BITS ZG629T : Dissertation

A Java Bytecode De-compilation Plug-In for Platform Independent Core-file Analyzers

Carried out at IBM India Software Lab

Pushkar N Kulkarni
2009HZ13463

Outline of the talk

- *Introduction*
  Background
  Objectives

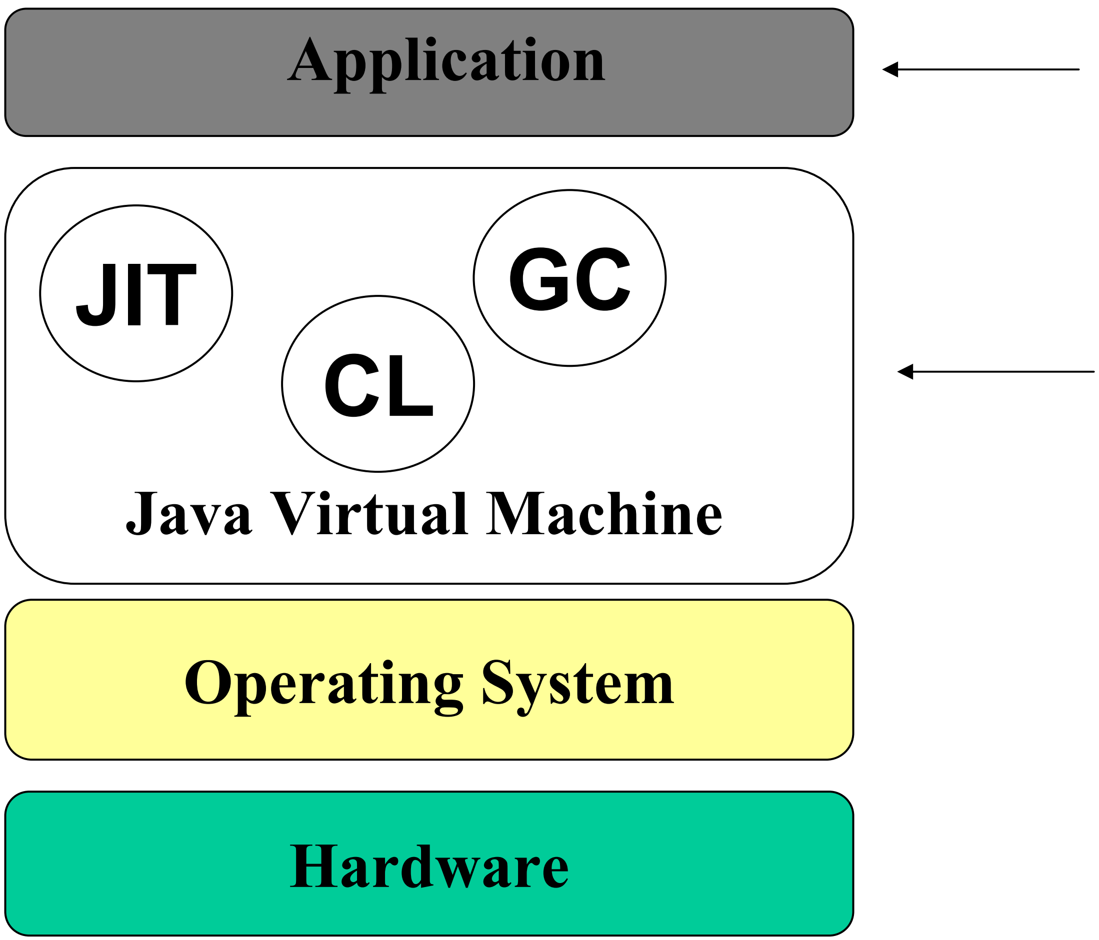
- *Discussion*
  Requirements
  Architecture and Design considerations

- *Conclusion*
  Examples
  Limitations & Future Work

Introduction - Background

- Java gained much fame from *platform independence*
- Platform independence is imparted by an abstract machine called the *Java Virtual Machine (JVM)*
- The JVM is another process running on your operating system
- Java applications run on top of (or inside) the JVM

