2011

The Tyranny of the Vital Few: The Pareto Principle in Language Design

Victor L. Winter

James L. Perry

Harvey Siy

Satish Srinivasan

Ben Farkas

See next page for additional authors

Authors
Victor L. Winter, James L. Perry, Harvey Siy, Satish Srinivasan, Ben Farkas, and James McCoy

Journal of Software Engineering and Applications, **2011, 4, 146-155**
doi:10.4236/jsea.2011.43016 Published Online March 2011 (http://www.SciRP.org/journal/jsea)

Scientific Research

The Tyranny of the Vital Few: The Pareto Principle in Language Design

Victor Winter[1], James Perry[1], Harvey Siy[1], Satish Srinivasan[1], Ben Farkas[2], James McCoy[2]

[1]University of Nebraska at Omaha, Omaha, Nebraska, USA; [2]Sandia National Laboratories, Albuquerque, New Mexico, USA.
Email: {vwinter,jperry,hsiy,smsrinivasan}@unomaha.edu, {bdfarka,jamccoy}@sandia.gov

Received January 26th, 2011; revised February 26th, 2011; accepted February 28th, 2011.

ABSTRACT

Modern high-level programming languages often contain constructs whose semantics are non-trivial. In practice however, software developers generally restrict the use of such constructs to settings in which their semantics is simple (*programmers use language constructs in ways they understand and can reason about*). *As a result, when developing tools for analyzing and manipulating software, a disproportionate amount of effort ends up being spent developing capabilities needed to analyze constructs in settings that are infrequently used. This paper takes the position that such distinctions between theory and practice are an important measure of the analyzability of a language.*

Keywords*: *Java Type Resolution, Transformation, Code Migration, SCORE Processor

1. Introduction

In 1906 the Italian economist Vilfredo Pareto observed that 80% of the land in Italy was owned by 20% of the population [1]. Since then similar observations have been made in a wide variety of disciplines including the field of computer science. This property is often referred to as the 80/20 *rule*, the *Pareto principle*, or the *law of the vital few*. The essence of the 80/20 rule is that effort and payoff are inversely related.

In software development, the 80/20 rule has been used to characterize a wide range of activities ranging from 1) the effort associated with testing software, to 2) the effort associated with the implementation of software features. On October 2, 2002 Microsoft CEO Steve Ballmer wrote a 3-page memo which contained the following:

"One really exciting thing we learned is how, among all the software bugs involved in reports, a relatively small proportion causes most of the errors. About 20 percent of the bugs cause 80 percent of all errors, and—this is stunning to me—*one percent of bugs cause half of all errors*." [2]

In this paper, we make a similar observation about complex constructs that sometimes find their way into the design of programming languages. One practical implication of this observation is that when developing tools for analyzing and manipulating software, a disproportionate amount of effort is spent on handling "corner cases" of language constructs which are theoretically justified but infrequently used in practice.

Analyzability is an important part of the usability of a language. Programming languages are chiefly designed such that programs can be efficiently translated into machine instructions, a process that typically involves a significant amount of local (e.g., statement-level) analysis. However, such designs could pose complications for objectives that require a precise understanding of the overall program structure or behavior, such as optimization, automated verification, and refactoring. Pointers in C provide an archetypal example of language design leading to analyzability problems as it complicates the process of accurately identifying data dependencies [3].

We see an increasing need for tools that perform sophisticated analysis of programs. The expense of developing new software necessitate adapting and reusing existing software as much as possible. In order to do so, it is often necessary for existing programs to be suitably modified from their original purpose or environment. For example, programs need to be optimized to take advantage of new processor architectures. They need to be restructured to improve maintainability. And they need to be migrated to different platforms. This phenomenon especially hits home in the area of embedded systems

*This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

programming where additional tools are often needed to facilitate the reuse of existing software.

The analyzability of the language will dictate how much effort will be spent to develop dependable tools to support such activities. Relatively newer languages such as Java may be easier to analyze than C but problems still persist, as we will illustrate in this paper. We see the Pareto Principle at work in that most Java language constructs are easy to analyze, but there are a few cases that take a significant amount of effort to analyze correctly.

We illustrate this using our experiences on a project to migrate the Java Standard Edition (SE) library to an embedded processor which runs a restricted implementation of the JVM. The processor supports most but not all the standard bytecodes. Thus, source code in the Java library that compiles to unsupported bytecodes needed to be suitably modified. To support this migration process, we developed a Java analysis and manipulation tool based on the TL system [4] that performs a source-to-source transformation to eliminate code that compiles to unsupported bytecodes.

