JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 3, No. 4, July–August 2004

Reuse and Abuse

Susan Eisenbach, Department of Computing, Imperial College London
Chris Sadler, School of Computing Science, Middlesex University

As programming languages and software development paradigms have evolved, so has our conception of what is meant by the term 'software reuse'. In this paper we track these shifts in meaning up to the era of component frameworks. Our concerns, initially related to the maintenance of shared libraries, have become a study of *dynamic evolution*.
The designers of modern programming languages and runtime systems have devoted considerable efforts to ensuring that today's software systems can, in some circumstances, be updated incrementally through the mechanism of dynamic linking. We examine those circumstances and the situations where they do not apply. We describe tools designed to support dynamic evolution on behalf of component developers, and to help their clients to benefit from it.

1 INTRODUCTION

'Software reuse' has been a confusing term for software developers. Over time, the word has taken different meanings which seemed to promise us different things. Initially, it implied some kind of code scrap-heap where a developer could find some bits and pieces to help get a job done. Then, with the advent of multi-user, time-sharing computers, it came to mean just another shareable resource. Nowadays, most people would say that ('reusable') code has been purpose-built to be supplied to multiple *client* developers for incorporation into their own applications. So really we just mean 'usable'.

The term 'supplied to' implies some sort of *packaging*, or, in the words of Szyperski [31], a 'unit of deployment'. The term 'incorporation into' implies some sort of *interface* or [31], a 'unit of composition'. We need to consider how to achieve these twin goals.

White-box distribution The earliest method of 'reusing' code was simply to ship the sources. The client developer could incorporate the supplied code easily as it stood or otherwise adapt it to the particular requirements of the application. If the code was adapted however and the resulting application failed to run correctly, this raised questions about responsibilities and liabilities implicit in the 'contract' between the two developers. When the original developer corrected or upgraded the *service* code, the client developer had to decide whether to incorporate (and possibly adapt) the new version or stick with the old. With a large number of

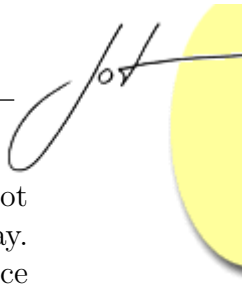
clients, an unsupportable multiplicity of *versions* could result. In addition to the issues of deployment and composition mentioned earlier, it seems it is necessary to consider the issue of *sustained maintenance* – how to support multiple clients whilst both they and the original service provider undergo change. Perhaps this is another meaning for 'reuse' – the ability to keep on using your application whilst the software it depends upon is upgraded. In this sense, white box distribution is, for all practical purposes, a disposable, one-off solution.

Black box distribution The abstract datatype and the concept of information hiding were conceived originally to prevent clients from adapting or otherwise misusing supplied service code in such a way as to compromise its integrity. The mechanism of separate compilation was thus developed to allow black box distribution via the 'executable subroutine library' which can be linked to the client application at build-time. This technology flourished in the 1970's. Limitations in memory and storage capacity led to the development of memory-resident (dynamic) link-libraries so that the application image size could be minimised and the library code shared between multiple concurrently active clients. At the time it seemed that this type of 'sharing' was the main interpretation of the re-use concept. The introduction of a new version of a library could have been a substantial undertaking because all the client applications would have to be re-built in order to re-establish their offsets into the *dynamic link-library* subroutines (loosely called *dlls*). Because subroutine libraries were not upgraded very frequently and because much of the process had been practically automated by means of *make-file* systems, this scheme was entirely feasible on the managed mainframe and minicomputer systems of the 1970's.

During the 1980's however, this state of affairs was put paid to by a number of developments, including in particular, the invention of the personal computer; the development of interactive graphics interfaces; and the adoption of object-oriented programming.

The personal computer Because the personal computer is generally just *used* and not really *managed*, the careful, professional construction and periodic reconstruction of 'the system' cannot be relied upon for updating and upgrading. Instead, applications are installed willy-nilly from a CDROM or the Internet, and at every installation a new set of dlls will be dumped into the library. Eventually (occasionally immediately) dlls which certain applications depend upon will be overwritten by new, incompatible versions. Those applications will fail unless the original dlls are re-installed at which time any subsequently-installed applications may in turn fail. The system will have entered what the Microsoft system support staff have termed 'Dll Hell'. [1, 25, 24]

Interactive graphics Managing multiple windows on the screen, and performing all the other complicated event-driven tasks required of an interactive graphics system, involves the client and its service-providing dlls in far more complex interactions than the simple subroutine call-and-return. Where it is necessary



for the two parties to exchange 'state' information in this manner, it is not really practicable for the dll to service multiple clients in the conventional way. The concept of code-sharing ceases to be a sensible runtime affair and hence becomes a design-time issue once again, as it was in the white-box days.

Object-orientation Object-oriented developers encapsulate their designs in classes and interfaces. When client developers *adopt* a class by instantiating objects, they directly (re)use the original developer's implementation; but when they *adapt* the class by subclassing, they reuse parts of the original design together with any code that's not over-ridden. Finally, when they implement an interface, it is largely the design that is reused. As a consequence of this multiplicity, when an object-oriented application executes, a heterogeneous assemblage of objects will be loaded. All their methods must mutually be able to access each other's in-scope fields and pass messages back and forth.

In the real world we may distinguish between the client developer and an array of service developers, but inside the machine, all objects are equal. Just making this work once is a clever trick towards which systems designers have directed much ingenuity. Making sure it works, run after run, build after build, under sustained maintenance throws up some subtle new problems. This paper describes the ways in which systems and language designers have tried to overcome these hurdles and approached the challenge of finding another solution to the problem of reuse.

Section 2 begins with an explanation of the fragile base class, the original maintenance time problem that arose when object-orientation collided with traditional compilation and linking techniques. Section 3 covers similar ground but in relation to more recent (dynamic) linking mechanisms, and sketches a rationale for the tools described in Section 4 and 6. DEJAVU, in section 4, is a prototype server-side environment for supporting small-scale Java class libraries. Because it was explicitly designed as a component composition and deployment environment, we have described the .NET architecture in some detail in Section 5 before outlining the design of SNAP, a client-side utility for reconfiguring .NET applications, in Section 6. Section 7 discusses some related work as reported in recent literature.

