ABSTRACT

Dynamic linking provides a way to share code components across applications and across the network. But managing these relationships as new code versions are developed gives rise to a range of problems well recognized by system support personnel. Operating system and language designers attempt to design solutions to these problems into the latest versions of their products and we have applied some tests to see if they have succeeded.

1 INTRODUCTION

The dynamic link library (DLL) was invented to allow applications running on a single system to share code. Sharing code in this way confers two distinct benefits. In the first place applications both when running in memory and when stored on disk are likely to be smaller because the duplicated code has been eliminated. In the second place, applications accessing common code share a common functionality so they all do certain things in the same way. This is good for the users – who get a reliable and predictable user interface. It is also good for the operating system, which can provide access to certain facilities through narrow and well-defined interfaces.

On the whole, DLLs work very well, as long as once written, they never get altered. As soon as one gets altered, a variety of problems may arise.

1. Any new application that gets built will not work unless the new version of the DLL is present. For this reason, at setup time, it is usual to download the required DLL, overwriting any DLL present with the same name.

2. If the new version is not backward compatible with old version, then as soon as the new version appears on the system, some of the old applications are likely to fail.

3. If the system user builds an application that actually requires the original version, this will be downloaded and overwrite the newer version, so a whole set of recently installed applications will fail (unless the old version is forward compatible!).

So, for any individual system that uses DLLs, as soon as something is changed (typically by building a new (or upgraded) application) it is only a matter of time before the user will encounter “DLL hell” [p00,p01,w98]. Many Microsoft support personnel report [a00] DLL hell as the single, most significant user problem that they are called upon to respond to, and Microsoft has struggled to find a way to design its way around the problem. The first step was to introduce a versioning system so that, when a new application was installing itself, it could check that its version of the DLL was not an earlier version than the version currently installed on the system [s98]. If you are a responsible software developer and you are confident that your software will work on anybody’s system, then maybe you will check the version numbers. If, on the other hand, you are a ‘rogue’ developer, or you believe that your customer should pay you if your product runs and pay somebody else to solve his system problems, then you are better off ignoring the version checks.

The next design solution cropped up in Windows 2000. The approach was on two fronts. Firstly, a few thousand DLLs were earmarked as protected DLLs, which the file protection system (WFP) will guard. If, when a new application is being built, an attempt is made to download a DLL of an earlier or later version than the currently installed one, the setup system will ignore this. If there is no currently
installed version of the DLL, the setup system will install the appropriate version from the home system’s \winnt\..\DLL cache, regardless of which version the application actually requires.

Protected DLLs can only be updated by a service pack update. Thus the computer system user (via Microsoft) controls the DLLs, which exist on his system and developers must take responsibility for delivering capability on whatever systems their customers have. This is fair enough for Microsoft and developers closely associated with Microsoft, whose functionality (via the DLLs) can be protected in this way. Everybody outside this circle loses the benefits of the ‘common functionality’.

The second front opened by Windows 2000 loses the benefits of sharing. So-called ‘side-by-side’ DLLs can be created by associating a specific instance of a specific version of a DLL with an application. Other applications may use the same version, but will not be sharing the same instance. Alternatively, they may use a different version, so the functionality will cease to be common. These mechanisms help to solve the problem of DLL hell and make life more peaceful for Microsoft support personnel, but they do so at the expense of some of those sharing advantages that DLLs were originally invented to exploit.

Along with some other researchers[...] we have studied dynamic linking from the point of view of a different kind of sharing – namely, software re-use through subclassing. This is not about applications sharing code on the same system but about developers sharing code distributed across the network. This approach offers the same advantages of commonality of functionality as DLL’s and, for those object-oriented languages that permit it, exploits a similar sort of dynamic linking mechanism. However, it also suffers from the same sort of maintenance and versioning problems as DLL hell.

We looked specifically at the evolution of Java programs distributed in the manner described above[es99,es01]. We considered clients that were binary compatible with an evolving class library. Those clients will link with the new version of the library without requiring any re-compilation so the user can reasonably expect the client to run correctly. However, we found some cases where although the client continued to run, it was not affected by the modifications to the library. We called these blind clients.

