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

BITS ZG629T: Dissertation

By

KULKARNI PUSHKAR
(2009HZ13463)

Broad Academic Area of Work comes under

Compiler Construction

Dissertation work carried out at

IBM India Software Lab

Submitted in partial fulfillment of M.S. Software Systems degree program

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE
PILANI (RAJASTHAN)

October 2011

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

BITS ZG629T: Dissertation

By

KULKARNI PUSHKAR
(2009HZ13463)

Broad Academic Area of Work comes under

Compiler Construction

Dissertation work carried out at

IBM India Software Lab

Submitted in partial fulfillment of M.S. Software Systems degree program

Under the Supervision of

GIREESH PUNATHIL
Advisory Software Engineer,
IBM India Software Lab

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE
PILANI (RAJASTHAN)

October 2011

CERTIFICATE

This is to certify that the Dissertation entitled *A Java Bytecode De-compilation Plug-in for Platform Independent Core-file Analyzers* and submitted by *Kulkarni Pushkar* having ID-No. *2009HZ13463* for the partial fulfillment of the requirements of M.S. Software Systems degree of BITS Pilani, embodies the bona fide work done by him/her under our supervision and Guidance.

GIREESH PUNATHIL
Advisory Software Engineer
IBM India Software Lab
Bangalore

SHIRISH KUNCOLIENKAR
Advisory Software Engineer
IBM India Software Lab
Bangalore

Place: Bangalore
Date:

ACKNOWLEDGEMENT

This piece of work has been the result of an effort of four years under the excellent guidance of my team-leader, **Gireesh Punathil**. Gireesh has been responsible for inculcating the art of reverse engineering in me, something that is very much essential in the maintenance of middle-ware like the Java Virtual Machine. He has been very patient in listening to each doubt rising in my mind and providing explanations with an unchanged enthusiasm.

Shirish Kuncolienkar has always helped me put together fragmented views of understanding to get the complete picture. His approach of looking at things taught me to think from head to toe. The ideas that we have discussed together have helped me in widening my scope of thinking. His understanding of the Java Class Library has always been a source of motivation to me.

I would like to thank Gireesh and Shirish whole-heartedly, because without them I would not even be able to conceptualize this piece of work. I also wish to express my gratitude to **Ajay Kumar Sami** and **Sadananda Aithal** who have spent a lot of time with me, helping me to get along with Eclipse and the dump-reader API used here. Lastly, I wish to thank the **Late Prasanna Kumar Kalle** for inducting me into the world of the Java Virtual Machine and the Just-In-Time Compiler. He demonstrated that core dumps are information treasures and that any problem could be understood by studying them.

Kulkarni Pushkar

ABSTRACT

IBM ships its own Java Virtual Machine Environment with all its products built on top of Java. This has put Java to an extensive amount of use and has been challenging JVM engineers with various kinds of failures. Failures in the Java Virtual Machine may sometimes manifest as failures in the Java application sitting on top. At such instances, service engineers prefer to understand the application method which apparently saw the failure. Problem determination usually begins with the comprehension of the structure of this method. This is more pronounced in the case of the JIT compiler which compiles methods on a per-method basis. Any failure in the JIT compiler often demands the understanding of the bytecode sequence that was being compiled. Till date, we have been comprehending method structures by studying and understanding bytecodes. This process is tedious and slows down the pace of debugging. If the structure of the method very readily available, it could expedite the process of problem determination and resolution. This idea of a de-compilation plug-in for a core file analyzer tool, presents a solution for the problem mentioned above. Bytecodes of a method may be read using the API defined by the core file analysis tool. These bytecodes and other method information can then be processed to generate Java code. In this process, we shift from a stack-based nature of the bytecodes to Java code. This shift is achieved by a three-step process. The first two processes define their own Intermediate Language and work with it. The first of the two is closer to bytecodes and the second is closer to Java code. Finally the third phase involves the construction of a parse tree. A depth-first walk of this parse tree will reveal the Java code of this method.

Keywords: Java**,** Java Virtual Machine, Bytecodes, Core file, Core file Analyzer, de-compilation, de-compiler

Table of Contents

Table of Figures

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

1. Introduction

1.1 Background

Java has gained fame in the past decade mostly because of the feature of platform independence. Java code, written and compiled on one platform, can be executed on other platforms. The abstract machine, well-known as the **Java Virtual Machine** imparts this platform independence to Java code. Just like a physical computing machine, the JVM also defines its own instruction set. The Java compiler translates Java code into JVM Instructions, also known as **Bytecodes**. A Java file is the unit of compilation. Upon compilation, a binary file called **class file** is generated. Inside the class file, the Java methods can be found in the form of bytecodes. The class file also encapsulates a lot of other information like the local variables, class fields, **class hierarchy** information and exception information.

A **class**, in simple words, represents a data-type at runtime. As and when different types are encountered by the execution engine, we resort to a lazy binding strategy of that type through an activity called **class loading**. Class loading is the act of resolving an encountered data-type and involves the loading the all data in the respective class file on the disk, into the JVM process memory, in appropriate structures. The Java Virtual Machine beneath a running Java application will usually have hundreds to thousands of classes loaded in its memory.

The code within a class, which is divided into units called **methods**, also gets loaded in the process of class loading and is represented in bytecodes. Application execution primarily begins with bytecode interpretation through an **interpreter**. As apparent, the interpreter is a layer between the application code and the underlying machine (the operating system & the hardware). It is the interpreter that helps achieve the platform independence. However, this layer also adds to the response

time bringing down the performance numbers of the Java application. To counter this bitter fact, computer scientists have invented the concept of **Just-In-Time compilation**, or **Dynamic Compilation**. Dynamic compilation involves the identification of methods which are being extensively utilized and translating to native machine instructions, subsequently peeling off the interpreter layer for these methods. We do not compromise on platform-independence with this kind of runtime compilation.

