Evolution of Distributed Java Programs

Susan Eisenbach, Chris Sadler, and Shakil Shaikh

[1] Department of Computing, Imperial College, London, UK SW7 2BZ,
[sue, sas97]@doc.ic.ac.uk
[2] School of Computing Science, Middlesex University, London, UK N14 4YZ,
c.sadler@mdx.ac.uk

Abstract. A major challenge of maintaining object-oriented programs is to find a means of evolving software that already has a distributed client base. This should be easier for Java developers than for most, because dynamic linking has been designed into the runtime system.
It turns out however that things are not so straightforward as they seem, since a given modification can leave a remote client in one of a number of states, not all of which are tolerable, let alone desirable. In this paper we attempt to delineate these states, and to consider ways of avoiding the worst of them. We describe our utility, which offers library developers a transparent version control system, to protect their remote clients.

1 Introduction

In this paper we consider the choices faced by a programmer who wishes to develop code for use by a community of heterogeneous and dispersed application developers. We refer to the code as a *library* and to its developer as the *library developer*. The users of the library we call the *client developers*.

The main issue of concern in this paper is the maintenance, or evolution of the library over time. We envisage that from time-to-time, the library developer will modify the code in the library and that different groups of client developers will initially join the user community at different points in its lifetime (different *generations*). We make a number of assumptions which it will be useful to clarify here:

1. The library developer is concerned to support all clients, no matter what generation of the library they are currently using.
2. The library developer has no idea which parts of the library are being used. The consequences of this is that, whenever a modification is made, no matter how obscure, the library developer must consider the *potential* impact, whether or not there are any clients who will *actually* be affected.
3. When the library developer makes a modification, he or she knows what is to be achieved, knows how to achieve it, and modifies the code in such a way as to achieve the desired result. We say such modifications are *intentional* and *effective*. Many library developers can and do make modifications which produce unintentional effects or which are not effective. We are not trying to solve this problem – if we could solve it, we would eliminate much of software maintenance altogether.

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 51–66, 2002.

4. When a library developer has modified the library, they will want to make it available to client developers and they will take any steps necessary to do so.
5. When a client developer needs to modify the client code as a result of a modification made to the library, the re-coding will be done with full knowledge about the nature of the library modification and the developer will continue re-coding until the modification has been successfully accommodated.

An early choice facing the library developer is how to distribute the code. On the one hand there is a very mature software engineering technology, which involves delivering a *static* version of the library on a physical medium or, with improved communications technology, by downloading from a network. The library developer can take care of his (or her) clients by imposing a good version control system [26] and offering frequent upgrades. Once downloaded, the software resides on the client system. For most of the history of software development this has been the standard method of distribution. It is still in use today (for example, by Sun Microsystems for delivering updated versions of the Java Development Kit [21]).

On the other hand, improved communication technology has given rise to a number of developments in software technology, which offer the library developer some alternatives. These include the concept of *object request brokering* [4,5,2]. Here the library developer never loses control of the library since the code to be executed resides at the library developer's site. Clients wishing to make use of the library must connect with the system and the library functions will be executed there. Provided that the integrity of the software interface is not compromised, this arrangement offers the library developer considerably more control over the immediate functionality of the library. Instead of waiting for the next 'release' to implement bug-fixes or enhancements, the library developer can implement these *in situ* to the immediate benefit of all concerned.

However, there are some disadvantages. Firstly, the library developer must have a system sufficiently powerful to deliver reasonable performance for an unknowably large community of client developers *and* their application users, and must be willing to pay for this. This is the *processor problem*. Secondly, if the system goes down or there are network problems, applications will not run at all, so all clients developers and users will be vulnerable – the *downtime problem*. Lastly, object request brokering doesn't allow for the software re-use that comes about through mechanisms that permit sub-classing (and sub-interfacing).

In this paper we consider *dynamic loading* which lies between the extremes of static library distribution and object request brokering. Dynamic libraries have a long history of providing support to executables at the operating system level [20]. However, these were usually distributed statically. Today's most prominent manifestation, the Windows dynamic link library (DLL) allows for a more incremental approach to updating, but imposes some restrictions and acknowledges some problems [1] compared with a more static approach to linking. Modern object oriented programming languages (specifically Java) incorporate

similar dynamic loading capabilities into the runtime system and it is this technology we investigate here.

