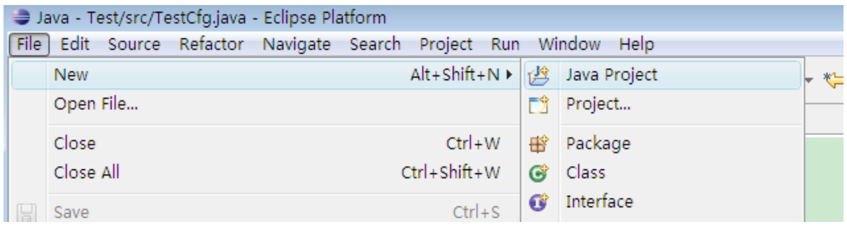
Displaying analysis result as tags

You can display the result of your analysis as tags. To do so, simply add a Tag to any Host. After Eclipse has executed your analysis, it willdisplay the results stored in the tag when you hover over the tagged element with your mouse.

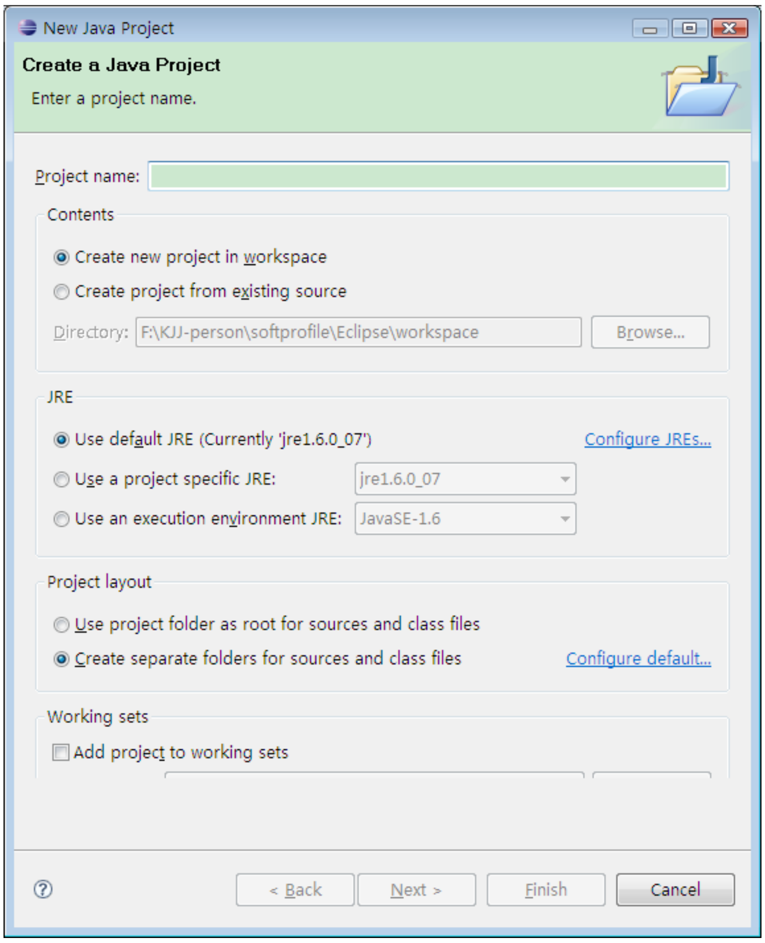
[from: http://www.bodden.de/tag/soot-tutorial/]

Setting up a project so that Soot can work with it

➡ Choose: File->New->Java Project



Now you get the following window:



New Java Project

Create a Java Project

Enter a project name.

Project name:

Contents

[✓ Create new project in workspace]

Create project from existing source

Directory: F:\KJJ-person\softprofile\Eclipse\workspace Browse...

JRE

[✓ Use default JRE (Currently 'jre1.6.0_07')] Configure JREs...

Use a project specific JRE: jre1.6.0_07

Use an execution environment JRE: JavaSE-1.6

Project layout

Use project folder as root for sources and class files

[✓ Create separate folders for sources and class files] Configure default...

