Classbox/J: Controlling the Scope of Change in Java

Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz

Classbox/J: Controlling the Scope of Change in Java

Alexandre Bergel
Software Composition Group
University of Bern – Switzerland
www.iam.unibe.ch/∼scg

Stéphane Ducasse
Language and Software Evolution Group
LISTIC – University of Savoie, France &
Software Composition Group
University of Bern – Switzerland
www.listic.univ-savoie.fr/

Oscar Nierstrasz
Software Composition Group
University of Bern – Switzerland
www.iam.unibe.ch/∼scg

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. *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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.5 [**Programming Languages**]: Object-oriented Programming

General Terms

Language, Design

Keywords

Module, Package, Open-classes, Class extension

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 [17] or Smalltalk [14]. For instance, Open Classes [25], AspectJ [1] and Hyper/J [28] 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 [38], Caesar [24], gbeta [11], and Nested Inheritance [26]), 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 [4,5]. 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

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

Preprint of the ACM OOPSLA 2005 Version
OOPSLA'05, October 16–20, 2005, pp. 177-189, San Diego, California, USA.

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 [5] and subsequently formally described [4]. 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. The contributions of this paper are:

- A proof-of-concept implementation of classboxes for statically typed languages. Classbox/J consists of a minimal extension of Java: (i) package import clauses are made transitive, and (ii) packages are able to refine imported classes and export these classes to other packages.
- The original classbox model is extended with a mechanism enabling refinements to access prior definitions. The Swing refactoring towards classboxes motivates the need to invoke original methods from their redefined bodies.
- Presentation of a large case study in which (i) the limits of subclassing are clearly identified, and (ii) classboxes are used to remove code duplication and incoherence in the class hierarchy.

In Section 2 we use the Swing case study to point out anomalies that can arise when subclassing is used to introduce significant crosscutting changes. In Section 3 we present the model of classboxes for Java. In Section 4 we present an example illustrating how classboxes support the implementation of cross-cutting changes. In Section 5 we apply classboxes to refactor Swing. In Section 6 we describe our Java implementation of classboxes. In Section 7 we provide a brief overview of related work. In Section 8 we conclude by summarizing the presented work.

2. SWING/AWT ANOMALIES

Using subclassing to incorporate crosscutting changes often introduces serious drawbacks such as duplicated code and mismatches between the original and the extended class hierarchy. We illustrate these problems by analyzing Swing [34], the Java standard framework for building GUIs. We first describe the Abstract Window Toolkit (AWT [2]) and its relationships with the Swing framework. Then we show how inheritance is used to share properties between classes. Finally we identify some important drawbacks of the Swing design.

2.1 AWT and Swing History

In its first release launched in 1995, Java included AWT 1.0, a framework for building graphical user interfaces. AWT evolved rapidly in version 1.1 to provide a better event handling mechanism.

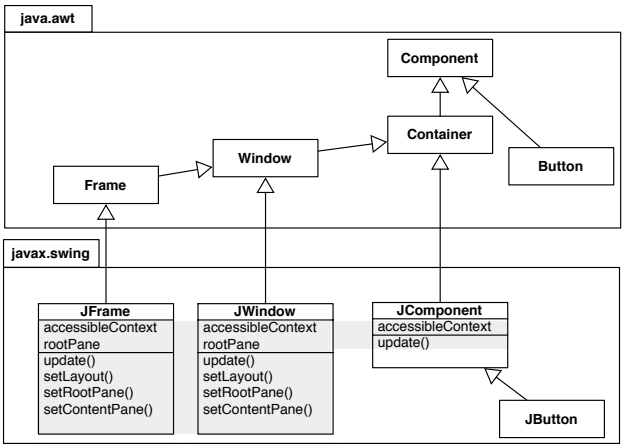

Figure 1: Swing is a GUI framework built on top of AWT. Fields and methods shown in JFrame, JWindow and JComponent are duplicated code (gray portion). More than 43% of JWindow is duplicated in JFrame.

AWT is close to the underlying operating system, therefore only a small number of widgets are supported to make code easier to port. In its latest version AWT consists of 345 classes and contains more than 140,000 lines of code.

Release 1.2 of the Java Development Kit included a completely new GUI framework named Swing. Swing contains 539 classes and more than 260,000 lines of code. This GUI framework is built on top of AWT. It provides a "pluggable look and feel", double buffering and more widgets. A small subset of the core of AWT (Component, Container, Frame Window), and Swing is depicted in Figure 1.

In AWT, the root of the graphical widget hierarchy is Component. It provides the essential functionalities of the GUI framework. JComponent is the base class for most of the Swing widgets. The core of Swing is defined by subclassing the core classes of AWT. Each Swing widget can be a container for other widgets, so JComponent inherits from Container. All the widgets except top-level containers (like windows and frames) inherit from JComponent. The classes JFrame and JWindow inherit from Frame and Window, respectively.

The AWT and Swing class hierarchies guarantee certain properties and behavior. In the AWT framework (i) a widget is a component – every widget inherits from Component, (ii) a frame is a window – Frame is a subclass of Window. On the other hand, the Swing framework has the following properties: (i) a Swing widget is not necessary a Swing component because not all of the Swing classes inherit from JComponent, (ii) a Swing frame is an AWT frame and an AWT window: JFrame inherits from Frame which has Window as its superclass, (iii) a Swing window is an AWT window: JWindow inherits from Window.

2.2 Problem Analysis

Subclassing and refinement relationships are fundamentally different: the former results in a new class containing the incremental changes to its parent class, whereas the latter results in the creation of scope within which the original class is changed. As pointed out by Findler *et al.* [12] and Torgersen [35] under *the extensibility problem*, subclassing does not solve the problem of adding new

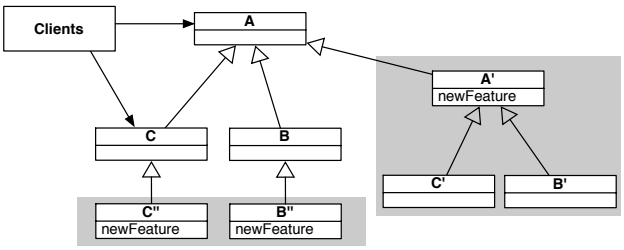

Figure 2: Two strategies (gray portions) to introduce changes without impacting existing clients

operations to a class without having to modify or recompile the original program component and its *existing clients*.

In Java, if we wish to extend the class Component by subclassing, without impacting existing clients, we can use either of two strategies (see Figure 2): either we build a completely new hierarchy derived from the root of the old hierarchy, duplicating old features in the new hierarchy, or we derive new classes from the leaves of the original hierarchy, duplicating the new features.