2 THE FRAGILE BASE CLASS PROBLEM

In a typical C++ implementation, a class's method declarations are indexed in a virtual function table (vtable) which subsequent classes reference in order to invoke the method [19]. Thus, a class Aircraft is declared as follows:

//Aircraft.h	vtable index

class Aircraft{
  virtual int GainHeight();
};
	0

Source files that include `Aircraft.h` are then compiled. Code is generated assuming 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	vtable index
`class Aircraft{`	
`  virtual void SndMsg(faultNo);`	0
`  virtual int GainHeight();`	1
`};`	

Any source files that include `Aircraft.h` that are not re-compiled after this modification will use at runtime 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 representation of the function in the vtable has lost the signature of the function — it is merely 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	vtable index
`class Aircraft{`	
`  virtual int LoseHeight();`	0
`  virtual void SndMsg(faultNo);`	1
`  virtual int GainHeight();`	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

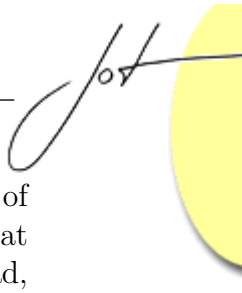
```
if (height < CRITICAL) GainHeight();
```

would crash the plane and not just the computer!

This is known as the *fragile base class problem* [17]. 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.

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. It is possible to construct implementations, which avoid or ameliorate the problem [7] but only at the cost of increased complexity



of the implementation and, in some cases, of reduced object-oriented capabilities of the language[11, 26]. By contrast, a Java *class loader* does more of the work at load-time and thus relaxes this stringent constraint on the Java compiler. Instead, for each *reference* in the source code, the Java compiler embeds *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,
- for field references, appropriate (symbolic) type information,
- for constructor invocations, symbolic references to the types of the parameters,
- 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 to resolve the references *dynamically*. 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 actually alter the link information in any way, then the loader will be able to link this new binary with all the old binaries. Such modifications 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 Table 1 [11, 26, 16].

Note that the word 'compatibility' here refers strictly to the ability of the binaries to link and run together. There is no guarantee that the new modifications will not introduce semantic errors at runtime. This has led to a distinction being drawn between *syntactic* binary compatibility (as defined above) and *semantic* binary compatibility where the program's integrity and/or the intentions of the modifier are somehow guaranteed [16]. Semantic binary compatibility will not be considered further here, and binary compatibility will be taken to mean syntactic binary compatibility.

The binary compatibility of Java programs is a by-product of Java's dynamic loading mechanism. Even though the modified classes may link correctly, a binary compatible modification can affect the client in unexpected ways. For example, sometimes a client may not experience the effect of the change until it is re-compiled. We have named these *blind* clients. Some examples of how blind clients can come about in Java and also in C# are given in Appendix A. In other cases the client's sources may not re-compile without error. We have termed these *fragile* clients, and Appendix B shows some ways in which these can come about.

	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 runtime mechanism for finding methods/fields in a class hierarchy.

Table 1: Binary compatible modifications

	Runtime		
(Re-)build time	*No effect*	*Effect*	*Error*
No effect	blissful ignorance	forgetful?	
Effect	blind	blissful	binary
Error	blind and fragile	fragile	incompatible

Table 2: Modification effects

Modification Effects

In order to understand the impact of different modifications and to exercise some control over the emergence of undesirable consequences, like blind and fragile clients, it is necessary to consider not only the nature of the modification (whether it is binary compatible or not) but also the phase at which the modification makes itself felt – immediately, at runtime, or subsequently at the next client build-time. [1]

There are three possibilities:

1. the modification may make no impact whatever (No Effect);
2. the modification may emerge as intended (Effect);
3. the modification may break the system (Error).

So we can classify any individual modification in terms of its effect as shown in Table 2. Ignoring the 'binary incompatible' modifications, which are important but not interesting, we can see the categories into which blind and fragile clients may fall.

In another category the client experiences the modification but, after being rebuilt, finds that the effect vanishes. We have not been able to find an example where this phenomenon occurs as a result of a *single* modification. Just in case somebody else does, we would like to propose the term 'Forgetful Client' for this category.

Two categories have wholly satisfactory outcomes – firstly the case where the client never experiences the modification (blissful ignorance) should occur when the modification involves adding service facilities which are intended only for new clients. In the second case (blissful) the modification is experienced immediately and it persists after the client is rebuilt. This is desirable when errors in the service module have been repaired and the corrections need to be painlessly propagated to the client applications. These two categories are extremely important because they offer service developers the opportunity, under certain circumstances, of *field-upgrading* their software at client sites without requiring any action or even awareness on the part of their clients and, most especially their clients' users. This phenomenon has been termed *dynamic evolution*.

[1] For static loading it is buildtime that happens immediately, and runtime that is subsequent.

Support for dynamic evolution

Developers need to be aware of the fact that their modifications may have different effects on their clients and that these may impact at different points in their applications' life-cycles. Having such advanced knowledge is a necessary condition for the developer to manage the deployment of service modifications and hence achieve satisfactory dynamic evolution. However, it is not sufficient. To achieve this we need tools capable of exploiting this knowledge in the service either of the developer or the client.

The precise nature of such tools depends to a large extent on the architecture of the runtime systems currently available to developers. Thus, for the Java Virtual Machine (JVM), we designed DEJAVU (the Distributed Evolution for JAVa Utility), a tool targeted at the small or medium developer who needs to maintain a modest class-library and to support clients across a network.

By contrast, SNAP (for Strong-Name Assembly Propagator) was designed to work at the sites of clients, managing upgrades by exploiting services embedded in Microsoft's .NET framework. These tools are described in detail in subsequent sections. Although the tools are radically different, they share some very broad design goals:

1. We wanted to work within the scope of programming languages as they are currently defined, and not to propose new structures, functions or programming conventions as these would serve to pass the problem on to the developers themselves.

2. We rejected any requirement which demanded that clients should subscribe to some server-side register for the purpose of achieving version control as this would pass the problem on the to clients.

3. We stipulated that the client application's *users* should experience transparency with regard to the modifications instituted by our tools. This means that it is imperative to deal sensibly with blind and fragile clients and to preempt the introduction of binary incompatible modifications.