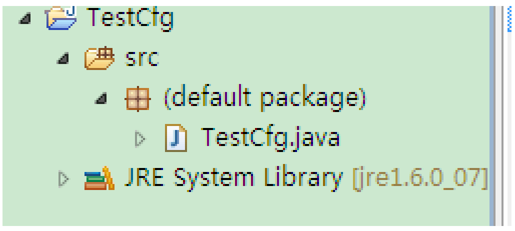
Working sets

Add project to working sets

< Back Next > Finish Cancel

- Give the project a name, take TestCfg for example
- Click Finish

Now you get a project in your workbench as following:



```
TestCfg
    src
        (default package)
            TestCfg.java
    JRE System Library [jre1.6.0_07]
```

The java file named TestCfg.java(Note: this is only an example):

```
public class TestCfg {
    public void TestFor()
    {
        int i;
        int a[][]=new int[3][3];
        for(i=0;i<3;i++)
        {
            a[i][i]=0;
        }
    }
}
```

Transforming files between different (intermediate) representations

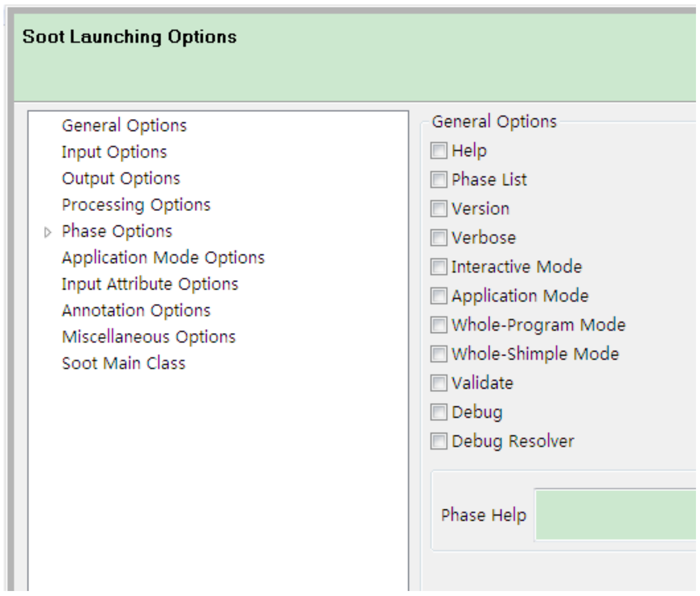
The eclipse plugin makes it easy to convert Java source files or bytecode files into the Jimple or Grimp intermediate representations, and even to convert back from those representations into source code or bytecode.

Soot places the generated files into a folder “sootOutput” in the current directory. It also shows its progress in a console view.

Use soot in Eclipse to get a cfg

- Right click the java file(TestCfg.java as above) as input file
- Choose Soot->Process Source File->Run soot…

Now you get the soot option window:


Soot Launching Options

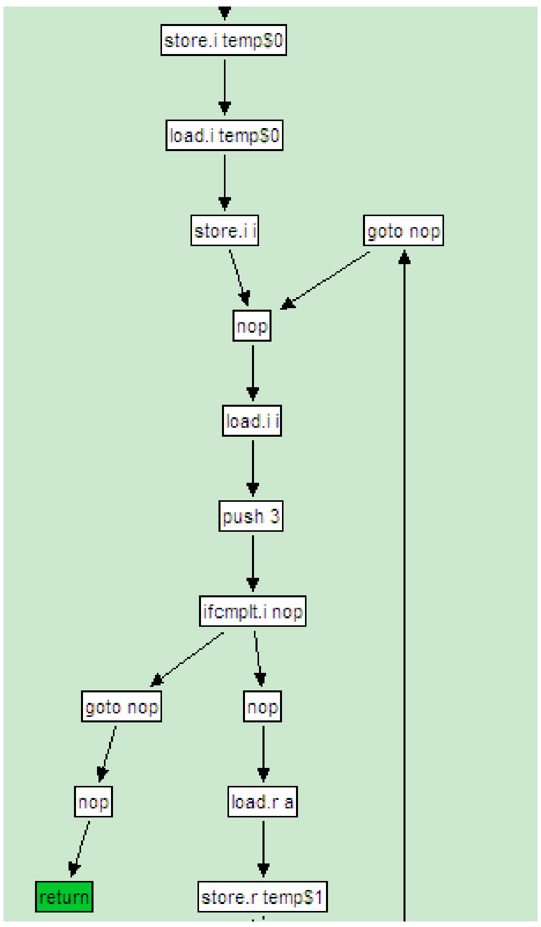
General Options
Input Options
Output Options
Processing Options
Phase Options
Application Mode Options
Input Attribute Options
Annotation Options
Miscellaneous Options
Soot Main Class