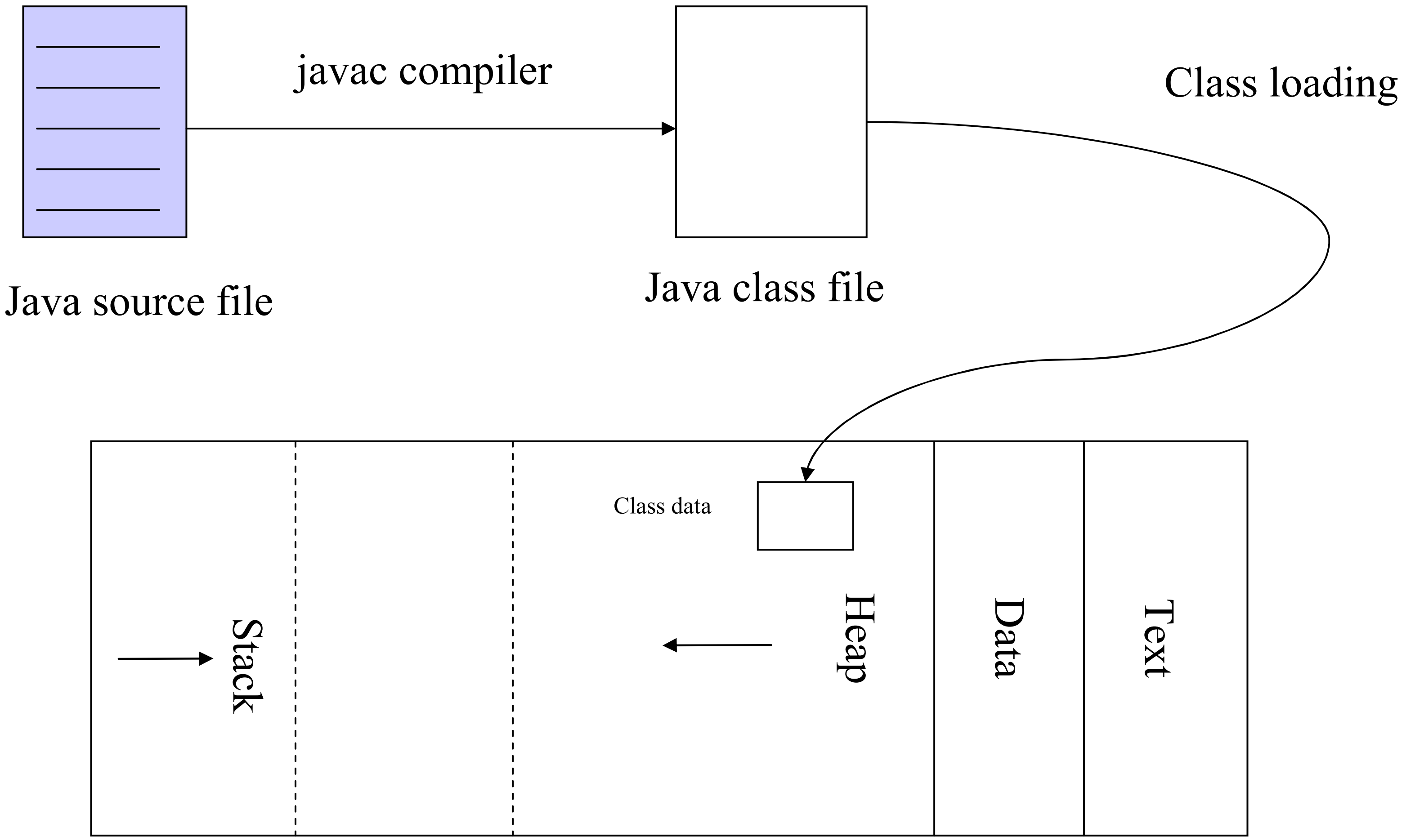
Introduction - Background



All the Java classes!

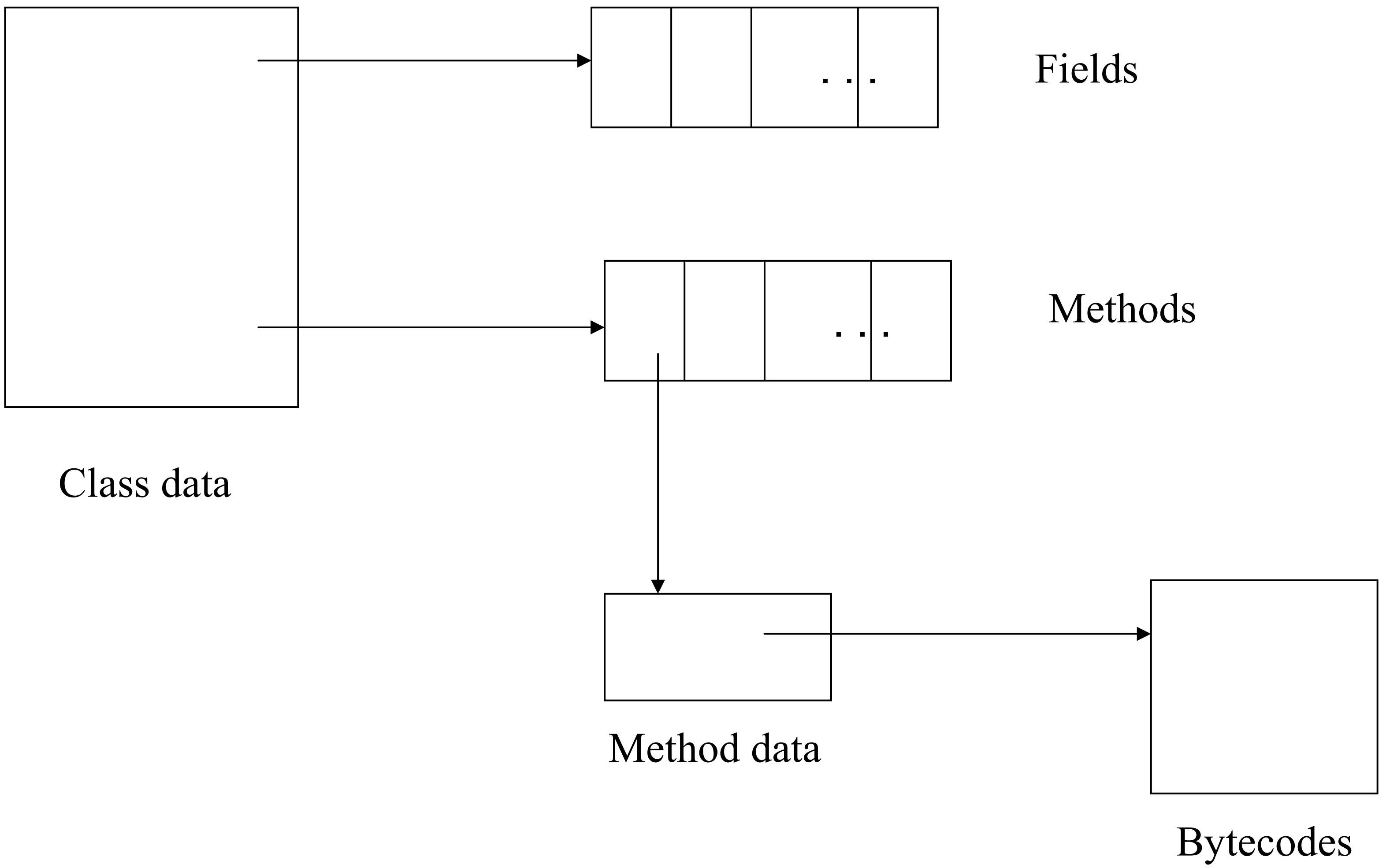
The JVM process.
GC – Garbage Collector
CL – Class-loader
JIT – Just-In-Time Compiler