As part of the analysis, we needed the ability to resolve type uses (e.g., in field declarations and class extensions) to their definition, or *canonical form* [5]. We argue that type resolution in Java exhibits the Pareto principle where resolution in a large number of settings is straightforward, but where corner cases exist in which type resolution is deceptively difficult. For example, analyzing a significant portion of the Java SE library (e.g., *java.util*, *java.lang*, *java.io*, etc.) reveals that roughly 99% of type uses in field declarations and class extensions involve only *simple identifiers* (*i.e.*, identifiers that do not contain "dots"). On the other hand, the remaining 1% needed a disproportionate amount of effort to analyze correctly as they involved precise and meticulous interpretations of the semantics of Java. The difficulty in correctly interpreting the resolution rules in these cases is highlighted by the fact that even well-established IDEs (and possibly some versions of Java compilers) interpret them incurrectly. These findings are reported in Section 5.

The remainder of the paper is organized as follows: Section 2 gives an overview of embedded systems and the challenges they pose to software development. In this section, the SCORE processor is also introduced and its limitations summarized. Section 3 describes a project whose goal is to migrate several core classes of the Java SE library to the SCORE platform and gives an overview of our in-house tool development. Section 4 introduces Java type resolution issues. Section 5 presents our findings with respect to Java's type resolution semantics. Section 6 presents related work and Section 7 concludes.

2. Background and Motivation

2.1. Embedded Processors

The computing power of modern micro-processors and micro-controllers has opened the door to increasingly complex embedded applications. These embedded systems have impacted virtually every aspect of our lives ranging from medical devices to automobiles to flight control systems. Historically, embedded systems programming was done using a low-level language such as assembly language. However, the complexity of modern embedded applications is such that high-level languages and tool support must be employed in order to effectively develop embedded software.

Ideally, software for embedded systems would be developed using the latest mainstream (*i.e.*, commercially available) techniques and methodologies. Software would be designed, developed in a high-level language, tested on a general-purpose computing platform and then "simply" ported to the targeted embedded system. In this setting, simulators could also be used to support analysis of the targeted micro-controller/processor. The advantages of this type of an approach are: 1) a large user base is employed to vet the technology (e.g., compiler correctness), 2) high-level mainstream programming languages reflect best-practices (e.g., type safety), and 3) tools (e.g., Eclipse) are cost-effective and have extensive capabilities requiring a development effort measured in terms of person-years.

The primary impediment to the adoption of the software development approach described in the previous paragraph is the gap that exists between micro-processors/controllers used in embedded systems and the general-purpose computing platforms on which the software would be developed. This gap arises because the hardware in embedded systems is oftentimes subjected to heavy constraints. In addition to economic forces, physical limitations place strict bounds on various hardware attributes such as volume, weight, and power usage. As a result, the computational capabilities of embedded platforms generally represent a scaled-back version of the more general purpose computing platforms found on a PC. From the perspective of software development, the gap between general-purpose platforms and embedded platforms can be bridged in two ways: 1) develop special-purpose high-level languages (e.g., a non-standard dialect of a general-purpose high-level language) and tools supporting software development for a specific embedded platform, or 2) limit software development to a subset of a general-purpose high-level language.

2.2. The SCORE Processor

The SCORE processor [6] is a hardware implementation

of the JVM [7] being designed at Sandia National Laboratories, that is similar to the Java Card [8,9], for use in resource-constrained embedded applications. **Table 1** gives an overview of the features not supported by the SCORE. We would like to mention that the SCORE does support general exceptions, just not run-time assertions.

3. Project Context

A project is underway to develop a methodology for effectively migrating Java code bases to restricted JVMs. such as the SCORE and Java Card. The goal is to then use this methodology to migrate (as needed) targeted portions of the core Java SE library (e.g., *java.util* and *java.math*) to the SCORE thereby enriching the environment for SCORE application programming.

Our approach to migration consists of two primary activities: *removal* and *re-implementation*. Removal is a fully automatic process in which source code is analyzed and unwanted dependencies are removed. Re-implementation is a predominantly manual effort that focuses on developing Java code having a desired functionality to replace code having unwanted dependencies. Previously, we have developed a source-to-source transformation-based lightweight code migrator [10,11] and we are using lessons learned from the development of this prototype to guide our current development efforts. In these earlier publications a more detailed discussion can be found regarding the nature of removal and re-implementation.