General Options
- [ ] Help
- [ ] Phase List
- [ ] Version
- [ ] Verbose
- [ ] Interactive Mode
- [ ] Application Mode
- [ ] Whole-Program Mode
- [ ] Whole-Shimple Mode
- [ ] Validate
- [ ] Debug
- [ ] Debug Resolver

Phase Help

- Click General Options, then choose Interactive Mode
- Click Run

Then you get the following “Control flow graph”(Note: the following picture only shows part the graph):



To get more details, please refer to this website:
http://www.sable.mcgill.ca/soot/eclipse/ca.mcgill.sable.soot/doc/concepts/concepts.html.
http://www.sable.mcgill.ca/soot/soot_in_eclipse_howto.html

An example: liveness

How to use soot to do our own analysis is what we concern mostly. Here we give you an example.

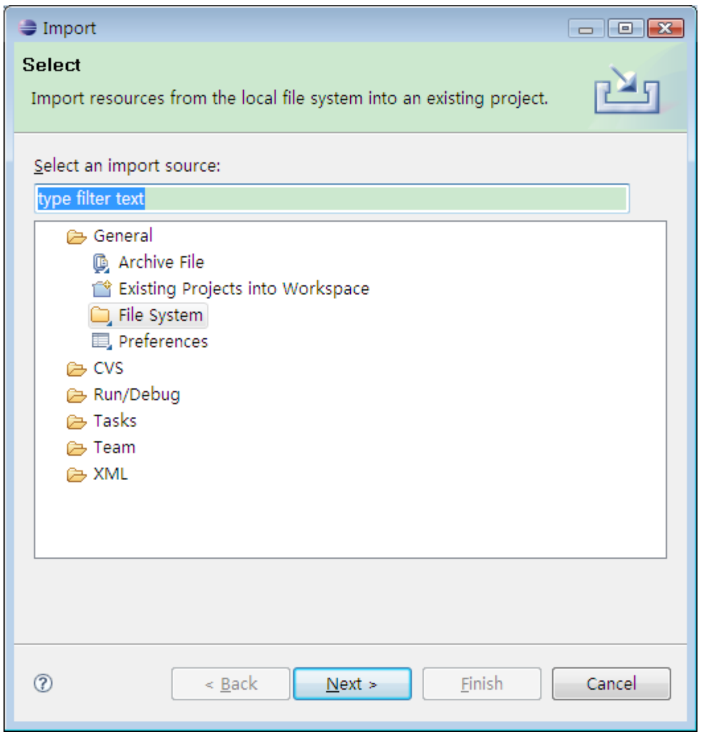
Step 1: create a project within Eclipse for testing

Step 2: create another project for your analysis code

Suppose you have created an empty project named liveness for your analysis.
Here we use some example codes from BRICS, hence first of all, you must import those java files. Do the following step by step.

- Right click the project icon
- Choose import

Now you get to this point:



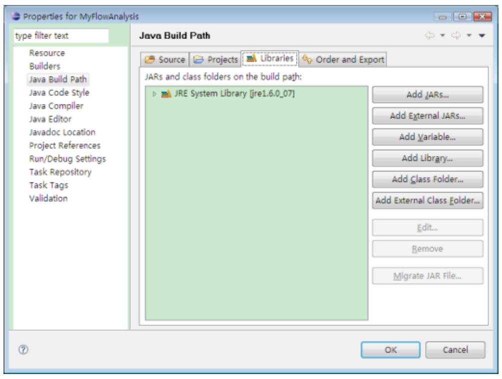
- Choose file system then click Next
- Browse to get your target java files
- Select your files for analyzing
- Click Finish

Because your codes may depends on some packages from soot, you must add the soot path into the current build path.