In section two we describe how Java's dynamic loading mechanism can give the library developer more control over the immediate functionality of his (or her) library and also some of the pitfalls which relying on this may entail. The main contribution of this paper is made in section three where we develop a scheme to help the library developer keep track of the different modification effects that can arise as the library evolves, and in section four which describes the design and development of a utility conceived to assist library developers in their task in such a way that even solves the downtime problem. In section five we report on other recent work, which has addressed this problem and in section six give some indications of where we want to take this work in the future.

2 Using Remote Libraries

2.1 Dynamic Loading

In most programming language environments linking occurs at compile-time and at runtime the system loads the complete binary. In Java environments, the compiler embeds only symbolic references into the binary and the Java Virtual Machine (JVM) uses this information to locate and load individual classes and interfaces 'on demand' – that is, *dynamically* [12,9]. This makes for a much more complex loading process but there are advantages:

- There is a faster start-up because less code needs to be loaded (at least initially). In particular, there is 'lazier' error detection since exceptions only need to be thrown when there is an actual attempt to link with unsafe code.
- At runtime, the program can link to the latest version of the binary, even if that version was not available at the time of compilation.

It is this last feature that we are interested in and it makes Java sound like the perfect solution for library developers with remote clients. Java is pretty good, but it is not perfect. In the first place, the designers of the JVM could not (quite rightly) just leave the loading mechanism at that – between loading the code and executing it, the Verifier must be run to ensure that any changes that may have been made to the code do not break the type checks that were originally performed by the compiler. If they do a *link error* occurs.

Secondly, even if the Verifier can be persuaded to accept a modified library class, it is possible to introduce modifications which will

- not be 'felt' at all by the client application;
- compromise the safe execution of the client application;
- put the code in such a state that further recompilation on the part of the client developer will result in failure – that is, an executable application cannot be re-created from the existing sources.

2.2 Binary Compatibility

If the Verifier detects no link errors at run time we say that the (new) library is *binary compatible* with the client [12]. Every individual modification that led to the existence of the new version must have been such as to maintain binary compatibility and so is a *binary compatible modification*.

Binary compatible modifications have been the subject of some study [10,11,7,8]. The way binary compatibility works is that at compile-time the compiler embeds symbolic references (not binary code!) into the client binaries. These references record the location of the library binaries together with type information (field types and method signatures). When the library source is modified and re-compiled, if the symbolic references are unchanged the new binary can still link to previously compiled clients.

Figure 1 lists some important types of modifications, which do not interfere with the symbolic references. There are also (see Figure 2) significant and common modifications that *do* interfere with symbolic information and which must be assumed to be binary *incompatible* with old clients.

Binary compatibility is a powerful concept, which, through the mechanism of dynamic loading, can offer the library developer a great deal of support in propagating the benefits of library evolution directly to clients. However, library developers should not be too comforted by binary compatibility because there are a number of traps waiting for unwary old clients.

Any modification made to anything private.
Any modifications which improve performance or correct errors without modifying field types, method signatures or the positions of classes, interfaces or members within the class hierarchy.
Any new classes, interfaces or members. Any modification which relaxes control over the class hierarchy. Thus `abstract` $->$ non-abstract permitting instantiation, `final` $->$ non-final permitting subclassing and any increases in the accessibility of classes, interfaces and members (`private` $->$ `protected` $->$ `public`).
Any modification, which moves a field up the class hierarchy. At runtime, the system attempts to resolve unresolved field references by searching up the class hierarchy.

