Changing Java Programs

Susan Eisenbach
Department of Computing
Imperial College
London, UK SW7 2BZ
+44 020 7594 8264
sue@doc.ic.ac.uk

Chris Sadler
School of Computing Science
Middlesex University
London, UK N14 4YZ
+44 20 8411 2705
c.sadler@mdx.ac.uk

ABSTRACT
The promises of object-orientation and distributed computing could be delivered if the software we needed were written in stone. But it isn't, it changes. The challenge of distributed object-oriented maintenance is to find a means of evolving software, which already has a distributed client base.

Working within this scenario, we observe how certain object-oriented language systems seek to support differing client requirements and service obligations. In particular, we examine how the Java Language Specification (JLS) facilitates the concept of binary compatibility, a useful property, but one that may introduce a class of clients who dare not re-compile! Following a suggestion in the new draft JLS, we describe our tool to manage distributed version control and we formulate some proposals for future developments.

Keywords
Distributed maintenance, binary compatibility, Java, version control, dynamic loading

1 INTRODUCTION

Distributed object-oriented maintenance faces two problems, which cannot be solved using the methods of traditional software maintenance. The first of these comes about through object-orientation; the second through distribution.

Object-oriented software systems are designed to promote design sharing and software re-use. Design sharing is brought about by the class-object system where suppliers provide class definitions, which client programs use to create objects; and also in languages like Java, by the interface mechanism[6,7] where suppliers define methods which the client programs must implement. Software re-use comes about through the mechanisms that permit sub-classing (and sub-interfacing) so that clients can adapt supplied base classes to their own purposes, either adding new fields and methods or by shadowing and overriding existing ones.

In this way it is possible to build more-or-less elaborate hierarchies of inheritance classes, which also need to be managed and controlled. In C++ multiple inheritance is permitted[16]. In Java, a class can only *extend* one other class. Even so, to control the hierarchy, it is necessary to define keywords to prevent instantiation (**abstract**), to prevent further subclassing (**final**) and to restrict access at various levels within the hierarchy (**public**, **protected**, **private**, default).

Sharing code in these ways implies separate compilation and linking. Program development environments need to provide mechanisms for resolving references across compilation units, and maintenance regimes need version control systems to manage separate but interdependent development.

Object-oriented developers don't talk about program maintenance very much. They talk about the "evolution" of classes and systems, evoking the ideas of inheritance hierarchies mentioned above[23]. However, there is a maintenance problem for object-oriented developers that comes from using different generations of a system. Consider field shadowing. A base class may define a variable in a particular way and may have a sub-class used by several (first generation) clients. As a part of the evolution of the subclass, a new variable with the same name (but a different type) can be declared there. Second generation clients will differ in type from the first, and when the whole system is rebuilt, the first generation clients will need to be modified so that they can 'find' the shadowed field. The same will be true for overridden methods; the same may be true for the modifications made to the inheritance hierarchy, e.g. altering access rights.

As a system evolves, it accumulates more clients, but every modification can split the client population into two or more different types of clients who need to respond differently to the new generation of system. In keeping with the evolutionary theme, we term this effect *client speciation*.

The standard way to modify (or evolve) systems is to make new binaries by compiling all the sources together or using a makefile[10,21] system to track dependencies and re-compile dependant sources. As we have seen above, this may be a nuisance for some species of client, especially in an 'evolutionary' type object-oriented environment, but it can be managed technically by means of good version control[20] in the same way that subroutine libraries have for the past thirty years.

Distributed computing however raises a further complication. Not only may there be different clients (and clients of the clients) but they will be developing and running on different machines (and perhaps different types of machines) at geographically dispersed locations[17,21]. The prospect of getting hold of a complete set of sources is nil. Even the prospect of discovering a set of dependencies across a distributed set of suppliers and clients so that a kind of distributed makefile can be produced is daunting. In any case, this is rather against the spirit of client-server computing, which generally tries to free clients from the necessity of identifying themselves to the server for any purpose beyond the immediate transaction, and which consequently frees the server from any responsibilities beyond those of the immediate transaction. This "anonymous/autonomous" philosophy forms a basis for the design of Java's dynamic loading mechanism, which we describe together with some examples of the different client species which it can generate, in Section 3.

