Scoped and Dynamic Aspects with Classboxes

Alexandre Bergel – Stéphane Ducasse

Software Composition Group — IAM, University of Bern, Switzerland
Language and Software Evolution Group — IAM, Université de Savoie, France

ABSTRACT. *Atomically introducing changes to a group of classes is a challenging task. In addition, certain applications require that changes be applied dynamically without shutting down and restarting the application. In this paper we present an extension of classboxes to make them dynamic. A classbox is a kind of module that supports class extension (method addition and redefinition, and instance variable addition). Class extensions and definitions defined in a classbox represent an aspect. A classbox cross-cuts different classes by defining various extensions. In addition, with classboxes, aspects are dynamically applied to, removed from and "hot-swapped" in a system. Such aspects may crosscut a large number of classes which are extended by adding or redefining existing methods and adding new instance variables. Finally the aspects are scoped* i.e., *changes defined by a classbox are only visible inside this classbox and its clients.*

RÉSUMÉ. *Introduire des modifications à un ensemble de classes de façon atomique est une tâche difficile. De plus, certaines applications nécessitent que ces modifications soient appliquées à l'exécution sans interruption. Nous présentons le modèle des classboxes, une sorte de module offrant des extensions de classes et additions de variables d'instance. Les extensions et définitions contenues dans un classbox représentent un aspect. Un classbox s'applique de façon transverse à différentes classes en définissant différentes extensions. De plus, les classboxes permettent d'appliquer, de retirer et d'échanger dynamiquement des aspects. Un aspect peut s'appliquer sur un grand nombre de classes en ajoutant et en redéfinissant des méthodes et en ajoutant des variables d'instances. Finalement les aspects ont une visibilité restreinte, c'est-à-dire que les modifications apportées par un classbox sont visibles uniquement dans ce classbox et ses clients.*

KEYWORDS: *aspect, dynamic adaptation, open-classes, class extension, module system.*

MOTS-CLÉS : *aspect, adaptation dynamique, classe ouverte, extension de classe, système de module.*

RSTI – L'object – 11/2005. Programmation par aspects, pages 53 à 68

1. Introduction

While it is possible to design an application to be adaptable in specific ways, by using approaches such as the Strategy design pattern [GAM 95], it is difficult, if not impossible, to anticipate all the ways in which applications may need to be adapted.

Dynamically adapting a running application means changing its behavior without stopping and restarting it [POP 03, RED 02, POP 02, OLI 99, MAL 00, DMI 01, CLI 00]. Mobile devices and embedded systems often require dynamic adaptation [SAR 04]. However this has to be specified statically, at compile time. Recently, there has been a growing interest in using dynamic aspect-oriented techniques [POP 02], class replacement [MAL 00, DMI 01] and class extension [CLI 00] to support dynamic adaptation. However, existing implementations either limit the kinds of changes that can be applied, or require a modified virtual machine.

Without a scoping mechanism, composing several aspects that extend a class is delicate (especially when these aspects overlap each other) and requires a dedicated language (Hyper/J [OSS 00]). Assimilating an aspect as a scope that bounds the visibility of its extension makes composition easier: aspects that conflict with each other can be used at the same time because their definitions are visible in different scopes.

Our approach to dynamically applying aspects is based on classboxes [BER 03]. A classbox is a kind of module that supports scoped definitions of a class extension. This paper describes an extension to classboxes (i) to support addition of instance variables, and (ii) to make them fully dynamic: they can be dynamically and atomically loaded and unloaded.

Within a classbox, classes are defined or imported from other classboxes. Imported classes can then be extended. These extensions consist in adding new instance variables, and adding and redefining methods. However, such extensions have bounded visibility: variables and methods defined on a class are visible *only* visible from the perspective of this classbox.

In this paper we motivate dynamic adaptation (Section 2). Then the classbox model is described by showing how it solves the adaptation problem (Section 3). Finally, some implementation issues are described (Section 4).

2. Dynamic Adaptation Needs

We motivate the need for dynamic adaptation and show how traditional approaches do not offer satisfactory solutions.