Swing illustrates an example of this problem. Swing is built on top of AWT and uses subclassing to extend AWT core classes with Swing functionalities. Since Java supports neither multiple inheritance nor class extension, this design leads, however, to the following severe consequences:

Duplicated Code. Due to the absence of an inheritance link between JFrame and JWindow, features defined in JWindow have to be duplicated in JFrame. In Swing, each widget can (i) describe itself (the accessibleContext variable refers to a description of the component) and (ii) support double buffering to provide smooth flicker-free animation (methods update(), setLayout(), ...). The source code of JWindow is 551 lines, and JFrame is 829 lines. As a result, 241 lines of code are duplicated between these two classes: 43% of JWindow reappears as 29% of JFrame.

Breaking Subtyping Inheritance. Whereas all AWT widgets are AWT components (because they inherit from Component), widgets defined in Swing can either be AWT or Swing components. Furthermore, the Swing design breaks the AWT inheritance relation: while a Window is a Component in AWT, a JWindow is not a JComponent in Swing. While a Button is a Component and JButton is a JComponent, a JButton is not a Button [19].

Explicit Type Checks and Casts. A Swing component is a container for other components. This is a feature obtained from Container by inheritance (JComponent is subclass of Container). Therefore types of subcomponents are Component, and not JComponent (the type of the collection of components is Component[]). The following code typifies what happens in Swing components:

```
public class Container extends Component {
  int ncomponents;
  Component components[] = new Component[0];
  public Component add (Component comp) {
     addImpl(comp, null, -1);
     return comp;
  }
  protected void addImpl (Component comp,
                          Object constraints, int index) {
     ...
     component[ncomponents++] = comp;
     ...
  }
  public Component getComponents(int index) {
    return component[index];
  }
}

public class JComponent extends Container {
  public void paintChildren (Graphics g) {
    ...
    for (; i > = 0 ; i--) {
      Component comp = getComponent (i);
      isJComponent = (comp instanceof JComponent);
      ...
      ((JComponent)comp).getBounds();
      ...
    }
  }
}
```

In the Swing framework numerous explicit type checks need to be performed to determine if a subcomponent is issued from Swing or from AWT. For instance, a JComponent needs to know if its subcomponents use double buffering or not. 16 type checks (... instanceof JComponent) and 25 casts to JComponent are performed in JComponent. In the whole Swing library, these numbers rise to 82 and 151, respectively.

3. CLASSBOX/J

A package can define new classes and it may refer to classes defined in other packages using an *import* clause. After importing a class, a package can either subclass it or reference it in a declaration. In pure Java, import statements are not transitive: a package p2 cannot import a class C from a package p1 if C was imported rather than defined in p1. In contrast to MultiJava [25], Hyper/J [28], CLOS [10] and Smalltalk [14], a Java package cannot add methods to a class defined in another package. Therefore a package can be adapted only by subclassing its member classes.

Classbox/J addresses these shortcomings by offering a means to refine classes within a well-defined scope.

3.1 Classbox/J in a Nutshell

Classbox/J is a module system for Java allowing classes to be refined with new class members, such as fields, methods and inner classes. A classbox in Classbox/J is essentially a Java package with the following three important differences: (i) imported classes can be refined by adding or redefining class members using the refine keyword, (ii) a class defined or imported within a classbox p can be imported by another classbox. This allows the import clause to be transitive, and (iii) a refined method can access its original behavior using the original keyword.

We illustrate Classbox/J with a small example based on the Swing case study.

Refining classes. Figure 3 illustrates two classboxes WidgetsCB and EnhWidgetsCB. WidgetsCB defines two classes Component and Button. EnhWidgetsCB imports them, refining Component with a new instance variable lookAndFeel and redefining the paint() method. These classboxes are implemented as follows:

```
package WidgetsCB;
public class Component {
  public void update () { this.paint(); }
  public void paint () { /* Old Code */ }
}
public class Button extends Component {
  public Button (String name) { ... }
```

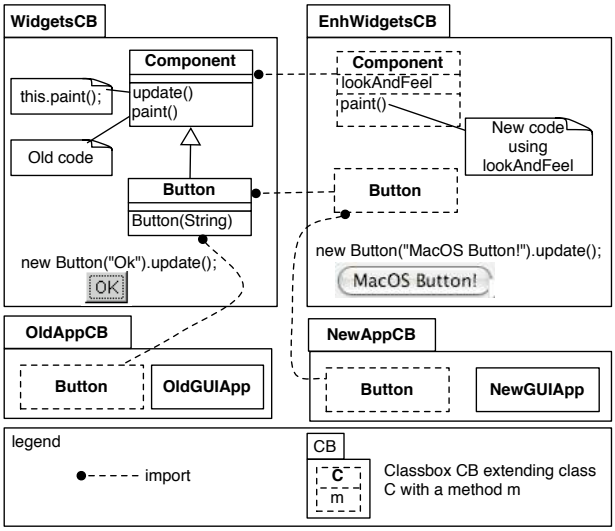

Figure 3: Two versions of classes Component and Button are used by two different clients OldAppCB and NewAppCB.

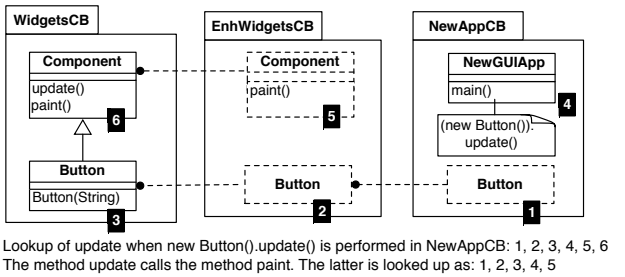

Figure 4: Locality of changes entails a new method lookup semantics. The numbers within the black boxes indicate the steps taken in looking up a message sent to a button.

```
}

package EnhWidgetsCB;
import WidgetsCB.Component;
import WidgetsCB.Button;
refine Component {
  private ComponentUI lookAndFeel;
  public void paint () { /* New code using lookAndFeel */ }
}
```

Refining a class conceptually defines a new version of it. In the previous example, two versions of Component *coexist* at the same time within the system in different scopes. The original version is accessible through WidgetsCB and the new version through EnhWidgetsCB. Class members refining an imported class are local to the refining classbox and to other classboxes that may import the refined class.

Transitive import. A class imported by a classbox can be transitively imported by other classboxes, whether this class is refined or not. For instance, a client of the new version of the widgets can be defined as:

```
package NewAppCB;
import EnhWidgetsCB.Button;
public class App {
  public static void main(String[] argv) {
    ... new Button().paint(); ...
  }
}
```

3.2 New Method Lookup Semantics

As shown in the previous section, class refinements have bounded visibility. Moreover, redefinitions have precedence over imported definitions. This behavior is obtained by a new semantics for method lookup. We illustrate this operationally.

Import over inheritance. Import statements between packages have to be taken into account when looking up a message. The main point is that the import clause has precedence over inheritance: before looking a method up in the superclass, the chain of imports has to be considered first.