Section 4 describes our binary compatible version control tool JVCS, while Section 6 reflects on how language designers and other software engineers are approaching this topic. To provide some preliminary motivation however, Section 2 describes how use of C++, which was not designed for distributed software development, can lead to unsound or unsafe 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[18]. 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 Aircraft 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 [6]. 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.

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[6] 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[7,8].

3 DYNAMIC LOADING AND LINKING

The fragile base class problem arises in C++ because the C loader requires that its binaries are direct memory images. By contrast, a Java *class loader* does more of the work at load-time and thus relaxes this stringent requirement for the Java compiler. Instead, for each *reference* in the source code, the Java compiler embeds carefully defined *symbolic* information into the binary. This must include:

- the name of the entity referred to (the target) together with sufficient qualifiers to locate it in the class/interface hierarchy,

	Binary Compatible Modifications	Reason
a	Correcting methods and constructors that previously threw unwanted exceptions, or otherwise failed. Enhancing the performance of methods and constructors.	There are no changes to the type information, so the client binaries will link to the new code.
b	Reordering type declarations or removing fields and methods whose accessibility is internal (i.e. private within a class, default within a package-assuming the whole package is re-compiled).	No client reference targets have been removed-only relocated. Since the link information is symbolic, the loader can still find them.
c	Inserting new classes or interfaces into the type hierarchies.	The name qualifiers held in the original client binaries point directly to the original targets – this is sufficient to locate them.
d	Adding new fields and methods to an existing class or interface.	New entities allow new contracts with new clients. Even where fields are shadowed and methods overridden, the old client binaries contain the locations of the original target.
e	Moving a method or field upward in the class hierarchy.	The method/field will still be found using the ordinary mechanism for finding methods/fields in a class hierarchy.

Figure 1: Binary Compatible Modifications

- for field references, appropriate (symbolic) type information,
- for constructor invocations, symbolic references to the types of the parameters (name not needed),
- for method invocations, symbolic references to the parameters and the return type.

At load-time the class loader uses this *link information* to locate targets and thus *dynamically* resolve the references. Naturally, the binary representations of these target entities must contain the corresponding type information for subsequent type-checking by the verifier.

When a particular class is modified and re-compiled, its link information will be generated anew. If the modifications made do not alter the link information in any way, then the loader will be able to link this new binary with all the old binaries. Program maintainers need to know whether or not the modifications they make will alter the link information. Those modifications that do *not* are termed *binary compatible* changes. The implication of binary compatibility is that clients can continue to run their original binaries and the effects of the modifications will be felt straight away. The most significant binary compatible changes are listed in Figure 1[5,7,8].

Binary compatibility is a by-product of Java's dynamic loading mechanism. Although it is a necessity for distributed clients and it can offer the possibility of 'painless' maintenance, it is a mixed blessing because it can complicate the maintenance task quite considerably. Software maintainers are accustomed to regarding the latest set of sources as some sort of master version – a "square one" you can always go back to when planned modifications or enhancements fail. However, it might be awkward (or even impossible) to compile all the sources together, or at least in synchronicity. With dynamic loading a binary compatible modification can affect your client base in a number of unexpected ways. For example, in some cases clients will not see the effects of changes until they re-compile. We call these *blind* clients. In other cases, the clients' sources cannot be re-compiled without errors. These we will call *fragile* clients.

The Blind Client

A blind client is a client binary that will link to a modified service binary without error, but which will not see the effect of the modification until re-compilation. This can occur in a number of situations.

Shadowed Fields

Adding a new field to an existing class is a binary compatible change. Where this field has the same name and type as another field farther up the class hierarchy, the new field *shadows* the old. However, previously compiled binaries will still be bound to the shadowed variable.