3.1. Tool Selection

Tool support for migration is predicated on the ability to analyze (Java) source code. Some analysis results can be obtained by leveraging existing software and tools such as: javac/javap, Doxygen, and Eclipse. These tools produce artifacts (e.g., xml files, Java classfiles, dot files) from which information can be mined. However, our experiences suggest that integrating existing tools can result in fragile systems where small changes to the analysis requirements can result in major overall tool re-implementation efforts. This occurs when information is desired that is not directly (or indirectly) available by a given tool.

Table 1. List of Java features not supported by the SCORE.

Feature	Keywords	Status
floating point	strictfp, float, double	unsupported
threading	synchronized, volatile	unsupported
serialization	Transient	unsupported
exceptions	assert	unsupported
reflection		unsupported
native methods	native	limited support

Limitations of third-party software provide the impetus for developing language migration capabilities in-house. In particular, we are developing migration capabilities within a transformation-based system called TL.

3.1.1. TL

The *TL System* [4,12] is a collection of functions providing general-purpose support for rewrite-based transformation over elements belonging to a (user-defined) domain language. In the TL System, a domain language is described by a tuple consisting of 1) an EBNF grammar and, 2) a lexical specification of tokens. TL is integrated with Standard ML in a relatively seamless manner. This enables a transformation to be expressed in a hybrid fashion as a mixture of rewrite rules and SML functions.

3.1.2. Bascinet

Bascinet[1] is a GUI front-end that has been developed for the TL System. Bascinet is implemented in Java and provides support for a variety of system-level functions. For example, a transformation can be applied to the entire contents of a folder (e.g., *java.lang*). Such application can be performed in a discrete or continuous fashion. When applied in a discrete mode, global state is not preserved between the application of a transformation to individual files (*i.e.*, one transformation application per file). When applied in a continuous mode, the application of a single transformation spans all the files in the selected directories. In Bascinet, one can also specify that a transformation should be applied only to files having certain extensions (e.g., .java).

TL and Bascinet are freely available and can be downloaded at [4].

3.2. Java Processing Infrastructure

At the time of this writing we have developed a Java infrastructure consisting of the following:

- A Java parser that preserves JavaDoc comments.
- A Java pretty-printer that formats Java parse trees in a manner conforming fairly closely to Java code conventions [13].
- A framework where it is easy to write transformations that gather a variety of metrics over a code base (e.g., number of classes, number of interfaces, lines-of-code, number of field declarations, etc.).
- A transformation that constructs an internal model of a Java code base. At present, we are only modeling class and field declarations. Our next step is to incorporate interfaces and enums into our model. Following that we will incorporate methods and

[1]Bascinet is a latest version of the HATS GUI (the previous IDE that supported TL programming).

constructors.

- A general ability to produce dot-files showing interesting information. These files can be viewed via ZGRviewer [14], which has been integrated into Bascinet. Currently, we are producing a dot-file showing the field declarations within classes as well as their inheritance relationships. The production of this dot-file is dependent on type resolution.

4. Type Resolution

The goal of *type resolution* is to convert a *type use U* occurring in a specific context *C* into its canonical form[2] $\mathcal{N}$.

$$(U, C) \xrightarrow{resolve} \mathcal{N}$$

Type uses can occur in a variety of contexts including: field declarations, "extends" directives, formal parameters of methods and constructors, casting operations, and local variable declarations. The file to which a context belongs is called its *compilation unit* (CU).

Consider the type uses associated with the declarations of *x*1 and *x*2 in the inner-class *A*1 (whose canonical form is *p*1.*A*.*A*1) shown in **Figure 1**.

The resolutions of *B*1.*C*1 and *C*2 yield:

$$(B1.C1,\ p1.A.A1) \xrightarrow{resolve} p1.A.B.B1.C1$$

$$(C2,\ p1.A.A1) \xrightarrow{resolve} fail$$

The explanation of these resolutions is as follows: *B*1.*C*1 is visible within *p*1.*A*.*A*1 because 1) the contents of *B* (e.g., *B*1) is inherited by *A*1, and 2) *A* and *B* belong to the same top-level class and hence have visibility over each others' private members. On the other hand, *C*2 is not visible within *A*1 because private classes are not inherited.

If one omits some technical details, the core functionality of Java's type resolution algorithm can be summarized in terms of the following set of resolution rules:

- *local-resolution rule*—Search locally for a declaration matching the type use.
- *super-resolution rule*—Search the inheritance hierarchy for a declaration matching the type use.
- *single-type import rule*—Search the single-type imports for an import matching the type use.
- *package-resolution rule*—Search all compilation units (CUs) belonging to the given package for a declaration matching the type use.
- *on-demand import rule*—Search the on-demand imports for an import matching the type use.
- *implicit import rule*—Search *java.lang* for an import matching the type use.[3]

[2]In Eclipse, focusing (F2) on a type use will resolve the type to its canonical form.