Fig. 1. Some Binary Compatible Modifications

2.3 Old Clients

New clients of course experience a 'virgin' library and are not our concern here. It is the existing clients that we want to try to help to keep up-to-date. Those old clients that are likely to experience link (and other) errors are considered in a subsequent section. In earlier work [11], we considered those old client developers who may be beguiled by error-free linking into believing that their users

Any modification, which removes (deletes) an accessible class or interface. If no old clients actually subclass or instantiate that class, they will in fact link without error. However, since the library developer cannot know which features and facilities his (or her) clients are using he (or she) must assume that all features and facilities are being used.
Any modification, which changes field types or method signatures in situ.
Any modification, which strengthens control over the class hierarchy. Thus non-abstract $->$ `abstract` preventing instantiation, non-final $->$ `final` preventing subclassing and decreases in the accessibility of classes, interfaces and members (`public` $->$ `protected` $->$ `private`).
Any modification, which repositions a field further down the class hierarchy.

Fig. 2. Some Binary Incompatible Modifications

1. will benefit immediately from recent modifications;
2. will run trouble-free; or
3. will be able to continue evolving their applications without difficulty.

In some cases a client binary will link to a modified library binary without error, but the client user will not experience the effect of the modification until re-compilation. We called these *blind* clients. Situations in which blind clients can occur include the introduction into the library of a shadowing field and the modification of a compile-time constant. In other cases a client binary will link to a modified library binary but will no longer compile without error. Examples of such *fragile* clients include the use of shadowing fields when the field type is changed, the introduction of a more limited access modifier and the introduction of a new method into an interface [11].

3 Modification Effect Analysis

In order that modifications should not lead to such surprises, it is important to understand the possible effects of all types of modifications. In this analysis of modification effect *outcomes* we distinguish between effects that will occur without any action on the part of the client developer, save simply executing the client code (*link-time effects*); and effects which occur after client developer action (i.e. re-coding and/or recompilation) – *compile-time effects*.

3.1 Link-Time Effects

Once the modification has been made, the library will be rebuilt and the next time the client runs it will be dynamically linked with the new library. There are three possible effects:

LE0: The client links without error and runs without any discernible modification effect;

LE1: The client links without error and the modification effect is immediately discernible;
LE2: The modification is not binary compatible and consequently a link error occurs.

In relation to 'smooth' evolution, clearly the states LE0 and LE1 are desirable in the sense that the immediate client execution is not compromised by the modification, whilst LE2 is undesirable.

3.2 Compile-Time Effects

At some point (probably immediately in the case of a link-time or run-time error) the client is likely to recompile. The compilation may be trouble-free or it may be troublesome (i.e. there are compilation errors). In the case of troublesome compilation, it is assumed in the model that the client developer will re-code until compilation succeeds, and that re-coding will be done in the light of full knowledge about the library modification and with a view to benefiting from the modification. This gives rise to three possible effects:

CE0: The client rebuilds without error and runs without any discernible modification effect;
CE1: The client rebuilds without error and the modification effect appears (or persists if it was previously discernible);
CE2: The client encounters a compilation error, and, after re-coding achieves the desired modification.

CE0 is a desirable outcome if the modification made was not intended for these clients at all, but for a new species of client. CE1 is also desirable. CE2 places the burden of achieving a desirable modification effect on the client developer. We delineate all possible states by combining all LE and CE states as in Figure 3.

3.3 Classifying Reaction Types

On the assumption therefore that the client is bound to use the current version of the library (as dynamic linking presupposes), any individual modification to a library class can result in one of nine distinct outcomes for the client. However, from the library developer's point of view they can be classified into four *reaction types*.

Type I. The client can link to the new library and continue operations without hindrance. This category covers four distinct outcomes. In some cases (00 and 11) there are no further implications for the library developer provided that the actual outcome is the desired one.

In the case of 00 for instance, if the library developer had introduced a new class, which was not intended for old clients, then the fact that the old clients would never see the class is a desirable outcome.