The Just-In-Time compiler acts on method bytecodes and converts them into machine instructions pertinent to the underlying platform. Here, a method is the unit of compilation. It involves fair to huge amounts of code optimization. The optimization algorithms work on an infinite number of bytecode patterns. It is seen that in some cases, these algorithms may produce results that end up in wrong machine code being generated. For example, a buggy optimization may end up reading an array past its end and cause a segmentation violation. The simple reason is that the resulting machine code is not "totally equivalent" to the existing bytecodes.

1.2 Need for the tool

It is often very difficult to **recreate** failures experienced by customers. The failure is hugely dependent on the application environment, which is difficult to be simulated. Problem determination hence begins with system **core file** analysis. Core files are created when a process goes down following the receipt of a signal from the kernel. Segmentation fault and illegal branch are two examples.

During the problem determination of these compiler issues, it is very essential to comprehend the structure of the method. Currently, the structure is comprehended by understanding the bytecode listing. The **platform-independent core file analysis tools** for the IBM JVM provide an interface to list the bytecodes of a given Java method.

1.3 Objectives

This project plans to improve the process of method structure comprehension by translating the bytecodes, to the best possible extent, to Java code. This reverse engineering of bytecodes is expected to improve the problem determination time for a wide range of customer problems including the Just-In-Time compiler problems.

The following could be stated as the prime objectives:

- Develop an efficient bytecode de-compiler where a method is the unit of de-compilation.
- Make the de-compilation back-end fully independent from the core file reading frontend. This should make the backend compatible with any front end core file reader API.
- Improve the problem determination times by improving the comprehension of Java methods available in system dumps in the form of bytecodes, by translating the latter to Java code. This is much required while troubleshooting Just-In-Time compiler problems.

2. Discussion

2.1 Requirements

The following can be listed as the fundamental requirements from this project:

1. A method to locate and list the bytecodes of a Java method along with other information such as **access specifications**, **exceptions** and **local variable** information, in a system core dump.
2. A method to convert the bytecode representation of a Java method to Java code.
3. A facility to spatially segregate the activities of information retrieval from the dump, as in (1), from the activity of translation to Java code, as in (2). This will impart an immense re-use potential to the translation method. This is mentioned in wake of the fact that there exists a wide array of core-file readers for JVM dumps. Segregation will separate the reader from the translator.
4. The translator is expected to recognize bytecode patters pertinent to all constructs used until Java 6.0. These constructs include:
    i. Expressions
    ii. Assignments
    iii. Method Invocations
    iv. Conditionals
    v. Loops
    vi. Try-Catch Blocks & Exception Throws
    vii. Synchronized Blocks
    viii. Switch Statements

Subsequent to the recognition, these bytecode patterns must be translated into the corresponding Java code.

2.2 Architectural Considerations

Requirement (3) talks about the importance of the segregation of the dump reading activity from the translation activity. A translator, once written, will not change and can be deployed in collaboration with different core dump readers. Considering this, we can present a very basic sketch of the architecture of our tool. This is the level 0 Data Flow Diagram.

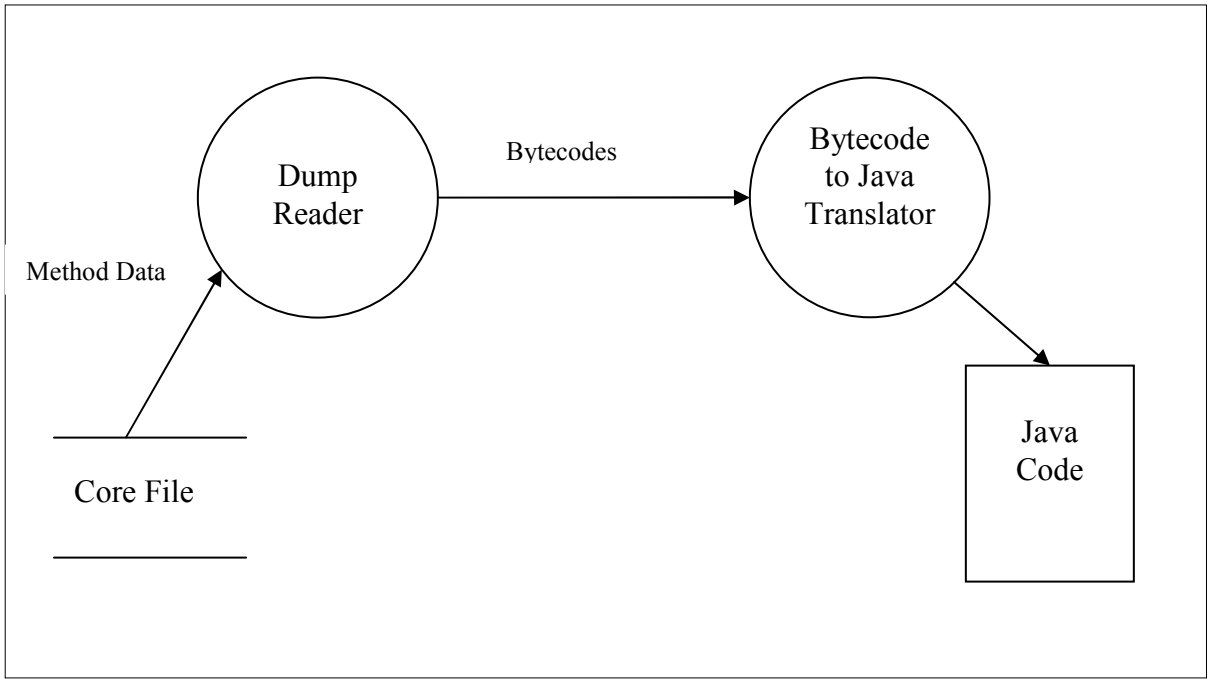

Figure 1: Level 0 Data Flow Diagram

A little more thought on the segregation front makes us realize that this modularity can be achieved only if the translator defines an interface or a set of services required for reading method data. These services are provided by the translator and used by the dump reader. With this thought

and with the above data flow diagram, we now present the basic system diagram.