Figure 4 illustrates the lookup of messages update() and paint(). When the message update() is sent to an instance of Button in the classbox NewAppCB, the lookup algorithm first searches for the implementation of update() in the classbox NewAppCB (1). This method is not defined in this classbox, therefore the lookup follows the chain of import (2). In EnhWidgetsCB, update() is not defined, so the lookup continues in WidgetsCB (3). In this classbox, the class Button is not imported anymore but defined in it. Therefore, update() is looked up in the superclass Component but starting from the source classbox (NewAppCB, in step 4). Because Component is not visible within NewAppCB and Button is imported from EnhWidgetsCB, the lookup continues to EnhWidgetsCB (5). The class Component is visible, but the method update() is not implemented. Finally the method is found in WidgetsCB. The method update() triggers the message paint(). In a similar way, the method paint() is looked up as in steps 1 through 5.

Note that defining new semantics for the method lookup algorithm does not necessarily mean that the virtual machine (VM) must be modified. As described in Section 6, the desired behavior can be obtained by inserting some code that performs dynamic run-time stack introspection where a method redefinition occurs.

Multiple imports. As illustrated in Figure 5, a diamond graph of imports may imply the use of different class refinements defined by several classboxes. In the classbox AppCB, sending the paint() message to an instance of LabelButton invokes the implementation of paint() on Component defined by WidgetsCB. In a similar way, sending this message to an instance of Button triggers the implementation brought by NewWidgetsCB on Component.

Accessing the original method. When a method is redefined, the original method is accessible using the construct original().

For instance, in the classbox EnhWidgetsCB the extension of Component could be:

```
refine Component {
  private ComponentUI lookAndFeel;
  public void paint () {
     if (lookAndFeel == nil) { original();}
        else { /* use lookAndFeel */ }
  }
}
```

The original() construct invokes the first method (*e.g.*,WidgetsCB .Button.paint() in Figure 6) in the import chain that was redefined by the method containing the expression original() (*e.g.*,EnhWidgetsCB.Button.pair

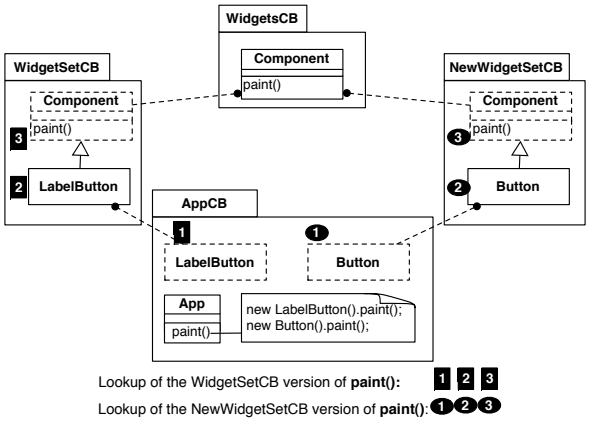

Figure 5: Within one classbox, different versions of the same class can be accessible. From AppCB sending the paint() message sent to a LabelButton triggers WidgetsCB's refinement, whereas sending it to a Button triggers NewWidgetsCB.

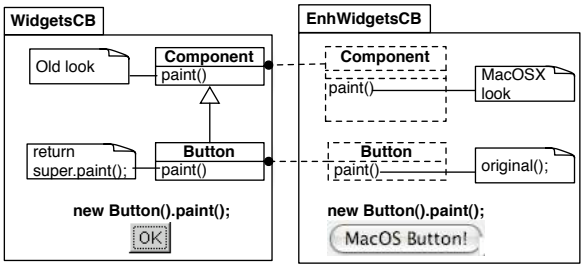

Figure 6: The original() construct invokes the hidden method but in the context of the classbox changes.

Note that in particular super invocation in the original methods takes into account potential changes introduced by the classbox containing the original() invocation, preserving that way the method lookup semantics of classbox. In Figure 6, new Button().paint() displays a button having a MacOSX look since, first the method WidgetsCB.Button.paint() is executed and the super invocation invokes EnhWidgetsCB.Component.paint().

It is precisely this kind of scenario, which arises frequently in the Swing case study, that has motivated the addition of the original mechanism into the classbox model.

3.3 Properties of the Model

The model of classboxes defined in the previous section exhibits several properties related to the visibility of refinements.

Locality of Changes. MultiJava [25] with its open-classes and AspectJ [1] with its inter-types allow class members to be defined separately from the class they are related to. Class members are not, however, contained in a unit of scope, therefore redefinition is not allowed and composition has to be explicitly stated. With classboxes, refinements of an imported class are visible to the refining classbox and to other classboxes that import this refined class. The refined class is a new version of the original class that coexists in the same system. Figure 3 shows two clients OldAppCB and NewAppCB using the old and new version of the widget framework. Any refinement introduced to WidgetsCB by EnhWidgetsCB does not impact OldAppCB. This is because changes are confined to EnhWidgetsCB and to other classboxes that may imported the classes it refines (*e.g.*,NewAppCB).

Precedence of redefinition. Redefined class members have precedence over the imported definition. EnhWidgetsCB redefines the method paint() for Component, thus hiding the previous definition. From this classbox and other classboxes that may import Component or its subclasses, the original definition of paint() is no longer accessible. Within the classbox EnhWidgetsCB or NewAppCB, sending the message update() or paint() to an instance of Button will trigger the new definition of paint().

Refinements along a chain of import. With classboxes, imports are transitive: a new version of an imported class can be re-imported. Figure 3 shows the class Button defined in WidgetsCB that is imported in EnhWidgetsCB and from this last, are imported in NewAppCB. From the point of view of an importing classbox, there is no distinction between a class that is defined or imported in the provider classbox (*i.e.*, classbox where the class is imported from). An imported class can always be refined and then re-imported, even multiple times over a chain of imports.

4. CROSS-CUTTING CHANGES

Refining a class is superficially similar to subclassing: a classbox can add new interfaces, fields, methods, static field, inner classes and constructors as well as redefine methods of an imported class. The key difference is that the changes are applied to the original class, not a subclass, but only within a well-defined scope. It is this feature that supports the introduction of cross-cutting changes. The following example shows how a *look and feel* feature is added to the root of a class hierarchy without breaking former clients, while propagating the refinements to collaborating classes. As shown in Figure 7, two classboxes WidgetsCB and FactoryCB define a base system which clients rely on. Since modifying these base classes would break these clients, changes cannot be directly applied to the classboxes WidgetsCB and FactoryCB, but are introduced in class-

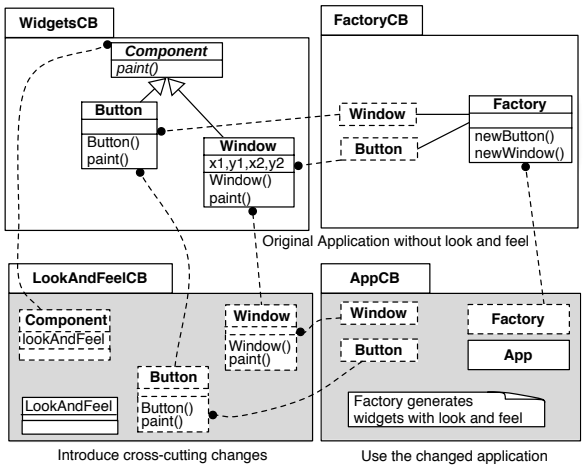

Figure 7: From the viewpoint of AppCB, refinements of the root of the hierarchy (Component) are propagated to the class Factory. This is a consequence of importing the version of widgets that have a look and feel.

box LookAndFeelCB and used by a new client in AppCB. The rest of this section shows how classboxes allow one to incorporate these changes without having to modify WidgetsCB and FactoryCB.