[3]The implicit import rule implicitly imports java.lang to all packages.

```
package p1;
public class A {
   class B {
      class B1 {
         private class C1 { }
      }
      private class C2 { }
   }

   class A1 extends B {
      B1.C1 x1; // private class C1 is visible
      C2 x2; // private class C2 is not visible
   }
}
```

Figure 1. Visibility rules involving inner-classes and inheritance.

- *direct-resolution rule*—Resolve the type use taking into account inheritance properties of the context.
- *absolute-resolution rule*—Resolve the type use without making any assumptions about the context in which it occurs.

Type resolution consists of a controlled application of these rules. For example, resolution must try the *single-type import rule* before trying the *on-demand import rule*. The on-demand import rule can be tried before the *implicit import rule*. It is worth mentioning that between the single-type and on-demand import rules is a *package resolution rule* that attempts resolution within the package but outside of the CU to which the context belongs.

On a more conceptual level, control over the application of resolution rules is governed by the following factors: 1) the presence/absence of a type declaration within a given context, 2) the visibility properties that hold between the context in which the type use occurs and the location where the type is declared, and 3) whether or not the resolution of a qualified id has partially succeeded (type resolution does not involve backtracking).

5. Findings

We have developed test cases that require type resolution to make a distinction between each type of rule. For example, the code in **Figure 2** distinguishes between the direct-resolution rule and the absolute-resolution rule. The interesting thing about this example is we have not been able to find a "real" Java program in which the distinction between these two rules is actually made. The code base we have (automatically) analyzed using our tool is the jdk1.6.0_18 which consists of over 2 million lines of code distributed across 7197 files. In the code base, the direct resolution rule is never needed and the absolute resolution rule is need only a tiny fraction of the time. In Section 5.1 we discuss some of our findings.

5.1. Metrics

At present we are focusing our migration efforts on a key


```
package p1;
class A extends p2.B.B1 {        // absolute rule
   p2.B.B1.B2   x1;        // direct rule
   p2.B.B1.B3   x2;        // direct rule
}

class A1 {
   p2.B.B1.B2   x1;        // unresolved type use
   p2.B.B1.B3   x2;        // absolute rule
}
```

```
package p2;
public class B extends p3.C {}    // absolute rule
```

```
package p3;
public class C {
   static public class B1 {
      protected class B2 {}
      public class B3 {}
   }
}
```

Figure 2. The distinction between the direct and absolute resolution rules.

subset of the Java SE library; specifically, the 23 packages whose prefixes can be matched with *java.io*, *java.lang*, *java.math*, *java.nio*, and *java.util*. The resulting code base comprises 13.7% (257,163 LOC) of the Java libraries. **Table 2** and **Table 3** provide metrics for the libraries we are currently targeting. These metrics were automatically collected by our tools.

Table 2 gives a summary of the use-frequency for selected resolution rules for all type uses that are currently modeled. Things worth noting include: 1) the direct-resolution rule is never used, 2) the use of the absolute-resolution rule is extremely rare, and 3) type uses within enums, interfaces and anonymous classes are (currently) not being modeled—this accounts for the discrepancy between the type use totals in **Table 2** and **Table 3**.

Table 3 gives a breakdown of the targeted libraries along a variety of metrics. Things worth noting include: 1) virtually all type uses in both fields and class extensions are simple identifiers (*i.e.*, identifiers that do not contain dots), and 2) it is extremely rare for a class declaration to occur within a method or constructor

5.2. Type Resolution Comparison

Type resolution is an essential function that underlies a variety of software development activities such as refactoring and various other code analysis capabilities provided by Eclipse [15], Netbeans [16] and IntelliJ IDEA [17]. In the majority of cases, type resolution is straightforward and has a semantics similar to the classical static scoping algorithm. However, implementing an algorithm capable of performing type resolution in all settings is deceptively tricky. Some cases are highlighted in this section.

5.2.1. Netbeans

Figure 3 shows an error in Netbeans involving the use of classes inside of methods. Java specifies that such classes are only accessible within the encompassing method. Also according to Java: a use of some class should first attempt to resolve to a corresponding class within the same scope. Thus when *C* extends *B* in **Figure 3**, *B* should resolve to *p*1.*A.foo*().*B*. However, Netbeans incorrectly resolves *B* to *p*1.*A.B*, while Eclipse and IntelliJ resolve it correctly. Netbeans printed "0", whereas the other tools printed "1".