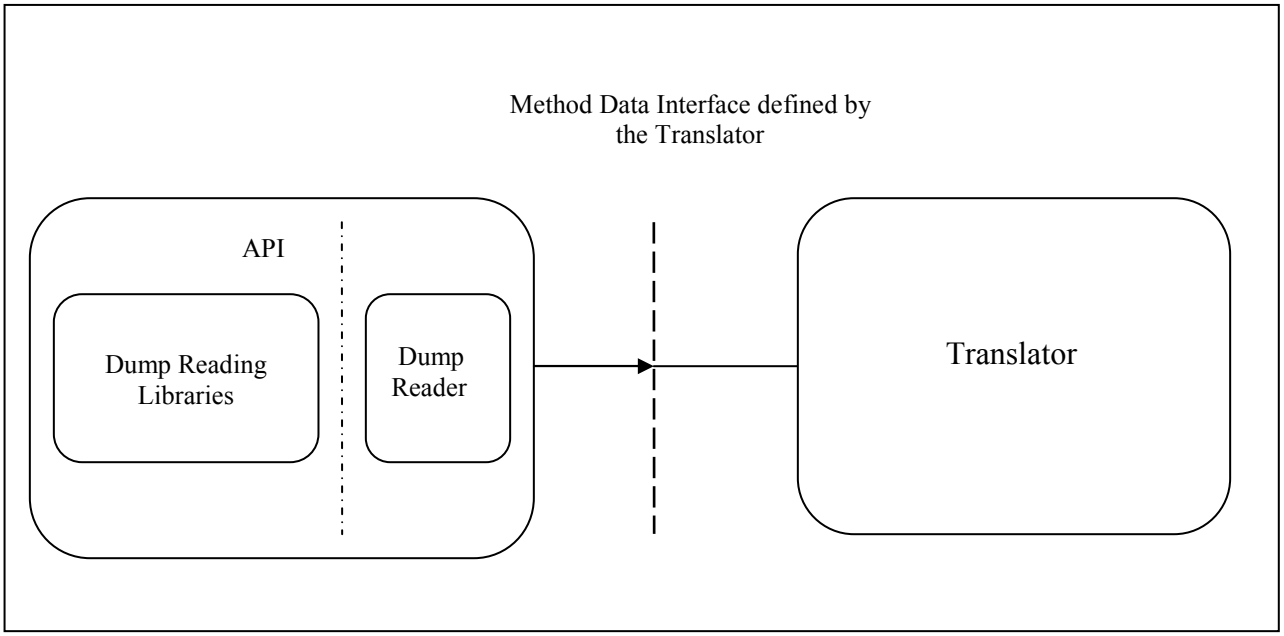

Figure 2: Basic System Diagram

The Service Interface defines all services necessary to facilitate the conversion of bytecode to Java code. Any dump reader that we wish to couple the translator with, should implement the services mentioned in this contract. A few examples of such services are:

i. Get the method name and signature

ii. Get the exception information

iii. Get the bytecode listing

iv. Get the method access specification

When the translator requests for the method data pertaining to a given method, the dump reader pumps this data into the translator. Subsequently the translator begins its job of translating bytecodes to Java code.

2.3 Architecture of the Translator

We now shift our focus to the task of converting bytecodes to Java code. Bytecodes are instructions defined by a stack based machine. Translation hence involves the conversion from a stack-based form of instructions to the Java programming language. This translation is carried out by converting the bytecodes to two types of **Intermediate Representations**. They are:

i. **Typed-Intermediate-Form**

This form represents each bytecode using the bytecode instruction, source operands and destination operands. At the end of this phase, we have linked all source operands to either valid programming objects (like local variable, constant) or to another instruction.

ii. **Untyped-Intermediate-Form**

This form removes the typed nature of the bytecodes. It also eliminates the bytecodes which are involved in stack-load operations only. These instructions are replaced by the actual sources of data. UIF instructions define a list of sources. We gradually move closer to the Java language, by shedding off the stack-based nature of the instructions.

The most important component of the Translator is the Code Generator which uses the Untyped-Intermediate-Form and creates a graph of Java grammar objects. Grammar objects represent grammatical units from the well-defined Java grammar. This graph, after construction, holds the structure of the Java method. A depth-first traversal of this graph can generate the entire Java method. With this background, we draw the Level 1 Data Flow Diagram for the Translator along with its System Diagram.