The following example shows some refinements defined with classboxes on a base system that (1) does not break clients that rely on the original definitions of this system, and that (2) propagate these refinements to collaborating classes defined in other classboxes.

Base system. The classbox WidgetsCB defines three classes: an abstract class Component and two subclasses Button and Window. The source code of this classbox is:

```
package WidgetsCB;
public abstract class Component {
   public abstract void paint();
}
public class Button extends Component {
   public Button () { }
   public void paint() {
      System.out.println("Button");
   }
}
public class Window extends Component {
   int x1, y1, x2, y2;
   public Window () { x1 = 50; y1 = 50; x2= 200; y2=200;}
   public void paint() {
      System.out.println("Window");
   }
}
```

New widgets are created using a factory. This factory is implemented in a separate classbox FactoryCB. When it was designed, the implementor of Factory relied on the version of the widgets obtained from WidgetsCB without any look and feel. The widget factory is defined as:

```
package FactoryCB;
import WidgetsCB.*;
public class Factory {
   public Button newButton () { return new Button(); }
   public Window newWindow () { return new Window(); }
}
```

Refinement of the base system. To introduce the changes that add a "look and feel" to the widgets, two new classboxes are added: LookAndFeelCB, which effectively defines the changes, and AppCB, which is a new client of the resulting system. In LookAndFeelCB the root class Component is refined with a lookAndFeel variable. In order for classes Button and Window to use this new variable added to their superclass, their constructor and paint() are redefined. These refinements are defined as:

```
package LookAndFeelCB;
import WidgetsCB.Component;
import WidgetsCB.Button;
import WidgetsCB.Window;
public class LookAndFeel {
  ...
}
refine Component {
   LookAndFeel lookAndFeel; // Variable added to Component
}
refine Button {
   public Button() { // Constructor redefined
     lookAndFeel = new LookAndFeel("ButtonMacOSX");
     original();   // Original constructor called
   }
   public void paint() { // Method paint redefined
      System.out.println(lookAndFeel.getName());
   }
}
refine Window {
   public Window() { // Constructor redefined
     lookAndFeel = new LookAndFeel("WindowMacOSX");
     original();   // Original constructor called
   }
   public void paint() { // Method paint redefined
      System.out.println(lookAndFeel.getName());
   }
}
```

A small application is built in the classbox AppCB. This classbox imports the class Factory from FactoryCB and the widgets having a look and feel from LookAndFeelCB. Now when the new application uses the factory to create widgets, it gets widgets with the look and feel as defined in the LookAndFeelCB classbox, whereas the clients of the original code defined in WidgetsCB are not impacted, *i.e.*, get widgets without look and feel. As AppCB imports the version of Window and Button with a look and feel, from the perspective of AppCB, this version of the widgets takes precedence over the one present in FactoryCB.

```
package AppCB;
import FactoryCB.*;
import LookAndFeelCB.*;
public class App {
   public static void main (String[] argv) {
      Factory f = new Factory();
      Window w = f.newWindow();
      Button b = f.newButton();
      //Display "WindowMacOSX" and "ButtonMacOSX"
      w.paint();
      b.paint();
   }
}
```

5. SWING AS A CLASSBOX

Because the mechanism provided by Java to specialize code is inheritance, Swing is built on top of AWT using subclassing. As already shown in Section 2 this extension of AWT is developed at a high cost: (i) properties defined in AWT according to the inheritance property are not valid in Swing anymore (*i.e.*, in AWT a

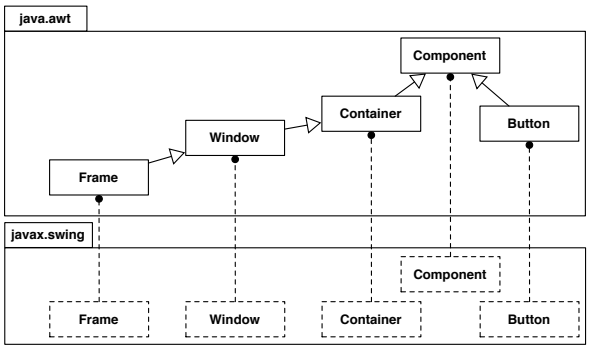

Figure 8: An ideal refactoring based on classboxes

Frame is a Window, but in Swing a JFrame is not a JWindow. Not all Swing widgets are JComponent), (ii) a serious amount of code is duplicated to emulate missing inheritance links in Swing (*i.e.*, 43% of JWindow is duplicated in 29% of JFrame), and (iii) Swing code is littered with explicit type checks.

Figure 8 shows an ideal situation where Swing would be extending AWT using classboxes. Obtaining such a situation would be possible if Swing would have been implemented by following the inheritance tree of AWT (*i.e.*, introducing a JContainer class) or if we could afford to perform a complete overhaul of Swing. Since Swing, however, is a large framework with complex logic we cannot rewrite it totally to obtain the situation depicted. In order to illustrate how classboxes offer a working solution, we refactored Swing as a classbox that refines AWT classes. In this section we first describe the new architecture of Swing made out of classboxes, then we present the results obtained, and finally we describe some issues that we encountered while refactoring.

5.1 Swing Refined from AWT Class

We focus on the refactoring of the core class JComponent, and then we describe how the classbox SwingCB is defined.

Component refactored in two steps. The goal of refactoring JComponent is to make the Swing version JComponent a refinement of the AWT version Component. As depicted in Figure 1, the class JComponent is a subclass of the AWT classes Container and Component. As Container is an intermediate class between JComponent and Component, the refactoring of the class JComponent is done in two steps, as illustrated in Figure 9:

1. *Incorporating the class Container in JComponent.* A Swing component has the ability to contain other components. Features defined by Container have first to be included in JComponent. Container defines 108 methods and 21 fields, however only a few of them have to be duplicated (32 methods related to container management (*e.g.*,add, remove) and events management, and 3 variables). We define this "enlarged" JComponent in the classbox SwingCB. This new class is a subclass of Component, which is imported in the new classbox SwingCB. JComponent overrides 22 methods in Container and most of the overriding methods do not perform any super call. For the methods in JComponent that perform a super call, the two implementations are simply merged.

2. *Making this new JComponent a refinement of Component* The inheritance link between JComponent and the imported Component is replaced by a refinement link.

