City University of New York (CUNY)
CUNY Academic Works

Open Educational Resources
Hunter College

2022

Eclipse, OSGi, and the Java Model

Raffi T. Khatchadourian
CUNY Hunter College

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/hc_oers/41
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Programming Languages/Software Engineering: Eclipse, OSGi, and the Java Model

Project 0, Part 2

Questions a and b

```
class Test {
 public static void main(String[] a) {
  int x = 5;
  int y = x + 6;
  System.out.println(y);
 }
}
```

< Application, LTest, main([Ljava/lang/String;)V >
4 v5 = binaryop(add) v3:#5, v4:#6 (line 4)

a. What is this instruction doing?
b. In the original source, computation occurs on both lines 3 and 4. Why in the IR is there only an instruction for line 4?

My solution to questions c and d

Add (static) fields to hold counts

```java
public class Main {

    /**
     * Total number of instructions seen.
     */
    private static int totalInstructions;

    /**
     * Total number of instructions in all methods that are branching statements.
     */
    private static int totalBranchingInstructions;
```

To the end of main(), add output statements

```
public static void main(String[] args) throws Exception {
	// …
	// output the total number of instructions (part c).
	System.out.println("Total instructions: " + totalInstructions);

	// output the total number of instructions (part d).
	System.out.println("Total branching instructions: " + totalBranchingInstructions);
}
```

Modify checkInstruction() to count *all* instructions seen

```
private static void checkInstruction(SSAInstruction instruction) {
        // increment the instruction count.
        ++totalInstructions;
```

Modify checkInstruction() to count *branching* instructions seen

```
private static void checkInstruction(SSAInstruction instruction) {
	// increment the instruction count.
	++totalInstructions;

	// if it's a branching instruction.
	if (instruction instanceof SSAGotoInstruction ||
		instruction instanceof SSAConditionalBranchInstruction ||
		instruction instanceof SSASwitchInstruction) {
		// output it to see what it looks like.
		System.out.println("Found branching instruction of type " +
			instruction.getClass().getName() + ": " + instruction);

		// increment the branching count.
		++totalBranchingInstructions;
	}
}
```

Import the new instruction types

```
import com.ibm.wala.ssa.SSAConditionalBranchInstruction;
import com.ibm.wala.ssa.SSAGotoInstruction;
import com.ibm.wala.ssa.SSASwitchInstruction;
// ...
class Main {
	// ....
}
```

Output Snippet

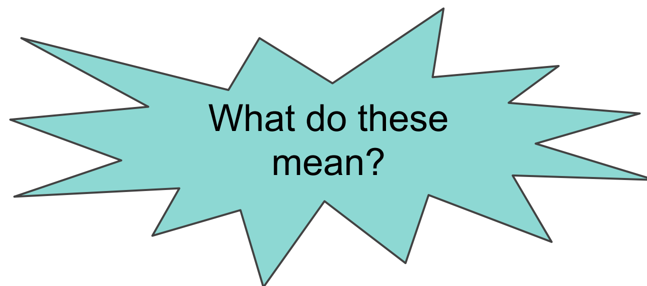
Processing instructions for application method: < Application, LJLex/CLexGen, packCode([C[C[CII)[C >

Found branching instruction of type com.ibm.wala.ssa.**SSAConditionalBranchInstruction**:
    conditional branch(ne, to iindex=63) 9,53

Found branching instruction of type com.ibm.wala.ssa.**SSASwitchInstruction**:
    switch 6 [0->102,1->107,2->112,3->122,4->127,5->132,6->117]

Found branching instruction of type com.ibm.wala.ssa.**SSAGotoInstruction**:
    goto (from iindex= 106 to iindex = 142)

...
Total instructions: **6535**
Total branching instructions: **1089**



A little bit about the Eclipse's Plugin Architecture

- The Eclipse architecture is an interesting case study in itself.
- Follows the conventions of the OSGi component architecture.
- Very dynamic!
  - Plugins are loaded at boot time.
  - Heavy use of *reflection* (actually an interesting research field for static analysis).
- Basically, *everything* is a plugin, even Eclipse core functionality.
  - They are “core” plugins.
  - Eclipse without any plugins is a mere shell.

Eclipse Plugin Extension Points and Extensions

- Plugins are configured through metadata:
  - plugin.xml
    - Extensions, extension points, etc.
  - MANIFEST.mf
    - Runtime stuff (classpath, i.e., where to find libraries), dependencies, version ranges.
- Plugins can have *extension points* and can *extend* other plugins.
- An extension point allows other plugins to extend functionality provided by the plugin.
  - For example, add a menu item to a context menu (right-click).
- A plugin extends the functionality of another plugin by *extending* its extension point.
- Many plugins are both *extensions* of other plugins and also have *extension points*.

Analysis and Transformation in Eclipse

Python?

- WALA is an extensive library of static analysis algorithm implementations.
  - We'll learn about these algorithms throughout the semester.
  - WALA can analyze a variety of software:
    - Android.
    - Javascript.
    - Etc.
  - WALA doesn't have great *transformation* capabilities.
    - It's mainly for analysis.
    - There is another project called Soot that is more ept at *bytecode* transforma
      WALA. We won't Soot in this course, though.
      - WALA used to be a little bit easier to work with within Eclipse since it originally has an OSGi-type architecture.
      - Probably something to do with both originating from IBM.
      - Since a major transition, my understanding is that OSGi metadata was removed.
- Eclipse supports source-to-source transformation on ASTs.

Scalpel?

PyDev?

Analysis and Transformation in Eclipse

- We’ve mainly looked at ASTs, but Eclipse really has *two* models for any given programming language.
- For Java, this is called the *Java Model*.
- It’s a domain model for everything up to Workspaces and Projects down to fields and methods.
- For example, an IProject is an interface representing *any* project in a workspace.
  - An IJavaProject is specific for Java.
- The JavaCore.create() is useful for obtaining IJavaElements (objects in the Java Model) without the UI (e.g., right-clicking on an object).
- The Java Model doesn’t allow transformations at the source level.
- However, there are useful analyses that can be generated, e.g., search engines, type hierarchies, and call graphs (more later on these).

JDT View

- The AST View plugin in Eclipse is useful for visualizing an AST.
- Likewise, the JDT View plugin is useful for visualizing the Java Model.
- These two models work hand-in-hand.
    - For example, you can build an AST node (instance of an CompilationUnit) from a file in the Java Model (instance of an ICompilationUnit).

Assignment 1: This week’s homework

Part 1: Use the Visitor Pattern for Project 0, Part 2

- Checking the types of each instruction can be cumbersome.
  - How do you know which types are available?
- Instructions in WALA support the visitor pattern!
  - But not as fully as we saw last week.
- Use [com.ibm.wala.ssa.IR.visitNormalInstructions(Visitor)]() in the main() method!
  - We don't need an array of all instructions (except to get the count of *all* instructions):
    ```
    // the instructions in the IR.
    SSAInstruction[] instructions = ir.getInstructions();
    ```
  - Instead, give the IR a *visitor* (for the *branching* instruction count).
- Subclass [com.ibm.wala.ssa.SSAInstruction.Visitor]().
  - "Default" implementations of all methods do nothing.
  - Only override the `visit()` methods for types you are interested in.
  - Add your own functionality.

Assignment1, Part 2: Learn how to work with ASTs

- Goals:
  - Appreciate the difference between ASTs and the instruction IR.
    - WALA generates instruction IR, Eclipse generates AST IR.
  - Learn how to build Eclipse plugins.
  - Learn how to discover, build, and analyze open-source projects.
  - Run your plugin on a project you found.
  - Collect various statistical data
- Sample output and resources available