Introduction - Background



Process layout (consider JVM)

Introduction - Background



Introduction - Background

- All your code written in Java is loaded onto the heap of the JVM process, as shown on the previous slide
- Bytecodes – JVM Instructions. Just like an x86 machine defines x86 instructions!
- Bytecodes are interpreted by the JVM
- JIT Compilation

Introduction - Background

- Sometimes, the JVM process may crash while it was executing certain Java methods
- Crashes & hangs can come up due to any reason – in the JVM, in the application or in the underlying OS
- However, if a thread has seen a failure and it was executing Java code, then we start with understanding what was being executed

Introduction - Background

- Methods of all loaded classes, at failure, are present on the heap and hence in the dumped core file
- Comprehension of Java (application) methods or at least their structures is a very common activity while debugging JVM failure.
- Currently, comprehension happens by reading bytecodes using special core file analyzers. We view bytecodes on a per method basis.

Introduction - Background

These are bytecodes for a method that calculates the area of circle!

```
+0      JBldc2dw               #15   // double 3.14
+3      JBdstore3
+4      JBdload3
+5      JBaload0
+6      JBdload1
+7      JBinvokevirtual        #8    // Test.getSquare(D)D
+10     JBdmul
+11     JBdstore               5
+13     JBdload                5
+15     JBreturn2
```

Reading bytecodes may be very difficult for large methods.

Introduction - Objectives

- *De-compile the method bytecodes, available in dumps, to Java code and present them to the reader, instead of bytecodes!*

```
public double areaOfACircle ( double radius ) {
     double PI = 3.14;
     double area = PI * getSquare ( radius );
     return area;
}
```

Introduction - Objectives

- *Make the de-compilation back-end independent from the dump-reading front end.*

  IBM has many platform-independent dump/core-file analysis tools for JVM dumps. All these tools should be able to use the de-compilation back end.

- *Hence improve method comprehension and problem determination times!*

Discussion - Requirements

- *A method to locate bytecodes for a given method, in a dump, and extract them – A Dump Reader*
  Other information like exception handling, local variables and access specifications should also be extracted.

- *A method to translate the extracted bytecodes to Java code – A Translator*
  This is what we call de-compiling.

Discussion - Requirements

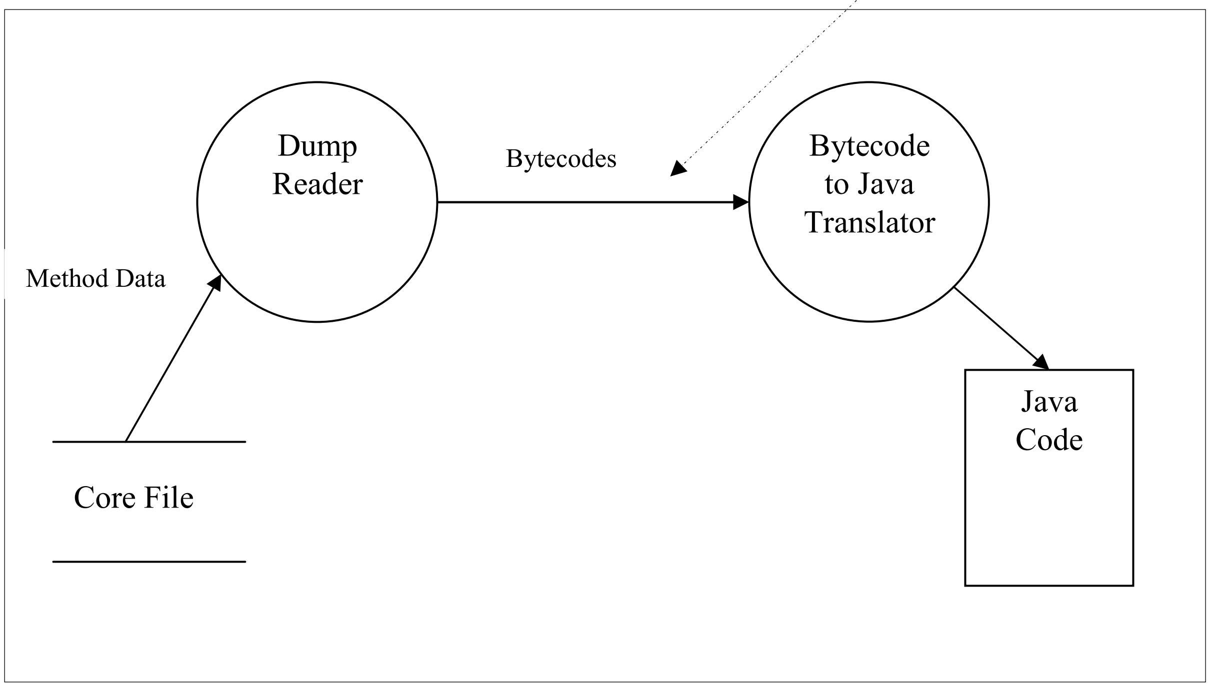
- *Segregation of the dump reader from the translator*
  This will impart immense re-use potential to the bytecode de-compiler.