Configure the build path

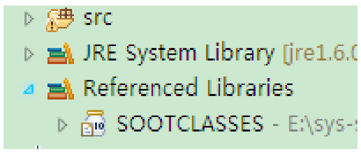
- Right click the project icon
- Choose: Build Path->Configure Build Path..
- Select the Libraries tab

Now what's on the screen is such a window:



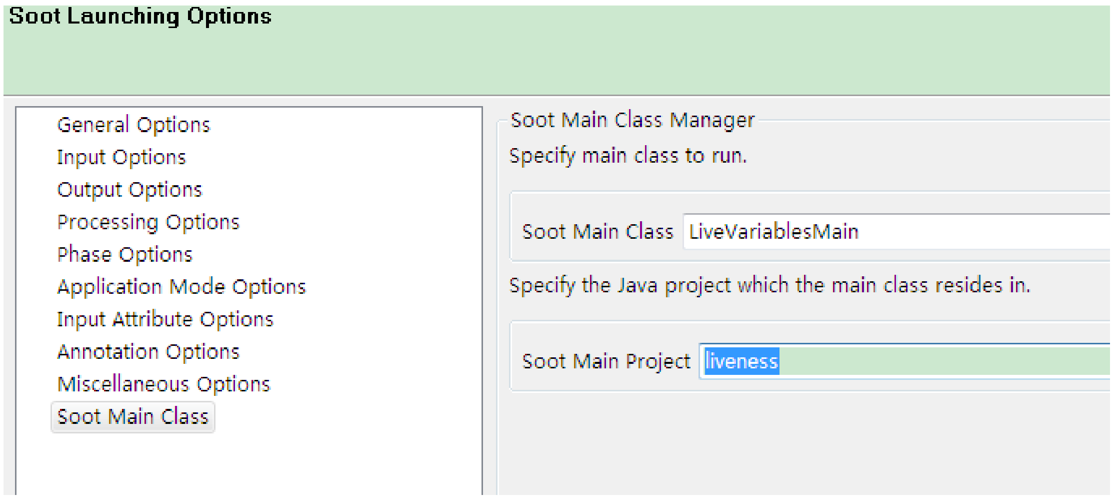
- Click Add Variables..
- Select SOOTCLASSES then click ok
- Click ok

Now you add a special lib to your working project.



Step 3: do variables liveness analysis with custom codes in Eclipse.

- Right click the test file in your first project
- Choose: Soot->Process Source File->Run soot…
- Select "Soot Main Class"
- Specify main class to run, input main class name and project name(eg. LiveVariablesMain and liveness)


Soot Launching Options

Soot Main Class Manager

Specify main class to run.

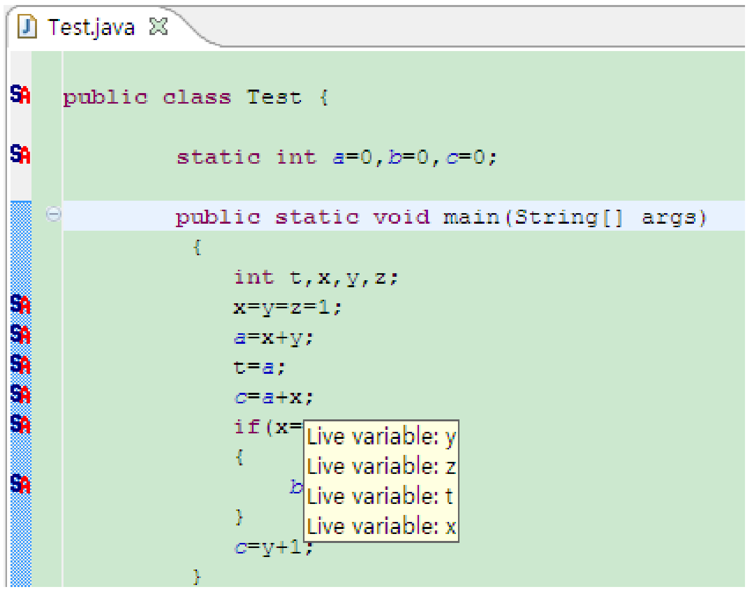
Soot Main Class: LiveVariablesMain

Specify the Java project which the main class resides in.