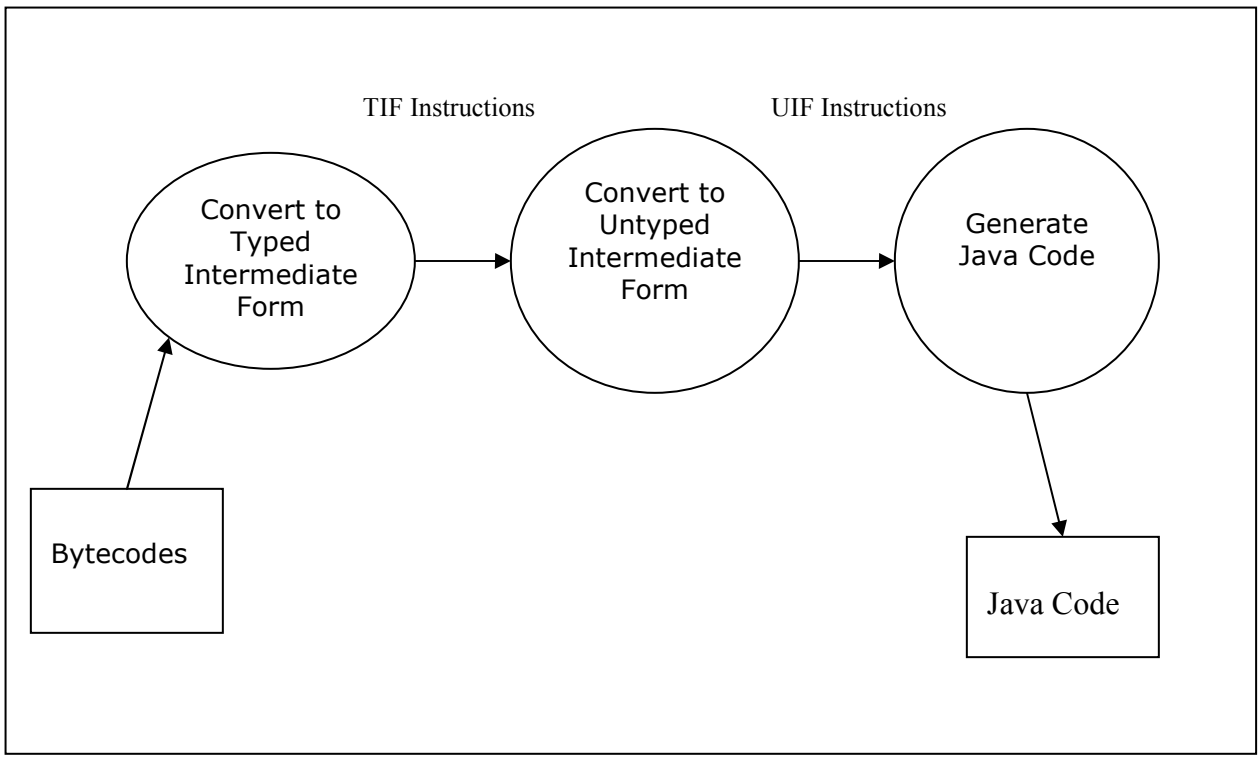

Figure 3: Level 1 Data Flow Diagram of the Translator Subsystem

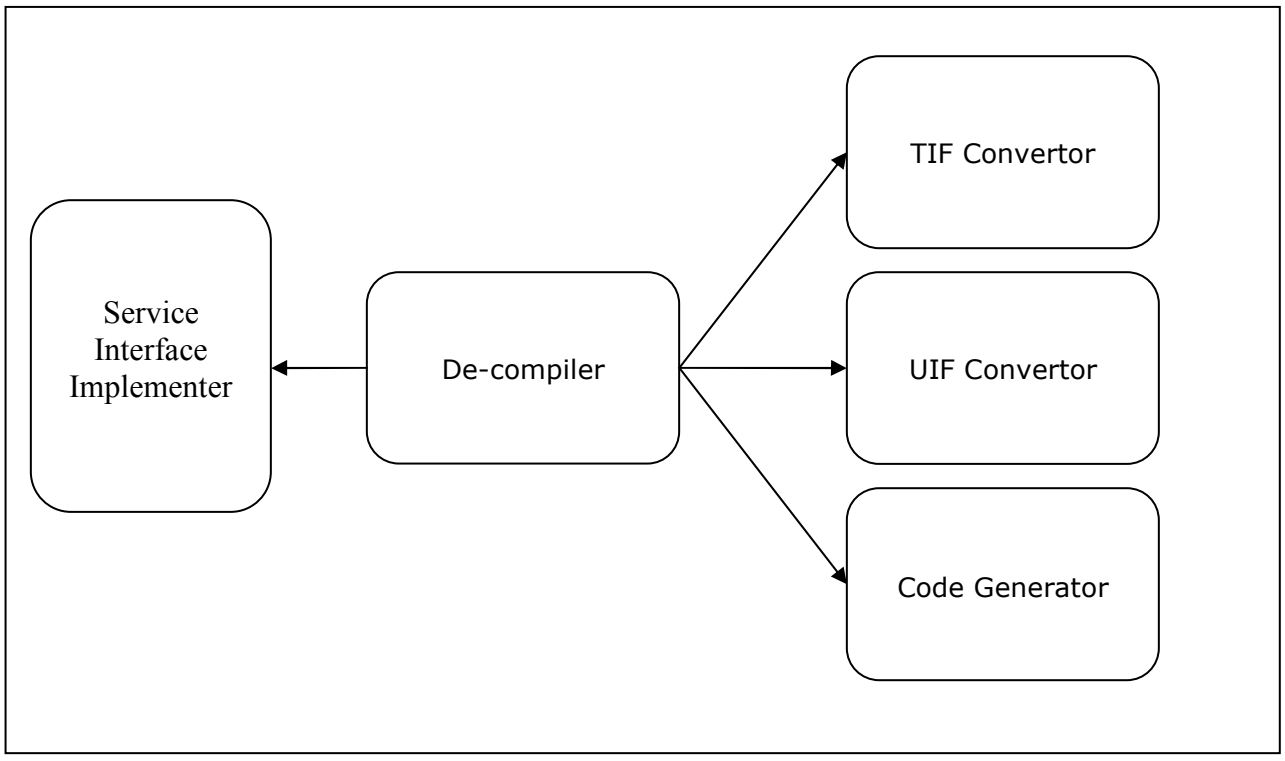

Figure 4: System Diagram of the Translator

2.4 Component Design

We now discuss in brief the design of each of the components mentioned in the overall system diagram and the one for the translator. At the end of this section, the reader should be able to comprehend how the components work together to pick up a bytecodes of a desired application method and translate them to the Java language.