2.1. *A Motivating Example*

While the primary function of a cell phone is to make phone calls, nowadays all cell phones provide numerous advanced display facilities (*i.e.,* colors, aliasing, 3D,

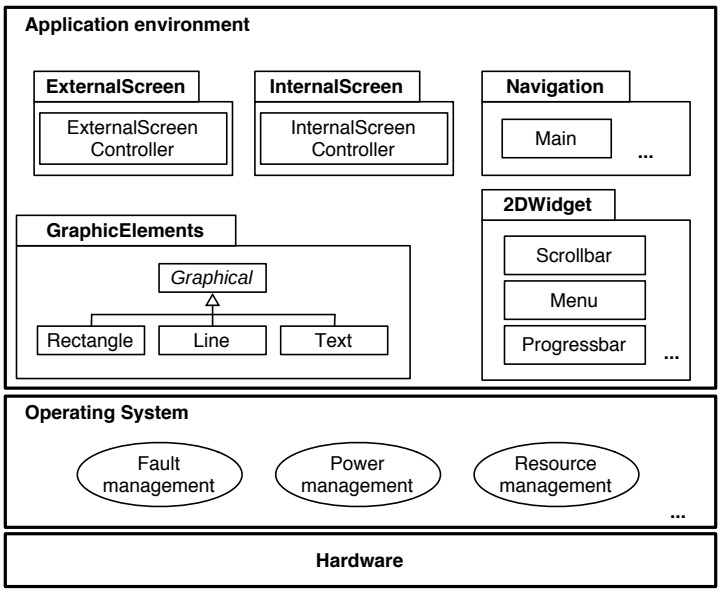

Figure 1. *The application environment contains 5 parts:* Navigation *defining the main application, two controllers related to the internal and external screen (*InternalScreenController *and* ExternalScreenController*),* 2DWidget *offering some graphical user interfaces widgets and* GraphicalElements *containing low-level graphical facilities*

animations, etc.). Such advanced features are considered to be *non-essential* and, in case of particular situations such as device overheating or battery power shortage, can be disabled to preserve the primary cell phone behavior [SAR 04]. During a phone call, disabling non-essential features to spare the battery must not interrupt the communication. Under these conditions software systems contained in a cell phone cannot tolerate being halted in order to be recompiled: they have to be adapted on the fly.

Figure 1 depicts the parts of a mobile cell phone related to its display capability. Often cell phones offer two LCD screens: an internal one only usable when the phone is open, and an external one displaying information about incoming calls. The external screen usually has fewer capabilities than the internal one. The cell phone's operating system provides an abstraction of the hardware to the application environment. It contains several modules related to fault management, power management and resource management.

The application environment contains (i) the user interface defined by the navigation application (Navigation), (ii) a controller for the internal screen and (iii) another for the external one, (iv) graphical user interface widgets like scroll bars, menus,

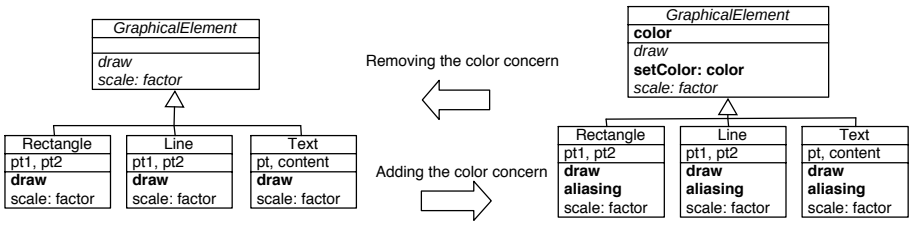

Figure 2. *Adding and removing a color concern throughout a hierarchy of classes (bold elements indicate variation)*

progress bars (2DWidget) using (v) a lower-level set of graphical elements used by the widgets such as texts, lines or rectangles (GraphicalElement).