Consider

```
class Coffee{String purity="pure Arabica";}
class Columbia extends Coffee{}
```

```
class SuesDiner{
   public static void main(String[] args){
       String cup = new Columbia().purity;
       System.out.println("Coffee - " + cup);
   }
}
```

SuesDiner is the client. When compiled and executed it outputs

```
Coffee - pure Arabica
```

Now suppose that Columbia were modified as follows

```
class Columbia extends Coffee{
   String purity="cut with chicory";}
```

The original purity has been shadowed and any newly compiled reference will be resolved in Columbia. Thus

```
class ChrisCafe{
   public static void main(String[] args){
      String quality=new Columbia().purity;
      System.out.println("This coffee is "
                          + quality);
   }}
```

will produce

```
This coffee is cut with chicory
```

However, SuesDiner is still bound to the old version of Columbia (see Figure 2) and so still displays

```
Coffee – pure Arabica
```

SuesDiner is blind to the modification until re-compilation, when it will be bound to the new purity.

Compile-time Constants

In Java, the keywords **final** and **static** are used together to denote a compile-time constant – one whose value is embedded directly into the binary everywhere it appears. Clearly, it is good programming practice to declare, as constant only those things that are truly constant, but if a maintainer were to change the value, none of the client binaries would see the change, even though they could link without error. The following example is taken from the *Java Language Specification* [7,8].

```
class Flags{final static boolean debug=true;}
class Test{
   public static void main(String[] args){
    if (Flags.debug)
          System.out.println("debug is true");
    else System.out.println("debug is false");
   }
}
```

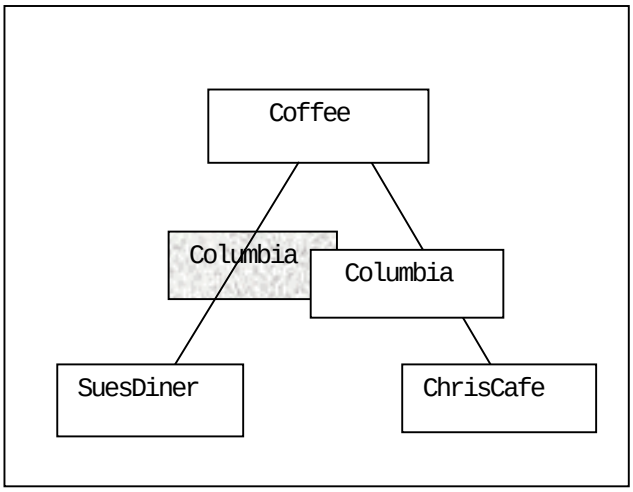

Figure 2: Class Relationships

When Test is compiled and run, the output

```
debug is true
```

is produced.

Suppose that Flags were modified so that debug=false and a new client Test1 written, identical to Test. If Flags and Test1 are compiled and run then the output

```
debug is false
```

will appear. However, when Test (which was not re-compiled) is run, it still produces

```
debug is true
```

If the maintainer realized the problem and modified Flags once again, so the keyword **final** were removed, then a new client, MyFlag say, could be written

```
class MyFlag{
   public static void main(String[] args){
      Flags.debug=!Flags.debug;
    }
}
```

Here, one client changes a value at will, whilst earlier generation clients, although they run without complaint, do not take any account of the change.

One of the main reasons for using **final** when the values are not truly constant is to prevent clients like MyFlag from overwriting values. In Java, this is better done by means of **private static** variables with bespoke access methods.

The Fragile Client

A fragile client is a client binary that will link to a modified service binary, but which cannot subsequently be re-compiled from a single set of sources. This situation can arise in a number of ways.

