Ephemeral Java Source Code

Susan Eisenbach
Department of Computing
Imperial College
London SW7 2BZ
se@doc.ic.ac.uk

Chris Sadler
School of Informatics
University of North London
London N7 8DB
c.sadler@unl.ac.uk

Abstract

In an object oriented, distributed environment, program maintenance, which has never been the most predictable task, becomes even more uncertain. Java's dynamic loading mechanism was developed to tackle some of the uncertainties. In doing so, it shifts the focus from the state of the sources to that of the binaries. This paper discusses some of the implications of that move.

1. Introduction

The languages of the seventies e.g. Ada[3,4] and Modula-2[11], were designed for the development of layered software systems. To this end their development environments implemented separate compilation and linking. Thus several developers could produce different compilation units (i.e. packages, modules) independently and the linker would ensure that the separately compiled units could work together. To achieve this, the units must be compiled in an order which reflects their dependencies so that a modification to a unit in a particular layer in the system requires the subsequent re-compilation of all the dependant units (clients) at that and subsequent layers.

Object-oriented software development introduces a new, more intimate type of dependency between compilation units. The concept of inheritance allows client applications to derive new (sub)classes based on original (base) classes defined elsewhere. At link-time references to inherited attributes and methods (i.e. dependencies) need to be resolved. This is not new. However, inheritance also permits the shadowing of inherited variables and the overriding of inherited methods. Whether an application accesses a base class variable or a derived class variable depends on the order in which the binaries were built, and likewise for methods.

Nevertheless, the layered software development system can deal with this satisfactorily for software systems where development occurs in a single consistent environment, with all sources available to all developers. On many systems, elaborate make-like utilities have been provided to track these dependencies and hence guarantee the integrity of the resulting application.

The development of distributed software raises some questions about this arrangement however, which potentially makes this problem intractable. For example:

- Who are the clients of this particular base class?
- How can we inform them that it has been modified?
- Can they get access to all the necessary sources (which may be on remote, heterogeneous systems) to re-compile?
- How can we co-ordinate the order of compilation to satisfy the base class clients, then the clients of the clients, etc.?
- Is there a version control system that can deal with this?

Development and maintenance regimes must abandon the safe ground where the source code is the ultimate fallback and learn to live with *ephemeral* sources.

In Section 2 we show how this requirement challenges "traditional" program development environments. In the two subsequent sections we discuss how the language specification of Java attempts to resolve this situation and what problems that raises. Finally we report results of a formal analysis, which give some reassurances to Java developers contemplating certain kinds of modification to their systems.

2. The Fragile Base Class Problem

In a typical C++ implementation, class method declarations are indexed in a *virtual function table* (vtable) which subsequent classes reference in order to invoke the method[8]. Thus, a class `Aircraft` is declared as follows:

```
//Aircraft.h

class Aircraft{
virtual int
GainHeight();
};
```

vtable index
0

Source files that include `Aircraft.h` are then compiled. Code is generated under the assumption that the virtual function `GainHeight` has vtable index 0.

Suppose that `Aircraft.h` is then modified to add another virtual function:

```
//Aircraft.h new version
class Aircraft{
virtual void
SndMsg(faultNo);
virtual int GainHeight();
};
```

vtable index
0
1

Any source files that include `Aircraft.h` that are not re-compiled after this modification will use at run-time the contents of entry `0` in the vtable for instances of class `C` in order to invoke `GainHeight`. But entry `0` will actually point to the code for `SndMsg`, and, when invoked from a client, a type violation will occur. The problem here is that the representation of the function in the vtable loses the *signature* of the function, replacing it merely with an offset in the table. The problems this can produce may be even more insidious. Say `Aircraft.h` was modified again:

```
//Aircraft.h new version
class Aircraft{
virtual int LoseHeight();
virtual void
SndMsg(faultNo);
virtual int GainHeight();
};
```

vtable index
0
1
2

Now the signature has been lost but not in such a way that type checking can spot it, so a client invocation of the form

```
if (height < CRITICAL) GainHeight();
```

would crash the plane and not just the computer!