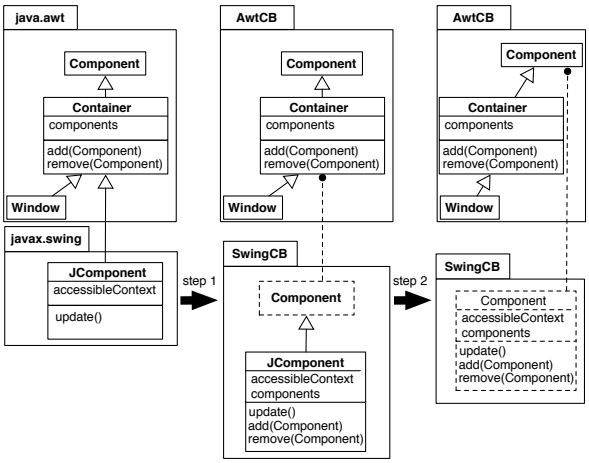

Figure 9: The refactoring of the AWT class Component is performed in two steps: (i) the intermediate class is merged to JComponent, then (ii) this merge becomes a refinement of the AWT class Component.

Swing as AWT refined. Figure 10 depicts the new architecture of Swing. Because the definition of a Java package is a valid definition of a classbox, the package java.awt is immediately turned into the AwtCB classbox: no modification is applied to AWT.

The classbox SwingCB imports the class Component, Window, Frame, and Button from AwtCB. These classes are refined with the Swing features.

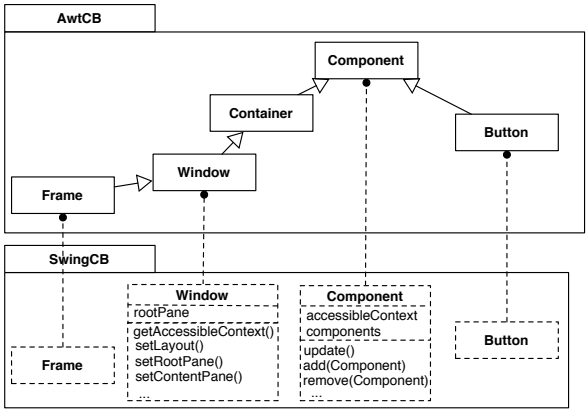

Figure 10: Swing refactored as a classbox.

5.2 Advantages with Classboxes

The Swing classes JComponent, JButton, JWindow and JFrame have been refactored as refinement of their AWT counterpart classes. The amount of code refactored is about 6,500 lines of code spread over these 4 classes. Designing Swing with classboxes has several advantages over the original implementation.

Inheritance coherence. The inheritance link defined in the AwtCB is fully preserved in the SwingCB. Therefore every Swing widgets, including frames and windows, are swing components. The relation "a frame is a window" stated by AWT is true in SwingCB.

Removed duplicated code. JWindow and JFrame are refactored into refinements of Window and Frame. As a result: Frame remains a subclass of Window in Swing and all the duplicated methods and variables related to the layout, root pane and content pane in JFrame are removed. The size of refined Frame is 29% less than the original JFrame.

Because JFrame and JWindow do not inherit from JComponent, the update() method defined by the latter had to be duplicated in JFrame and JWindow. With Swing as a classbox, this duplication is eliminated.

Explicit type checks avoided. Within the SwingCB classbox, a Swing component is a Component. Therefore, all the explicit type-checks and casts used in the original Swing to check if a subcomponent is a Component or a JComponent are useless.

Since the checks (... instanceof JComponent) are always true, downcasts from Component to JComponent are simply removed. The 16 type-checks (... instanceof JComponent) and 25 casts to JComponent were removed while refactoring the class JComponent (no such expressions are present in the other refactored classes).

5.3 Issues and Limits

Now we discuss the results obtained and the impacts on the packages in terms of their visibility.

Refactoring super calls. Several methods related to the content management in JWindow like remove(Component) and setLayout(LayoutManager) override methods defined in Window. These methods perform a check on a property of the root pane, then call the original definition using a super call. For instance, the definition of setLayout(LayoutManager manager) in JWindow is:

```
public void setLayout(LayoutManager manager) {
   if (isRootPaneCheckingEnabled()) {
      throw createRootPaneException("setLayout");
   }
   else {
      super.setLayout(manager);
   }
}
```

The expression super.setLayout(manager) triggers the implementation defined in the AWT class Window. Refactoring this overriding method into a refinement of Window implies that the original keyword must be used to invoke the original AWT definition. This scenario convinced us of the need to introduce the original() construct to the classbox model.

Need to enlarge visibility of some Swing classes. Replacing the Swing class JComponent by a refinement of Component enlarges the visibility of some classes that were in Swing. For instance, JComponent references Swing classes like AncestorNotifier, which are private to the javax.swing package. Swing classes that were private to Swing need to be visible outside their defining package.

Limitations of our refactoring. Unfortunately, removing the class JComponent would entail a major overhaul of Swing. The reason is that each method of the class javax.swing.plaf.ComponentUI refers to the name JComponent. Given our limited resources for this experiment, we confined this overhaul to the classes JWindow, JFrame and JButton. As a consequence, our version of Swing does not contain the pluggable look and feel.

Execution cost. With our current implementation of classboxes, the new method lookup semantics is about 22 times slower than the normal one. This result is obtained from triggering 10000 times the update() methods redefined in Component. This loop takes 1008 ms, whereas it is 45 ms for the same method directly implemented in this class. As explained in the following section, our implementation is rather naive. In our previous work with classboxes in Smalltalk [4], we were able to optimize the implementation so that the cost of the redefined method lookup is only 1.1 times slower (compared to 22 times slower with the Java version).

6. IMPLEMENTATION

We implemented a preprocessor that translates classbox definitions into pure Java files, which are then compiled using a classical compiler. While producing Java source files, classboxes are compiled away by producing a Java package for each classbox. Our implementation is freely available at www.iam.unibe.ch/∼scg/Research/Classboxes. It offers an executable `cbj` compiler similar to the `javac` compiler, where argument files are classbox-aware. Please note that this implementation is *naive* and serves only as a proof-of-concept for Java.

Our implementation handles three different ways of refining an imported class: (i) a new class member is added (*i.e.*, not redefined), (ii) a class member other than a method is redefined, and (iii) a method is redefined. The following sections examine each of these cases. We drew a distinction between redefined methods and other redefined class members because the former are dynamically looked up when messages are sent, but not the latter (which are statically bound). We then describe how the new method lookup semantics is implemented using dynamic introspection of the method call stack (Section 6.4). And finally we show how the transitivity of imports is handled (Section 6.5) and we present some limitations and possible improvements (Section 6.6).