Shadowed Fields

Consider the earlier example of a shadowed field. Suppose that Columbia were modified as follows

```
class Columbia extends Coffee{int purity=100;}
```

The original purity has been shadowed by a variable with the same name but a different type. The class ChrisCafe could be written as

```
class ChrisCafe{
   public static void main(String[] args){
      int percent=new Columbia().purity;
      System.out.println("Serving Arabica of
                purity " + percent + "%");
   }
}
```

and produces

```
Serving Arabica of purity 100%
```

when run, while if the original SuesDiner is executed, its version of purity is still bound to the Coffee class so that it will once again output (see Figure 2)

```
Coffee - pure Arabica
```

However, if SuesDiner were to be compiled the error

```
incompatible types
found   : int
required: java.lang.String
       String cup = new Columbia().purity;
                                  ^
```

will occur. Thus, if its two clients are to undergo their own maintenance, the Coffee service hierarchy cannot simultaneously honour its contracts with both.

Access Modifiers

Java gives developers a degree of control over some aspects of the classes they create. For instance, any field or method that is declared as **private** cannot be manipulated (or even seen) outside of its containing class. This being the case, clearly any modifications performed on **private** code or data are utterly binary compatible. The opposite of **private** is **public**, which gives access to all and sundry. Between the two there is a category **protected** that grants access within the class and within all sub-classes in the hierarchy, but not outside. This is an important control mechanism for developers – powerful methods can be provided to clients, but they can only be run in pre-determined contexts dictated by the class environment.

Some consideration needs to be given to the inheritability of this access control. If client developers can create sub-classes, should they be able to interfere with the accessibility determined by the original author? The Java compiler takes the view that they should not be able to restrict access beyond that determined by the original author. In practical terms this means that there will be a compile-time error whenever an attempt is made to shadow a **public** field with a **protected** one or to override a **public** method with a **protected** one.

This gives yet another way to make fragile clients. Suppose a developer creates a class

```
class Coffee{
  protected void adulterate(){
      System.out.println("add some milk");
   }
}
```

and a client writes

```
class Homebrew extends Coffee{
   protected void adulterate(){
      System.out.println("3 spoons of sugar");
   }
   public static void main(String[] args){
      Homebrew h = new Homebrew();
      h.adulterate();
   }
}
```

Homebrew produces the output

```
3 spoons of sugar
```

because adulterate from Coffee has been correctly and successfully overridden. If the author of Coffee decided to make his version of adulterate **public** rather than **protected** and were to re-compile Coffee, this would still link with the original binary of Homebrew, which would produce the same output. However, were Homebrew to be re-compiled, the compiler would not allow the protected adulterate to override the public method in Coffee. In spite of the JLS[8] assertion to the contrary, this makes for a *fragile* client.

Interfaces

As a final example of the fragile client, consider a situation where one programmer develops an interface and a second developer implements it in a class.

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

Class `Jet` implements the interface `StatusReport` by defining the method `ShowHeight`.

```
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+".");
   }
}
```

Then a program `JumpJet` is written which instantiates `vtol`, an object of type `Jet`.

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

Subsequently, the original programmer introduces a new method `ShowSpeed` into `StatusReport`. Adding a method to an interface is a binary compatible change (see Figure 19. Provided nothing else is re-compiled, JumpJet can still run. However, if Jet is recompiled the error

```
Jet should be declared abstract; it does
not define ShowSpeed(int) in Jet
```

occurs, so `Jet` is a fragile client. Moreover, until the 1.3 release of the Java Developers Kit (JDK), `JumpJet` was also fragile. If `JumpJet` were re-compiled with the original binary of `Jet`, the following error occurred

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

4 VERSION CONTROL FOR DISTRIBUTED JAVA DEVELOPMENT

Distributed object-oriented maintenance is problematic because a modification can affect different clients differently. Clients who built their run-time images before the modification was made may be separated through blindness or fragility, as we showed earlier, from those who linked to the latest generation of the service software. To be able to predict the effects on the various species of clients, maintainers need to know about the way that Java binaries are constructed – in particular, they need to know the precise nature of the symbolically recorded link information for all the different binary code components.