- *The translator should be able to recognize and convert to Java all constructs defined by the language as in Java 6.0*
  This includes Expressions, Assignments, Invocations, Conditionals, Loops, Try-Catch blocks, Switch statements and Synchronized blocks

Discussion – Architecture & Design Considerations

Level 0 Data Flow Diagram

Achieve separation here



Discussion – Architecture & Design Considerations

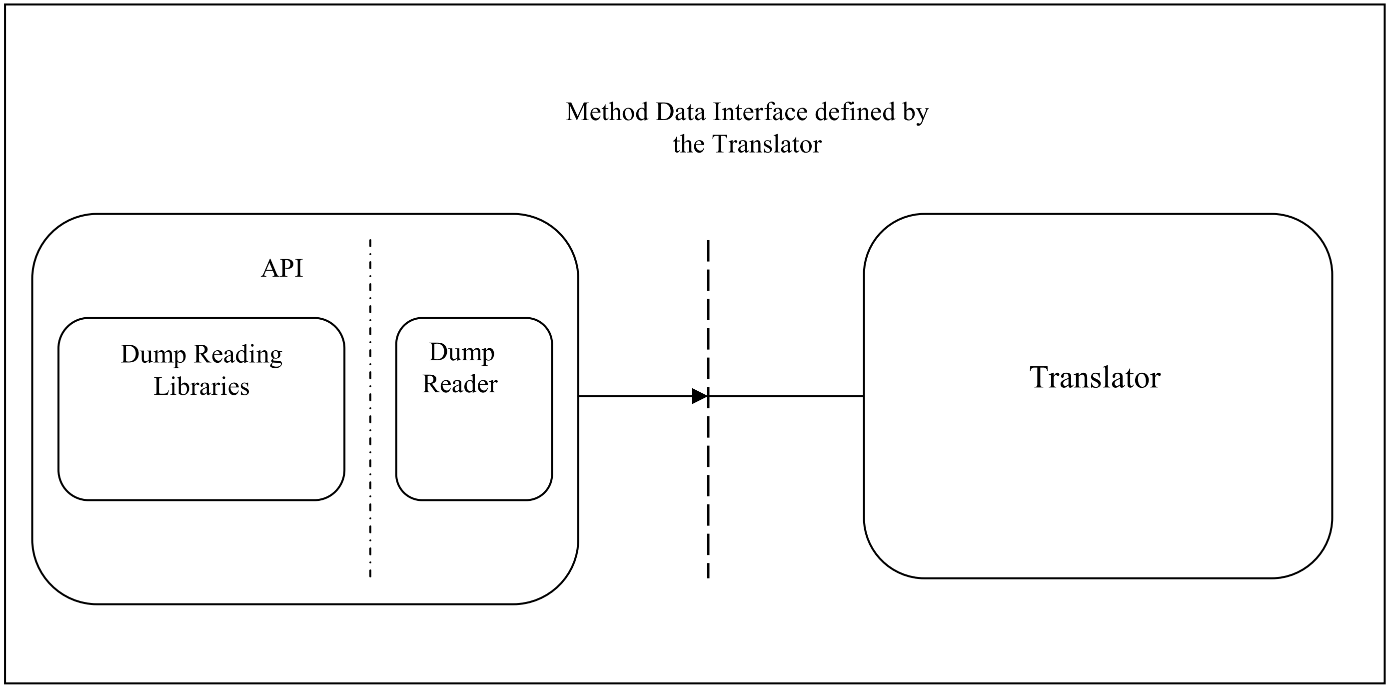
- How to segregate ?

  Define an interface that provides methods to pump data into the translator.

  The translator implements this interface and the dump-reader uses it.

  Any other dump reader, would only need to be passed the interface object.