6.1 Pure Class Member Addition

Class members that are new additions (not redefinitions) are inserted into the Java class without being modified. For instance, a classbox WidgetsCB defines an empty class Component, that is refined in a classbox EnhWidgetsCB.

```
//Classbox WidgetsCB
package WidgetsCB;
public class Component {
}
```

```
//Classbox EnhWidgetsCB
package EnhWidgetsCB;
import WidgetsCB.Component;
refine Component {
  private int color;
  public int color () {
    return color;
  }
}
```

When passed to our `cbj` preprocessor, the resulting Java package used to generate pure java bytecodes is:

```
package WidgetsCB;
public class Component {
  private int color;
  public int color () {
    return color;
  }
}
```

6.2 Redefinition of Class Members Other Than Methods

For class members that are not looked up (*i.e.*, variables, static fields, static initializations) a renaming is performed while compiling a classbox away. Classbox WidgetsCB defines a class Component that contains a variable color accessed by a method color1() and an inner class Color. This class is refined in a classbox EnhWidgetsCB with a new variable color, a method color2() and a new inner class Color.

```
//Classbox WidgetsCB
package WidgetsCB;
public class Component {
  Color color;
  public Color color1() {
    return color;
  }
  class Color {}
}
```

```
//Classbox EnhWidgetsCB
package EnhWidgetsCB;
import WidgetsCB.Component;
refine Component {
  Color color;
  public Color color2() {
    return color;
  }
  class Color {}
}
```

The resulted Java code gathers all the class members:

```
package WidgetsCB;
public class Component {
  WidgetsCBColor WidgetsCBcolor;
  EnhWidgetsCBColor EnhWidgetsCBcolor;
  public WidgetsCBColor foo() {
     return WidgetsCBcolor;
  }
  public EnhWidgetsCBColor bar() {
     return EnhWidgetsCBcolor;
  }
  class WidgetsCBColor { }
  class EnhWidgetsCBColor { }
}
```

6.3 Method Redefinition

Looking up methods that are redefined requires a new method lookup semantics (Section 3.2). When producing Java source code, method redefinitions are compiled into one method where each redefinition is contained in a if statement used to trigger the right definition according to the current position in the execution flow of the program (*cf.*, following section). The method paint() contained in the class Component is redefined in EnhWidgetsCB

```
//Classbox WidgetsCB
package WidgetsCB;
public class Component {
  public void update() {
    paint();
  }
  public void paint() {
    //Original paint
  }
}
```

```
//Classbox EnhWidgetsCB
package EnhWidgetsCB;
import WidgetsCB.Component;
refine Component {
  public void paint() {
    //Enhanced paint
  }
}
```

The pure Java source code produced contains only one paint() method that gathers the two implementations of the method.

```
package WidgetsCB;
public class Component {
  public void update() {
    paint();
  }
  public void paint() {
    if ( ClassboxInfo.methodVisible (
                ”EnhWidgetsCB”, ”Component”, ”paint”)) {
      //Enhanced paint
    }
    if ( ClassboxInfo.methodVisible (
                ”WidgetsCB”, ”Component”, ”paint”)) {
      //Original paint
    }
  }
}
```

ClassboxInfo is a generated class that (i) gathers some informations about the composition of classboxes needed at runtime like a description of the classboxes that were used to produce the Java code, and (ii) offers some methods useful to introspect the method calls stack. At runtime, when the update() method is invoked, one of the two implementations is executed according to the structure of classboxes inferred from the method calls stack.

6.4 Dynamic Introspection of the Method Call Stack

Whenever a redefined method is invoked, the method call stack is reified (by using the exception handling mechanism of Java) to build the structure of the classboxes.

```
//Classbox OldAppCB
package OldAppCB;
import WidgetsCB.Component;
public class OldApp {
  public static void main (String[] argv) {
    // Original paint method invoked
    new Component().update();
  }
}
```

When the main(...) method of the OldApp is invoked, before entering the paint() method the corresponding method call stack given by Java is:

```
WidgetsCB.Component.update() //Top of the stack
OldAppCB.OldApp.main() //Bottom of the stack
```

Using this stack reification and the information about the structure of classboxes kept in ClassboxInfo, the static method ClassboxInfo.methodVisible (“EnhWidgetsCB”, “Component”, “paint”) yields false, whereas ClassboxInfo.methodVisible (“WidgetsCB”, “Component”, “paint”) return true.

NewAppCB is a client of the refined Component:

```
//Classbox NewAppCB
package NewAppCB;
import EnhWidgetsCB.Component;
public class NewApp {
  public static void main (String[] argv) {
    // Enhanced paint method invoked
    new Component().update();
  }
}
```

In a similar way, before entering the paint() method, the method call stack is:

```
WidgetsCB.Component.update() //Top of the stack
NewAppCB.NewApp.main() //Bottom of the stack
```

Because the paint() method is redefined in the classbox NewAppCB, the new implementation has to be used: the static method ClassboxInfo.methodVisible (“EnhWidgetsCB”, “Component”, “paint”) yields true, whereas ClassboxInfo.methodVisible (“WidgetsCB”, “Component”, “paint”) return false.

6.5 Adapting Classbox Import to Package Import

Since class imports are transitive in Classbox/J, but not in plain Java, all transitive imports must be compiled away. In the resulting Java source code, each import statement must refer to the original package that defines this class.

For example, while producing the package corresponding to the classbox NewAppCB the import statement import EnhWidgetsCB.Component is translated into import WidgetsCB.Component because the class Component is defined in WidgetsCB.

6.6 Limitations and Possible Improvements

Since the current implementation is only intended to serve as a proof of concept, we feel it is important to raise a few points concerning the limitations of this prototype.

Native methods. A native method is a function written in a language other than Java. Only the signature of the method is declared within the Java class. Because such methods do not contain any Java code, they cannot be rewritten using the mechanism described above. As a consequence, native methods cannot be redefined.

Super call in a constructor. Constructors can be redefined as well as methods. Constructor redefinitions are compiled into one single constructor following the mechanism described in Section 6.3. This approach is, however, limited when a constructor performs a super call. Java enforces the constructor of the superclass to be executed *before* the constructor of the subclass: the super call has to be the first statement of the constructor. Therefore the body of a constructor cannot be embedded in a if statement.

Debugging facilities. Even with our current approach where classboxes are compiled away, information about classboxes needed to structure the system is available (class ClassboxInfo). This information is accessible with a debugger, however it is tedious to manually retrieve the defining classbox for a given class member. Development with classboxes would be more comfortable with a classbox-aware debugger.

Modifying the VM. Prior to this work, we implemented two versions of the classbox model in Smalltalk: (i) by implementing a new method lookup algorithm within the VM [5], and (ii) by using bytecode transformation and method context reification on a normal VM [4]. The cost of the former strategy is about 1.1 times slower and the latter is about 1.25 times slower (these figures were obtained by comparing the execution times of a normal Smalltalk application in a classbox and a plain environment).

The Java VM does not provide a bytecode that reifies the context of a method call. Therefore, the latter strategy cannot be implemented in Java. By modifying the Java VM to implement a new method lookup algorithm [5], we expect to achieve a similar speedup. Whereas with this approach we would need to modify the VM (which can be tedious), the advantage is that classboxes would be transparent in term of run-time cost.