To alleviate this problem, we are developing a tool, Java Version Control System, (JVCS) that will help service developers to keep track of their modifications in such a way as to anticipate or avoid subsequent client difficulties. Amongst other things, the tool can report on the binary compatibility or otherwise of modifications made to the source of a service class.

To ensure that the original and modified source files are syntactically correct, JVCS is configured to invoke an external Java compiler to create bytecode that is then checked. The checks are made between 'old' and 'new' versions.

JVCS systematically looks for modifications that *break* binary compatibility, so it needs to check the inverse of the 'rules' given in Figure 1. It starts with a look at the *overall integrity*, checking that no top-level classes or interfaces have been removed in their entirety, and that none have been introduced whose names clash with any already existing ones. This corresponds to item (c) in Figure 1.

All the remaining checks are done on a class-by-class basis. Firstly the class's position in the class hierarchy is checked to ensure that previous super classes are still visible and a similar treatment is given to each superinterface of the class.

JVCS then turns its attention to the fields of the class. Item (d) of Figure 1 reads

> "Adding new fields and methods to an existing class or interface (is a binary compatible change)."

so JVCS checks that no fields have been deleted and that any field appearing in both the current and previous versions is the same type. The same treatment is given to the methods of the class (this time checking signatures).

Whenever a check fails, a report is logged in the history file. The user obtains a check-by-check detailed result on the message screen, see Figure 3, or can browse the listing file later. For detailed debugging, it is possible to view the previous version (read-only) in another window, and, on closing the project it is possible to select which version of each file to save.

The future development of JVCS will incorporate checks for other binary compatible changes, in particular those dealing with changes in access modifiers. It may also be useful for users to be able to move directly to the relevant (highlighted) text in the editor whenever a binary compatible check fails.