This is known as the *fragile base class* problem [8]. A robust base class should allow previously compiled clients to run without re-compilation or error, since the inclusion of the new method (for *new* clients) should have no bearing on the original *contract*. The fragile base class however breaks the contract needlessly, dictating that tedious re-compilation will be required for every client application, each of which will need its own makefile.

In Java the concept of *dynamic loading* means that linking is a piecemeal activity that happens on a class-by-class basis at run-time. Each time a unit (a Java class, interface or package) is compiled, the compiler incorporates *type information* into the binary. When a given unit is compiled, the type information of any imported binaries is used in the type checking. When the interpreter runs, the same information is used to load the class dynamically and generate a run-time binding. The type information of any exporting binary is therefore accessed in two separate operations. The signatures of any methods are preserved in the type information so that the run-time binding always finds the correct method.

Should any modifications be made to the source code of the exporting unit and a new binary produced, if these modifications alter the type information accessed by the importing unit at compile-time, then the loading and binding process may fail. In this case, the Java developers will need to fall back on the sources and re-compile, just like anybody else. However, if the modifications do not alter the type information, then the loading and binding process will not be compromised. Such modifications are termed *binary compatible.*

The fragile base class problem arises out of the way that certain (most) C++ systems are implemented. It is possible to construct implementations, which avoid or ameliorate the problem[8] but only at a price, which increases the complexity of the implementation and in some cases may reduce the object oriented capabilities of the language[2]. The Java Language Specification[9] solves the problem by stipulating how implementations must behave. (Conforming implementations must record type information symbolically and are mandated to implement dynamic loading.) This development represents a new excursion into comparatively unknown territory and the Sun Java Software team have embarked on this with commendable candour [1,10].

The makefile regimes are designed to ensure that a set of sources will compile together. Dynamic binding (and binary compatibility) by contrast is concerned with whether a set of binaries will run together. In the next section we show how these two concepts can provide interference with each other.

3. Binary Compatibility – the Guarantee

The Java Language Specification[9] defines the term binary compatibility in a *declarative* manner, expressing the properties that binary compatible changes must guarantee:

"A change to a type is binary compatible with (equivalently, does not break compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error."

Ideally, one would like this to mean that if a program ran successfully before a change and the changed code compiles, then it would run successfully afterwards. However, it is impossible to constrain changes to such an extent, so it is worth examining in greater detail what the Java language designers want to achieve.

Consider the following example:

Developer L writes a class `Precipitation.java` and creates a subclass `Rain.java`. These are utilised in the program `Forecast.java` by Developer A.

Developer L

```
class Precipitation
 {String amount = "drought";}
class Rain extends Precipitation {}
```

Developer A

```
class Forecast {
 public static void main
 (String[] args) {
   String a = new Rain().amount;
   System.out.println(a);
 }
}
```

Because there is no `amount` in `Rain`, the compiler resolves the reference in `Precipitation`. When the program is executed it displays:

```
drought
```

Next the class `Rain.java` is altered by adding the instance variable `amount` of type `int`.

Developer L

```
class Rain extends
 Precipitation {int amount = 2;}
```

`Rain'.java` (say), is then re-compiled producing, `Rain'.class`. Since adding an instance variable is a binary compatible change, there is no need to re-compile the other classes and `Rain` can still honour its contract with `Forecast`. Thus if `Forecast.class` is executed, its version of amount is still bound to `Precipitation.class`, and upon execution, the program displays:

```
drought
```

Then Developer A decides to write another program `Weather.java`, which also uses amount from `Rain.class`. As `amount` is now an `int` not a `String` this is reflected in the declaration in `Weather.java`.