LE	CE	Description	Reaction Type
0	0	The client experiences nothing (ever).	I
0	1	The client experiences nothing until a trouble-free re-compilation makes the modification discernible.	I
0	2	The client must re-code in order to experience the modification.	II
1	0	The client experiences the modification but it disappears after a trouble-free compilation.	I
1	1	The client experiences the modification immediately.	I
1	2	The client experiences the modification but the client developer will need to re-code as soon as recompilation occurs.	II
2	0	The client experiences an immediate linking problem. On recompilation, the problem disappears, but the modification never appears.	III
2	1	The client experiences an immediate linking problem which a trouble-free compilation resolves.	III
2	2	The client experiences an immediate problem, which the client developer will need to re-code to fix.	IV

Fig. 3. Modification Effects

In the case of 11, the modification would be discernible for old clients. If the effect is intended (for example, if it achieved improved performance of an algorithm, or corrected a spelling mistake in an error message) then the outcome is desirable.

However, the library developer could introduce a modification, which caused old clients to compute incorrect results (for example, changing a method so that it returned financial data in Euros rather than Pounds). Although the modification is discernible, its effect is an undesirable one and further intervention is indicated.

For outcome 01, the modification will only become discernible after rebuilding. It may be that no action is required on the part of the library developer, but if the modification *corrects* an error, which threatens future runs of the client, it would be desirable for the client developer to deal with it urgently. Another case may arise when the developer introduces an unintentional name-clash. On re-compilation the client binary may link to the wrong entity. Since it was unintentional, the library developer is unlikely to be aware of this and the client developer will have to detect the error and recover.

Type II. The modification does not threaten the running of the existing client, but when the client code is re-built, the modification will compromise the re-

sulting binaries unless further client coding is done. Since it is not possible for the library developer to dictate precisely when the client recompiles, it would be safer if the client were to continue to link to the previous version.

Type III. Even though the re-build will be trouble-free, any execution before it is done will compromise the client. Once again, because the library developer cannot force the client to rebuild, a conservative policy dictates that the client should continue to link to the previous version of the library.

Type IV. The modification has compromised the client binaries – the client cannot link with the new library and cannot recompile without further coding. This is the least desirable scenario of all.

3.4 An Evolutionary Development Strategy

How can Java library developers evolve their classes without forcing their distributed clients into undesirable reaction states? One possibility is to restrict modifications so that they are only Type I. To help library developers to achieve this, we want to be able to determine, for any particular modification, whether

1. the client code will link and run (i.e. the modification is binary compatible, LE=0 or LE=1);
2. the client code will compile without error (CE=0 or CE=1);
3. the client code will execute correctly (LE=1, CE=1 and the discernible effect is desirable).

In any utility designed to assist Java library developers with distributed evolution, it should be possible to implement (1) and (2), although (3) will always be the developer's responsibility.

The restriction to Type I modifications is severely limiting for developers and we need to find a way to overcome this limitation. One idea is to devise a method of compromising the dynamic loading mechanism so that, under controlled conditions, potentially incompatible clients can link to earlier versions of the library (with which they are compatible). This can be incorporated relatively easily into any utility for the evolution of distributed Java programs. Another idea would be to develop techniques for embedding the code of undesirable modifications into structures that can mitigate their effects.

Finally, the discussion above identified several situations when it would be highly desirable for the library developer to be able to communicate with the client developer – for example to advise of an essential bug-fix which can be made discernible by means of a simple rebuild (01 or 21) or to warn not to rebuild (12). To achieve this it is not necessary to maintain a large contact list of clients or to require clients to go through a registration process, since every time a client runs, it 'touches base' with the library. We would like a Java evolution utility to help the library developer achieve this communication.

In the next section we discuss the design of a Distributed Evolution for Java Utility DEJAVU which has been developed to implement and experiment with some of the ideas discussed above.

4 DEJAVU – The Distributed Evolution for Java Utility

We set out to design a utility to help library developers and their client application developers to overcome the problems discussed above. To limit the scope and also to exploit the potential embedded in its design, we decided to restrict ourselves to distributed program development using Java. We assume furthermore that any modification made by the library developer is intentional (in the sense that it does what the library developer wanted it to do) and effective (in the same sense). A number of other stipulations were made to determine the scope of the utility:

1. All modifications are to be made to source code. No bit-twiddling of the binaries or other hacking can be permitted.
2. Evolution and access are to be mutually asynchronous. Any concept of 'releases' should be utterly transparent to clients.
3. Clients should be able to access and use the library without having to identify themselves or 'register' in any other way.
4. No changes to the Java language definition or to Java technical standards should be imposed.
5. Both library developers and client developers should be able to develop their code in standard Java development environments.
6. Library developers and client developers should not have to modify their programming style in order to use the tool.
7. Library developers and client developers should have access to version information.
8. The existence and operation of the tool should be transparent to all client application users.

