

On the Integration of the Classbox Concept in the .NET Framework

Extended Abstract

Andre Lokasari, Hua Ming, and Markus Lumpe

Department of Computer Science
Iowa State University
113 Atanasoff Hall
Ames, IA, 50011-1040, USA

`andli241, hming, lumpe@cs.iastate.edu`

TR #06-15
June 2006

Invited talk at Lang.NET 2006, Microsoft Corporation, Redmond, WA, July 31 — August 2.

On the Integration of the Classbox Concept in the .NET Framework

Andre Lokasari, Hua Ming, and Markus Lumpe
Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA-50011, USA

{andli241,hming,lumpe}@cs.iastate.edu

Bergel et al. [1] have recently proposed *classboxes*, a new module system that defines a packaging and scoping mechanism for controlling the visibility of isolated extensions to portions of class-based systems. Besides the “traditional” operation of *subclassing*, classboxes also support the *local refinement* of imported classes by adding or modifying their features without affecting the originating classbox. Consequently, the classbox concept provides an attractive and powerful framework to develop, maintain, and evolve large-scale software systems and can significantly reduce the risk for introducing design and implementation anomalies in software systems [1]. The key attributes of the classbox concept can be summarized as follows:

- A classbox defines an *explicitly named scope* within which classes, methods, and variables are defined.
- A classbox supports the *local refinement* of imported classes by adding and/or modifying their features without affecting the originating classbox.

We have developed an approach to seamlessly incorporate the classbox concept into the .NET framework [2]. More precisely, we have defined a “classbox-aware” dialect of the C# language in which we use a modified `using` directive to denote the local refinements of imported classes. Furthermore, we have also introduced the notions of *logical* and *physical* structure of a classbox. While these concepts do not change the underlying semantics of the classbox model, they provide us with an effective means to separate the program interface of a classbox from its implementation. The logical structure of a classbox defines a *namespace* to specify the import of classes, the introduction of subclasses, and the extension of classes. The physical structure of a classbox, on the other hand, identifies the assemblies that contain the executable code that is defined by the logical structure of a classbox.

A key aspect of our approach is that a growing number of modern programming systems compile program code into a platform-independent representation that is executed in a *virtual execution system*. In the case of the .NET framework this virtual execution system is the *Common Language Infrastructure* (CLI), which provides an *abstract machine* to execute *managed* code. The major benefit of a virtual execution system is that the concrete layout of classes is not specified. This decision rests with the implementation of the virtual execution machine or a corresponding just-in-time (JIT) compiler. In order to enable this technique, the CLI utilizes a combination of *Intermediate Language* (IL) bytecode and *metadata*. Metadata provides the means for

self-describing units of deployment and more importantly, it is metadata and not the IL-bytecode that defines the structure of classes and their underlying relationships.

Our implementation of “classbox-aware” C# exploits this special relationship between IL-bytecode and metadata in order to bind class extensions defined in a given classbox to their corresponding imported classes. Compiling a C# classbox takes place in two phases. In the first phase the compiler creates a new subclass with the same name for each explicitly imported class and incorporates the defined local refinements into it. However, creating a new subclass breaks the link between the imported class and clients of it in the importing classbox. To restore this link we use *code instrumentation* to incorporate the newly created subclass into the inheritance graph of clients in the importing classbox. More precisely, the second phase of the “classbox-aware” C# compiler performs a series of *metadata transformations* to reorganize the relationships between classes described in metadata. In other words, we imprint the logical structure of a classbox in the metadata of the assemblies representing the physical structure of that classbox.

Our approach relies solely on a *compile-time* mechanism to construct the structure of a classbox. Moreover, unlike the approach presented by Bergel et al. [1], which imposes a runtime overhead inbetween 25% to 60% on method calls, our technique does not add any runtime overhead to the resulting classbox assemblies. In addition, we can treat standard .NET assemblies as classboxes, that is, we can import classes originating from standard .NET assemblies into a newly defined classbox, apply some local refinements to those classes, and generate a classbox assembly that is backward-compatible with the standard .NET framework. As a result, we obtain a mechanism that supports the coexistence of non-classbox-aware and classbox-aware software artifacts in one system, which allows for phased and fine-grained software evolution approach.

References

- [1] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems & Structures*, 31(3–4):107–126, May 2005.
- [2] M. Lumpe. Using Metadata Transformations as a Means to Integrate Class Extensions in an Existing Class Hierarchy. Technical Report TR #06-02, Department of Computer Science, Iowa State University, Mar. 2006.