7. RELATED WORK

Over the last decade considerable research has focused on new ways to modularize or change a system. One main line of our work has been to keep the notion of class and package distinct. This has to be put in contrast with systems like virtual classes [21] or hierarchy inheritance [8] where classes and modules are unified under a common lookup algorithm operating on namespaces that serve as classes and modules.

The related work presented in this section can be classified according to five families: (i) class extensibility (class extensions, Unit, Jiazzi, open classes), (ii) module (MixJuice, MJ), (iii) alternative inheritance (mixins, virtual classes, nested and hierarchy inheritance), (iv) other approaches (AOP, namespaces).

Class Extensibility

Class extension. CLOS [17], Smalltalk [14] and Objective-C [27, 29] allow an already existing class to be extended with new methods or method redefinitions (not in Objective-C). These *class extensions*, however, are global, which leads to conflicts when two packages extend the same class with the same methods. The resolution policy usually adopted is that the last version of a redefined method is the one that will be globally used. As a consequence, only a single version of a class can be present in a running system. Classboxes make it possible for multiple versions of the same class to be present in the system at the same time.

Unit. MZScheme [13] offers an advanced module system in which a *unit* is the basic building block. A unit is a packaging entity composed of requirements, definitions and exports. Mixins are defined by creating within a unit a subclass of a class that will be provided by other units at linking time. Units have to be instantiated and composed with each other to form a program. Reusability and extensibility are expressed by recombining units. An application, made of units, can be recomposed and by aliasing new units can inserted. Units differ from classboxes since a unit acts as a black box: a class within a unit cannot be refined. Instead a new unit has to be provided and included in a recomposition. New mixins can be defined to extend a base system, but we fall again in all the problem related to using inheritance. Therefore not much would have been gained if Swing had been refactored with units.

Open classes. MultiJava [25] is an extension of Java that supports open classes and multiple method dispatch. An open class is a class to which new methods can be added. Method redefinitions are not, however, allowed: an open class cannot have one of its existing methods refined.

Jiazzi. The *unit* system of MZScheme has been ported to Java. Jiazzi [22] is an enhancement of Java that adds support for encapsulated code modules as unit. The main difference with MZScheme is that Jiazzi enables the creation of open classes that can be enhanced with new methods and fields without invasively modifying the original definitions or breaking their existing subclasses. This enables a modularization of cross-cutting concerns [23]. Refinements occur with links between units. The difference with classboxes are twofold: (i) classes defined in the same unit are tied together. Let's assume a class PointFactory and a class Point are contained into the same unit, and Point is imported and refined with a color feature in another unit. Because PointFactory is defined in the same unit that the colorless version of Point, even if PointFactory is also imported in the unit containing the color addition, there is no way for the factory to produce colored points. (ii) Refinement applications are implemented with subclassing, therefore an instance of Point produced in a unit is not an instance of the refined class Point.

Modules

Mixjuice. Mixjuice [16] defines difference-based modules, in which a module can refine a class defined in another module by adding new class members. A refined class constitutes a new version. Contrary to classboxes, with MixJuice multiple versions of the same class *cannot*, be present in the system at the same time.

MJ. MJ [9] is a module system for Java that provides a high-level interface to abstract low-level Java technical issues related to class loading. The focus of MJ is to support the deployment of different versions of the same package. As such with MJ changes cannot be added to existing classes. In MJ, a module contains the following information: (i) class definition, (ii) dependencies with classes offered by other modules, (iii) access control for this module's provided classes like class privacy and restriction for the clients in subclassing provided classes, and (iv) some initialization code.

By removing some technical limitations of the dynamic class loading mechanism related to the use of `CLASSPATH`, MJ allows multiple versions of a class to coexist at the same time within a system. These versions are referenced by different namespaces

(*i.e.*, classloaders), therefore, they are considered to be two different classes. New versions of a class cannot be propagated to formerly collaborating classes without modifying the original dependancies: modules are considered to be black boxes in which contained classes cannot be modified. This mechanism differs from classboxes because, for a given class, formerly collaborating classes can be reused with new versions of the original class. MJ cannot be used to refactor Swing to our new architecture since classes cannot be extended with new changes.

Alternative Inheritance

Virtual classes. Virtual classes were originally developed for the language BETA [18], primarily as a mechanism for generic programming rather than for extensibility [21]. Keris [37], Caesar [24], and gbeta [11] offer such a mechanism, where method and class lookup are unified under a common lookup algorithm. Virtual classes are not statically safe because they permit types of method parameter to change covariantly with subtyping. In a similar way that a method is looked up according to an instance, a class is looked up according to an instance (*i.e.*, an encapsulating class). With such a unification of method and class lookup, the role of a class is overloaded with semantics of packages and objects constructor. With classboxes, we keep the original meanings of class and package separate.

Hierarchy Inheritance. Cook [8] presents a use of inheritance as a derivation of modified hierarchies or other graph structures. Links between nodes in a graph are interpreted as self-references from within the graph to itself. By inheriting the graph and modifying individual nodes, any access to the original nodes is redirected to the modified versions. For example, a complete class hierarchy may be inherited, while new definitions are derived for some internal classes. The result of this inheritance is a modified class hierarchy with the same basic structure as the original, but in which the behavior of all classes modified that depend upon the classes explicitly changed is modified. Hierarchy inheritance is based on having a lookup of classes and on relationship between group of classes, whereas with classboxes, no class-lookup is involved and import is done at the class-level.

Nested inheritance. The Jx programming language [26] is an extension of Java where members of an encapsulating class or package may be enhanced in a subclass or subpackage. Packages may have a declared inheritance relationship. Nested classes in Jx are similar to virtual classes. Unlike virtual classes, nested classes in Jx are attributes to their enclosing class, not attributes of instances of their enclosing class. The difference with classboxes is that in Jx (i) inheritance is overloaded with import semantics, and (ii) a class is defined in only one classbox and can be extended by others, whereas with Jx classes are looked up according to the inheritance defined between packages and between classes.

Scala. Scala [32] is a statically-typed object-oriented and functional programming language developed at EPFL, the École Polytechnique Fédérale de Lausanne. Scala introduces a new concept to solve the extensibility problem (Section 2.2): *views* allow one to augment a class with new members. Views follow some of the intuitions of Haskell's type classes, translating them into an object-oriented approach. The scope of a view can be controlled, and competing views can coexist in different parts of one program. A view is statically applied by the compiler to satisfy type constraints. For instance, if a variable anA is of type A, the compiler would translate an expression var aB: B = a, which declares a variable aB of type B and initializes it with a reference to anA, as var aB: B = view(anA), where view is a method (or a function) provided by the programmer, taking an argument of type A and returning an object of type B. In Scala a conversion is done by using type information provided by the programmer whereas with classboxes the scope of change depends on the graph of classboxes involved in the computation.