From these indications we deduce the operation of the utility as follows:

1. The utility should accept versions of libraries from the library developer.
2. Having accepted a new version, it must compare it with the existing version. In making this comparison, it must detect significant modifications and must assess their impact at some level. (Initially Response Type I.)
3. The utility must report its findings to the library developer.
4. On the basis of the report (or otherwise), it must permit the library developer to publish the new version – that is to make it available to clients.
5. When a client developer wishes to compile, the utility must provide access to the appropriate library code.
6. When a client application attempts to link at runtime, the utility must determine whether or not the compile-time library version corresponds to the current runtime version. If not, it must ensure that the client application links to (and hence runs with) the appropriate (updated if necessary) version of the library.

7. The utility must be able to report to the client developer where and how the client application's most recently compiled version of the library diverges from the latest published version.

4.1 Architecture

Providing the client developer with a custom classloader, which is sensitive to the various versions of the library, could solve this problem. When this classloader detects a class that had been updated since the last build or run, it could analyze subsequent versions of the library to determine the most recent compatible one from which to load the class.

However, this is not a feasible solution for distributing libraries because of the way the current JVM works (see [24]). Either the client developer or the library developer would have to alter their programming style quite radically for such a system to work. (The client developer would need to use reflection to invoke methods. The library developer would need to establish fixed static interfaces). Instead we adopted the idea of updating at the level of the library as a whole. On the library developer side, we proposed a Controller component, which allows developers to maintain multiple versions of the library. One version is labelled 'latest' and this is the default version with which all new clients will link. Older clients will originally have compiled using an earlier version. The Controller needs to know, for any given client, whether the linked version is compatible with the latest version, and if not, which is the most recent version that it is compatible with.

The crucial question here is to determine what constitutes 'compatible' in these circumstances. In our system we define compatibility via a set of 'rules' governing one or more differences between versions of a class. If all the rules are satisfied the two versions will be considered compatible. By enunciating new rules and expanding the ruleset, we plan to extend the support, which the utility can provide to developers. The component of the system that applies Rules to two versions of a any library class is the RuleEngine. The overall architecture can be represented schematically as in Figure 4.

4.2 The Rule Engine

In order to drive the RuleEngine, an abstract Rule class was developed. This contains a number of fields and methods whose explanation will serve to describe the structure of the class.

1. `Vector newClasses, oldClasses`. When the RuleEngine attempts to apply a rule, it needs to compare two versions of a class library. The Vectors store all the classnames of each version, so each rule can be applied in turn to all the classes in the library.
2. `Vector ruleBreakingClasses`. Once it has been determined that a rule has been broken, the RuleEngine needs to be able to identify those classes where this has been detected.