4 DEJAVU – DISTRIBUTED JAVA VERSION CONTROL SYSTEM

The Java compiler has to identify an exact location in the file system from where, at runtime, the Java classloader can fetch the code pertaining to any particular class. If a class is modified therefore, the new version needs to be placed in the same location and thus it effectively *replaces* the original version. This arrangement cannot satisfy the developer whose library must, on the same system, service multiple clients with different versioning requirements.

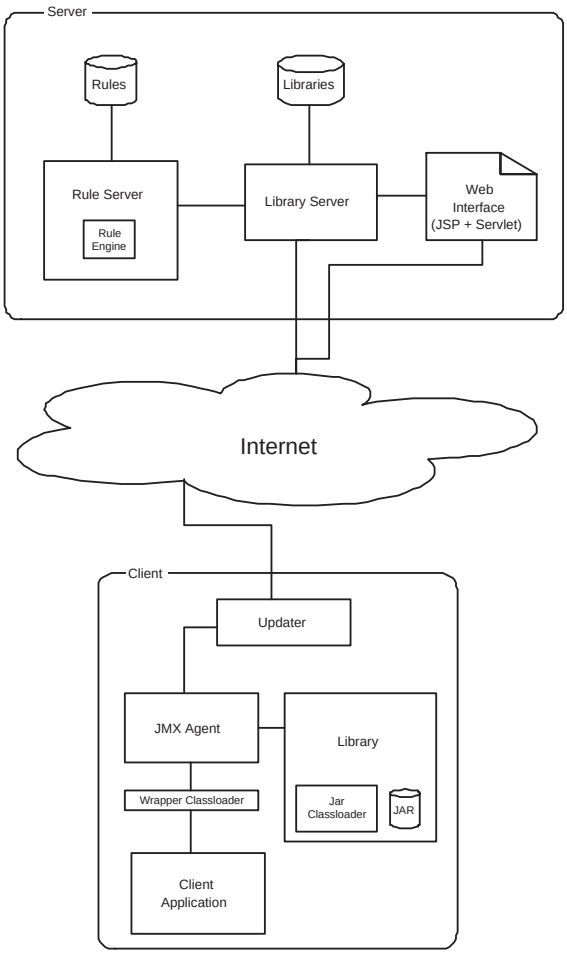


Figure 1: DEJAVU Architecture

The solution chosen in DEJAVU was to version at the level of the library as a whole (rather than to try to support different versions of a class within a single library), and to supply each client with a private copy of the library. [2] Because deployment is distributed, DEJAVU employs a client-server architecture, where the server hosts multiple versions of each library it manages and undertakes to upload an appropriate update when any client requests it. The upload consists of the entire library.

The server maintains an interactive interface for the developer. Via this a new version of a library can be downloaded. It can be explicitly tested for binary compatibility against any other version of the library present on the server, and the

[2] This solution is impractical where the library is very large or where there are a large number of different clients at the same site. An alternative solution could be to provide each client with a smarter customised classloader capable of detecting and fetching post-compilation versions of a service class. However, this would require client developers to program 'reflectively' and could, under some circumstances, make for very slow execution.

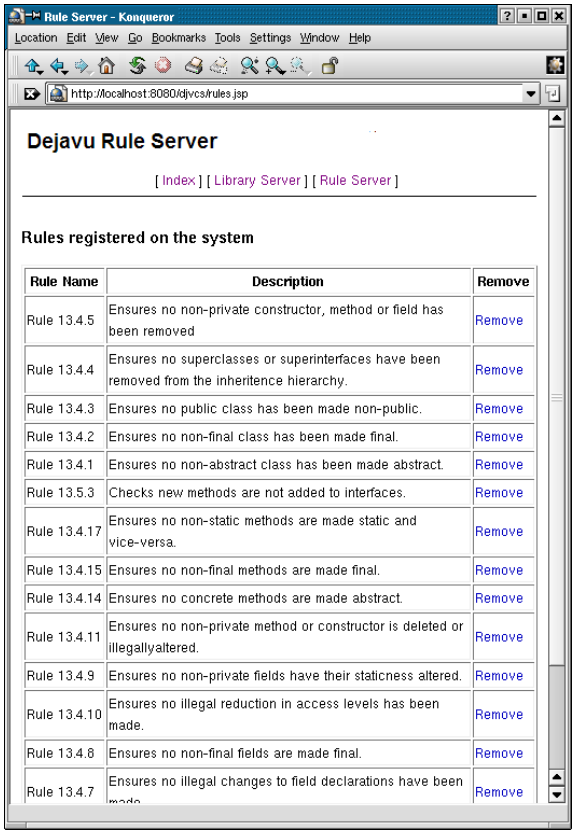


Figure 2: Rule Server Index

results of all the tests are stored. Additionally, redundant versions of the library can be removed in their entirety.

In order to use the library, the client must download an `Updater` component to manage communications with the server, and a `Library` JMX mbean. `Library` encapsulates the jar file of the client's current library version together with a custom classloader specifically configured for that version. Communications between the mbean and `Updater` and between the mbean and the client application are handled through a JMX agent – see Figure 1.

The client initiates an update request which the server fulfils by uploading the requisite jar file, if appropriate. `Updater` then initiates the update via the JMX agent. Once the old jar file has been replaced, a new classloader is constructed, configured for the new library, so that the next time the client application calls for a component, the new version will execute. If `Updater` is called whilst the client application is running then the upgrade will be a 'hot-swap'.

The server side of DEJAVU consists of a `LibraryServer` which encapsulates a repository of libraries incorporating all current versions and the results of all inter-version compatibility tests; and a `RuleServer` which the `LibraryServer` calls upon to conduct these tests – see Figure 1. The `RuleServer` uses reflection to test two

versions of a library against the binary compatibility requirements laid down in the Rules database. Currently these consist of the 30 binary compatibility 'rules' which appear in the JLS [18] (see, for example, Figure 2). Each rule is referenced by its section number within the JLS and is accompanied by methods which perform the actual tests. On the basis of the results, the RuleServer flags whether or not the two versions are binary compatible.

The Updater component of any given client can request one of two kinds of updates. The standard request supplies the server with the client's current library version number and the server will provide the most recent binary compatible jar file if one exists. If the new version incorporates any modifications which may make the client fragile these are flagged for the Updater's log, but the new version is still 'shipped'. The other kind of update is the maintenance request where the client developer wants the absolutely latest version of the library for the purposes of preventative maintenance, regardless of its compatibility status.