One thing especially bad about this bug is that it compiles without any warning, and running it from Netbeans gives the wrong answer, however using the jar file generated by Netbeans and running from the command line gives the correct answer.

Figure 4 shows an error in Netbeans—this time involving duplicate class declarations. In Java, one is not allowed to have two files with the same name within a package. This includes package-private classes within other java files.

However Netbeans does not catch a duplicate class name if it is hidden inside a different '.java' file. Fortuna-

Table 2. Syntax-based metrics for core Java library.

Resolution Rule	Count	Frequency
single-type import rule	102	2.50%
on-demand import rule	129	3.16%
implicit import rule	312	7.64%
direct-resolution rule	0	0.00%
absolute-resolution rule	11	0.27%
all other resolution rules	3530	86.43%
Total	4084	100.00%

```
package p1;
public class A {
    public static void main(String [] args) { foo(); }

    public static void foo() {
        class B { int x = 1; }
        class C extends B { }
            // B incorrectly resolves to p1.A.B
        System.out.println(new C().x);
            // outputs 0 instead of 1
    }

    static class B { int x = 0; }
}
```

Figure 3. Netbeans Bug I.

Table 3. Syntax-based metrics for selected core Java library.

	Java Library					Total
	java.io	java.lang.	java.math	java.nio.	java.util.	
Packages	1	6	1	5	10	23
Compilation Units (CUs)	84	169	7	139	232	631
Lines of Code (LOC) (includes comments)	29003	54890	9422	46862	116986	257163
Type Uses in Field Declarations (classes, enums and interfaces)						
Simple ids or primitives	392	676	84	215	1802	3169
Total type uses	395	685	85	215	1847	3227
Percentage of simple ids or primitives	99.2%	98.7%	98.8%	100%	97.5%	98.2%
Enums	0	4	1	0	2	7
Interfaces	12	29	0	7	51	99
Classes						
Inner classes (non-static)	4	0	0	0	103	107
Static nested classes	23	28	2	7	198	258
Classes within methods	0	1	0	0	0	1
Classes within constructors	0	0	0	0	0	0
Top-level classes	73	132	6	125	191	527
Total	100	161	8	132	492	893
Type Uses in Class Extensions						
Type uses consisting of simple ids	65	87	3	110	305	570
Implicit extensions to Object	34	72	5	19	180	310
Type uses consisting of qualified ids	1	2	0	3	7	13
Total	100	161	8	132	492	893
Percentage of simple ids or Object	99%	98.7%	100%	97.7%	98.6%	98.5%

```
package p2;
public class Main {
    public static void main(String [] args) {
        System.out.println(DuplicateClass.x);
    }
}

class DuplicateClass { static int x = 5; }
```

```
package p2;
class DuplicateClass { static int x = 10; }
```

Figure 4. Netbeans Bug II.

nately, Netbeans won't compile with this error. However, Netbeans will allow users to run the code from the IDE. And whichever file was most recently saved is the one which Netbeans will use during execution. (Thus printing "5" or "10" depending on which file was saved last).

It is important to note that Netbeans is supposed to use the same complier as Java: which makes the bugs we found more interesting. On the Netbeans forum, some of the bugs were thought to be errors in JDK1.7 javac that propagated to Netbeans (though Bug II appears to be a bug strictly local to Netbeans). The current released JDK at the writing of this paper is 'java 1.6update21'. Though at present, Java also has the 'JDK 7 project' which accessible but is still considered to be in the 'build' stages. Netbeans, since it is directly associated with Oracle and Java, is using a preliminary version of JDK 1.7 javac. We have found that both Netbeans 6.8 and the latest version 6.9 have Bug I, yet when we downloaded the current binaries for JDK7 (build 99), Bug I did not occur. We therefore assume that Netbeans used an early version of JDK7 that had the problem, but we expect that future

releases of JDK7, as well as Netbeans, will have fixed this problem.

5.2.2. Eclipse

Figure 5 reveals a bug in Eclipse. According to the Java specification, "fields obscure classes". Thus when given the choice between a class and a field with the same name in the context where they both could be used, resolution should favor the field over the class. This is what Eclipse appears to be doing. However, in this case the field *B* is not visible in package *p*1 since it's access is package-private. Therefore, *B* in the print statement should refer to the class. Unfortunately, Eclipse refers to the field (which is not visible), and results in the error shown.

Interestingly, IntelliJ has the same bug while Netbeans correctly identifies the class *B* as the resolvent.