Soot Main Project: liveness

- Click run

Wait for a moment, you will get the result.



```
public class Test {

        static int a=0,b=0,c=0;

        public static void main(String[] args)
         {
            int t,x,y,z;
            x=y=z=1;
            a=x+y;
            t=a;
            c=a+x;
            if(x=
            {
                b
            }
            c=y+1;
         }
```

```
Live variable: y
Live variable: z
Live variable: t
Live variable: x
```

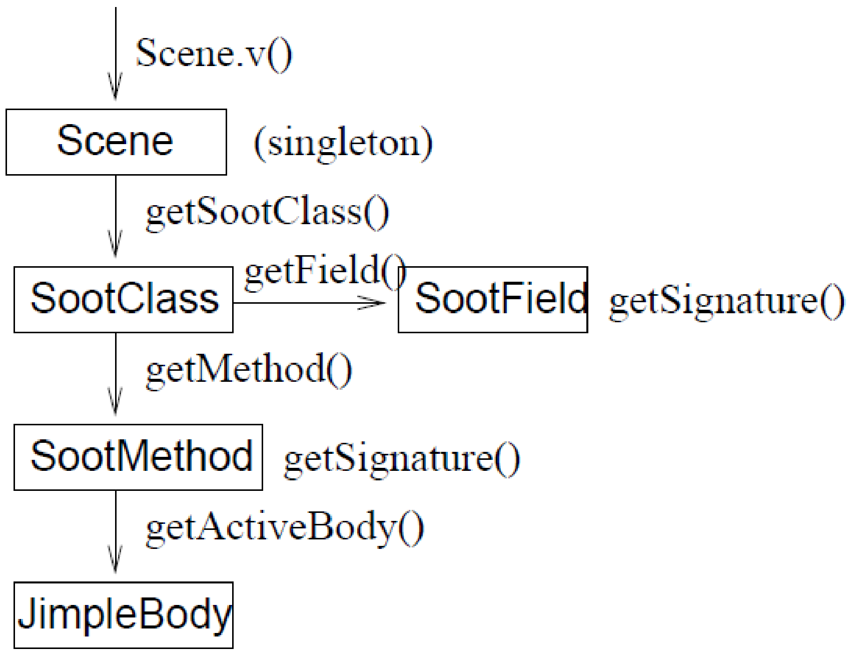
From the above picture we see it shows live variables when you hover over the tag with your mouse.

6. Fundamental Soot objects

Soot builds data structures to represent:

- Scene. The Scene class represents the complete environment the analysis takes place in. Through it, you can set e.g., the application classes(The classes supplied to Soot for analysis), the main class (the one that contains the main method) and access information regarding interprocedural analysis (e.g., points-to information and call graphs).
- SootClass. Represents a single class loaded into Soot or created using Soot.
- SootMethod. Represents a single method of a class.
- SootField. Represents a member field of a class.
- Body. Represents a method body and comes in different flavors, corresponding to different IRs (e.g., JimpleBody).

These data structures are implemented using Object-Oriented techniques, and designed to be easy to use and generic where possible.



6-1 Soot Classes

From: http://plg.uwaterloo.ca/~olhotak/cs744/4soot.pdf

For more information, refer to this tutorial: A Survivor’s Guide to Java Program Analysis with Soot.

Appendix

Soot's home:
(1)McGill University
http://www.sable.mcgill.ca/soot/
<<<麦吉尔大学,是加拿大唯一一所能与 University of Toronto 相提并论的大学。 >>>
(2)Another wonderful site
http://www.brics.dk/SootGuide/
<<<丹麦（Denmark）的计算机科学基础研究中心(BRICS)>>>

download soot:
http://www.sable.mcgill.ca/soot/soot_download.html

soot api doc :
http://www.sable.mcgill.ca/soot/doc/index.html
localfs://soot\sootall-2.3.0\soot-2.3.0\doc\index.html

soot tutorial:
http://plg.uwaterloo.ca/~olhotak/cs744/
http://www.sable.mcgill.ca/soot/tutorial/index.html
localfs:// soot\sootall-2.3.0\soot-2.3.0\tutorial\index.html

Soot Eclipse plugin:
General intro
http://www.sable.mcgill.ca/soot/eclipse/index.html
Installation guide
http://www.sable.mcgill.ca/soot/eclipse/updates/
Developing With Soot in Eclipse
http://www.sable.mcgill.ca/soot/soot_in_eclipse_howto.html
Using the plugin with a custom main class
http://www.sable.mcgill.ca/soot/eclipse/different-main.html
Plugin help
http://www.sable.mcgill.ca/soot/eclipse/ca.mcgill.sable.soot/doc/concepts/concepts.html