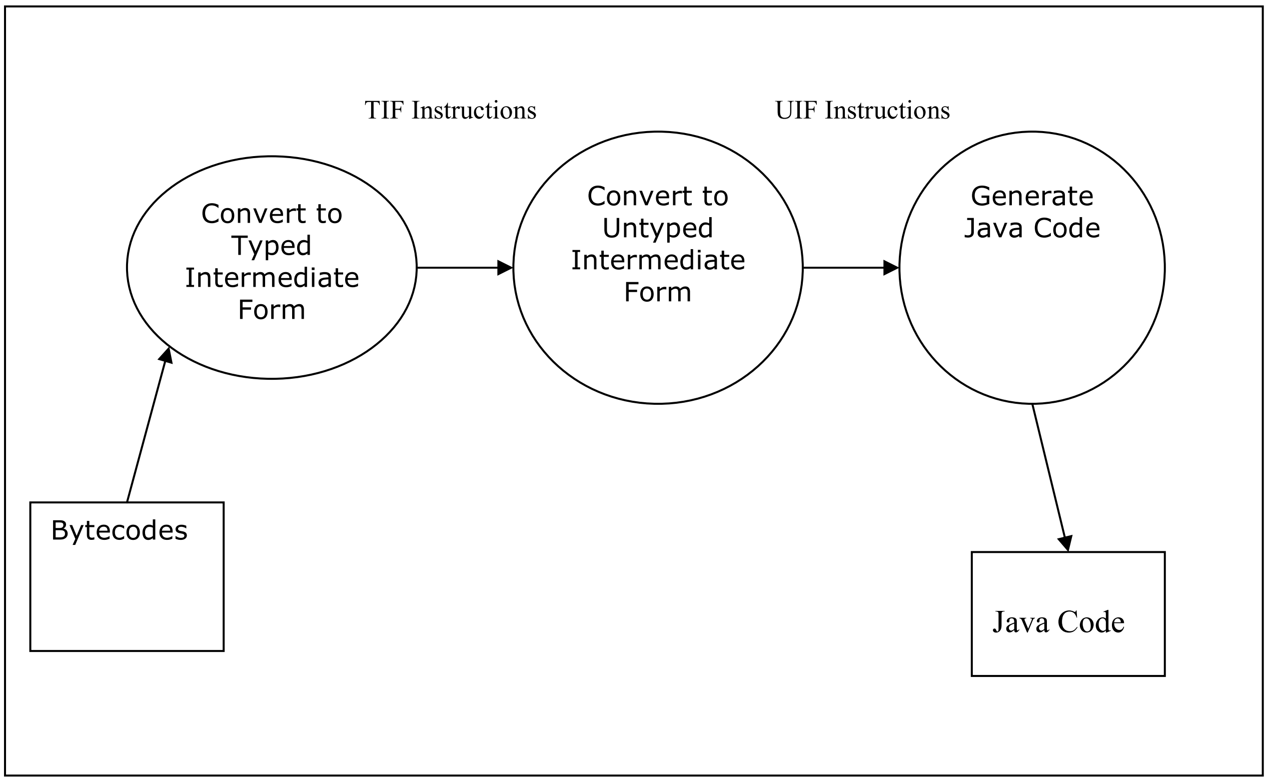
Discussion – Architecture & Design Considerations