Figure 6 reveals another bug in Eclipse. This time the problem occurred with import statements. The Java language specification [5] Section 7.5.1 states:

A single-type-import declaration d in a compilation unit c of package p that imports a type named n shadows the declarations of:

- any top level type named n declared in another compilation unit of p.
- any type named n imported by a type-import-on-demand declaration in c.
- any type named n imported by a static-import-on-demand declaration in c.

The specification [5] Section 7.5.2 goes on to say, "A type-import-on-demand declaration never causes any other declaration to be shadowed." Therefore, if an import-on-demand contains a class that is also in a single-type-import, the single-type-import would take precedence.

Consider for a moment package *p*1 in **Figure 6**; the import statements *p*2.*Bar*.*B* and *p*3.*Foo*.* imply that any use of *B* should come from *p*2.*Bar* and not from *p*3.*Foo*. However Eclipse does the opposite. In general Eclipse handles resolution involving import statements correctly, but the case shown in **Figure 6** is more complicated in two ways: 1) one of the imports is a *static import*, and 2) there exists field-class name clash. A static import allows a CU to import static members of a class, which can include: static fields, static methods, and even static classes. Thus, *import static p*2.*Bar*.*B* is importing both the class and the field *B*. In addition, *p*2.*Bar* contains a field and a class whose identifier is *B*. As was mentioned earlier, the rules of obscuring states that a field should be chosen over a class if they both could be used in this context and if they have the same name. It is true that fields and classes can be used inside a print statement; however, fields are not classes and therefore do not have a *.class* to be accessed. We believe that the cause of the error is because Eclipse first obscures the class *B* with the field *B*

```
package p1;
public class Main {
    public static void main(String [] args) {
       System.out.println("length = " + p2.A.B.length);
    }
}
```

```
package p2;
public class A {
    public final static class B {
       public final static String length = "hello world";
    }

    int [] B = new int[5];
}
```

Figure 5. Eclipse and IntelliJ Bug I.

```
package p1;
import static p2.Bar.B;
import p3.Foo.*;
public class TestClass {
    public static void main(String [] args) {
       System.out.println(B.class.getCanonicalName());
    }
}
```

```
package p2;
public class Bar {
    public static class B { }
    public final static String B = new String("");
}
```

```
package p3;
public class Foo {
    public class B { }
}
```

Figure 6. Eclipse Bug II.

inside *p*2.*Bar*. Then it sees the complete context requiring a class and the only class *B* it can find is within *p*3.*Foo*.* since the class within *p*2.*Bar* was obscured—this is incorrect—since it must be a class in this context, the field *B* should not obscure the class.

Netbeans, IntelliJ, and the Java compiler correctly determine first that the context requires a class—and therefore choose *p*2.*Bar*.*B* since it was imported using a single-type-import.

5.2.3. IntelliJ IDEA

Aside from the bug pointed out in **Figure 5**, **Figure 7** reveals another bug in IntelliJ regarding accessing private classes from subclasses. The Java language specification Section 8.4.8 states:

A class C inherits from its direct superclass and direct superinterfaces all non-private methods (whether abstract or not) of the superclass and superinterfaces that are public, protected or declared with default access in the same


```
package p;
import extra.C2;
public class A {
    class B1 {
        class C1 {
            private class D1 {}
        }
        private class C2 {}
    }

    class B2 extends B1 {
        C1.D1 x0;
        C2 x1;   // incorrectly resolves to A.B1.C2
    }
}
```

Figure 7. IntelliJ Bug II.

package as C and are neither overridden nor hidden by a declaration in the class.

Also, the specification states:

A private class member or constructor is accessible only within the body of the top level class that encloses the declaration of the member or constructor.

Based on this latter rule, class *B*2 in **Figure 7** should be able to access the private class *D*1 when qualified by the inherited class *C*1. On the other hand, class *B*2 should not be able to access the private inner class *C*2 in its superclass *B*1. Instead, the reference to *C*2 should resolved to the single type import *extra.C*2. IntelliJ correctly resolves the reference to *C*1.*D*1 as *A.B*1.*C*1.*D*1, but incurrectly resolves *C*2 to *A.B*1.*C*2, which should not be visible. Eclipse, NetBeans and the Java compiler correctly resolve both references.

5.3. Discussion

These findings illustrate the Pareto Principle at work in code analysis and manipulation. Type resolution is straightforward in most cases as an overwhelming number of type uses are simple identifiers. However, for the few instances where qualifiers are used, resolution quickly becomes complicated and there are a few cases where even established tools resolve types incorrectly. Such situations could lead to hard-to-detect faults when these incorrectly resolved types are further manipulated, e.g., through refactoring. For example, an operation to rename all occurrences of a particular type may cause the wrong set of occurrences to be renamed.