5 MICROSOFT .NET FRAMEWORK

.NET is a framework devised by Microsoft to promote the development of component-based applications and to enable their efficient and effective deployment [25, 12]. The core component of .NET is the Common Language Runtime (CLR) which is actually a runtime *environment*. The CLR accommodates the interoperation of components rendered into the Common Intermediate Language (CIL). Rather than interpreting these statements for a virtual machine (as with the Java Virtual Machine) the CLR uses a Just-in-Time (JIT) compiler to generate momentary native code for the local platform. Having a *common* intermediate language means that applications can compose components written in different source languages whilst the Platform Adaptation Layer (PAL), together with the Framework Class Library, makes .NET applications potentially highly portable across Windows platforms. Because C# differs from Java and because our previous studies focussed exclusively on Java, we examined the binary compatibility potential of C#. C# still causes blind and fragile clients although there are some language-dependent differences [13].

A number of the .NET design goals [30] are particularly relevant to dynamic evolution:

1. "Resolve intertype dependencies at runtime using a flexible binding mechanism." This is what makes it *dynamic*.

2. "Design runtime services to ... gracefully accommodate new inventions and future changes." This is what is meant by *evolution*.

3. "Package types into portable, self-describing units." 'Portable' means that the packages are effectively components, as defined earlier, and in .NET are referred to as *assemblies*. The 'self-describing' means that the CLR need

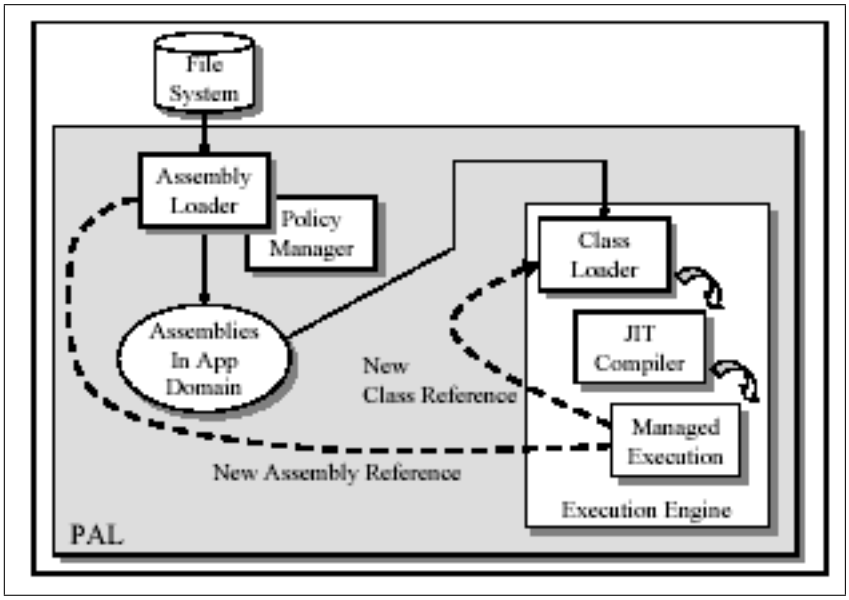


Figure 3: CLR Loading

not depend on a registry to compose components at runtime. Instead, each assembly incorporates a *manifest* which lists the resources provided by the assembly and the nature and locations of any resources it depends on from external assemblies.

4. "Ensure isolation at runtime, yet share resources." This implies a DEJAVU type repository that can hold multiple versions of a component and a runtime system that can correctly select and use the different versions appropriate for each application. In .NET this repository is known as the Global Assembly Cache (GAC) and the runtime capability is referred to as 'side-by-side' operation. The runtime address space is divided into Application Domains (*appdomains*). At runtime an Assembly Loader loads assemblies from secondary storage (or a URL) into the appropriate appdomain on demand. Assemblies loaded from the GAC are loaded into the Shared Domain where they are accessible to any running application. Other domains exist for system services and application-specific classes. Security checks are performed during the loading process.

5. "Execute code under the control of a privileged execution engine ... ". The CLR execution engine in Figure 3 performs 'managed execution' of code produced by the JIT compiler. When an executing object references an unloaded *class*, the ClassLoader loads the class from the appdomain for compilation. If the reference is to an unloaded *assembly*, the request is passed to the Assembly Loader. Code verification is performed during JIT compilation.

A .NET assembly that is capable of entering the GAC and participating in side-

by-side operations must be identified by a *strong-name*. A strong-name incorporates a name; a four-part version number divided into <major>, <minor>, <build> and <revision> parts; a 'culture' and a public key ID originating from the assembly author. Any two assemblies are regarded as distinct if there is a variation in any one of these. Thus even if two providers simultaneously update the same assembly with the same version number increment, the public key information still allows the system to distinguish between them. Assemblies which are not in the GAC (and hence are not intended for side-by-side operation) need not be strong-named. [3]

6 SNAP – STRONG-NAME ASSEMBLY PROPAGATOR

Rational Component Management

The strong-name assemblies in the GAC, which are so uniquely specified, are equally uniquely referenced in the metadata of dependent (client) assemblies. Thus, every time a client runs, it will only ever request the exact service assembly it was built against. Any improvements arising in future versions will by default be lost to the client until its next rebuild. This is deliberate:

> "Historically, platform vendors forced users to upgrade to the latest version shipped. Software developers . . . were (solely) responsible for resolving any resulting incompatibilities" [23]

and the result was dll Hell!

Instead, the default behaviour can be overridden because the Assembly Loader consults a sequence of XML 'policy' files which can be used to redirect the load operation. So

> "the .NET Framework team (puts) complete control in the hands of system administrators and developers who use the framework . . . " [23].

[3] According to the .NET documentation [30] the CLR loader will try to match the version specified in the Manifest with the component. If it can't find the exact version, the CLR will look for a close substitute version with the same major and minor numbers. The build number is considered possibly compatible, and the revision number is also called the Quick Fix Engineering number (QFE), and considered compatible. If the manifest doesn't find the exact match, it will load a version that differs only by the revision or build number if it is newer.