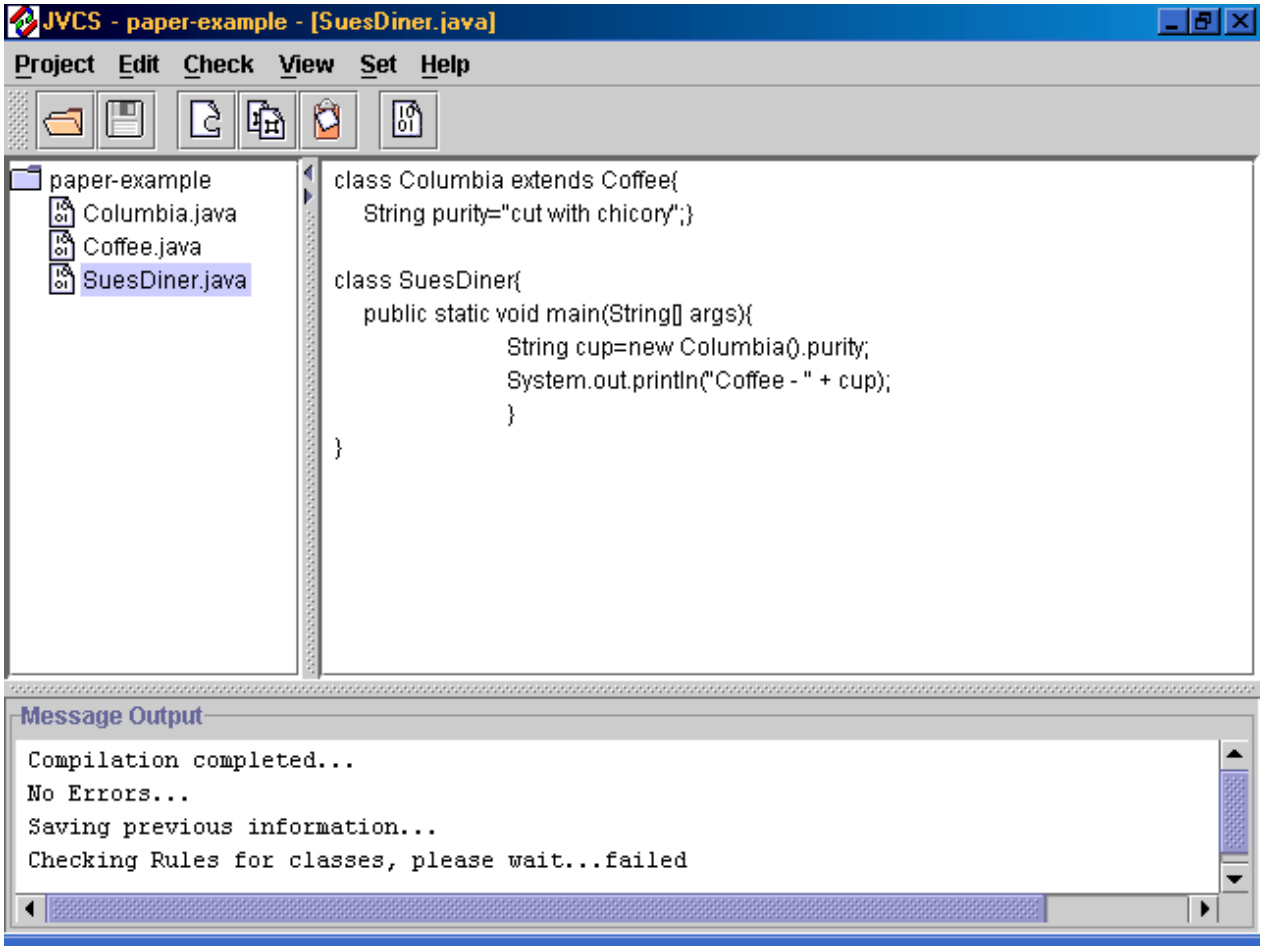

Figure 3: Java Version Control System Screenshot

5 RELATED WORK

The work described here arose directly out of more theoretical work done in collaboration with Sophia Drossopoulou[3,4]. In this work formal models were developed that distinguish between independent sets of sources and binaries since they enjoy a separate existence. It was shown that type correct binaries will execute safely (that is they will not produce a type error at run-time). 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[4].

From the formal definition it was possible to explore the exact nature of binary compatibility[3,4]. As binary compatibility was conceived to assist with the development of distributed libraries [7,8], we examined its effect on evolving libraries. We showed that if a change is binary compatible then some larger system incorporating the change won't suddenly break. We also showed is that one can make a sequence of binary compatible changes and the resultant program will still be binary compatible. Were these properties not true the whole concept of binary compatibility would be fairly pointless, but it is nice to know that they are provably correct, especially since they aren't specifically mentioned in the Java Specification[7,8]. Other formal work on distributed versioning has been done by [22], but this work does not consider the issue of binary compatibility.

Other related work falls into three categories – firstly there are groups who alter code delivered to them to get it to behave as desired. Then there are groups, primarily software engineers, working on distributed configuration management; and finally groups, primarily language designers, who are investigating support for dynamic loading.

Altering Library Binaries

[25,1,14] have done work on altering previously compiled code. Such systems enable library code to be mutated to behave in a manner to suit the client. Although work on

altering binaries preceded Java [25] it really came into its own with Java since Java bytecode is quite high level, containing type information. In both the Binary Component Adaption System [14] and the Java Object Instrumentation Environment [1] class files are altered and the new ones are loaded and run. One of the main purposes of this kind of work is extension of classes for instrumentation purposes but these systems could be used for other changes.

We have not taken the approach of altering library developers' code because it makes the application developer responsible for the used library code. Responsibility without source or documentation is not a desirable situation. There is also the problem of integrating new library releases, which the client may find it difficult to benefit from.

