Using Metadata Transformations to Integrate Class Extensions in an Existing Class Hierarchy

Markus Lumpe

Department of Computer Science
Iowa State University
Ames, IA 50011, USA
lumpe@cs.iastate.edu

Abstract. Class extensions provide a fine-grained mechanism to define incremental modifications to class-based systems when standard subclassing mechanisms are inappropriate. To control the impact of class extensions, the concept of *classboxes* has emerged that defines a new module system to restrict the visibility of class extensions to selected clients. However, the existing implementations of the classbox concept rely either on a "classbox-aware" virtual machine, an expensive runtime introspection of the method call stack to build the structure of a classbox, or both. In this paper we present an implementation technique that allows for the structure of a classbox to be constructed at compile-time by means of metadata transformations to rewire the inheritance graph of refined classes. These metadata transformations are language-neutral and more importantly preserve both the semantics of the classbox concept and the integrity of the underlying deployment units. As a result, metadata transformation provides a feasible approach to incorporate the classbox concept into programming environments that use a *virtual execution system*.

1 Introduction

It is generally accepted that the inheritance relationships supported by mainstream object-oriented and class-based languages are not powerful enough to express many useful forms of incremental modifications. To address this problem, several approaches have emerged (e.g., Smalltalk [10], CLOS [22], MultiJava [6], Scala [21], or AspectJ [13]) that focus on a particular technique: *class extensions*. A class extension is a method that is defined in a packaging unit other than the class it is applied to. The most common kinds[1] of class extensions are the *addition* of a new method and the *replacement* of an existing method, respectively.

However, a major obstacle when specifying class extension is that their embodied changes have global impact [2]. Moreover, even if a system allows for a modular specification of class extensions (e.g., MultiJava [6] or AspectJ [13]), it may not support multiple versions of a given class to coexist at the same time. To remedy these shortcomings, Bergel et al. [1, 2] have recently proposed *classboxes*, a new module system that defines a packaging and scoping mechanism for controlling the visibility of isolated

[1] Bracha and Lindstrom [3] have also presented a *hide* operator that renders a method of a class invisible to clients of that class.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 290–306, 2006.

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 those systems [2].

At present, there exist two implementations of classboxes in Smalltalk [2] and a restricted prototype in Java [1]. The first Smalltalk implementation relies on a modified, "classbox-aware" virtual machine in which a dedicated graph search algorithm implements local rebinding of methods. The second implementation uses a combination of bytecode manipulation and a reified method call stack to build the structure of a classbox. This technique is also applied in Classbox/J [1], an implementation of classboxes for the Java environment. In Classbox/J, a preprocessor translates each method redefinition into a `if` statement that uses a `ClassboxInfo` object to determine, which definition to call in the current context.

Common to all three implementations is that the integration of class extensions occurs at runtime by means of a specially-designed method lookup mechanism. This implementation scheme adds a significant execution overhead to redefined methods. For the Smalltalk implementations, for example, this overhead is generally in-between 25% to 60%, compared to the "normal" method lookup [2]. Similarly, the method lookup of redefined methods in Classbox/J is on average 22 times slower than the normal method lookup [1].

In this paper we present an alternative implementation strategy that uses *metadata transformations* to integrate class extensions into a given class hierarchy. More precisely, we present a "classbox-aware" dialect of C# that defines a minimal extension to the C# language in order to provide support for the classbox concept, and Rewire.NET, a metadata adapter that implements a *compile-time* mechansism to incorporate the local refinements defined in a classbox into their corresponding classes. This approach allows us to treat standard .NET assemblies as classboxes, that is, we can import classes originating form 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 and therefore allows for phased and fine-grained software evolution approach.

Our approach to incorporate the classbox concept into the .NET framework uses *code instrumentation* [4, 5, 12, 14, 15] to *rewire* the inheritance graph of a class hierarchy in order to build the structure of a classbox. This approach preserves the original semantics of the classbox concept while moving the process of constructing the structure of a classbox from runtime to compile-time. Furthermore, the application of metadata transformations allows us to use the standard method lookup mechanism for redefined methods. No dynamic introspection of the method call stack is required.

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*. The virtual execution system provides an *abstract*