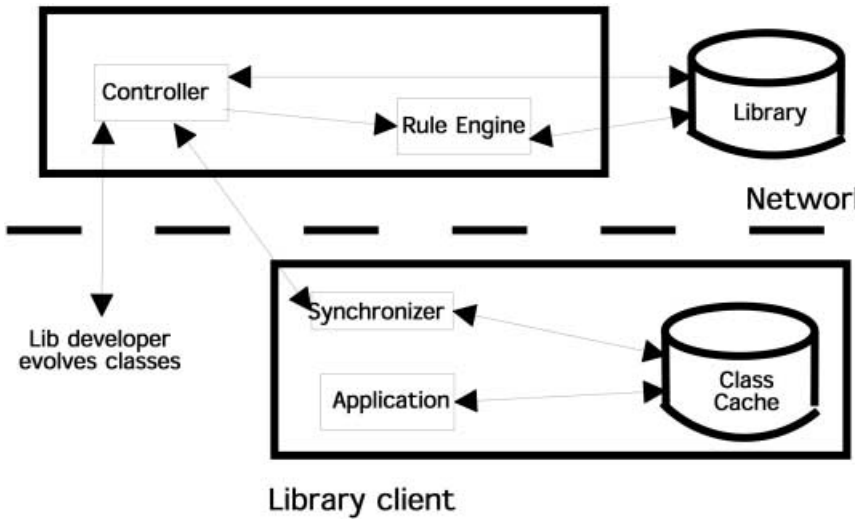

Fig. 4. Tool architecture

3. `String rule, ruleName, ruleExp`. These are designed to store specific reporting data for each specific rule, respectively the rule's operation, name and a description of the nature of the breakage.
4. `void setClasses (Vector nc, Vector oc)`. This method establishes the classnames of classes in the two versions of the library to be compared.
5. `static Class extract (Class c, Vector v)`. This method is used to extract corresponding classes from the two libraries.
6. `abstract Boolean checkRule()`. This method applies the (concrete) rule to the current set of library versions.
7. `abstract String diagnose()`. Rreturns a diagnostic message when a rule is broken.
8. `abstract Vector getDetails()`. Returns ruleBreakingClasses to report where the rule was broken.

Some rules cover classes as a whole – for example, there are rules to detect deleted public classes; classes that have become abstract, final and other than public; and a rule to determine whether the inheritance chain has been changed. Others are concerned with members of classes - deleted public fields and methods; deleted constructors; reduction of access to fields and methods.

Using this scheme, a set of such Rules can be implemented and passed to the RuleEngine, which iterates through, reporting its findings. The RuleEngine works with java.lang.reflect to examine and compare class versions. To do this the relevant classes must be loaded in the normal way. However the routine classloader provided with the JVM [18] is not capable of loading two classes with the same name from different locations. To bypass this a server-based custom classloader was developed capable of loading named classes from arbitrary locations.

4.3 The Controller and Synchroniser

On the client side, the system provides a Synchronizer component, which must be invoked as a part of the client application initialization. The client developer must start with an initial version of the library, downloaded into a local ClassCache, and containing the `Synchroniser` class `ClassSynchr` with the method `syncClasses()`. It is this method that must be invoked early in the execution of the client application. At compile time the client code will link to the local version of the library and build the binaries accordingly. At runtime, the call to `syncClasses()` will create a connection to the remote `RuleServer` to test to see if the library needs updating. If it does then the entire library is replaced by the latest compatible version before loading continues once again from the local ClassCache.

In order to implement this, it is essential that the `syncClasses()` enquiry (about version compatibility) can be quickly dealt with. To enable this a number of persistent 'settings' files are maintained by the utility. These are

1. RuleSettings. This file is maintained by the `RuleServer` and covers all versions of the library as a whole – thus it records a list of all 'currently published' versions.
2. ServerVersionSettings. This file resides in the directory of each version. It stores the version number together with a record of the latest compatible version and also the latest version with which it has been compared.
3. LocalVersionSettings. This records the most recently copied local version of the library.

The Controller provides the interface for all the users of the system. It must be able to

1. accept updated versions of the library and, when requested, publish these.
2. interact with the `ClassSyncr` object of each client to determine whether a new library version is required and, if so, which one to download.

To manage multiple simultaneous clients, the `Controller` is implemented by a Controller/Slave pattern. Library developer requests cause a `RuleRemoteObj` to be created. This manages the upload of the new library files and invokes the `RuleEngine` to generate the version compatibility data.

This is not an ideal arrangement since the library developer cannot deduce the precise compatibility circumstances governing the current modifications in any interactive way, but must instead wait until the new library has been uploaded. For this reason the upload occurs in two stages. First the library is uploaded. Then it can be checked by the RuleEngine and a report for the library developer generated. Only then may the library developer 'publish' the new version, making it available to external clients. Correspondingly, on receiving a request from a remote ClassSyncr object, the controller creates a slave RuleRemoteCL to retrieve and examine the LocalVersionSettings file of the requester and choose an appropriate library version to download to the remote ClassCache. This can be represented schematically as in Figure 5.