2.2. *Dynamic Adaptation*

A typical scenario of dynamic adaptation occurs when the fault manager receives a low battery event triggered by the hardware. The power consumption has to be reduced, so the power manager asks the resource manager to reduce its needs by downgrading some non-essential facilities like the display [SAR 04]. When the battery is full and the power consumption is not restricted, the display uses advanced animations and 3D rendering to display widgets to offer a more attractive display to the end-user. Reducing the display facility in case of power shortage (less than 20 % of the battery left) consists in keeping the internal screen fully active but removing the animation and using a colored two dimensional rendering on the external screen. When the lower threshold of 8 % approaches, 3D is removed from the internal screen and the external screen becomes monochrome.

Bringing changes to a hierarchy. Switching from a 3D rendering to a 2D rendering is done by changing the set of methods defined on the widgets. However, if the graphical elements have to be monochrome, the color concern applied to the hierarchy has to be removed from the existing hierarchy. This modification consists in removing an instance variable and a method setColor: color from a class root of the hierarchy, and providing a draw method for each of the graphical elements (subclasses of GraphicalElement). These new methods draw do not use the color variable.

Figure 2 presents a typical problem of extension. It shows the changes brought to a hierarchy consisting in adding or removing a color concern. Adding the concern means adding an instance variable color and a method setColor: color to the abstract class GraphicalElement. Also, each of the subclasses is extended with draw and aliasing methods that use the color variable. Removing this concern means removing the aliasing method, replacing the colored draw method by the monochrome one (that does not use the color variable), and removing this variable.

Key problems. Dynamic adaptation of an application consists in changing the definition of the application to adapt its configuration according to its context and environment at a given moment in time. As illustrated by the above example the following points have to be addressed:

– **Dynamicity.** These changes have to be applied dynamically: a cell phone cannot be interrupted, stopped and then restarted. For instance changing the display configuration during a phone call should not be perceived by the user. Changes have to be loaded when they are necessary and unloaded at runtime to spare memory and battery consumption [SAR 04].

– **Small runtime.** The runtime system has to be kept small and simple. This means that a compiler should not be part of the runtime. Changes should be made without requiring any source code.

– **Changes crosscut several classes.** Adapting a part of a system requires some dynamic changes, such as adding or removing piece of state (instance variable) or fragment of behavior (set of methods), that have to be applied to a set of classes.

2.3. *Scoped Changes*

Due to space limitations, redundancies in the system have to be avoided. Even if one screen is colored and not the other, there should be only one hierarchy of graphical elements used by the widgets. This is achieved by scoping the changes applied to the application. Only some particular changes, like a color concern applied to a set of classes, are accessible within one part of the system (*e.g.,* internal screen) whereas these extensions might not be visible within another part (*e.g.,* external screen).

Key problems. Changes required by different parts of a system have to be scoped separately to avoid conflicts when applying different changes at the same time and to limit the visibility of the changes to the parts of the system that actually needs them.

2.4. *Limitation of Subclassing*

Extending a class by creating a subclass imposes some constraints on clients of the extended classes [FIN 98, BER 03]: references contained in a client have to be updated. Extending a hierarchy using subclassing also leads to code duplication and static type issues (*e.g.,* unwanted type casts). Figure 3 shows how the base system containing the graphical hierarchy is extended based on subclassing with a color concern. It is assumed that we only consider here single inheritance. Each graphical element adds the instance variable color, redefines the method draw (to take the color into account), and adds the methods setColor: color and scale: factor.

This leads to several drawbacks:

– **Duplication of code.** The color variable and the setColor: color method are duplicated as many times as there are graphical elements. The consequence is that