Developer A

```
class Weather {
 public static void main
 (String[] args) {
   int a = new Rain().amount;
   System.out.println(a);
 }
}
```

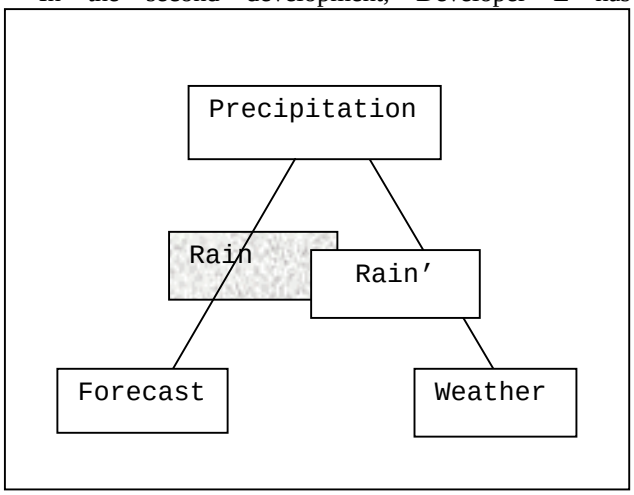
When `Weather.java` is compiled (producing `Weather.class`) and executed it displays:

```
2
```

There are thus two versions of `amount` available at the same time. `Forecast.class` is using a string from `Precipitation.class` and `Rain'.class` provides an integer to `Weather.class`. (See figure below.) As long as there is no re-compilation both programs will execute satisfactorily.

Class Relationships

In the second development, Developer L has



introduced an inconsistency between the source code (`Precipitation.java`, `Rain'.java` and `Forecast.java`) and the running binary code of the program `Forecast` (`Precipitation.class`, `Rain.class`, `Forecast.class`).

Now if one re-compiles the entire `Forecast` program a type error

```
Incompatible type for declaration
Can't convert int to java.lang.String
```

occurs on compilation of `Forecast.java`. If developer L alters `Rain'.java` to return it to the original `Rain.java` then `Forecast` can be re-built but trying to re-build `Weather` yields another incompatible type.

It is important to be clear about what binary compatibility guarantees, and what it does not. A developer such as L can maintain facilities provided for a number of clients without requiring them to re-compile their applications for each new version, provided the

changes are binary compatible ones. So library developers can fix minor errors in their code and application developers can benefit from these improvements without intervention.

The guarantee is only about binaries though. As soon as any client does try to re-compile, he may find that re-compilation is not possible.

4. Binary Compatibility – the List

The Java Language Specification also provides an *imperative* definition, listing changes which, given the guarantee, appear intuitively to be binary compatible. The following is an abbreviated version of the given list[9]:

(a) adding a new class or interface to a program, as long as the name of the new type is not the same as that of any existing type;
(b) changing the direct super-class of a class, as long as all direct or indirect super-classes continue to be direct or indirect super-classes;
(c) changing which are the direct super-interfaces of a class, as long as all direct or indirect super-interfaces continue to be direct or indirect super-interfaces;
(d) adding a field to a class;
(e) adding a method to a class;
(f) changing a method body in a class, or changing the names (but not the types or ordering) of the formal parameters of a method;
(g) adding a method to an interface.

This is a more utilitarian definition since developers can check whether contemplated modifications feature here and, if they do, proceed with confidence. However, some care still needs to be exercised. Consider the following example:

In step 1, an interface `StatusReport` is constructed, which declares a method `ShowHeight`. Then the class `Jet` is created where `ShowHeight` is implemented. Finally, the object `vtol` is instantiated in the application `JumpJet` and this executes successfully.

```
interface StatusReport {
void ShowHeight (double amount);
}
```

```
class Jet implements StatusReport {
double height; int speed;
Jet (double h, int s) {
 height = h; speed = s;
 }
public void ShowHeight (double h) {
 System.out.println("Flying at " +
                  h+". ");
 }
}
```

```
class JumpJet {
public static void main
 (String [] args) {
 Jet vtol = new Jet(5000, 550);
 vtol.height = vtol.height –  50;
 vtol.ShowHeight(vtol.height);
 }
}
```

In step 2 a new method `ShowSpeed` is added to the interface `StatusReport`. This is listed as a binary compatible change (see (g)). `JumpJet` links and runs without error.

```
interface StatusReport {
void ShowHeight (double amount);
void ShowSpeed (int amount);
}
```