Investing significant additional efforts in testing and fixing such problems in code analysis and manipulation tools exacerbates an already complicated development process for embedded processor environments. As embedded processors are used to control safety- and business-critical systems, there is little margin for error for software development, let alone the tools used to support the development process.

6. Related Work

6.1. Library Migration

Rayside and Kontogiannis [18] discuss a process to extract Java library subsets for supporting embedded systems applications by removing unused components from the library. They have the capability to produce library subsets having certain properties: 1) a space optimized subset, 2) a partial space optimized subset, and 3) a space reduced subset. The production of a subset is application specific with the space optimized subset being the most aggressive. The space optimized subset is created by removing all fields and methods that are not referenced by a given application. This is slightly different than the migration goals we are pursuing in which we want to universally prohibit access to fields and methods depending on features that are not supported by the target platform (*i.e.*, the SCORE). Furthermore the class loader for the SCORE [19] already has similar removal capabilities to the space optimized subset produced by Rayside and Kontogiannis. In particular, when processing the class files for a given application the class loader for the SCORE removes all methods (but not fields) that are not referenced.

6.2. Java Type Resolution

Type resolution can be generalized into identifier lookup where the use (or access) of an identifier (e.g., variables, types, methods, etc.) is matched to its declaration. Shafer, *et al.* [20] discussed identifier lookup issues in the context of automatic support for identifier renaming in Java programs. Identifier renaming requires precise identification of lookups and accesses. Making the prerequisite conditions too weak would lead to unsound renaming (the renaming is carried out incorrectly). Making the conditions too strong would disallow certain potential renaming operations from being carried out. Complex name lookup rules and addition of new language features exacerbates the difficulty of defining correct renaming rules. To address the shortcoming, the authors proposed a reverse lookup strategy, an approach which is very similar to a name lookup implemented for a compiler. The inverted lookup rules ensures the preservation of the binding structure between identifier access and declaration and that preconditions that are too weak are avoided. Secondly, the direct correspondence between the inverted lookup rules and direct access makes it possible for the refactoring tools to cope with evolution in a language. In comparison, our process of transforming a type use to its canonical form is similar to defining the lookup and access function binding, though it is unclear whether their solution differentiates between direct and absolute references.

The introduction of parameterized types or generics further complicates the type resolution process. Our type resolution algorithm currently leaves type arguments alone. To correctly resolve type arguments, the ability to perform type inference is needed. Just as implementations of type resolution have problems, we take note that type inference algorithms can also work incorrectly, as Smith and Cartwright [21] point out in the case of the Java 5 compiler. Some of the problems arise from conscious engineering decisions and others from the heuristic nature of the algorithm. Their solution, while not completely backwards compatible with the current language definition, solves many of the bugs and makes it possible to address extensions like first-class intersection types and lower-bounded type variables.

7. Conclusions

In this paper we have argued that the type resolution rules for Java are complex and make semantic distinctions along some dimensions that virtually never arise in practice. Nonetheless, the level of dependability needed by critical embedded applications necessitate that tools supporting the development processes of such applications must be able to handle such cases when they do occur.

Our experiences with Java type resolution lend support to the premise that an extreme case of the Pareto principle applies to language design. The presence of the Pareto principle has a number of implications on the analyzability of a language:

- Infrequently used parts of a language are not as comprehensively vetted by the programming community at large as their mainstream counterparts. As a result, in such settings compilers and development tools are not as mature as they are in more standard settings.
- Infrequently used parts of a language are also deceptive with respect to level-of-effort estimates for tool development. For example, it may be relatively easy to develop a prototype that correctly analyzes a construct in a standard setting. When (casually) tested, this prototype may successfully analyze the construct in virtually all cases. However, the ease in developing this prototype does not imply that development of a comprehensive analysis of the construct, handling all corner cases, also requires a similar level of effort. When the Pareto principle is in effect, a tool developer doesn't know how hard it is to develop a tool for analyzing a construct until they have implemented it in its entirety. Extrapolating the Balmer memo from Section 1 suggests half of the development effort may lie in the implementation of the last one percent.

In this article, we argued that if a programming language contains language constructs that are complex and infrequently used, then this can present significant obstacles to the development of reliable tools for that language. We believe this kind of problem is a serious one to which language designers of the future should be more cognizant. With this in mind we introduce the following new language design principle.