Mixin Layers. A collaboration-based design [15, 36] aims at supporting large-scale refinements. A *collaboration* is a set of *roles* applied to a set of *participant objects*. Collaborations are layered linearly to form an application. In mixin layers [33], Smaragdakis and Batory represent a collaboration as a C++ template, a role as a mixin [6], and a participating object as a class. A layered application that uses mixin layers is open to changes by adding new collaborations. However, for an application that is not layered, mixin layers do not offer a satisfying solution to support unanticipated changes.

Feature-oriented programming. Feature-Oriented Programming is the study of feature modularity in product-lines [30]. AHEAD [3, 20] is an approach to Feature-Oriented Programming (FOP) where a base system is regarded as a constant and refinements intended to be added are functions adding features to this base system. A refinement is a function that takes a program as input and produces a refined program as output. FOP advocates program construction as a set of functions applied to a base system. New changes are modeled as new functions. Contrary to classboxes, AHEAD does not support multiple versions of the same class living in the same system.

Generic type. Torgersen [35] uses generic type extensions of C# and Java to solve the extensibility problem in a secure and type safe manner. His solutions rely on the use of F-bounds [7] and wildcards in the declaration of type variable to make them type-safe when a system is extended with new data-types and operators. However, use of generic type has to be foreseen prior to apply an extension, as a consequence, this approach does not fit to support unanticipated changes.

Other Approaches

Aspect-oriented programming. Hyper/J [28] is based on the notion of *hyperspaces*, and promotes composition of independent *concerns* at different times. Hyperslices are building blocks containing fragments of class definitions. They are intended to be composed to form larger building blocks (or complete systems) called *hypermodules*. A hyperslice defines methods for classes that are not necessarily defined in that hyperslice: class members are spread over several hyperslices. With its notion of inter-type, AspectJ [1] allows class members to be separated from the class definition by being defined in an aspect. Whereas with classboxes a class can be refined in two classboxes with two method having the same name, with Aspect/J conflicts are not allowed: two aspects cannot define two methods having the same name on the same class. This kind of extension does not allow redefinition and consequently does not help in supporting unanticipated evolution.

Sister namespaces. In Java, a class type is uniquely identified at runtime by the combination of a class loader and a fully qualified class name. The same class loaded into two different class loaders (*i.e.*, namespaces) has two distinct types [31]. Let's assume that two classloaders N1 and N2 load the same class C. One instance of the class C in the classloader N1 cannot be regarded as an in-

stance of C in a second classloader N2 because they have different types. This is identified as the *problem of the version barrier*. *Sister namespaces* [31] relax the version barrier between application components by defining the notion of binary compatibility and extending the type checker. Sister namespaces make the exchange of instance of different class versions possible across classloaders by relaxing the type checker. To be compatible, two class versions has to be "close enough", whereas with classboxes a class can be refined with any kind of class members.

8. CONCLUSION

Classboxes address the problem of delimiting visibility of a change to a restricted scope in order to avoid conflicts with other changes and to avoid impacting clients that should not be affected. In a classbox, classes can be defined, classes can be imported from other classes, and class members can be defined for any classes visible (*i.e.*, defined or imported) in this classbox. Classboxes offer an elegant way of bringing some unanticipated changes over a system while delimitating the impact of these changes.

In this paper, we present the Java implementation of this model by adding a small number of constructs to Java. A classbox is a Java package where imported classes can be refined with new class members and imported classes that are refined or not to be reimported in other classboxes. Having a Java version of the model shows that classboxes can be applied to a statically-type language like Java.

By refactoring Swing, we stress-tested the classbox model by applying it to a large case study. Our new version of Swing removes (i) the incoherence in the original Swing hierarchy and (ii) the code duplication that was introduced due to the limitations of the Swing inheritance hierarchy. Moreover, while refactoring, we found the need to extend the classbox model with a new construct that allows a previous definition of a redefined method to be accessed.

As a future work we plan to enhance the notion of refinement in order to enable the use of classboxes as a way to express general changes that can be applied to a system (and not just additions or redefinitions of class members).

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1).

We also like to thank Gabriela Arévalo, Marcus Denker, Günter Kniesel, Sean McDirmid, Eric Tanter, Matthias Rieger, Klaus D. Witzel for their valuable comments and discussions.

9. REFERENCES

[1] AspectJ home page. http://eclipse.org/aspectj/.

[2] Awt api. http://java.sun.com/j2se/1.3/docs/api/java/awt/package-summary.html.

[3] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *Proceedings ESEC/FSE-11*, pages 48–57, New York, NY, USA, 2003. ACM Press.

[4] 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, May 2005.

[5] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag.

[6] G. Bracha and G. Lindstrom. Modularity meets inheritance. Uucs-91-017, University of Utah, Dept. Comp. Sci., Oct. 1991.

[7] P. S. Canning, W. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, Sept. 1989.

[8] W. R. Cook. *A Denotational Semantics of Inheritance*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[9] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: a rational module system for Java and its applications. In *Proceedings OOSPLA 2003*, pages 241–254. ACM Press.

[10] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[11] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[12] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 94–104. ACM Press, 1998.

[13] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[14] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[15] I. M. Holland. Specifying reusable components using contracts. In O. L. Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 287–308, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[16] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer Verlag.

[17] S. E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 1989.

[18] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, Cambridge, Mass., 1987.

[19] W. LaLonde and J. Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, Jan. 1991.

[20] R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technlogies. In *Proceedings ECOOP 2005*.

[21] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*,

volume 24, pages 397–406, Oct. 1989.

[22] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, Oct. 2001.

[23] S. McDirmid and W. C. Hsieh. Aspect-oriented programming with jiazzi. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 70–79, New York, NY, USA, 2003. ACM Press.

[24] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.

[25] T. Millstein, M. Reay, and C. Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings OOSPLA 2003*, pages 224–240. ACM Press.

[26] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *: Proceedings OOPSLA 2004*, pages 99–115. ACM Press.

[27] The objective-c programming language. http://developer.apple.com/documentation/Cocoa/-Conceptual/ObjectiveC/index.html.

[28] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000.

[29] L. J. Pinson and R. S. Wiener. *Objective-C*. Addison Wesley, 1988.

[30] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 419–443, Jyväskylä, June 1997. Springer-Verlag.

[31] Y. Sato and S. Chiba. Loosely-separated "sister" namespaces in java. In *Proceedings ECOOP 2005*.

[32] Scala home page. http://lamp.epfl.ch/scala/.

[33] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 11(2):215–255, Apr. 2002.

[34] Swing api. http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-summary.html.

[35] M. Torgersen. The expression problem revisited — four new solutions using generics. In M. Odersky, editor, *Proceedings ECOOP 2004*, LNCS, Oslo, Norway, June 2004. Springer-Verlag.

[36] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings OOPSLA '96*, pages 359–369. ACM Press, 1996.

[37] M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

[38] M. Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, University of Lausanne, EPFL, 2003.