System diagram

Discussion – Architecture & Design Considerations

Designing the Translator

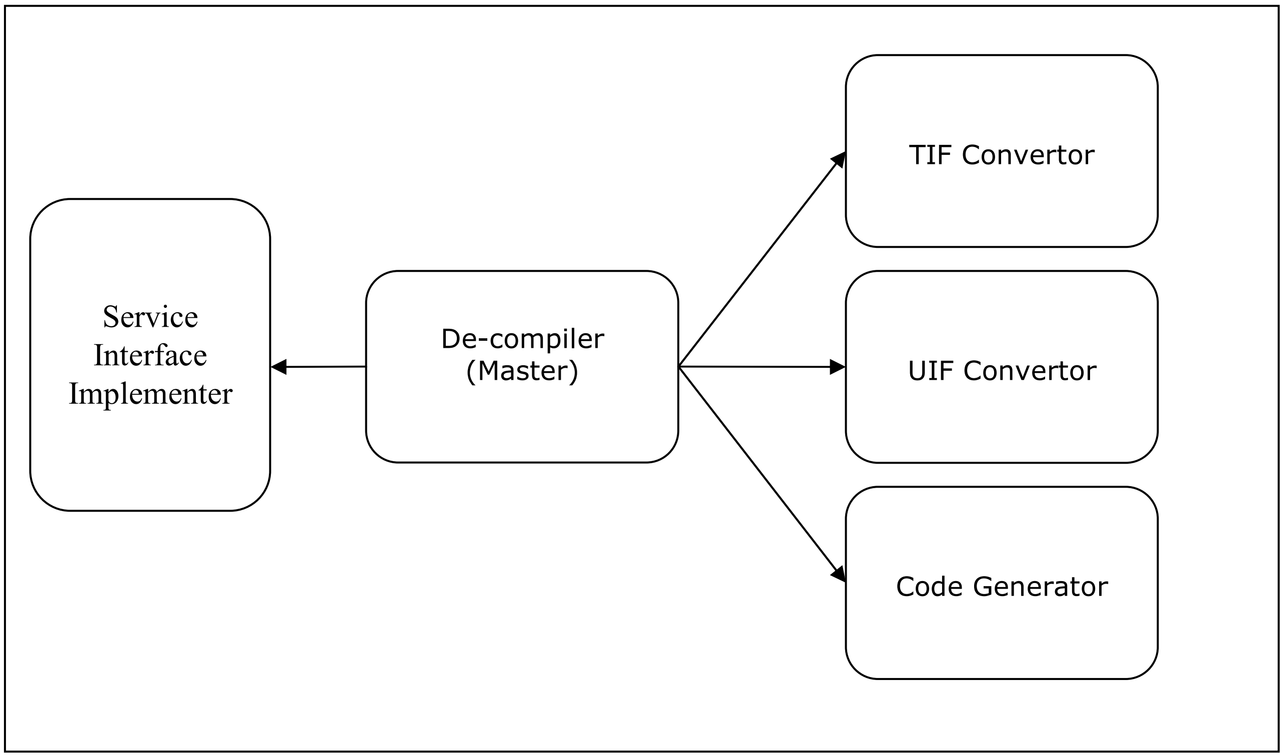


TIF – Typed-Intermediate Form

UIF – Untyped-Intermediate Form

Discussion – Architecture & Design Considerations

With the previous DFD in mind, we define a few components for the Translator.



Discussion – Architecture & Design Considerations

- We now describe the three important components:

  1. TIF Convertor

     Converts bytecodes to Typed-Intermediate-Form

  2. UIF Convertor

     Converts TIF to Untyped-Intermediate-Form

  3. Code generator

     Creates a parse tree from the UIF

Discussion – Architecture & Design Considerations

TIF Convertor

- Bytecodes are instructions for a stack-based machine.

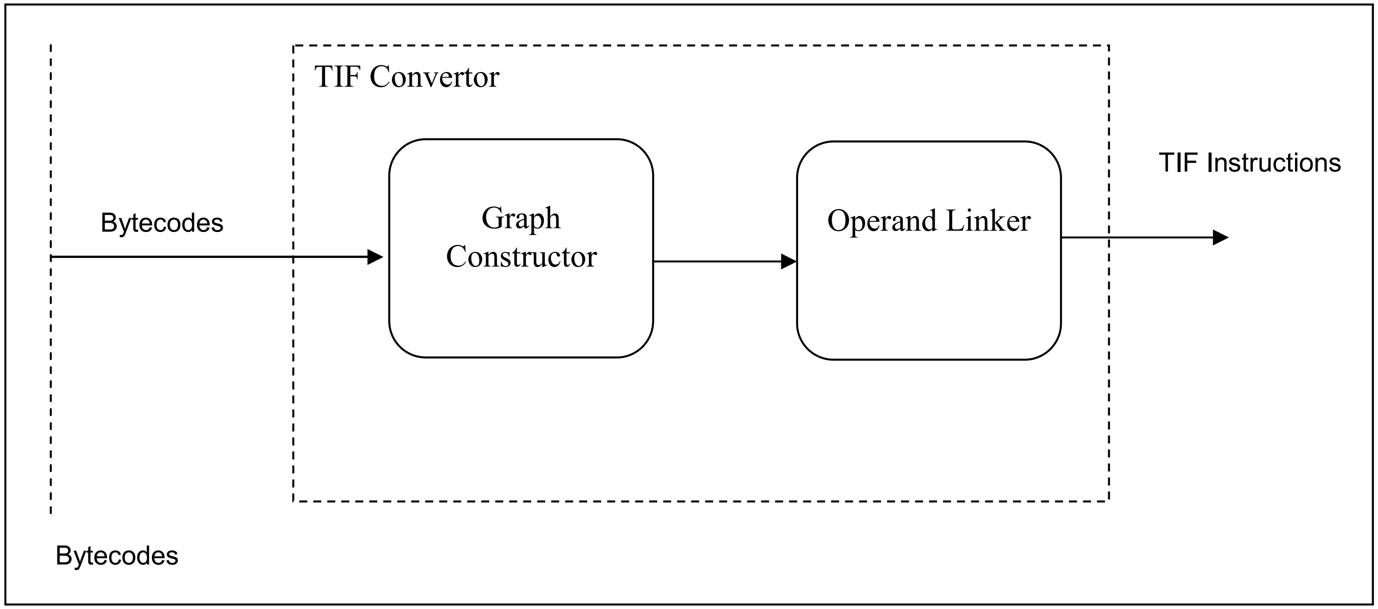
  The TIF Convertor reads bytecodes and maps all the stack-operands to either local variables, or to constants, or to the results of other TIFs.

- Bytecodes have type-information and this is maintained in this form.

- The result is a fully-linked TIF graph, with no stack-operands.