We also found cases where the client would run correctly until it was re-compiled – i.e. it was binary compatible but not actually source compatible. This is an uncomfortable position for the client maintainer to be in, and we termed these clients fragile clients.

Microsoft’s latest object-oriented language is C# and it is designed to run on the .NET platform[c#02,m01,wh02]. The Common Language Runtime manages dynamic linking by referring to metadata (called the manifest) of each component (called an assembly). By enforcing version support in the manifest, and by enabling side-by-side running of assemblies at the platform level,.NET seeks to overcome the problem of DLL hell. Each manifest contains a four-part version number (major.minor.build.revision) that the CLR uses to enforce version policy. The CLR policy decides what versions are compatible or incompatible when an application goes to execute a file. According to the papers on the Microsoft website, by default, 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 we have observed changes to this Microsoft policy. By experimenting with assemblies, it can be seen that the QFE is disabled by default in the beta 2 / final .NET releases. The versioning algorithm used by the Assembly Resolver of the common language runtime has been changed. For public assemblies, an exact version match is now required. The resolver no longer loads the latest build & revision numbers. The "Runtime Settings Schema" document under the ".NET framework Configuration File Schema" section of documentation shows that the UseLatestBuildRevision XML element has now been removed from the config files, and that it is suggested for administrators to use explicit redirection instead. That Microsoft have changed their policy for choosing which version to load, seems to imply that programmers may have given changes quick fix status when loading them might cause problems. Whether the current algorithm for choosing which dll to load is trouble free only time will tell. We decided to test C# versions of our Java test programs to see whether our blind and fragile clients had been eliminated as well.
2 SHADOWED FIELDS

Blind Clients

Adding a new field to an existing class is a binary compatible modification in both Java and C#.
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.

In the example in Figure 1, Coffee and Columbia have been written by a library developer and
SuesDiner by a client developer. The Java is compiled with j2sdk1.4.0 [j02] and the C# compiled into a library assembly
component using the commands

csc /t:library coffee.cs
   /reference:coffee.DLL sue.cs

When either SueDiner.class or sue.exe are executed, they output

   Coffee - pure Arabica

   Coffee - cut with chicory

![Figure 1 : Inheriting from a superclass](image)
Figure 2: Overriding a field

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

```csharp
namespace Coffee {
   public class Columbia : Coffee {
      new public int purity = 100;
   }
}
```

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

```csharp
namespace Coffee {
   public class VladsVenue {
      public static void Main(string[] args) {
         string percent = new Columbia().purity;
         System.Console.WriteLine("Serving Arabica of purity " + percent + ">%.");
      }
   }
}
```

Figure 3: Overriding a field with a different type

Now suppose that Columbia were modified as in figure 2. The original purity has been shadowed and any newly compiled reference will be resolved in Columbia. Then executing ChrisCafe 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

In both the Java and the C# versions, SuesDiner is blind to the modification until re-compilation, when it will be bound to the new purity.

When Sun released a beta compiler jdk1.4, it exhibited different behaviour. Using the beta version executing SuesDiner produced

> Coffee – cut with chicory

Whereas running sue.exe still prints

> Coffee – pure Arabic

It is puzzling why Sun decided to revert to binding the field cup in SuesDiner to that available at compile time rather than that available at runtime.

Two differences between the Java and the C# versions of this code can be seen from the code in figure 1. Whereas Java uses the directory structure for visibility in C# what is visible is included as a namespace. The default access modifier in Java is `public` whereas it is `private` in C#. Both of these features make C# programs longer than the equivalent Java code.

The Fragile Client

A fragile client is a client binary that will link to a modified library binary, but which cannot subsequently be re-compiled from a single set of sources. Starting with the example in figure 1 suppose that the library developer modifies Columbia as in figure 3. The original purity has been shadowed
by a variable with the same name but a different type. The Java Language Specification categorizes this change as a binary compatible change since it is just the addition of a field to a class. The class \texttt{VladsVenue} (in figure 3) could be written by a client application developer and when executed (either the Java or the C#) produces

\begin{verbatim}
Serving Arabica of purity 100%.
\end{verbatim}

when run, while if the original \texttt{SuesDiner} is executed, its version of \texttt{purity} is still bound to the \texttt{Coffee} class so that it will once again output