Dump Reader

The Dump Reader is responsible for reading the contents of a Java Virtual Machine Core Dump or Core file. The De-compiler component mentioned in Figure 4 invokes the Dump Reader. The Dump Reader has access to the Method Data Interface. It utilizes this interface to pump method data for a given method into the translator. This component may use any one of the available dump-reading libraries written for JVM dumps.

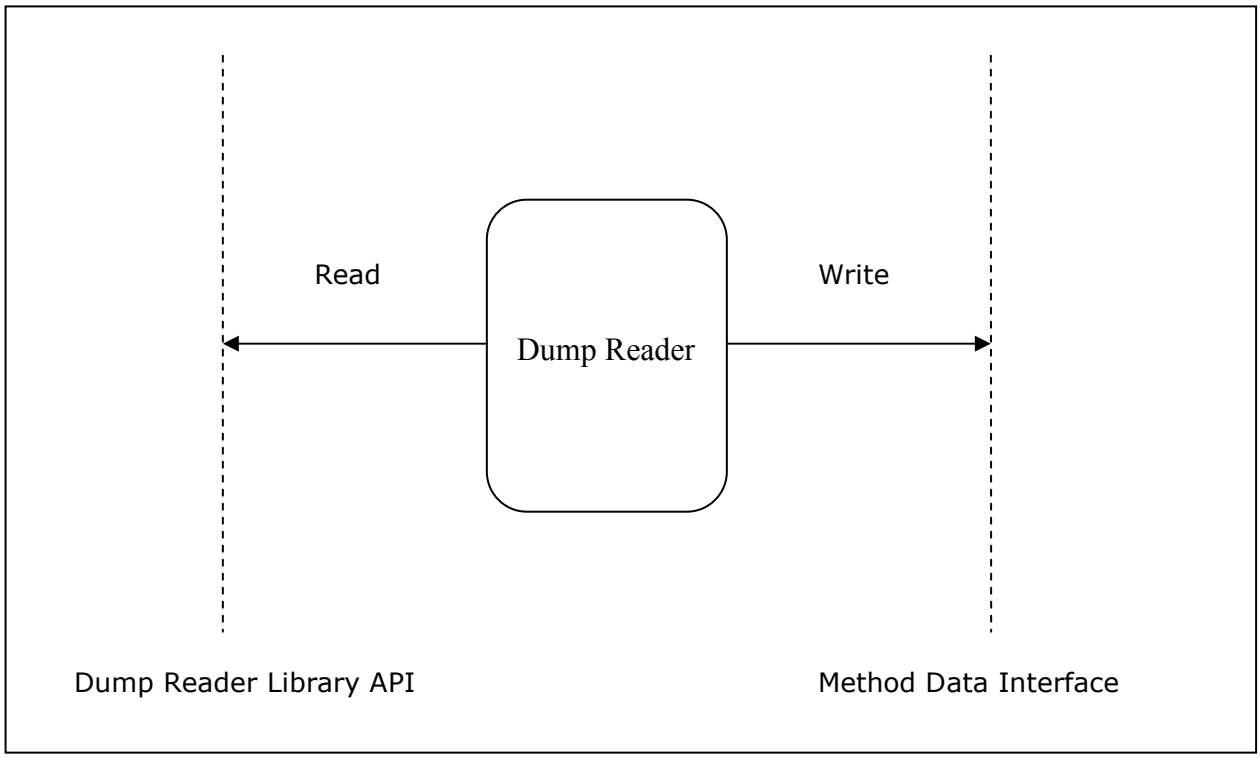

Figure 5 Dump Reader

De-compiler

Unlike what the name may suggest, this component is the master component which invokes the other components in the desired manner. It does the following activities, in the same order as mentioned below.

i. Initializes and invokes the Dump Reader

ii. Initializes the TIF Convertor and invokes a series of operations

iii. Initializes and invokes the UIF Convertor

iv. Invokes the Code Generator

TIF Convertor

In a nutshell, the Typed-Intermediate-Form Convertor reads method information from the Method Data Interface and constructs a bytecode graph. It then executes an algorithm to link source operands and destination operands. These linkages are utilized by the UIF Convertor to switch from a stack-based instruction mode to a list-based instruction mode.