Design Principle 1. ***Regular Use****—a language design principle such as uniformity or generality is achieved only to the extent that it is used by the programming community for that language.*

Generality and uniformity are commonly accepted principles belonging to the theory of good language design [22]. However, to truly achieve its goals such theory should be reflected in practice: *A design principle not used is not a design principle at all.*

REFERENCES

[1] V. Pareto, "Manuale di Economia Politica," Piccola Biblioteca Scientifica, Milan, 1906. English Translation by A. Schwier, "Manual of Political Economy," Kelley Publishers, New York, 1971.

[2] S. Ballmer, "Connecting with Customers," 2002. http://www.microsoft.com/mscorp/execmail/2002/10-02customers.mspx.

[3] L. Hendren and G. Gao, "Designing Programming Languages for Analyzability: A Fresh Look at Pointer Data Structures," *Proceedings of the International Conference on Computer Languages*, Oakland, 1992, pp. 242-251. doi:10.1109/ICCL.1992.185488

[4] V. L. Winter, "The TL System," 15 November 2010. http://faculty.ist.unomaha.edu/winter/TL/TL_index.html

[5] J. Gosling, B. Joy, G. Steele and G. Bracha, "The Java Language Specification," Third Edition, Addison-Wesley, Boston, 2005.

[6] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach and V. L. Winter, "The SSP: An Example of High-Assurance System Engineering," *Proceedings of the* 8*th IEEE International Symposium on High Assurance Systems Engineering* (*HASE*), Tampa, March 2004, pp. 167-177.

[7] T. Lindholm and F. Yellin, Ed., "The Java Virtual Machine," Second Edition, Addison-Wesley, Boston, 1999.

[8] Z. Chen, "Java Card™ Technology for Smart Cards: Architecture and Programmer's Guide," Prentice-Hall, Upper Saddle River, 2000.

[9] Oracle Sun Developer Network, "Java Card Platform Specification 2.2.2," 15 November 2010. http://java.sun.com/javacard/specs. html

[10] V. L. Winter and J. Beranek, "Program Transformation Using HATS 1.84," In: R. Lämmel, J. Saraiva and J. Visser, Ed., *Generative and Transformational Techniques in Software Engineering* (*GTTSE*), LNCS Vol. 4143, Springer, Berlin, 2006, pp. 378-396. doi:10.1007/11877028_15

[11] V. L. Winter, A. Mametjanov, S. E. Morrison, J. A. McCoy and G. L. Wickstrom, "Transformation-Based Library Adaptation for Embedded Systems," *Proceedings of the* 10*th IEEE International Symposium on High Assurance Systems Engineering* (*HASE*), Dallas, November 2007, pp. 209-218. doi:10.1109/HASE.2007.38

[12] V. L. Winter, "Stack-Based Strategic Control," *Preproceedings of the* 7*th International Workshop on Reduction Strategies in Rewriting and Programming*, Paris, June 2007.

[13] Oracle Corporation, "Code Conventions for the Java Programming Language," 1999. http://www.oracle.com/tech-network/java/codeconvtoc-136057.html

[14] E. Pietriga, "A Toolkit for Addressing HCI Issues in Visual Language Environments," *IEEE Symposium on Visual Languages and Human-Centric Computing* (*VL/HCC*), Dallas, September 2005, pp. 145-152.

[15] Eclipse, "Eclipse," 21 July 2010. http://www.eclipse.org

[16] Oracle, "NetBeans," 21 July 2010. http://www.netbeans.org

[17] JetBrains, "IntelliJ IDEA: The Most Intelligent Java IDE," 12 October 2010. http://www.jetbrains.com/idea

[18] D. Rayside and K. Kontogiannis, "Extracting Java Library Subsets for Deployment on Embedded Systems," *Science of Computer Programming*, Vol. 45, No. 2-3, November-December 2002, pp. 245-270. doi:10.1016/S0167-6423(02)00059-X

[19] S. Morrison, "SSP Class Loader Responsibilities," Technical Report (Internal), Sandia National Laboratories, Albuquerque, 2005.

[20] M. Shafer, T. Ekman and O. de Moor, "Sound and Extensible Renaming for Java," *Proceedings of the* 23*rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (*OOPSLA*), Orlando, 2008, pp. 277-294. doi:10.1145/1449764.1449787

[21] D. Smith and R. Cartwright, "Java Type Inference is Broken: Can We Fix It?" *Proceedings of the* 23*rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (*OOPSLA*), Orlando, 2008, pp. 505-524. doi:10.1145/1449764.1449804

[22] K. Louden, "Programming Languages: Principles and Practice," Cengage Learning, 2003.