However, by experimenting with assemblies, we found that the QFE appeared to have been disabled by default in the official .NET releases that we looked at. The versioning algorithm used by the Assembly Resolver had been changed. For public assemblies, an exact version match was required, the resolver no longer loaded the latest build and revision numbers. The "Runtime Settings Schema" document under the ".NET framework Configuration File Schema" section of documentation showed that the UseLatestBuildRevision XML element had been removed from the config files, and it suggested that administrators use explicit redirection instead. This seems to imply that programmers may have given changes quick fix status to avoid compatibility problems when loading.

One policy file is the Application Configuration file and another is the Machine Configuration file, and these give some (manual) control to the client-side system administrator. On the developer side, there is a Publisher Policy file. Taken together, these files can no doubt provide a dynamic evolutionary pathway for any application provided that all parties with write-access to the policy files have full information about component dependencies and versions (that is, they know the information in the metadata and will act on it).

The aim of SNAP [15] was to exploit the evolutionary facilities offered by .NET to provide software support to application developers who wanted to migrate the benefits of service component updates to their own products. However, this evolution cannot occur at application loadtime (as with DEJAVU) because the assemblies are 'locked' into the appdomain when they are loaded and any attempt to interfere with the policy files looks to .NET's security system very like a 'spoofing' attack. Instead we envisaged a separate maintenance phase in which the user directs the tool to update the GAC and/or the policy files.

It will be useful to list some terms and concepts derived from the theoretical model [14] we derived:

Component A *Component* is a uniquely (strong-) named entity which requires a set of *import* services, provides a set of *export* services and maintains a set of *required* components (which collectively export those services which the component imports).

Cache A *Cache* is a set of components.

Coherent A cache is *coherent* if every service required by each component in the cache can be provided by one or more other components in the cache.

Add A component can be *added* to a coherent cache provided it is not already there and providing that adding it does not compromise the coherence property of the cache (i.e., it cannot require any service which is not, a priori, provided).

Remove A component can be *removed* from a coherent cache provided that this does not compromise the coherence of the cache (i.e., there must be other components that provide the services provided by the target component).

Where a sequence of 'physical' components (components actually stored in the cache) represents the temporal evolution of a single 'logical' component (a component designed to export a specific set of services), it is *assumed* that the ordering of the sequence can be deduced by examining component (strong-)names. [4]

[4] This should be possible if the 4-part .NET version number is used consistently. For our work we assumed a simple linear sequence.

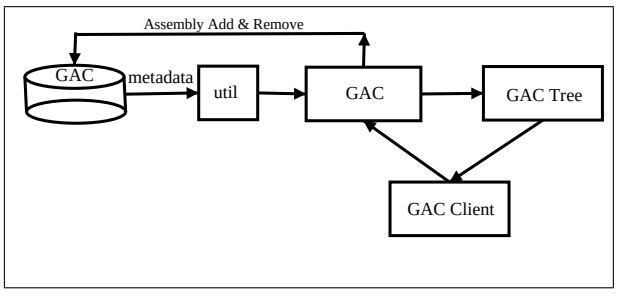


Figure 4: GAC Administration via SNAP

Tool Requirements

A tool to manage GAC assemblies in the way envisaged should:

1. allow a system administrator to *add* an assembly to the GAC. The GAC is not a flat file-system, so this requirement involves creating an appropriately-named path to a folder containing the relevant .dll file together with an .ini file holding a copy of some of the manifest.

2. allow a system administrator to remove an assembly from the GAC. It is not possible to apply a simple coherency test here since the target assembly may be explicitly referenced in the *required* metadata of another GAC assembly and hence not removable. If the target is explicitly referenced in an external application, it would be necessary to rebuild the application before it could be coherently removed.

3. allow an application developer to configure an application so that it utilises the most recent binary compatible version (relative to that application). This requirement involves creating and/or maintaining an Application Configuration file. Note that this operation obviates the need to rebuild the application mentioned above.

System Design and Use

In order to avoid the appdomain locking problem, it is necessary to extract the GAC assembly metadata without loading the assembly. This function is performed by an assembly called util which parses the byte-streams of each dll in the GAC and constructs a table recording each assembly's strong-name, its required assemblies and a list of its strong-named clients.

This table is accessed by an assembly called GAC whose role is to encapsulate the GAC. GAC exports methods which allow other components to gather assembly data and to Add and Remove assemblies from the GAC.

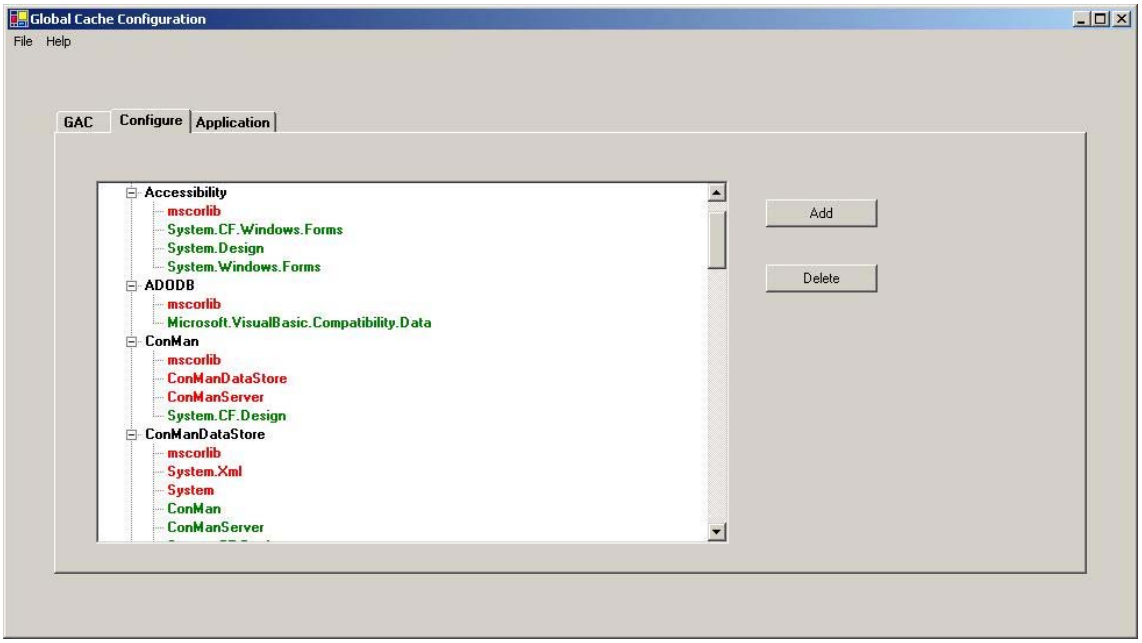


Figure 5: GAC Configuration

GACClient is the user-interface for system administrators, through which they can view the GAC and intra-assembly dependencies, via GACTree; and Add and Remove assemblies via GAC. Figure 4 illustrates the component configuration for system administrators and Figure 5 shows a screenshot of the tool (labelled Configure in SNAP). All the assemblies in the GAC are listed and the listing can be expanded to show each assembly's dependencies. The dependencies are colour-coded to distinguish between the required assemblies (red on the screen) and client assemblies (green on the screen).