In step 3 a method is added to `JumpJet`. This is a binary compatible change (see(e)), yet the compiler, when it tries to compile the constructor, outputs:

```
Class Jet is an abstract class. It
can't be instantiated
```

Actually it doesn't matter what change is made to `JumpJet`. Any attempt to re-compile it will produce the error message.

```
class JumpJet {
void SpeedStuff(Jet j){ ... }
public static void
 main (String [] args) {
Jet vtol = new Jet (5000, 550);
   ...
SpeedStuff(vtol);
 }
}
```

Since `Jet.class` hasn't been touched since step 1, the requirement that every abstract method in an interface must be implemented[9] has been bypassed. So although the modification was binary compatible and `StatusReport` still honours its contract with `Jet`, it makes `Jet` break its contract with its clients, like `JumpJet`.

List-type definitions have their advantages because they make things explicit. However, there are also disadvantages since the list may be incomplete or incorrect.

5. Formal Underpinnings

We have seen that, in a distributed, dynamically-linked environment, it can be difficult to predict what the outcome of a particular system-build might be. To try to

pinpoint the expectations which might be engendered by particular source modifications and a particular order of compilation, it is useful to adopt a formal model[6]. In doing so we note that

i. independent sets of sources and binaries seem to enjoy a separate existence – and so need to be distinguished in the model,
ii. the "context" (i.e. the precise state of the symbol table) within which compilation takes place, needs to be taken into account.

Thus, in the model:

Concept	**Formal Analogue**
Java source code compilation units ($Java_s$)	p_s (source code)
Java binaries ($Java_b$)	p_b (binary code)
Contexts of separate compilation (i.e. identifiers in scope)	Γ (environments)

Although Java is actually compiled to byte-code the Java Language Specification[9] does not require this. What it does require, is that a binary program be enriched with symbolic type information so that it can disambiguate expressions. So in the `Precipitation` example `amount` in `Forecast` (in $Java_b$) could be `amount[Precipitation]`, whereas `amount` in `Weather` (in $Java_b$) could be `amount[Rain]`.

Both $Java_s$ and $Java_b$ have type systems. The judgement

$$\Gamma \mid\!\!- p_s \ \Diamond$$

indicates that p_s type checks in the environment Γ. Similarly, the judgement

$$\Gamma \mid\!\!- p_b \ \Diamond$$

indicates that p_b is type correct in the environment Γ. Finally,

$$\Gamma \mid\!\!- e_b : T$$

indicates that the binary expression e_b has type T in the environment Γ.

Given an environment Γ, a binary program p_b and a binary expression e_b such that $\Gamma \mid\!\!- p_b \Diamond$ and $\Gamma \mid\!\!- e_b$:T then the outcome of evaluating e_b in the context of p_b will be one of the following :

- a value of type T
- a raised exception
- non-termination.

Hence type correct binaries will execute safely. If one of the list of binary compatible changes is made on a type correct program then it can be shown that the resulting program is still type correct[7].

From the formal definition it was possible to explore the exact nature of binary compatibility[5,7]. As binary compatibility was conceived to assist with the development of distributed libraries in mind[9], it is important to look at the properties that affect the way libraries evolve.

The first property that was proven is that binary compatibility scales up. So establishing that a change is binary compatible for a small number of classes automatically establishes that the change is binary compatible for any program that contains these classes [5,7]. This is fortunate because it provides assurance that if a change is binary compatible some larger system incorporating the change won't suddenly break.

Another property proven is that one can make a sequence of binary compatible changes and the resultant program will still be binary compatible[5,7]. Were this property not true the whole concept of binary compatibility would be fairly pointless, but it is nice to know that it is provably correct, especially since it isn't specifically mentioned in the Java Specification[9].

Binary compatibility is preserved over libraries. Library developers can make binary compatible changes to the particular compilation unit they are working on as long as they are aware of the changes that others make on their code. Programmers can apply independent changes and expect their programmer to link as long as they were working on different fragments. So independent libraries may be modified separately, in binary compatible ways and linking will not be compromised.