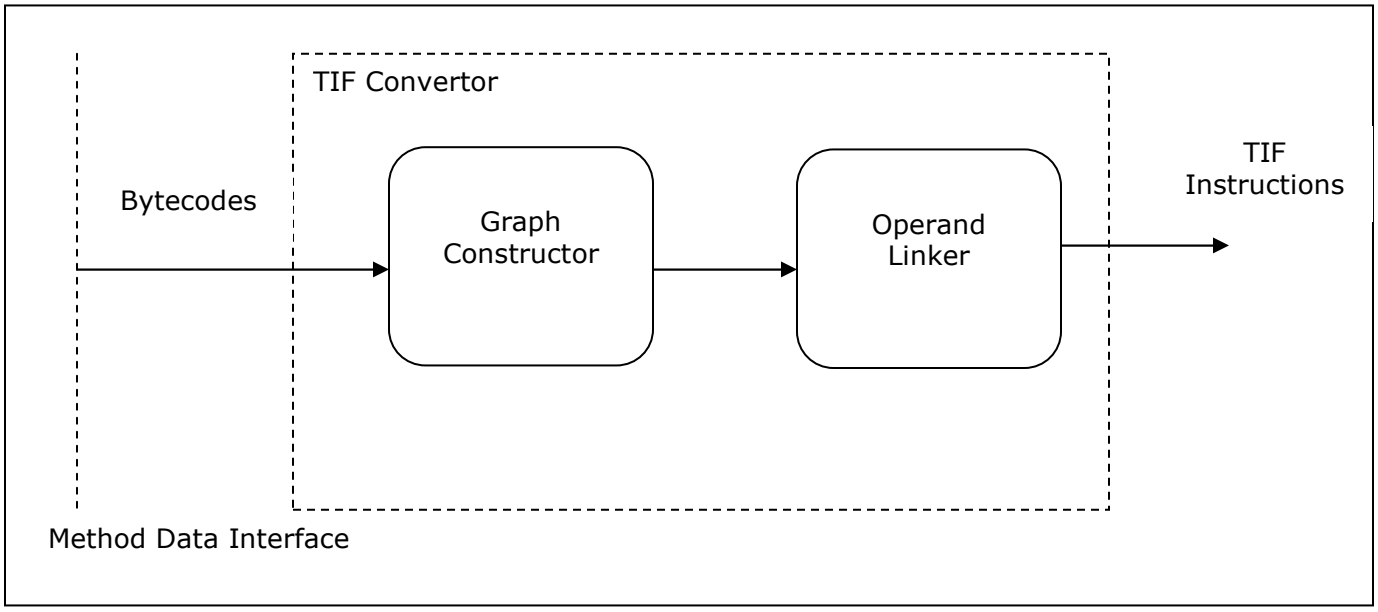

Figure 6 TIF Convertor

UIF Convertor

This component works on the TIF Graph and removes the stack-based nature of the instructions. UIF Instructions have operands in the form of a list. Duplicates of operands do not exist. The UIF Convertor is also responsible for naming local variables and evaluating their types. If the local variable information is available in the dump, we skip the generation of names for local variables.

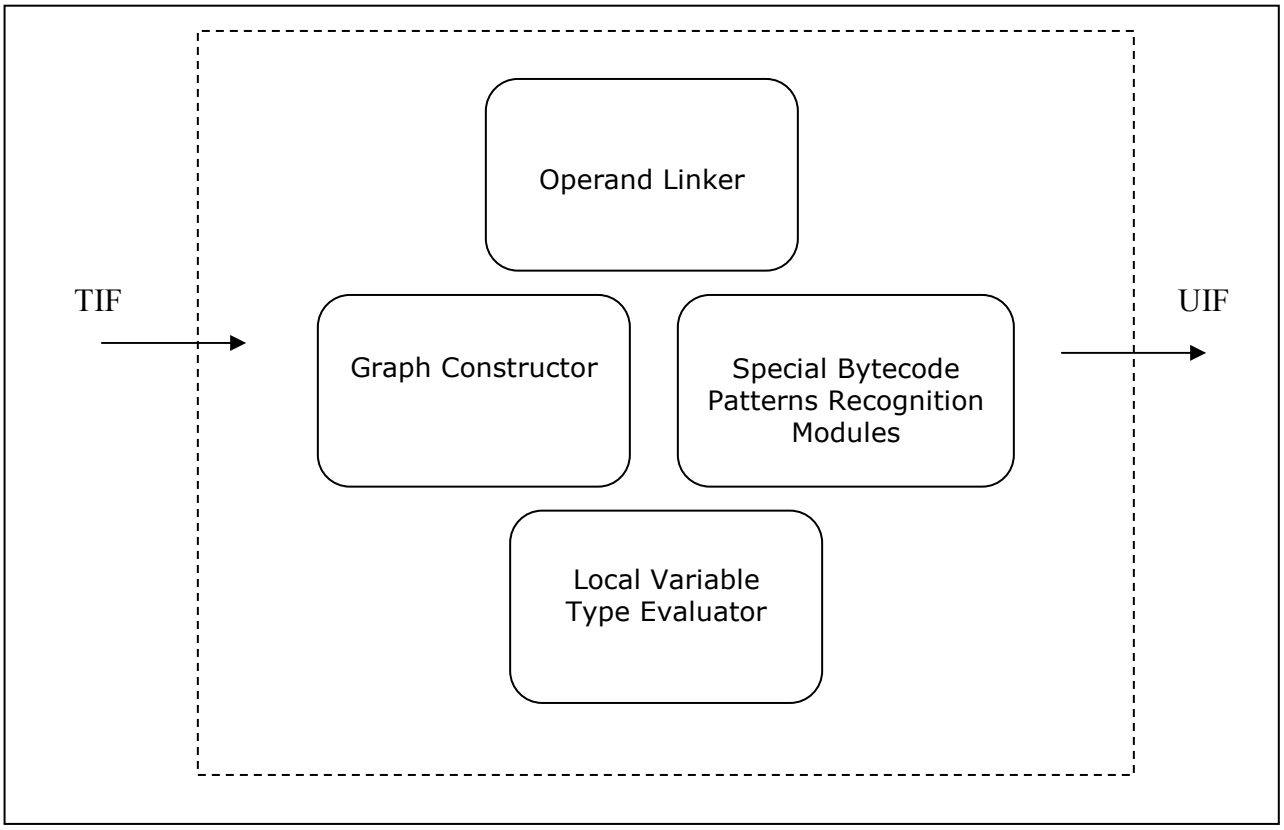

Figure 7 UIF Convertor

Special Bytecode Patterns Recognition Modules are dedicated for detecting specific patterns in the UIF Graph. An example is the construction of comparison expressions from multiple IF statements. Another example is the detection of the ternary operator, which very much looks like an IF instruction in bytecode.