Configuration Management

Often configuration management is about configuration per se – technical issues about storage and distribution; or management per se – policy issues about what should be managed and how. Network-Unified Configuration Management (NUCM)[11,12] embraces both in an architecture, which incorporates a generic model of a distributed repository. The interface to this repository is sufficiently flexible to allow different policies to be manifested.

World Wide Revision Control (WWRC)[16] and its successor, World Wide Configuration Management (WWCM)[13] provide APIs for a web based client-server system. They are built around the Configuration Management Engine (CME) which extends the Karlsruhe Revision Control Engine (RCE), to implement what is effectively a distributed project in the sense used in Section 4. CME allows elements of a project to be arranged in a hierarchy and working sets to be checked in and out, and the project as a whole can be versioned so several different versions may exist concurrently.

In this paper we have mostly discussed binary compatible maintenance. The changes discussed should propagate without requiring explicit management. Where there are modifications, which are not binary compatible or in other words, which will need substantial rebuilding, Configuration Management systems such as those described will be necessary.

Language Design

On the language design side, the new draft of the Java Language Specification[8] shows substantial changes in Chapter 13, *Binary Compatibility*, although the software specification has not changed markedly. Instead, the authors have been much more explicit about the precise format of the binary file, presumably deeming it necessary to give fuller information to potential Java compiler writers.

For example in the original version the binary requirements for methods and constructors are bundled together even though their behaviour is different[7]. In the new edition[8], methods and constructors have been separated.

The designers of C#[9] also recognized that versioning is a problem that they needed to tackle and in particular that compile-time constants present a binary compatibility problem. C# has a keyword **`readonly`** for constants whose values are determined at run-time.

When one method overrides another, it is ambiguous whether this was a deliberate override or an accidental name clash. The C# compiler has keywords **`new`**/**`override`** to help make this distinction and warns developers when the ambiguity is detected. Early versions of the C# documentation had an empty chapter entitled *Versioning*.

Finally Visual Basic[19] allows server dlls to be installed with a binary compatability switch turned on. This allows server bug fixes (equivalent to category (a) in Figure 1) without having to re-compile all clients.

6 CONCLUSION AND FUTURE WORK

In this paper we have seen how a modification made to a service class in a distributed object-oriented scenario can place previous clients of that class into one of five situations. First, the modification may be an enhancement designed for use by other clients and of no concern. At the other extreme, the modification may be fundamental and require an immediate rebuild. Between these there are situations where the modification painlessly takes effect (the lucky client?), where it takes effect at the next rebuild (the blind client), or where its effect is to compromise the next rebuild (the fragile client). Software developers who support remote clients need assistance in determining what effects their modifications will have.

We have given some examples of how blind and fragile clients can arise in distributed Java applications and a description of a prototype tool, which can be used to examine the binary compatibility of Java modifications.

Finally, the following quotation from the JLS section 13.2 [8] raises an intriguing possibility:

> "Producing a consistent set of source code (following a binary compatible change) requires providing a qualified name or field access expression corresponding to the previous meaning."

Can we satisfy two (or more) sets of clients by retaining the "previous meanings" and arranging for each client only to see whatever meaning it needs? This would allow old clients to re-compile without error. It will be necessary to examine the circumstances where this device may or may not be desirable. It will also be necessary for

the client binary to identify which version of the service binary it was built with. These are topics for further investigation.

Acknowledgements
We acknowledge the financial support of the EPSRC grant Ref GR/L 76709. This work is based on more formal work done with Sophia Drossopolou and David Wragg. We thank the JVCS implementation team Y. Lam, K. Lin, Y. Gojali, C. Xu and D. Woo for their contributions to the tool.