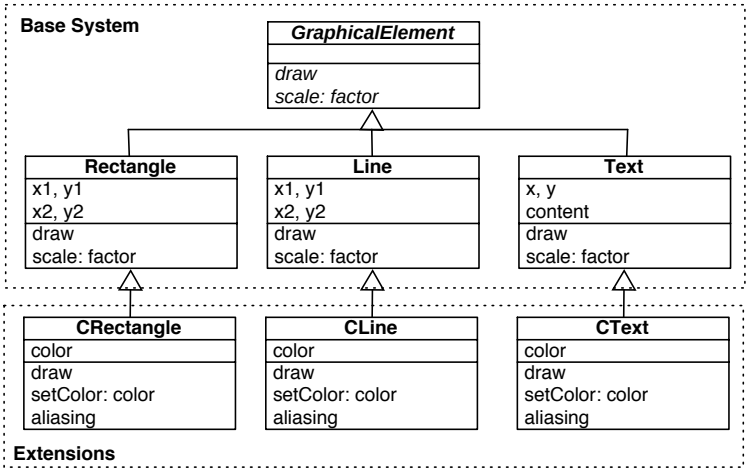

Figure 3. *Extensions by subclassing*

maintenance becomes harder and more error-prone. For instance a future evolution that makes the color concern evolve would also be duplicated.

– **Clients need to be adapted.** The reuse of existing classes (originally intended to be used with the first version of the library) implies some adaptations because these classes refer to original classes (Rectangle, Line, etc.) and not the subclasses representing the extensions (CRectangle, CLine, etc.).

– **Statically typed languages.** In Figure 3 each of the nodes is subclassed, leading to a large number of type casts in order to ensure type safety. For instance the aliasing method can be used only through the type of the extension (CRectangle, CLine, and CText) and not from the type defined by GraphicalElement. By using single inheritance, subclassing the class GraphicalElement defines a new hierarchy with ExtendedGraphicalElement at the top, duplicating the classes Rectangle, Line, and Text.

3. Supporting Dynamic Adaptations With Classboxes

3.1. *Classboxes*

The classbox model [BER 03] is a module system for dynamically typed object-oriented languages that limits the visibility of class extensions. A *class extension* reshapes a class by adding new instance variables and adding or redefining some methods. Classboxes restrict the visibility of the class extensions defined by a classbox to

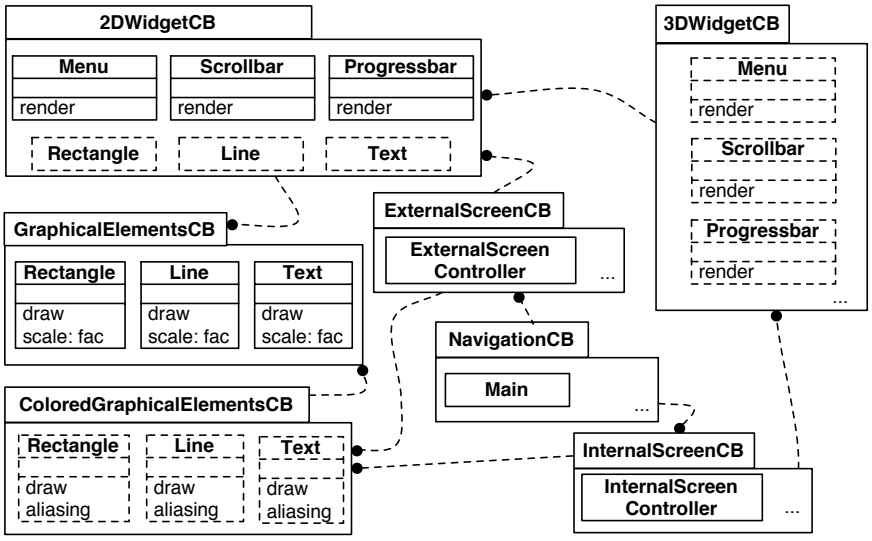

Figure 4. *Example of configuration where the internal screen is monochrome using 2D widgets and the external screen is colored with 3D widgets*

the classbox itself and to its clients *i.e.,* other classboxes importing the present one. A classbox is a unit of scoping (*i.e.,* it acts as a namespace).

A *classbox* is a module where (i) classes can be defined, (ii) classes from other classbox can be *imported*, (iii) classes can be extended by adding instance variables and by (iv) adding or redefining methods.

Figure 4 shows two classboxes, GraphicalElementsCB [1] and ColoredGraphicalElementsCB. The first one defines three classes. Imported classes are represented using dashed boxes and the original definition is designated using a dashed curved line. For instance 2DWidgetCB imports the classes Rectangle, Line and Text from GraphicalElements without extending them. All the classes visible in 2DWidgetCB are imported by 3DWidgetCB (shown by the dashed curved line between 2DWidgetCB and 3DWidgetCB) and then classes Menu, Scrollbar and Progressbar are extended with a method render.