The requirement for the Application Developer (requirement 3) involves the comparison of different versions of an assembly to discover whether the later one is binary compatible with the earlier one *relative* to the application in question. In order to do this it is necessary to establish whether the particular types, fields and methods imported by the application match those exported by each version.

This information is extracted using reflection. The export metadata is easily available via the 'Managed' Reflection API; however when it comes to import metadata, the Managed Reflection API does not reveal the token values needed to compare with the corresponding exports. Therefore, the 'Unmanaged' Reflection API is used to examine the assembly header files, and a Wrapper assembly was devised to bring this unmanaged component into the managed fold. The comparison is done in the BinaryCompatibilityChecker assembly (Figure 6), starting with the application imports and recursing through the entire dependency tree. Any legitimate redirections are recorded in an Application Configuration file which is eventually deposited in the application's source directory.

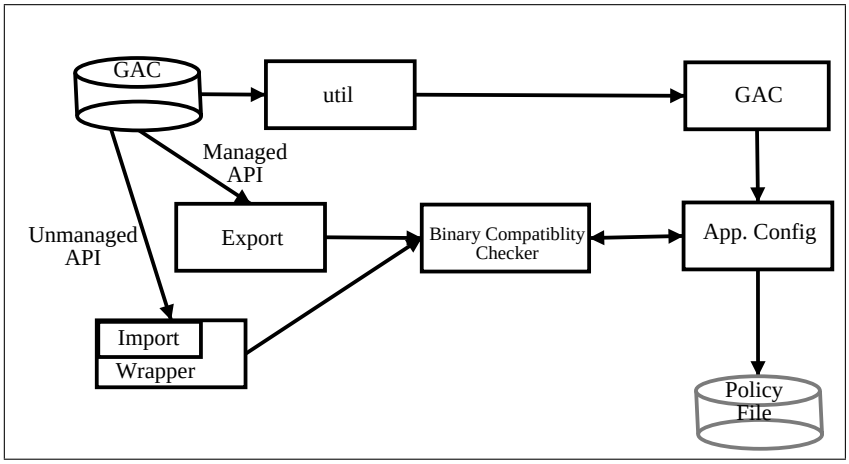


Figure 6: Application Configuration Client

7 RELATED WORK

As observed by Stuckenholz [29], advances aimed at improving the evolutionary processes in component development can occur in three areas — amongst the concepts embodied in the underlying component model; within the scope and constructs of the programming languages used to design and build components, or in the frameworks within which components are deployed, composed, and executed on systems. Accordingly, this section seeks to document some of the recent work in each of these categories.

Some of this work has been motivated by the problems of component adaption or dynamic update. Component adaptation is concerned with getting components to collaborate through interfaces which are not directly compatible. This is usually managed through*connectors* which may be components in their own right, or intelligent channels of one sort or another. Dynamic update (or hot-swapping) is concerned with updating elements of an application while it is running.

Most researchers seem to share our reluctance to attempt to seek consensus on a component model more elaborate than that defined by Szyperski [31]. However, Steyaert et. al. [28] have proposed *reuse contracts* as a means of making the component developer's design commitments explicit, and Mezini [22] developed a *smart composition* model as an extension of the Smalltalk runtime which could detect, at the client site, modifications that rendered a service module incompatible. Brogi et. al. [6] have proposed that behavioural information should be specified in component interfaces in the form of *sessions*. By giving these sessions a formal description it is possible to generate automatically the 'response' session (dual) of an idealized interacting counterpart component. This gives a formal definition to the term *compatibility* and makes it possible to determine whether two components are indeed compatible. They go on to show how in the case of incompatible components, it is possible to generate an *adapter* component to act as a go-between [4]. The creation

of such adaptors could be one way for an evolving component to continue to serve former clients.

Other formal work on dynamic linking [26, 10, 9] has been extended by Malabarba et. al. [21] to derive a definition of a (potentially type-safe) *dynamic class* that can be hot-swapped. The definition of type-safety used here is marginally more restrictive than the Java binary compatibility definitions given earlier [18], largely because it does not deal explicitly with some of the Java accessibility modifiers, such as `final`. Other workers [27, 5] recognized that the evolutionary behaviour of components can be classified at the level of the component model by versioning. In particular Brada [5] has used the SOFA CDL to classify modifications according as to whether the new specification was

- $=$ an exact match of the old one
- $\supset$ a specialization of the old one
- $\subset$ a generalization of the old one
- $\triangle$ a mutation of the old one

Since the SOFA specification is decomposed into Provided, Behavioral, and Required parts, it is possible to distinguish between degrees of compatibility as follows:

Compatibility	Provided	Behavioral	Required
Equivalent	$=$	$=$	$=$
Strongly Compatible	$\supseteq$	$\subseteq$	$\subseteq$
Conditionally Compatible	$\supseteq$	$\supseteq / \triangle$	$\supseteq / \triangle$
Incompatible	$\subset / \triangle$		

Brada's scheme is held to be the most complete analysis of component versioning to date [29].

Ideally key elements of any component model would be explicitly incorporated into general programming languages, although in reality this is not the case [8]. Researchers thus have the option of proposing language extensions where broad acceptance is highly unlikely, or of proposing a special approach to program development, supported by an appropriate class library. This was the strategy taken in Section 4 with DEJAVU. Although their motivation is different, the same line was taken by da Silva et. al. [8] who have developed a Java connector package which, if incorporated with an application whose implementation embeds the appropriate interface, can generate standardized connector components to manage inter-component interactions. They plan to automate the process whereby a component-based architectural description can be mapped to a Java implementation.