7 REFERENCES

1. G. Cohen, J. Chase, and D. Kaminsky, *Automatic Program Transformation with JOIE*, In Proc. of USENIX Annual Technical Symposium, New Orleans, Jun. 1998.
2. 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.
3. 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, Jul. 1999, http://www-dse.doc.ic.ac.uk/projects/slurp/.
4. 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/.
5. S. Eisenbach and C. Sadler, *Ephemeral Java Source Code*, IEEE Workshop on Future Trends in Distributed Systems, Cape Town, Dec. 1999.
6. G. Cohen, J. Chase, and D. Kaminsky, *Automatic Program Transformation with JOIE*, In Proc. of USENIX Annual Technical Symposium, New Orleans, Jun. 1998.
7. 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.
8. 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, Jul. 1999, http://www-dse.doc.ic.ac.uk/projects/slurp/.
9. 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/.
10. S. Eisenbach and C. Sadler, *Ephemeral Java Source Code*, IEEE Workshop on Future Trends in Distributed Systems, Cape Town, Dec. 1999.
11. I. Forman, M. Conner, S. Danforth and L. Raper, *Release-to-Release Binary Compatibility in SOM*, OOPSLA'95 Proceedings,October 1998.
12. J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.
13. J. Gosling, B. Joy, G. Steele and G. Bracha, *The Java Language Specification Second Edition, in draft*, Addison-Wesley, 2000.
14. A. Hejlsberg and S. Wiltamuth, *C# Language Reference*, http://msdn.microsoft.com/ vstudio/ nextgen / technology/csharpintro.asp, June 2000.
15. *Help Files: UNIX: Make*, http:// www.me.berkeley. edu/ decf/ help/ unix/make.html, 1998.
16. A. Hoek, D.M. Heimbigner, and A.L. Wolf, *A Generic, Peer-to-Peer Repository for Distributed Configuration Management*, ACM 18th International Conference on Software Engineering, March 1996.
17. A. Hoek, D.M. Heimbigner, and A.L. Wolf, *Versioned Software Architecure*, 3rd International Software Architecture Workshop, Orlando, Florida, November 1998.
18. J. J. Hunt, F. Lamers, J. Reuter and W. F. Tichy. *Distributed Configuration Management Via Java and the World Wide Web*, In Proc 7th Intl. Workshop on Software Configuration Management", Boston, 1997.
19. R. Keller and U. Hölzle. *Binary Component Adaptation*. Proc. of the European Conf. on Object-Oriented Programming, Springer-Verlag, July 1998.
20. T. Lindholm and F Yellin, *The Java™ Virtual Machine Specification*, http: // java.sun.com/ docs/ books/ vmspec/2nd-edition/ html/ChangesAppendix.doc.html.
21. J. Reuter, S. U. Hänßgen, J. J. Hunt, and W. F. Tichy. *Distributed Revision Control Via the World Wide Web*, In Proc. 6th Intl. Workshop on Software Configuration Management", Berlin, Germany, March, 1996.
22. D. Rogerson, *Inside COM Microsoft's Component Object Model*, Microsoft Press, 1997.
23. B. Stroustrup, *The C++ Programming Language (3rd ed.)*, Addison Wesley Longman, 1997.
24. I. Salmre, *Building, Versioning and Maintaining Visual Basic Components*, Microsoft Corp., http: // msdn.microsoft.com/library/techart/msdn_bldvbcom. htm, February, 1998.
25. W. Tichy. *RCS: a system for version control*, Software---Practice & Experience, 15(7):637--654, July 1985.

26. *Writing Makefiles*, http:// www.gnu.org/ manual/ make/ html_chapter/ make_3.html.

27. P. Sewell, *Modules, Abstract Types, and Distributed Versioning*, Proc. of Principles of Programming Languages, ACM Press, London, Jan. 2001.

28. B. Strulu, *TOSCA: TINA Open Service Creation Architecture*, http:// www.algo.com.gr/ acts/ dolphin/ cd-rom/html/ tosca/ tosca_d3/ tosca_d3.html, 1997.

29. Steve Vinoski, *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*, IEEE Communications Magazine, February, 1997.

30. R. Wahbe, S. Lucco, and S. Graham. *Adaptable binary programs*. Technical Report CMU-CS-94-137, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, Apr. 1994.