Code Generator

This component converts the list of UIF instructions into a graph of Java grammar objects. Each object is suitably annotated, so that a mere depth-first walk of this graph generates the Java code. There is no modularization in this component. A hierarchy of Java Grammar Elements is defined and used by this component, during code generation.

2.5 Implementation Details

The de-compiler plug-in has been programmed in Java using a dump-reader API defined for the IBM Java Virtual Machine. The code has been divided into four broad packages, namely:

1. **Dump Reader** – Classes required for dump-reading
2. **Main** – Classes which define components of the translator
3. **Instructions** – Classes defining the various types of instructions used in the TIF and UIF
4. **Operands** – Abstractions of operands E.g. Local Variable, Constant
5. **Grammar** – Abstractions of Java grammar elements

The plug-in is written for dumps generated by IBM Java 6.0 and is deployed on the same version of Java on Windows XP. Eclipse is being used as the development environment.

3. Conclusions

3.1 Executing the Plug-in

Every Java method of a loaded class is represented internally through a metadata structure. The input to the plug-in is the address of the desired method's metadata structure. This structure holds all the essential metadata including the code in the form of bytecodes. All core dump analysis tools provide commands that report the method metadata structure address, given its name (with wildcards).

The de-compiler is a plug-in in one of IBM's core file analyzers for Java and is invoked through a command. The format of the command is:

dec <address>

3.2 Sample outputs

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;
 }
```

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;
        }
 }
```

The example above exposes one of the limitations of the de-compiler. The else block is differently laid out. Nevertheless, the semantic essence is maintained.

Example 3: Ternary Operator

Java Method

```
public int terOpTest ( int a, int b, int c )
{
       int d = (a > b) ? ((b > c) ? b :c) : a;
       int e = (b > c) ? a+b : b+c;
       return (d < e ) ? d : e;
 }
```

Output of the de-compiler

```
public int terOpTest( int ig, int ih, int ii )
{
    int ij = ig > ih ? (ih > ii ? ih : ii) : ig;
    int ik = ih > ii ? (ig + ih) : (ih + ii);
    return (ij < ik ? ij : ik);
}
```

Example 4: Array Operations & Exceptions

Java Method

```
public void arrayAccess ( char [][] charArray, int index )
{
     try {
          charArray[0][index] = 'a';
     } catch ( ArrayIndexOutOfBoundsException e )
     {
         e.printStackTrace ( );
      }
}
```

Output of the de-compiler

```
public void arrayAccess ( char [][] aArr, int ib )
{
     try {
          aArr[0][ib] = 'a';
     } catch ( ArrayIndexOutOfBoundsException cObjA ) {
         cObjA.printStackTrace ( );
     }
}
```

Example 5: Loops

Java Code

```
public int loopExample ( int [] intArray )
{
    int sum = 0;
    for ( int i = 0 ; i < intArray.length ; i++ )
        sum += intArray[i];
    return sum;
}
```

Output of the de-compiler

```
public int loopExample ( int [] aArr )
{
    int ib = 0;
    int ic = 0;
    while ( ic < aArr.length )
    {
        ib = ib + aArr[ic];
        ic = ic + 1;
    }
    return ic;
 }
```

3.3 Limitations

Limitations can often be categorized into those that are real limitations and those that present opportunities. We have several lined up in the latter. Here they are:

1. Short-hand assignment operators are not identified.

   a += 1 would be shown as **a = a + 1**

2. Pre/Post Increment/Decrement operators not identified.

   b = a[--i] would be shown as

   i = i – 1

   and

   b = a[i]

3. Distinguishing the while-loop from the for-loop.
4. Detecting the complement operator **~**
5. 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.

The real limitations, identified as of now, are:

1. 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.
2. **Static variables** which are also **final** are merely represented as constants internally. There is no way to retrieve the names of these class variables.

4. Summary

The biggest hurdle during the comprehension of customer problems in the cases of crashes and hangs is a lack of comprehension of the finest context of the failure. By “finest” I mean the vicinity of the failure. In Java applications, these failures may show up as if they were in an application method. In such cases, it is necessary that the service engineer understands the failure context fully. Method structure comprehension is hence a very crucial tool to expedite problem determination. Java methods are present in the form of bytecodes and they can be large. It is difficult to comprehend bytecode sequences that are large. There are around 200 bytecodes in use and studying huge sequences can be a pain.

This piece of work presents a solution for the method comprehension problem. The solution is trivial! De-compilation of bytecodes can create Java code for the methods and this can be easy to read and comprehend. Understanding the structure of a few methods involved in the failure can lead to complete problem determination. This has been my experience. Here we try to make de-compilation a per-method activity and present Java code to the user on a per-method basis. The per-method approach has been chosen because it strikes a balance between two problems. On one hand, engineers servicing the Java Virtual Machine are curious to know why a particular application sent the former for a toss. On the other hand, there is the problem of Intellectual Property Security, which prevents us from understanding customer’s code. This work tries to present code in fragments, on demand. Moreover, the concealed local variable information also hides a lot of sensitive information. This de-compiler is hence a good tool to assist JVM Engineers in maintaining this complicated piece of software.

5. Future Work

There is a lot of scope for future work. The limitations discussed in the first part of section 3.3 are actually optimization opportunities. Optimization in this context means making the Java code as compact as possible. Future work can hence include:

1. Recognition and exploitation of available optimization opportunities, making the resulting code as compact as possible.
2. Improvement in the local variable name generation algorithms. 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**.
3. The current implementation does not consider performance at all. A code review to improve the memory-efficiency is recommended. Time-efficiency is not critical here.
4. Java 7 added some new syntax to the Java language. The grammar need to embrace these additions.