The figure illustrates a configuration in which the internal screen uses 3D widgets whereas the external screen uses 2D widgets. Because classes defined by ColoredGraphicalElementsCB are imported in ExternalScreenCB and InternalScreenCB, widgets used by both screens are colored.

1. CB designates a Classbox.

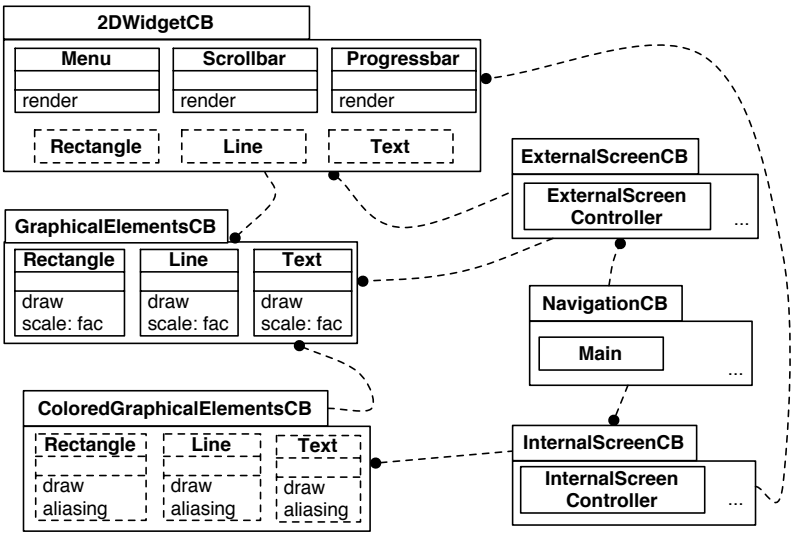

Figure 5. *Rearranged classboxes: the external screen is monochrome and the internal one colored*

3.2. *Dynamic Application of Classboxes*

Classboxes can be installed and un-installed at runtime. There is no need to stop and restart a running system. No source code is required. A classbox can be dynamically swapped by a new classbox or a set of new classboxes: the new classbox(es) must provide at least the same set of classes.

Rearrangement of classboxes. When the amount of energy left in the battery enters a different range, connections between the classboxes are modified and classboxes that are not used anymore are unloaded. For instance with less than 8 % of the battery power left the internal screen uses colored 2D widgets and the external one monochrome 2D widgets: the 3D facility becomes unnecessary. Classboxes are then rearranged (Figure 5) to make the internal screen use colored 2D widgets and the external one use monochrome 2D widgets: InternalScreenCB now imports the widgets from 2DWidgetCB, ExternalScreenCB imports monochrome graphical elements from GraphicalElementsCB. The classbox 3DWidgetCB becomes unnecessary and is unloaded from the memory to save some energy.

On-the-fly Method Switching. If a method is replaced while it is activated, then previous calls continue with the former definition while any new call triggers the new definition. This is an approach similar to Java's [DMI 01] where all invocations of old methods are allowed to complete, whereas all method calls initiated after class redefinition go to the new method. This is also true if a method calls itself after being