Discussion – Architecture & Design Considerations

TIF Convertor



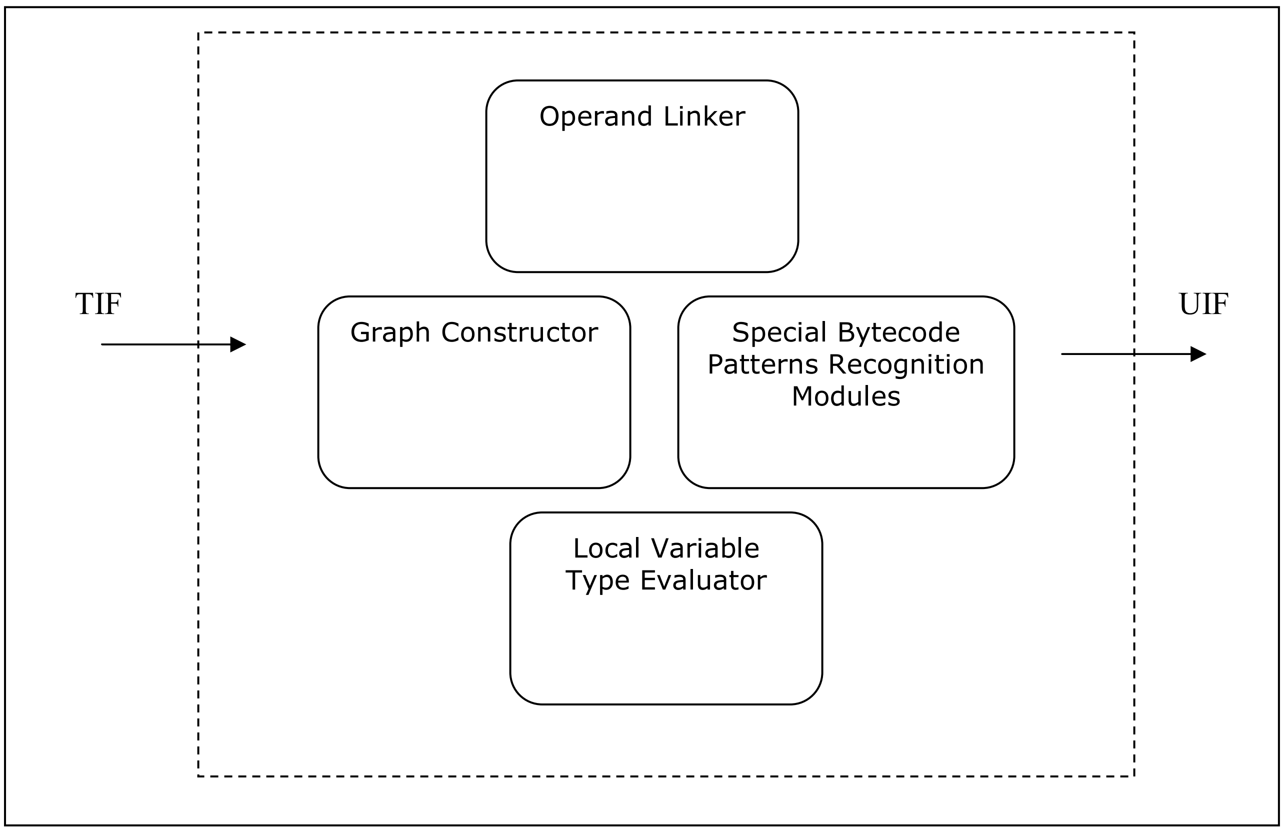
Discussion – Architecture & Design Considerations

UIF Convertor

- The Typed-Intermediate-Form instructions are them reduced to the Untyped-Instruction-Form.
- The type-information is shed, after deducing the types of local variables
- New operand linkages are established
- Special bytecode patterns are searched for and decompiled – eg. the ternary operator

Discussion – Architecture & Design Considerations

UIF Convertor



Discussion – Architecture & Design Considerations

Code Generator

- The Code Generator, fits the UIF Instructions into a parse-tree.

- The parse-tree uses a customized Java grammar which is semantically equivalent to the actual grammar.

- A hierarchy of Java-grammatical elements has been defined. The Code Generator uses this hierarchy.

