Classboxes – Controlling Visibility of Class Extensions

Dr. Alexandre Bergel: Trinity College Dublin, 2 Dublin, Ireland
Tel: +353-877449165, E-Mail: Alexandre.Bergel@cs.tcd.ie
Dr. Alexandre Bergel is currently a Research Fellow at LERO – the Irish Software Engineering Research Center, and Trinity College Dublin, in Ireland. He designed and conceived the classbox model system during his PhD. In 2006 Dr. Bergel received the Ernst Denert Software Engineering Award.

Keywords: Software engineering, module system, software evolution

MS-ID: Alexandre.Bergel@cs.tcd.ie
Abstract
Unanticipated changes to complex software systems 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.

A class extension is a method that is defined in a module, but whose class is defined elsewhere. Class extensions offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. Unfortunately existing approaches suffer from various limitations. Either class extensions have a global impact, with possibly negative effects for unexpected clients, or they have a purely local impact, with negative results for collaborating clients. Furthermore, conflicting class extensions are either disallowed, or resolved by linearization, with subsequent negative effects.

To solve these problems we present classboxes, a module system for object-oriented languages that provides for behavior refinement (i.e. method addition and replacement). Moreover, the changes made by a classbox are only visible to that classbox (or classboxes that import it), a feature we call local rebinding.

We present an experimental validation in which we apply the classbox model to both dynamically and statically typed programming languages. We used classboxes to refactor part of the Java Swing library, and we show two extensions built on top of classboxes which are (i) runtime adaptation with dynamically classboxes and (ii) expressing crosscutting changes.

Zusammenfassung


Als Lösungsansatz stellen wir classboxes vor, ein Modulsystem für objekt-orientierte Sprachen das Verhaltensverfeinerungen anbietet, d.h., Redefinition und Zusatz von Methoden. Die Veränderungen einer classbox sind nur für diese oder andere classboxes, die diese importieren, sichtbar. Dieses bezeichnen wir als sogenanntes local rebinding.

Wir präsentieren eine experimentelle Bewertung in der wir das classbox Prinzip auf statisch und dynamisch typisierte Programmiersprachen anwenden. Teile der Java Swing Bibliothek wurden mit diesem Prinzip refactored. Zusätzlich stellen wir zwei Erweiterungen vor, die auf dem classbox Prinzip aufbauen, die Adaption einer Anwendung mit dynamischen classboxes und die Definition von crosscutting Ausdrücken.

Widmung
D.3.3 [Programming Languages]: Language Constructs and Features; D.1.5 [Programming Languages]: Object-oriented Programming
It is well-established that object-oriented programming languages gain a great deal of their power and expressiveness from their support for the open/closed principle \cite{Mey88}: classes are closed in the sense that they can be instantiated, but they are also open to incremental modification by inheritance.

Nevertheless, classes and inheritance alone are not adequate for expressing many useful forms of incremental change. For example, modern object-oriented languages introduce modules or packages as a complementary mechanism to structure classes and control visibility of names.

We focus on a particular technique, known as class extensions, which addresses the need to extend existing classes with new behaviour. Smalltalk \cite{GR89}, CLOS \cite{Pao93}, Objective-C \cite{PW88}, and more recently MultiJava \cite{CLCM00} and AspectJ \cite{KHH+01} are examples of languages that support class extensions. Class extensions preserve class identity when extended, whereas class inheritance implies creation of new classes. Class extensions offer a good solution to the dilemma that arises when one would like to modify or extend the behaviour of an existing class, and subclassing is inappropriate because that specific class is referred to, but, one cannot modify the source code of the class in question. A class extension can then be applied to that specific class.

Classboxes are a modular approach to class extensions that solve traditional composition issues of class extensions such as conflict (two class extensions that refer to the same methods but with different implementations for example). A classbox is a kind of module with three main characteristics:

- It is a unit of scoping in which classes, global variables and methods are defined. Each entity belongs to precisely one classbox, namely the one it which it is first defined, but an entity can be made visible to other classboxes by importing it. Methods can be defined for any class visible within a classbox, independently of whether that class is defined or imported. Methods defined (or re-defined) for imported classes are called class extensions.

- A class extension is locally visible to the classbox in which it is defined. This means that the extension is only visible to (i) the extending classbox, and (ii) other classboxes that directly or indirectly import the extended class.

- A class extension supports local re-binding. This means that, although extensions are locally visible, their effect extends to all collaborating classes. A classbox thereby determines a namespace within which local class extensions behave as though they were global. From the perspective of a classbox, the world is flattened.

The model of classboxes exhibits several properties related to the visibility of class extensions:

- Locality of changes – Class extensions of an imported class are visible to the refining classbox and to other classboxes that import this extended class. The extended class is a new version of the original class that coexists in the same system.

- Precedence of redefinition – Redefined class members have precedence over the imported definition.

- Coexistence of several class versions – Extending a class conceptually defines a new version of it.

Classboxes provide an efficient mechanism to define change and model evolution of software program. Consider, for example, the development of Swing, a GUI package for Java that was built on top of the older AWT package. Because subclassing is used to incorporate Swing-related changes, serious drawbacks such as duplicated code (e.g., 43\% of JWindow is duplicated in JFrame) and mismatches between the original and the extended class hierarchy (left part of Figure 1).

Swing has been refactored into the SwingCB classbox (right part of Figure 1). SwingCB imports the classes Frame, Window, Component, and Button from the AwtCB classbox. Those classes are extended with the features from the original JFrame, JWindow, JComponent, and JButton classes. For instance, the imported Component class is extended with some new variables (e.g., accessibleContext, components) and new methods (e.g., update(), add(Component)). Inheritance relationships defined in AwtCB are preserved in the SwingCB classbox. For instance, in SwingCB, Frame is a subclass of Window, itself a subclass from Component. As a result, the code duplication in the original Swing has been removed in the classbox version of Swing.

The work on classboxes made numerous significant contributions.

- First-class environment module calculus: Understanding the multitude of module systems requires a common foundation in which differences between various semantics are expressed. We define a module calculus for this purpose. Because the notion of namespace is implicitly associated to module, this calculus makes the notion of environment a first-class entity [BDN05].

- Analysis of a large library: An analysis of a large and widely used Java library (Swing) is used to define criteria for a better mechanism to deal with changes.
This analysis points out code duplications, broken inheritance and explicit type checking, which justifies the need of having at the language level constructs to express changes and how they can be applied [BDN05b].

- Scoped class extensions: Scoping facilities to deal with changes by means of class extensions are provided by means of a new module system, classboxes [BD05b, BDW03a, MB03d].

- Strategies to efficiently implement scoped class extensions: A description of three implementations of classboxes is proposed. Two of them are based on the Smalltalk dialect Squeak whereas the third one is made in Java: (i) modification of the Squeak virtual machine, (ii) use of reflective capabilities of Squeak, and (iii) manipulating source code and reifying method call stack in Java. Benchmarks are provided for each of these implementations [BDW03b, BDN05].

  Dynamic classboxes: Uniform and expressive mechanism to support crosscutting changes made of class extensions [BD05a]

  Classboxes triggered numerous interest among the community in programming languages [LS05, LS06, BHCC06a].

Literatur