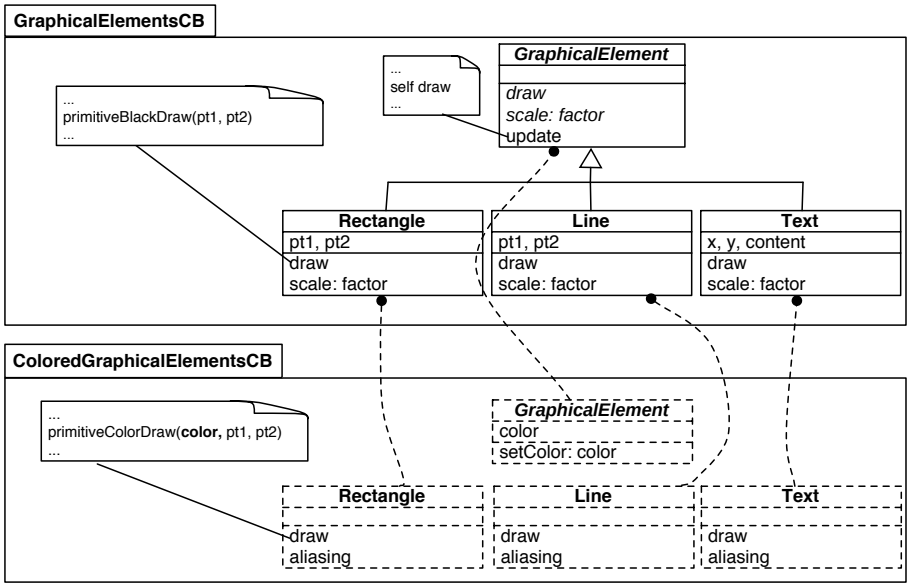

Figure 6. *Adding a color concern over a graphical elements hierarchy*

changed. The call invokes the new method definition, and then goes back to the old definition to be continued.

3.3. *Class Extension*

We define a *class extension* to be a modification of the behavior of a class (by adding or redefining a method) or an extension of its state (by adding new instance variables).

Importing a class makes it visible in the importing classbox scope. A classbox can import classes from several other classboxes and extend them. Extended classes can also be imported and then extended again by other classboxes. Therefore classboxes support the definition and application of changes to several classes distributed over several classboxes.

The solution of the extensibility problem presented in Section 2.2 using classboxes is represented in Figure 6. GraphicalElementCB defines the original framework. ColoredGraphicalElementCB imports each class defined in GraphicalElementCB and extends it. The abstract class GraphicalElement is extended with a new color instance variable and a mutator for it (setColor: color). Each subclass of this class is extended with the methods draw (to take the color into account) and aliasing.

Class extensions are characterized by five features: (i) A class can have its behavior modified by adding or redefining a method and (ii) new variables can be added to a class, (iii) these extensions are local to the classboxes that define them, (iv) a classbox that imports an extended class sees these extensions but (v) local extensions take precedence over imported code (*local rebinding*).

Adding or Redefining Methods. The behavior of an imported class (*i.e.,* defined in another classbox and imported) can be modified by defining new methods or redefining existing ones. For instance in Figure 6 the classbox ColoredGraphicalElementCB defines some new methods on the imported class. Each of the concrete classes has a new method (aliasing) and the method draw is redefined. Note that adding a new method to a class makes it visible to its subclasses. This is the case for example with the method setColor: color which is added to the class GraphicalElement. Within the classbox ColoredGraphicalElementCB, instances of the class Rectangle understand this method.

Adding New Instance Variables. Within a classbox, new instance variables can be added to imported classes. In the classbox ColoredGraphicalElementCB a new instance variable color is added to the abstract class GraphicalElement. This variable is accessible from new methods such as setColor: color and in new methods added to GraphicalElement's subclasses. The draw methods implemented on Rectangle, Line and Text can freely refer to color. Adding a variable that already exists to an imported class *hides* the previous declaration and the value held by this previous declaration becomes inaccessible. This is useful for an evolving classbox to not raise conflicts (*i.e.,* variables clashes) with classboxes that depend on it.

Locality of Extension. A classbox extending a class bounds the visibility of elements it contains to itself and to classboxes that import the extended classes. This means that extensions brought to a class are *only* accessible from the classbox that defines them and to other classboxes that import the extended classes. A classbox that extends a class with new instance variables and methods does not impact other clients that rely on the original definition of this class. Therefore several classboxes adding the same variable or redefining the same method do not conflict with each other.

Importing Extended Classes. Any class visible in a classbox (even if extended) can be imported by other classboxes. Therefore a chain of classboxes can incrementally add new extensions to a class. From the point of view of a client of a classbox, there is no difference between a class that is imported and a class that is defined in the source classbox. Any extension can be applied to it.