\begin{verbatim}
Coffee - pure Arabica
\end{verbatim}

However, if \texttt{SuesDiner} were to be compiled the errors:

\begin{verbatim}
Java:   incompatible types
        found : int
        required: java.lang.String
        String cup = new Columbia().purity;
\end{verbatim}

\begin{verbatim}
C# :    Cannot implicitly convert type 'int' to 'string'
\end{verbatim}

will occur. So in either language, if its two clients are to undergo their own maintenance, the \texttt{Coffee} library hierarchy cannot simultaneously honour its contracts with both.

So both Java and C# behave identically for this situation. They both treat fields and methods differently. If a programmer uses a field from a class and that field is actually retrieved from the class’ superclass, then if later on the class is extended to include the requested field, the field from the superclass will still be accessed, rather than the new field. If later on the programmer recompiles all the code, only then will the new field become visible.

3 COMPILE-TIME CONSTANTS

In Java, the keywords \texttt{final} and \texttt{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 example in figure 4 is taken from the \textit{Java Language Specification} [7,8].

When \texttt{Test} is compiled and run in the Java program, the output

\begin{verbatim}
debug
\end{verbatim}

is produced.

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

\begin{verbatim}
!debug
\end{verbatim}

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

\begin{verbatim}
debug
\end{verbatim}

If the maintainer realized the problem and modified \texttt{Flags} once again, so the keyword \texttt{final} were removed, then a new client, \texttt{MyFlag} say, could be written. 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 \texttt{final} when the values are not truly constant is to prevent clients like \texttt{MyFlag} from overwriting values. In Java, this is better done by means of \texttt{private static} variables with bespoke access methods.
Java

class Flags {
    final static boolean debug = true;
}

class Test {
    public static void main(String[] args) {
        if (Flags.debug) 
            System.out.println("debug");
        else 
            System.out.println("!debug");
    }
}

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

class Flags {
    final static boolean debug = false;
}

class Test {
    public class Flags {
        public readonly static bool debug = true;
    }
}

C#

namespace Test {
    public class Flags {
        public readonly static bool debug = true;
    }
}

namespace Test {
    public class Flags {
        public readonly static bool debug = false;
    }
}

namespace Test {
    public class Test {
        public static void Main(string[] args) {
            Flags.debug = !Flags.debug;
        }
    }
}

namespace Test {
    public class Test {
        public static void Main(string[] args) {
            if (Flags.debug) 
                System.Console.WriteLine("debug");
            else 
                System.Console.WriteLine("!debug");
        }
    }
}

namespace Test {
    public class Test {
        public static void Main(string[] args) {
        }
    }
}

In contrast to Java, C# has no final keyword. C# prevents the overwriting of values through a readonly keyword. This keyword marks a field as constant after being initialized. A readonly field can only be changed by the instance constructor (if the field is non-static) or by the type initializer (if the field is static).

The first Java example in this section was ran in C# where the final keyword was replaced by a readonly keyword, and (as expected) experienced no problems. A change in Flags is immediately propagated to the Test client, without recompiling Test. The field cannot be changed, as was done in the Java example containing MyFlag. When we tried altering the value of Flag we couldn’t. Compiling the client assembly MyFlag throws a compile-time error with the message

A static readonly field cannot be assigned to, except in a static constructor or a variable initializer

<table>
<thead>
<tr>
<th>Field Contract Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initonly</td>
<td>Marks a constant field</td>
</tr>
<tr>
<td>Literal</td>
<td>A metadata field. No memory is reserved for this field</td>
</tr>
<tr>
<td>Static</td>
<td>Static field</td>
</tr>
<tr>
<td>Notserialized</td>
<td>Field is not serialized with other fields of the type</td>
</tr>
</tbody>
</table>

Figure 4 : Changing the value of a constant

Figure 5 : Field contract attributes supported in the CLI
This mechanism, along with a similar mechanism for non-static readonly fields, successfully prevents a maintainer from overwriting a constant field.