6. References

There is no literature available on the reverse engineering of bytecodes. Research papers, too, aren't available. The following sources of information help in understanding the fundamentals.

1. Inside the Java 2 VM – Bill Venners

   http://www.artima.com/insidejvm/ed2/

2. The Java VM Spec

   http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

3. The Java Grammar

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

7. Glossary

API

Application Programming Interface

Bytecodes

Like any other hardware machine architecture, the Java Virtual Machine also defines its own instruction set. These instructions are known as bytecodes. Internally, Java methods are actually represented as a stream of bytecodes.

Class file

The **javac** compiler which is shipped with the Java Standard Development Kit compiles Java classes to class files. These files have the definition for one public class.

Class hierarchy

Polymorphism is a hallmark of object oriented programming. This quality allows classes to inherit data and operations from other classes. This leads to a lot of code reuse! When a class A inherits another class B, a parent-child relationship is established between A and B. A class that does not inherit from any other class, by default, inherits from the Object class. Hence, all the defined classes form a tree and this is called class hierarchy.

Class Loading

A defined class is nothing but an abstract type. In the primitive languages, types were always known to the runtime. In the case of Java, the Virtual Machine is made aware of a particular abstract type when it is encountered

for the first time during code execution. This revelation happens through Class Loading. During class loading, the class file present on the disk or on a network is loaded into the JVM memory.

Core File or Core Dump

A running process has its own virtual memory. On the arrival of certain signals (indicating or not indicating failure), the operating system may decide to halt the execution of a process and present the diagnostics in the form of a snapshot of the process memory at that instance. The snapshot is a binary file known as a core file or a core dump.

Dynamic Compilation

Dynamic Compilation is the concept of translating one form of code to other at runtime, most of the times, alongside the application execution. This concept exists only in languages that come with their own runtime environments. Java is an example.

Final (keyword)

A final variable cannot be reassigned. It represents a constant. Similarly, a final method cannot be redefined and a final class cannot be inherited.

Interpreter

Java has gained popularity because it is a platform independent language. The Java Virtual Machine holds a bytecode interpreter, which executes one bytecode at a time. This interpreter has gifted Java with the much-talked quality of platform independence.

Intermediate Representation

Traditionally, compilers are known to have defined an intermediate language or an intermediate representation of the code that flows into them. The objectives of having an intermediate language can be two fold. It can encourage a gradual shift from the source language to the target language. It can also provide a standard platform to execute compiler optimization algorithms.

Just-In-Time Compilation or JIT Compilation

See Dynamic Compilation.

Java Virtual Machine or JVM

The Java Virtual Machine is the platform on which a Java application executes. It is synonymous with Java Runtime Environment. The JVM includes the interpreter, the class loader and the garbage collector. The JVM is nothing but just another process on the operating system.

Local Variable Table

The **javac,** when invoked with the **–g** switch, collates the information pertinent to local variables in methods and stores it in a table called Local Variable Table. This table holds information such as variable name, type and visibility limits with respect to bytecode indices. The IBM JVM does not load this table into its memory by default.

Recreation (or failures)

Recreation is a very crucial activity of reproducing a reported failure in another environment. Problem recreation is the best step to problem determination.

Static Variables

When thinking from the OO perspective, one may wish to define attributes to classes - something which holds good for all object instances of that class. Such attributes can be defined through static variables.

TIF

Typed Intermediate Form – see Section 2.2

UIF

Untyped Intermediate Form – see Section 2.2

Checklist of items for the Final Dissertation Report

This checklist is to be attached as the last page of the report.

This checklist is to be duly completed, verified and signed by the student.

1.	**Is the final report properly hard bound?** (**Spiral bound or Soft bound or Perfect bound reports are not acceptable.)**	Yes / No
	Is the Cover page in proper format as given in Annexure A?	Yes / No
2.	Is the Title page (Inner cover page) in proper format?	Yes / No
3.	(a) Is the Certificate from the Supervisor in proper format? (b) Has it been signed by the Supervisor?	Yes / No Yes / No
4.	Is the Abstract included in the report properly written within one page? Have the technical keywords been specified properly?	Yes / No Yes / No
5.	Is the title of your report appropriate? The title should be adequately descriptive, precise and must reflect scope of the actual work done.	Yes / No
6.	Have you included the List of abbreviations / Acronyms? Uncommon abbreviations / Acronyms should not be used in the title.	Yes / No
7.	Does the Report contain a summary of the literature survey?	Yes / No
8.	Does the Table of Contents include page numbers? (i). Are the Pages numbered properly? (Ch. 1 should start on Page # 1) (ii). Are the Figures numbered properly? (Figure Numbers and Figure Titles should be at the bottom of the figures) (iii). Are the Tables numbered properly? (Table Numbers and Table Titles should be at the top of the tables) (iv). Are the Captions for the Figures and Tables proper? (v). Are the Appendices numbered properly? Are their titles appropriate	 Yes / No Yes / No Yes / No Yes / No Yes / No
9.	Is the conclusion of the Report based on discussion of the work?	Yes / No

10.	Are References or Bibliography given at the end of the Report? Have the References been cited properly inside the text of the Report? Is the citation of References in proper format?	Yes / No Yes / No Yes / No
11.	Is the report format and content according to the guidelines? The report should not be a mere printout of a Power Point Presentation, or a user manual. Source code of software need not be included in the report.	Yes / No

Declaration by Student:

I certify that I have properly verified all the items in this checklist and ensure that the report is in proper format as specified in the course handout.

Place: Bangalore **Signature of the Student**

Date: **KULKARNI PUSHKAR**

2009HZ13463