Information Technology

Classboxes – Koontrollierte Sichtbarkeit von Klassenerweiterungen

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: Heft: 46/3 (2007)	Alexandre.Bergel@cs.tcd.ie	28th February 2007

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

Unerwaretete Veraenderungen in komplexen Systemen fuehren zu Anomalien wie z.B. dupliziertem Code, suboptimalen Vererbungsbeziehungen und die Zunahme von runtime downcasts. Refactoring bietet nicht immer eine Loesung, zumindest nicht in allen Stufen der Sofware Evolution.

Eine Klassenerweiterung ist eine Methode, die in einem Modul deklariert, aber deren Klasse an anderer Stelle definiert ist. Klassenerweiterungen sind eine elegante Moeglichkeit um bestehende Klassen inkrementell zu veraendern wenn Veerbung nicht anwendbar ist. Existierende Ansaetze zeigen einige Nachteile auf. So haben zum Beispiel Klassenerweiterungen globale Einfluesse mit negativen Folgen fure die Benutzer dieser Klasse.

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

Wir praesentieren 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. Zusaetzlich stellen wir zwei Erweiterungen vor, die auf dem classbox Prinzip aufbauen, die Adaption einer Anwendung mit dynamischen classboxen und die Definition von crosscutting Ausdruecken.

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* [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 [GR89], CLOS [Pae93], Objective-C [PW88], and more recently MultiJava [CLCM00] and AspectJ [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 redefined) 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 rebinding*. 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 clasbox 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 [BDN05a].

– *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.

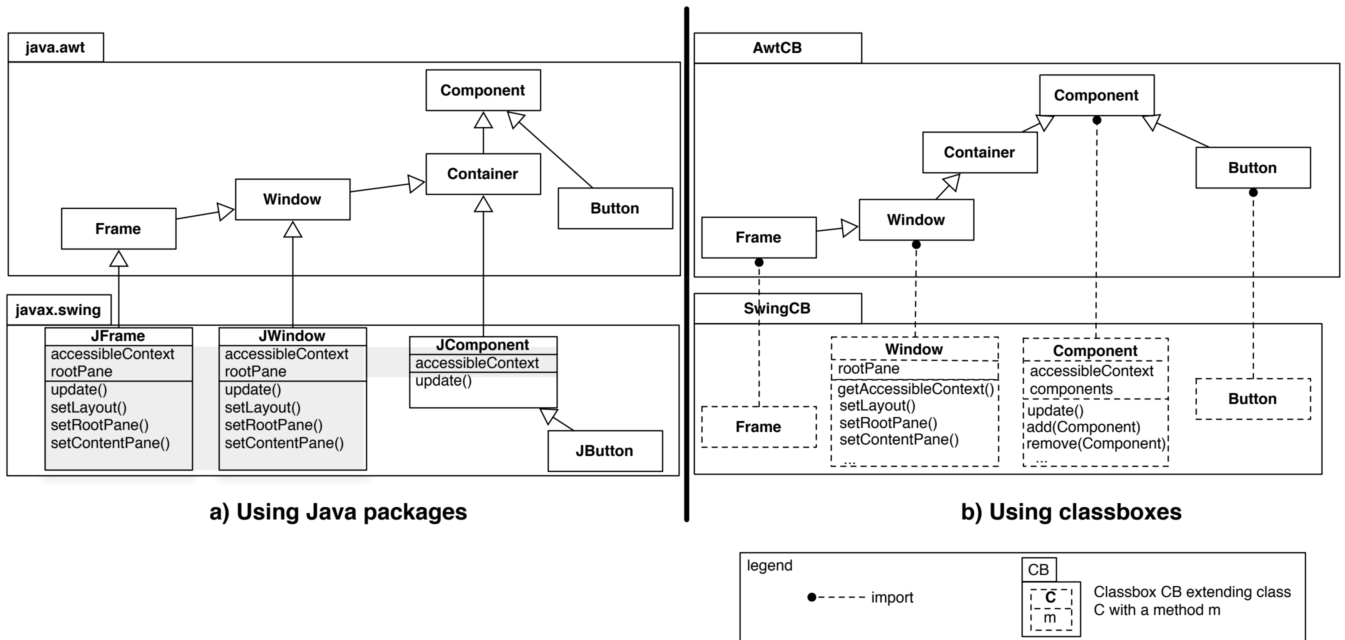

Bild 1: Swing is a GUI framework built on top of AWT. a) Swing contains large portion of duplicated code (gray portion) and inheritance defined in Swing mismatches the one defined in AWT (*e.g.,* an AWT frame is an AWT window, however a Swing frame is *not* a Swing window). b) The classbox version of Swing removes all the code duplication because the classbox SwingCB defines a new view of AwtCB instead of using inheritance.

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, MBCD05].

– *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, BDNW05].

– *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

[BDW03b] Alexandre Bergel, Stéphane Ducasse, and Roel 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. Best Award Paper.

[BDW03a] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. The Classbox module system. In *Proceedings of the ECOOP '03 Workshop on Object-oriented Language Engineering for the Post-Java Era*, July 2003.

[BDNW05] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.

[BDN05b] Alexandre Bergel, Stéphane Ducasse, and Oscar 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, San

Diego, CA, USA, 2005. ACM Press.

[BDN05a] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 2005. To appear.

[BD05b] Alexandre Bergel and Stéphane Ducasse. Supporting unanticipated changes with Traits and Classboxes. In *Proceedings of Net.ObjectDays (NODE'05)*, pages 61–75, Erfurt, Germany, September 2005.

[BD05a] Alexandre Bergel and Stéphane Ducasse. Scoped and dynamic aspects with Classboxes. *RSTI – L'Objet (programmation par aspects)*, 11(3):53–68, 2005.

[BHCC06a] Alexandre Bergel, Robert Hirschfeld, Siobhàn Clarke, and Pascal Costanza Aspectboxes – Controlling the Visibility of Aspects. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2006)*, September 2006.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

[LS05] Markus Lumpe, and Jean-Guy Schneider Classboxes – An Experiment in Modeling Compositional Abstractions using Explicit Contexts In *Proceedings of ESEC '05 Workshop on Specification and Verification of Component-Based Systems (SAVCBS '05)*, September 2005.

[GR89] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[LS06] Markus Lumpe, and Jean-Guy Schneider On the Integration of Classboxes into C# In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, March 2005.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[MBCD05] Florian Minjat, Alexandre Bergel, Pierre Cointe, and Stéphane Ducasse. Mise en symbiose des traits et des classboxes : Application à l'expression des collaborations. In *Proceedings of LMO 2005*, volume 11, pages 33–46, Bern, Switzerland, 2005.

[Pae93] Andreas Paepcke. User-level language crafting. In *Object-Oriented Programming : the CLOS perspective*, pages 66–99. MIT Press, 1993.

[PW88] Lewis J. Pinson and Richard S. Wiener. *Objective-C*. Addison Wesley, 1988.