Discussion – Architecture & Design Considerations

Code Generator

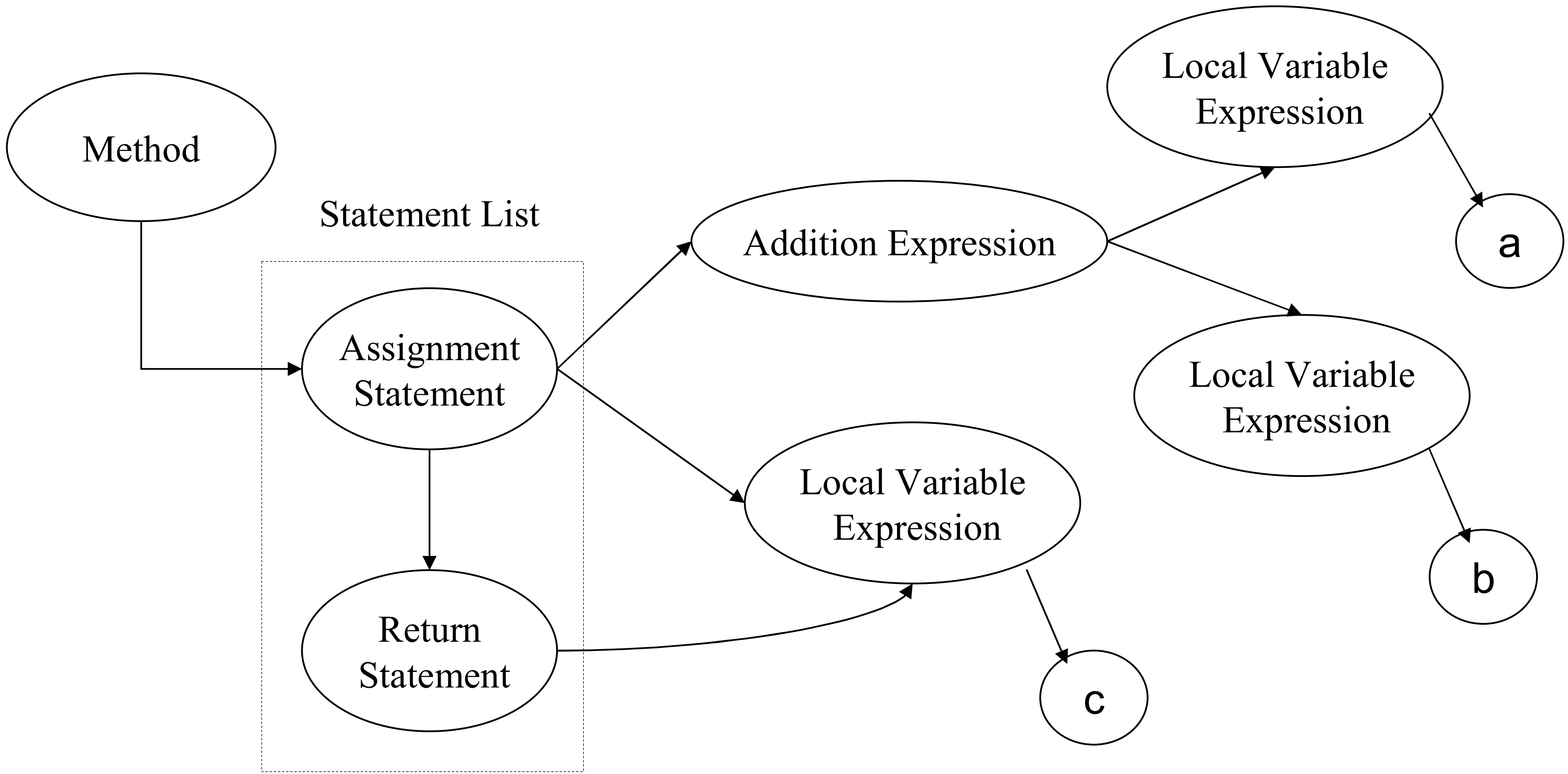
- After the parse-graph is built, a depth-first walk of this tree will generate the Java code.

Example: If we had the method,

```
int sum ( int a, int b )
{
  int c = a + b;
  return c;
}
```

… this could be generated from a depth-first walk of the tree given on the next page.

Parse-Tree Example



Example 1 – Simple Method

Java method:

```
public double areaOfACircle ( double radius )
{
        double PI = 3.14;
        double area = PI * getSquare ( radius );
        return area;
}
```

Output of the de-compiler

```
public double areaOfACircle ( double dd )
{
        double de = 3.14;
        double df = de * getSquare ( dd );
        return df;
}
```

Example 2 – IF statement

Java Method:

```
public void testIFELSE ( int a, int b )
{
        if ( a > b )
            a = b;
        else if ( a < b )
            b = a;
        else
            a = b = a+b;
}
```

Example 2 – IF statement

Output of the de-compiler:

```
public void testIFELSE ( int ia, int ib )
{
        if ( ia > ib )
            ia = ib;
        else if ( ia < ib )
            ib = ia;
        else
        {
            ia = ia + ib;
            ib = ia + ib;
        }
}
```

Limitations & Future Work

The current limitations which could be corrected in future are:

- ***Recognition of post/pre increment and decrement operations.***

Currently, t = a[++i] will be written as
i = i + 1;
t = a [i];

These, however, are semantically equivalent.

Limitations & Future Work

- ***Distinguishing the while-loop from the for-loop***
  This is not a limitation per se.

- ***Detecting the complement operator ~***

- ***Booleans true and false are internally represented as 1 and 0 respectively.***
  Detecting when 1 and 0 should be converted to true and false is still not implemented.

Limitations & Future Work

- ***Generated local variable names make no sense.***
  For example, if there exists:
  int ik = getAge ( )
  Here **ik** is a randomly generated variable. A closer look will reveal that this variable could be renamed as **age**

- Java 7 added some new syntax to the Java language. The grammar need to embrace these additions.

Limitations & Future Work

Real limitations:

- The **local variable table** is not present in dumps for most of the times. The random names generated for local variables may not make sense and act as blockades in method comprehension.
- **Static variables** which are also **final** are merely represented as constants internally. There is no way to retrieve the names of these class variables.

References

Inside the Java 2 VM – Bill Venners
http://www.artima.com/insidejvm/ed2/

The Java VM Spec
http://java.sun.com/docs/books/jvms/second_edition/htVMSpecTOC.doc.html

The Java Grammar
http://java.sun.com/docs/books/jls/first_edition/html/19.doc.html

Thank you!