Bialek et al. [2, 3] have addressed the problem of dynamic update. Their analysis identifies five problem areas that all hot-swapping solutions must address, as follows:

1. *Compatibility* – This refers to the problem areas we have been considering under the designation of syntactic and semantic binary compatibility.

2. *Class Identity* – This refers to the need to be able to maintain (and recognize) multiple versions of a component.

3. *Class Unloading* – This is largely a runtime issue.

4. *State Transfer* – Another runtime issue.

5. *Performance* – This refers to the runtime overhead which dynamic evolution might incur. This can be an issue for systems like DEJAVU, but it is a secondary consideration here.

On the whole dynamic update clearly embraces almost all of the concerns of dynamic evolution, together with a number of runtime specific issues. Since it tends to involve only a single runtime instance, however, hot swapping researchers tend to consider neither the problem of serving multiple clients on a single system nor of coping with distributed systems, and these are the essence of dll hell.

In their paper, Bialek et al. [3] outline a method of partitioning monolithic applications into independently updateable 'components'. Each component is enveloped in a *mediator* wrapper, which provides the Class Identity sought in 2 above, and helps to accommodate multiple component versions at runtime (see 3). In addition, a modified Java classloader is employed which, by altering the class bytecode, manages inter-partition transactions via proxies. By this means the State Transfer problems of 4 can be solved.

Directly motivated by hot-swapping considerations, Malabarba et al. [21] describe their DVM (Dynamic (Java) Virtual Machine). This works with a custom `DynamicClassloader` to allow hot-swapping of so-called *dynamic classes* which have been carefully (and formally) defined to satisfy the Compatibility criteria (see 1). Unlike the standard model, `DynamicClassloader` can *reload* or *replace* previously-loaded classes, thus tackling Class Identity (see 2). Firstly, the DVM works with the classloader to implement these features and also tackles the issues under 3 (using a mark-and-sweep update algorithm allied to garbage-collection considerations). The paper also considers performance issues mentioned at 4 together with a number of additional security issues raised by the implementation of hot-swapping technology (since malicious code can fairly easily masquerade as an 'update' to some existing class, especially in a distributed environment).

Another example of the framework approach is described by Kon et al. [20] who have introduced an *automatic configuration* architecture. The primary motivation for this is concerned with the deployment of (largely) hardware resources, but the framework provides mechanisms for dynamic evolution and even (in principle) hot-swapping. Their architecture implements a component repository similar to early versions of SNAP [15]. Each component incorporates metadata which documents pre-requisite specifications – although there is no automatic means of constructing this metadata.

At runtime, the metadata for each component is used to construct a unique *component configurator*. This is an active component in its own right that manages the dependencies exhibited by its allied component and that helps to synchronize resource usage amongst components via a number of system-wide services. The active elements of each configuration need to be manually programmed. The configurators play a role similar to connectors except that each component has its own configurator. The Auto Configuration service permits the association of upgrades only to those application currently requiring the upgrade (so-called 'pull' technology). This leads to significant efficiencies because it is not necessary to download the upgrade to all nodes and, since it is demand-driven, it is also not necessary to keep a register of all of a component's clients. Although dependencies can be specified in a number of different ways, it is not stated in the paper how the directory service distinguishes between different versions of the same component, or establishes which version is the most appropriate candidate for any client.

Stuckenholz [29] also proposes a component repository in which multiple versions of interdependent evolving components are organized as a *version readability graph*. Any application's component dependencies can be represented as a subgraph of this directed cyclic graph, and any upgrade will involve finding a more up-to-date compatible, or at least minimally incompatible, subgraph. Considerations of how this can be achieved lead him to an *intelligent component swapping* system. Work on this approach is continuing.

8 CONCLUSION

The problems that emerge from the field of dynamic evolution overlap somewhat with those of component adaptation and hot-swapping. Solutions to all these sorts of problems can be approached through the elaboration of our conceptual model of components per se; through refinements to our programming languages; or through enhancements to our component frameworks.

The tools described in this paper fall into the last category. DEJAVU is a server-side repository that stores successive generations of an evolving Java class library, whilst SNAP provides some GAC housekeeping facilities for .NET clients. Their efficacy will be tested and their functionality improved with future field testing.

Software maintenance is still a complex, expensive and comparatively manual business compared with many other areas of software engineering. The development of supporting tools can ameliorate this situation. However, more work needs to be done in the areas of *semantic* binary compatibility and of *automatic* versioning before these tools are likely to receive much recognition or acceptance.

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 Drossopoulou and the Slurp research group. We thank all the Imperial College students who over the years have implemented our ideas, helping us to greater understanding. These include Shakil Sheikh, Vladimir Jurisic, Miles Barr, and Dilek Kayhan.

REFERENCES

[1] R. Anderson. The end of dll hell. In *MSDN Magazine*, http://msdn.microsoft.com/, January 2000.

[2] Robert Bialek and Eric Jul. A framework for evolutionary, dynamically updatable, component-based systems. In *The 24th IEEE International Conference on Distributed Computing Systems Workshops*, pages 326–331, Hachioji, Tokyo, Japan, March 23-24 2004.

[3] Robert Bialek, Eric Jul, Jean-Guy Schneider, and Yan Jin. Partitioning of Java Applications to Support Dynamic Updates. In *11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 616–623, 2004.

[4] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74(1):45–54, 2005.

[5] Premysl Brada. Component change and version identification in sofa. In J. Pavelka, G. Tel, and M. Bartoek, editors, *SOFSEM'99: Theory and Practice of Informatics: 26th Conference on Current Trends in Theory and Practice of Informatics*, volume 1725 / 1999 of *LNCS*, Czech Republic, 1999. Springer-Verlag GmbH.

[6] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Behavioural types and component adaptation. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology: 10th International Conference, AMAST 2004*, volume 3116 / 2004 of *LNCS*, Stirling, Scotland, UK, July 2004. Springer-Verlag GmbH.

[7] G. Cohen, J. Chase, and D. Kaminsky. Automatic Program Transformation with JOIE. In *USENIX Annual Technical Symposium*, 1998.