There are also properties that might have held but can be shown not to[5,7]. In particular, programmers may not apply independent binary compatible changes to the same code and expect the code to link. Another possible property, which can be shown not to hold, is that two binary compatible changes can be folded into one.

6. Conclusion

In conclusion, traditional software development and maintenance is predicated by full access to sources supported by make-file utilities, source code versioning systems and other facilities external to the run-time system. Distributed software development and maintenance forces us to relinquish these and seek new ways to manage our programs. Java's dynamic loading mechanism solves the fragile base class problem. It also introduces the idea of binary compatibility. We have

shown that this can be a mixed blessing. On the plus side some binary compatible modifications have no effect at all on the application binaries while others allow a developer to make incremental corrections and improvements which are immediately passed on the client application. On the minus side binary compatible changes can surreptitiously introduce incompatibilities between sources and otherwise play havoc with the program semantics.

Binary compatibility offers developers a short-term solution, allowing them to fiddle with their code without immediately having irate customers on the phone. The long-term, systematic maintenance regime however needs something better. At a minimum, it would be useful to find ways to define and distinguish between:

i. modifications which have no effect on existing client binaries;
ii. modifications which propagate corrections and improvements to existing binaries;
iii. modifications which preserve source compatibility;
iv. more severe modifications.

A future trend in distributed computer systems is likely to involve finding out how best to support such developments.

7. Acknowledgements

We acknowledge the financial support of the EPSRC grant Ref GR/L 76709 and the School of Informatics and Multimedia at University of North London. This work is based on more formal work being done with Sophia Drossopolou and David Wragg. We thank Sophia Drossopoulou, David Wragg and Mark Skipper for feedback.

8. References

[1] G. Bracha, *Packages break Strong Locality*, http://www-dse.doc.ic.ac.uk/~sue/packages.html, February, 1999.

[2] M. Campione, K. Walrath, A. Huml,

The Java Tuorial: A Practical Guide for Programmers, http://java.sun.com/docs/books/tutorial/java/.

[3] M. Dausmann, S. Drossopoulou, G. Persch and G. Winterstein, *A Separate Compilation System for Ada*, Proc. GI Tagung: Werkzeuge der Programmiertechnik, Springer Verlag Lecture Notes in Computer Science, 1981.

[4] *Department of Defense, Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815 A, 1983.

[5] S. Drossopoulou, S. Eisenbach and D. Wragg , *A Fragment Calculus -- towards a model of Separate Compilation, Linking and Binary Compatibility*, IEEE Symposium on Logic in Computer Science, July 1999, http://www-dse.doc.ic.ac.uk/projects/slurp/.

[5] S. Drossopoulou, S. Eisenbach and D. Wragg , *A Fragment Calculus -- towards a model of Separate Compilation, Linking and Binary Compatibility*, IEEE Symposium on Logic in Computer Science, July 1999, http://www-dse.doc.ic.ac.uk/projects/slurp/.

[6] S. Drossopoulou, S. Eisenbach and S. Khurshid, *Is the Java Type System Sound?* , Theory and Practice of Object Systems, Volume 5(1), p. 3-24, 1999.

[7] S. Drossopoulou, D. Wragg and S. Eisenbach, *What is Java Binary Compatibility?*, OOPSLA'98 Proceedings, October 1998, http://www-dse.doc.ic.ac.uk/projects/slurp/.

[8] I. Forman, M. Conner, S. Danforth,L. Raper, Release-to-Release Binary Compatibility in SOM, OOPSLA'95 Proceedings,October 1998.

[9] J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[10] T Lindholm and F Yellin, *The Java™ Virtual Machine Specification*, http: //java.sun.com/docs/books/ vmspec/2nd-edition/html/ChangesAppendix.doc.html.

[11] *Modula-2 (Base Language)*, ftp://titania.mathematik.uni-ulm.de/pub/soft/modula/standard , SO/IEC 10514-1:1996,1996.