Within a classbox all the visible classes have to have different names. However, a diamond graph of imports may imply the use of different class extensions defined by several classboxes. This occurs when two subclasses of two versions of a common superclass are imported in the same classbox. As illustrated in Figure 7, in the classbox AppCB, sending the draw message to an instance of LabelButton invokes the implementation of draw on GraphicalElement defined by WidgetSetCB. In a sim-

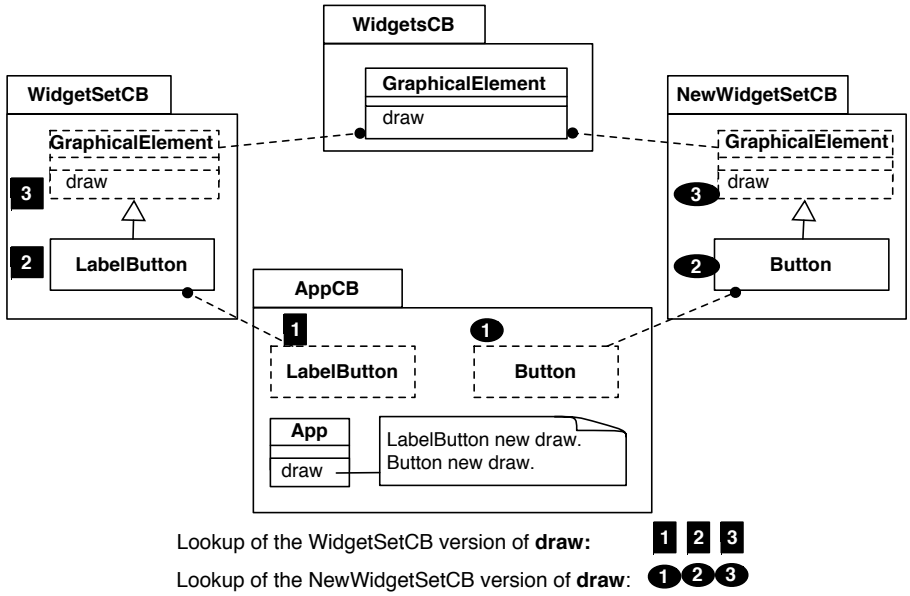

Figure 7. *Within one classbox, different version of the same class can be accessible. From AppCB sending the draw message to a LabelButton triggers WidgetSetCB's refinements, whereas sending it to a Button triggers NewWidgetSetCB's*

ilar way, sending this message to an instance of Button triggers the implementation brought by NewWidgetSetCB on GraphicalElement.

Local Rebinding. This is a key property of the classbox model. Redefining some methods in a classbox gives them priority over the imported methods whenever a message is sent. From the viewpoint of this classbox, it is as if the redefinition were global: any message that refers to the former methods triggers the new definition.

Figure 6 shows one example of local rebinding. The classbox GraphicalElementsCB defines a hierarchy with GraphicalElement as the root. This class contains an abstract method draw and a concrete method update that calls the method draw. The method draw is then overridden in each of the subclasses.

The classbox ColoredGraphicalElementsCB imports each class defined in the first classbox and extends them by redefining the method draw. Invoking update *within* ColoredGraphicalElementsCB calls the locally redefined method draw. This is an illustration of the local rebinding facility.

4. Implementation

Our current implementation is made in Squeak [ING 97], an open-source Smalltalk implementation. In this section we discuss some central implementation points and give some performance results.

Adding or Redefining Methods. In Smalltalk, it is trivial to dynamically modify methods of a class. All the methods of a class are contained in an instance of the class MethodDictionary. This instance lives in memory (and not inside the VM) which enables it to be manipulated as any object. Adding a new entry or replacing the value of a key in a method dictionary is performed without any recompilation.

Locality. Extensions are confined to the classbox that defines them and to other classboxes that import the extended classes. This is achieved by having two dictionaries (methods and instance variables) attached to each classbox for each class. The method lookup algorithm has a new semantics to take classboxes into account. This is achieved *without* any modification of the virtual machine. It rather uses a message passing control (in Squeak, messages can be reified by replacing a compiled method with an object that understands the message run: selector with: arguments in: object) and some reflection mechanisms (using the pseudo-variable thisContext the complete method call stack is accessible).