[8] Moacir C. da Silva Jr, Paulo A. de C. Guerra, and Cecilia M. F. Rubira. A java component model for evolving software systems. In *IEEE International Conference on Automated Software Engineering, ASE'03*, volume 18. IEEE, 2003.

[9] S. Drossopoulou. An Abstract Model of Java Dynamic Linking, Loading and Verification. In *Types in Compilation*, September 2001.

[10] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java Binary Compatibility? In *Proc. of OOPSLA*, pages 341–358, 1998.

[11] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 5(1), January 1999.

[12] F. Redmond (ed.). Microsoft .NET Framework. In *MSDN*, http://msdn.microsoft.com/ netframework/, January 2004.

[13] S. Eisenbach, V. Jurisic, and C. Sadler. Feeling the way through DLL Hell. In *The First Workshop on Unanticipated Software Evolution USE'2002*. http://joint.org/use2002/proceedings.html, June 2002.

[14] S. Eisenbach, V. Jurisic, and C. Sadler. Managing the Evolution of .NET Programs. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems, FMOODS'2003*, volume 2884 of *LNCS*, pages 185–198. Springer-Verlag, Nov. 2003.

[15] S. Eisenbach, D. Kayhan, and C. Sadler. Keeping Control of Reuseable Components. In *IFIP/ACM Working Conference on Component Deployment*, LNCS. Springer-Verlag, June 2004.

[16] S. Eisenbach and C. Sadler. Ephemeral Java Source Code. In *IEEE Workshop on Future Trends in Distributed Systems*, December 1999.

[17] S. Danforth I. Forman, M. Conner and L. Raper. Release-to-Release Binary Compatibility in SOM. In *Proc. of OOPLSA'95*, October 1995.

[18] G. Steele J. Gosling, B. Joy and G. Bracha. *The Java Language Specification*, pages 251–273. Addison Wesley, 2 edition, June 2000.

[19] J. Reuter J. J. Hunt, F. Lamers and W. F. Tichy. Distributed Configuration Management Via Java and the World Wide Web. In *In Proc 7th Intl. Workshop on Software Configuration Management*, 1997.

[20] Fabio Kon, Jeferson Roberto Marques, Tomonori Yamane, Roy H. Campbell, and M. Dennis Mickunas. Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems. *Software: Practice and Experience*, 35(7):667–703, May 2005. John Wiley & Sons, Inc. Publisher.

[21] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361, London, UK, 2000. Springer-Verlag.

[22] M. Mezini and K. J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proc. of OOPSLA*, pages 97–116, 1998.

[23] Microsoft. Versioning, Compatibility and Side-by-Side Execution in the .NET Framework. In *MSDN Flash Newsletter*, http://msdn.microsoft.com/netframework/technologyinfo, 2003.

[24] M. Pietrek. Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, October 2000.

[25] S. Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, November 2001.

[26] S. Eisenbach S. Drossopoulou and D. Wragg. A Fragment Calculus: Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *Logic in Computer Science*, pages 147–156, 1999.

[27] P. Sewell. Modules, Abstract Types, and Distributed Versioning. In *Proc. of Principles of Programming Languages*. ACM Press, January 2001.

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

[29] Alexander Stuckenholz. Component evolution and versioning state of the art. *SIGSOFT Softw. Eng. Notes*, 30(1):7, 2005.

[30] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly Press, 2003.

[31] C. Szyperski. *Component Software – Beyond Object Oriented Programming*. Addison-Wesley / ACM Press, 2 edition, 2002.

A BLIND CLIENTS

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 the following situation which can arise when programming in C#.

```
public class Coffee{
  public string purity="pure Arabica";
}
public class Columbia :  Coffee{
```

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

`SuesDiner` is the client. When compiled and executed it outputs `Coffee - pure Arabica` Now suppose that `Columbia` were modified as follows

```
public class Columbia :  Coffee{
  new public string purity = "with chicory";
}
```

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

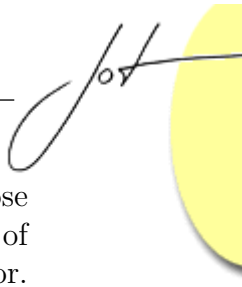
```
public class ChrisCafe{
  public static void main(string[] args){
    string quality=new Columbia().purity;
    System.Console.WriteLine("This coffee is " + quality);
  }
}
```

will produce `This coffee is cut with chicory`. However, `SuesDiner` is still bound to the old version of `Columbia` 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`.

In object-oriented languages fields are shadowed and methods over-ridden. The runtime system employs a *dynamic* dispatch mechanism to locate the hierarchically closest method for any given object and this evades the blind client problem. Versions of the Java Development Kit subsequent to v.1.4.0 resolve field references similarly so that Java clients can not become blind through shadowed fields.

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 [18].

```
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");
  }
}
```

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.

C# has no `final` keyword. Instead, the `readonly` keyword marks a field as constant after initialization. When the above code is translated into C# and executed, the `Test` client always displays the current (constant) value.

B FRAGILE CLIENTS

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

```
public class 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

```
public class ChrisCafe{
  public static void main(string[] args){
    int percent=new Columbia().purity;
    System.Console.WriteLine("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 `Coffee - pure Arabica`.

However, if `SuesDiner` were to be compiled the error

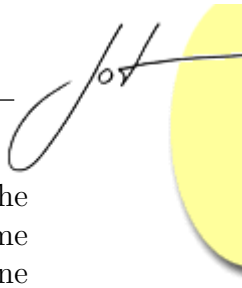
```
Cannot implicitly convert type 'int' to 'string'
```

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 Java 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 assertion to the contrary [18], this makes for a fragile client.

In C# the client developer has the choice of using the keyword `override` (the local method will conform to the inherited access restrictions) or `new` (the local method can define its own access parameter).

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. 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.
```

In the equivalent C# test, the JumpJet assembly throws a TypeLoadException at runtime so it is not really a binary compatible change.

ABOUT THE AUTHORS

Susan Eisenbach is a Professor in the Department of Computing, Imperial College London. She can be reached at s.eisenbach@imperial.ac.uk. See also http://www.doc.ic.ac.uk/~ sue.

Chris Sadler is a Principal Lecturer in the School of Computing Science, Middlesex University, London. He can be reached at c.sadler@mdx.ac.uk. See also http://www.cs.mdx.ac.uk/staff/profiles/c_sadler.html.