Controlling the Scope of Change in Java with Classboxes

Alexandre Bergel

LERO &
Distributed Systems Group, Trinity College Dublin, Ireland

Abstract. Introducing changes in complex software systems that were anticipated can introduce anomalies such as duplicated code, suboptimal inheritance relationships and a proliferation of run-time downcasts. Refactoring to eliminate these anomalies may not be an option, at least in certain stages of software evolution. *Classboxes* are modules that restrict the visibility of changes to selected clients only, thereby offering more freedom in the way unanticipated changes may be implemented, and thus reducing the need for convoluted design anomalies. In this paper we demonstrate how classboxes can be implemented in statically-typed languages like Java. We also present an extended case study of Swing, a Java GUI package built on top of AWT, and we document the ensuing anomalies that Swing introduces. We show how Classbox/J, a prototype implementation of classboxes for Java, is used to provide a cleaner implementation of Swing using local refinement rather than subclassing.

1 Introduction

Programming languages traditionally assume that the world is consistent. Although different parts of a complex system may only have access to restricted views of the system, the system as a whole is assumed to be globally consistent. Unfortunately this means that unanticipated changes may have far-reaching consequences that are not good for the general health of the system. Consider, for example, the development of Swing, a GUI package for Java that was built on top of the older AWT package. In the absence of a large existing base of clients of AWT, Swing might have been designed differently, with AWT being refactored and redesigned along the way. Such a refactoring, however, was not an option, and we can witness various anomalies in Swing, such as duplicated code, sub-optimal inheritance relationships, and excessive use of run-time type discrimination and downcasts.

In this paper we argue that unanticipated changes are better supported when we abandon the principle of the consistent world-view. *Classboxes* offer us the ability to define a local scope within which our world-view is refined without impacting existing clients. Classboxes can collaborate to control the scope of change in a way that can significantly reduce the need for introducing anomalous design practices to bridge inconsistencies between the old and the new parts of a system.

In recent years, numerous researchers have proposed better ways to modularize code in such a way as to allow a base system to be easily extended, following the philosophy behind CLOS or Smalltalk. For instance, Open Classes, AspectJ and Hyper/J allow class members to be separately defined from the class they are related to. They do

not, however, permit multiple versions of a class to be present at the same time. Other approaches, like virtual types (as in Keris, Caesar, gbeta, and Nested Inheritance), allow multiple versions of a given class to coexist at the same time: classes are looked up much the same way that methods are. These mechanisms, however, only allow one to refine inner classes inherited from a parent class. Refinement divorced from inheritance is not supported.

We have previously proposed classboxes as a means to control the scope of change in the context of Smalltalk [2, 3]. A classbox is essentially a kind of module which not only provides the classes it defines, but may also import classes from other classes and *refine*[1] them by adding or modifying their features.

There are three key characteristics to classboxes:

- A classbox is a *unit of scoping* within which classes and their features (*i.e.*, fields, methods, inner classes) are defined, imported and refined. Each class is always *defined* in a unique classbox, but it may be imported and refined by other classboxes. Refinements are either new features or redefinitions of features.
- A refinement is *locally visible* to the classbox in which it is defined. This means that the change is only visible to (i) the refining classbox, and (ii) other classboxes that directly or indirectly import the refined class.
- A local refinement has precedence over any previous (*i.e.*, imported) definition or refinement. This means that, although refinements are locally visible, their effect impacts all their collaborating classes. A classbox thereby determines a namespace *within* which local class refinements behave *as though they were global*. From the perspective of a classbox, the world appears to be consistent.

Classboxes were first introduced with an implementation in Smalltalk [3] and subsequently formally described [2]. In particular, we were able to demonstrate that classboxes could be implemented efficiently in a dynamically-typed language with minimal run-time overhead. In this paper we demonstrate how classboxes can be applied effectively to control unanticipated change in a large, industrially-developed application framework written in a statically-typed language, namely Java. A longer description of Classbox/J can be found in our previous work [1].

References

1. A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.
2. A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, Dec. 2005.
3. A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003.

[1] In the literature, such modifications are usually termed "extensions", but to avoid confusion with Java's *extends* keyword, we refer instead to "refinements".