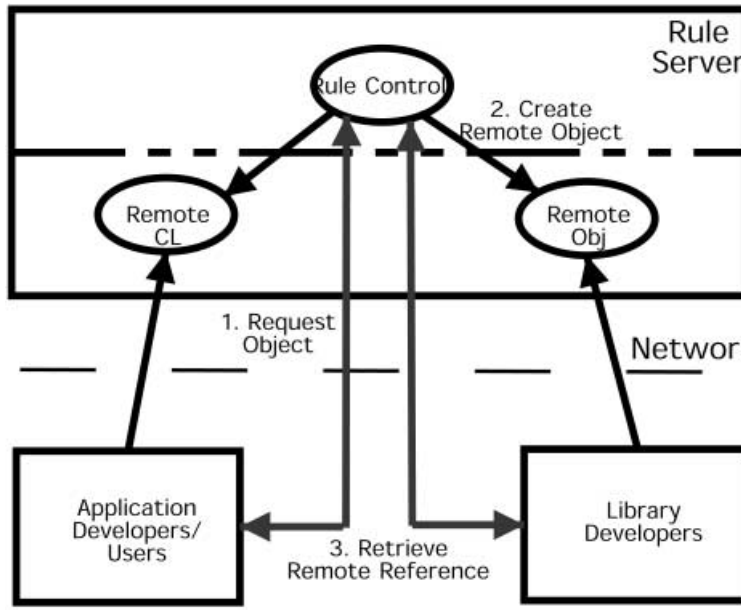

Fig. 5. Rule controller and slave objects

5 Related Work

The work described here arose directly out of theoretical work on binary compatibility done in collaboration with Drossopoulou and Wragg [7,8]. As binary compatibility was conceived to assist with the development of distributed libraries [10,11], we examined its effect on evolving libraries. Drossopoulou then went on to model dynamic linking in [6] and we looked at the nature of dynamic linking in Java [9]. We have looked at the problems that arise with binary compatible code in [10] and built a less powerful tool, described in [11]. Other formal work on distributed versioning has been done by Sewell in [23], but this work does not consider the issue of binary compatibility.

Other groups have studied the problem of protecting clients from unfortunate library modifications. [25] identified four problems with 'parent class exchange'. One of these concerned the introduction of a new (abstract) method into an interface. As discussed in [11], this is a 02 modification effect since the client will not compile correctly until the method has been implemented. The other issues all concern library methods which are overridden by client methods in circumstances where, under evolution, the application behaviour is adversely affected. To solve these problems, *reuse contracts* are proposed in order to document the library developer's design commitments. As the library evolves, the terms of the library's contract change and the same is true of the corresponding client's contract. Comparison of these contracts can serve to identify potential problems.

Mezini [19] investigated the same problem (here termed horizontal evolution) and considered that conventional composition mechanisms were not sophisticated enough to propagate design properties to the client. She proposed a *smart* composition model wherein, amongst other things, information about the library calling structure is made available at the client's site. Successive versions of this information can be compared using reflection to determine how the client can be protected. These ideas have been implemented as an extension to Smalltalk.

[17,3,27] 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 [27] it came into its own with Java since Java bytecode is high level, containing type information. In both the Binary Component Adaptation System [17] and the Java Object Instrumentation Environment [3] 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 not benefit from.

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) [13,14] embraces both in an architecture, incorporating a generic model of a distributed repository. The interface to this repository is sufficiently flexible to allow different policies to be manifested.

World Wide Configuration Management (WWCM) [15,22] provide an API for a web based client-server system. It is built around the Configuration Management Engine (CME) to implement what is effectively a distributed project. 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 changes that should propagate without requiring such explicit management. Where there are more major modifications, which will need substantial rebuilding, Configuration Management systems such as those described will be necessary.

6 Conclusions and Future Work

We have attempted to address the problems faced by a library developer who wishes to support remote clients by means of Java's dynamic loading mechanism. Although we started by considering binary compatibility as a key criterion for safe evolution, our analysis indicates that the situation is more complicated and that there is a greater variety of modification effects. We have developed a

utility that helps to manage some of the issues raised and which we believe has the potential to be extended to cover a wider range of the issues.