Since all code written in .NET-compliant languages is compiled into an intermediary language (IL) representation, the keywords applied to types and type members in a .NET-compliant language are translated into metadata attributes. These attributes are defined in the Common Language Infrastructure specification – a superset of features required by all .NET-compliant languages. Figure 5 lists the CLI field contract attributes.

For example, the Java field declaration

```java
final static boolean debug = true;
```

translates into the following CLI metadata declaration

```java
field public static literal bool debug = bool(true)
```

The CLI literal attribute, along with the bool(true) metadata initializer, stores the value of a static field in its metadata declaration. The literal attribute can be applied only to static fields.

### 4 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 open access. 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 library developer creates a class Coffee and a client developer then writes Homebrew as in figure 6. When the Java program Homebrew is executed it produces the output

3 spoons of sugar

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

Figure 7: Restricting access to library classes in C#

C# was designed in the light of the difficulties that Java developers might have with access modifiers. Inherited methods can be hidden using the keyword new. Method hiding results in non-virtual method invocation.

Changing the Coffee class to make the adulterate method public rather than protected leads to the same problem experienced by Java programmers – re-compiling the client assembly will result in compile-time error and the message

Cannot change type access modifiers when overriding ‘public’ inherited member

C# supports virtual methods. This makes use of method overriding, resulting in method dispatches based on runtime object types.

When a developer creates Coffee and a client then writes Homebrew in the example in figure 7, then any change to the Coffee class to make the adulterate method public rather than protected would cause no problems. The example programs can be successfully recompiled unlike in Java or in C# when virtual and override methods have been used. So the C# compiler does generate compile-time errors when a type member is replaced with a type member with same signature but more restricted access.
It is interesting to note that the CLI metadata grammar provides a `final` method contract attribute. This attribute is applicable only to methods marked as `virtual` and is used to prevent the virtual methods from being overridden by methods of the same signature defined in subclasses. Clearly, this is a useful feature for maintainers who wish to prevent library users from modifying method functionality.

### Figure 8: Adding a method to an interface

It is interesting to note that the CLI metadata grammar provides a `final` method contract attribute. This attribute is applicable only to methods marked as `virtual` and is used to prevent the virtual methods from being overridden by methods of the same signature defined in subclasses. Clearly, this is a useful feature for maintainers who wish to prevent library users from modifying method functionality.

## 5 INTERFACES

As a final example of the fragile client, consider a situation (see figure 8) where one programmer develops an interface and a second developer implements it in a class. Class `Jet` implements the interface `StatusReport` by defining the method `showHeight`. Then a program `JumpJet` is written which instantiates `vtol`, an object of type `Jet`.

Subsequently, the original programmer introduces a new method `showSpeed` into `StatusReport`. Adding a method to an interface is a binary compatible change[JLS]. 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(double) in Jet class Jet implements StatusReport{
```

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 CLI, modifications to interface specifications have a severe effect on binary compatibility.
The C# test consists of three assemblies. Assembly JumpJet references assembly Jet. Both of these assemblies reference (depend on) assembly Status. Executing the JumpJet assembly will print

Flying at 4590

Adding a showSpeed method to interface StatusReport in assembly Status, and recompiling Status leads to the following problems:

1. Executing the JumpJet assembly throws a TypeLoadException because the method showSpeed could not be loaded in the existing declaration of type Jet

2. Re-compiling the Jet assembly throws a compile-time error because the interface member showSpeed is not implemented by type Jet

This example demonstrates that the Common Language Runtime detects failure of a type to fully implement an evolved interface specification. The Java solution, which ignores methods added to interfaces until a program tries to access the new method produces fragile clients.

6 CONCLUSIONS

We looked for evidence of blind and fragile clients in C#. Although our Java examples are not exhaustive, they do show similarities and differences between the two languages. C# has avoided the problem of compile-time constants by introducing the readonly keyword and better run-time checking helps to provide improved interface integrity. Nonetheless, it also seems clear that the Common Language Runtime resolves fields and methods differently, to the detriment of shadowed fields.

One day maybe a language will be designed whose syntax elegantly avoids the pitfalls and whose runtime system can guarantee the integrity of distributed systems. In the meantime, we need to design tools to help library and application developers to achieve these goals.

7 REFERENCES