In contemplating the further development of our project we see a number of ideas, which could serve to enhance the utility. These include:

1. Support for library development at the package level [16]. We have considered a library to be simply a hierarchy of classes. The Java language designers envisage libraries more in terms of Packages and we need to consider what issues this raises.
2. Library clients. It is possible that the client developer is developing a library for use by downstream clients. This leads to a multi-layered development model and we need to consider how our utility might work when a downstream client tries to link to several libraries each linked to a different version of the original.
3. Touching Base. Every time a client application runs, dynamic loading causes it to 'touch base' with the library's home system. There are many ways in which this communication could be exploited, such as
   (a) transmitting warnings and other updating information;
   (b) collecting information about the numbers of users and frequency of use of different versions of the library;
   (c) managing licenses and intellectual property data;
   (d) charging users on a 'per use' basis. Sun is developing support for this via its JMX (Java Management Extensions) specification [16]. We need to see whether this technology can serve our purposes.

Acknowledgements

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

References

[1] R. Anderson, *The End of DLL Hell*, Microsoft Corporation, http:msdn.microsoft.comlibrarytechartdlldanger1.htm, January 2000.
[2] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer, *SOAP: Simple Object Access Protocol*, http://msdn.microsoft.com.
[3] G. Cohen, J. Chase, and D. Kaminsky, *Automatic Program Transformation with JOIE*, USENIX Annual Technical Symposium, New Orleans, 1998.
[4] *CORBA*, http://www.corba.org/.
[5] DCOM, http://www.microsoft.com/com/tech/DCOM.asp.
[6] S. Drossopoulou, *An Abstract Model of Java Dynamic Linking*, Loading and Verification, Types in Compilation Montreal, September 2001.

[7] 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/.

[8] 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/.

[9] S. Eisenbach and S. Drossopoulou, *Manifestations of the Dynamic Linking Process in Java*, June 2001, http://www-dse.doc.ic.ac.uk/projects/slurp/dynamic-link/linking.htm.

[10] S. Eisenbach and C. Sadler, *Ephemeral Java Source Code*, IEEE Workshop on Future Trends in Distributed Systems, Cape Town, Dec. 1999.

[11] S. Eisenbach and C. Sadler, *Changing Java Programs*, IEEE Conference in Software Maintenance, Florence, Nov. 2001.

[12] J. Gosling, B. Joy, G. Steele and G. Bracha, *The Java Language Specification Second Edition*, Addison-Wesley, 2000.

[13] D. Hoek, 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.

[14] D. Hoek, M. Heimbigner, and A.L. Wolf, *Versioned Software Architecture*, 3rd International Software Architecture Workshop, Orlando, Florida, November 1998.

[15] 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.

[16] *Java Management Extensions (JMX)*, java.sun.com/products/JavaManagement/, Jul. 2000.

[17] R. Keller and U. Holzle. *Binary Component Adaptation*, Proc. of the European Conf. on Object-Oriented Programming, Springer-Verlag, July 1998.

[18] T. Lindholm and F. Yellin, *The Java(tm) Virtual Machine Specification*, http:java.sun.comdocsbooksvmspec2nd-editionhtmlChangesAppendix.doc.html.

[19] M. Mezini, *Maintaining the Consistency of Class Libraries During Their Evolution*, Proc. of OOPSLA, 1997.

[20] J. Peterson and A. Silberschatz, *Operating System Concepts*, Addison Wesley, 1985.

[21] *Products and APIs*, http://java.sun.com/products/.

[22] J. Reuter, S. U. Hanssgen, 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.

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

[24] S. Shaikh, *Distributed Version Control for Java*, June, 2001, http://www-dse.doc.ic.ac.uk/projects/slurp/.

[25] P. Steyaert, C. Lucas, K. Mens and T. D'Hondt, *Reuse Contracts: Managing the Evolution of Reusable Assets*, Proc. of OOPSLA,1996.

[26] W. Tichy. *RCS: A System for Version Control*, Software – Practice and Experience, 15(7):637–654, July 1985.

[27] 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.