Updating Instances. When new instance variables are added to a class the natural question regards existing instances of the modified class [RIV 97, SER 99]. These need to be adapted by having their size augmented by the number of new variables added. These variables are initialized with an empty value (nil). The state of a class is atomically extended. No thread or external signal can interrupt this operation. This is necessary to preserve the consistency of the running system.

Adapting Bytecodes. According to the usual set of bytecodes used in virtual machines, an object instance variable is accessed using an offset. For instance the method scale: factor in the class Rectangle accesses the pt1 instance variable by referring to it through the first field of the object.

Adding a color instance variable in GraphicalElement triggers an adaptation process of the scale: factor method because the first field now corresponds to the color instance variable.

The variable color has to be placed at the first offset in the class GraphicalElement. This guarantees that all the objects issued from a class hierarchy have a similar and homogeneous variable structure. Such a linearization of variables is obtained from manipulating the bytecodes of all methods (including extensions) of GraphicalElement and adapting the bytecodes that represent accesses (reads and writes) to instance variables.

Performance. With our current implementation, the cost of installing a classbox that extends 100 classes with 500 methods is about 3560 milliseconds. Uninstalling it takes about 500 milliseconds. In comparison, loading the same amount of code in

Java takes 190 milliseconds. We are confident that the initial performance figures for installing and uninstalling classboxes can be significantly improved by using a binary storage format rather than a textual one.

Having added dynamicity to classboxes does not bring any additional runtime cost when invoking a method that is added by only one classbox. Calling a normal method (result of a method addition) does not bring any overhead. However, there is an overhead of about 10 % when calling a redefined method [BER 05]. This overhead is due to the local rebinding facility: when sending a message, the proper method implementation is computed according to the classbox where it is invoked from (based on the method call stack). Note that no overhead is introduced with class instantiation.

Performance figures obtained with classboxes have to be contrasted with those of other mechanisms for dynamic adaptations of behavior. IguanaJ [RED 02] is 24 times slower than a normal JVM to call a method and return from it. Object creation in IguanaJ is 25 slower than a classical Java VM. Most of the other runtime reflective architecture like PROSE [POP 02] or Guaraná [OLI 99] give similar figures.

5. Related Work

Dynamic adaptation has been addressed by many researchers. We classify them according to three different techniques: static code adaptation, dynamic code adaptation and dynamic extension and replacement of classes.

Static Code Adaptation. Many mechanisms supporting aspect-oriented programming are based on source code transformation (AspectJ [KIC 01], Hyper/J [OSS 00], Knit [EID 02], EAOP [DOU 02], AspectC# [GAL 01, KIM 02], AspectR [Asp]). Operating directly at the source code level has many advantages. For instance AspectJ handles a wide range of join points (calling a method and returning from it, accessing and modifying a field, throwing exceptions, initializing object, etc.). Hyper/J allows a very fine degree of composition to be specified when composing concerns. In Aspectual Components [OVL 02][LIE 99] a collaboration defines fields and methods that can be compiled separately from the base application. Binary adaptations are performed using a pre-compilation phase.

These systems support encapsulation of cross-cutting concerns intended to be installed on a base system at system implementation. Therefore, they do not provide any support for dynamic adaptation.

Dynamic Code Adaptation. With PROSE [POP 02] aspects can be woven and unwoven at run-time. PROSE allows advices to be joined on a smaller number of join points such as accessing/modifying methods and entering/returning methods. However, it does not offer any feature related to class extension.

Guaraná [OLI 99] defines a reflective architecture based on metaobjects supporting dynamic method switching. Even if it supports dynamic application changes, the definition of the class itself is static: dynamic evolution is done at the metalevel.

MetaclassTalk [BOU 00] goes further than Guaraná by using the ability of Smalltalk to dynamically create/modify/remove classes. However, the scope of modification is global.

By providing true delegation, Lava [KNI 00] supports dynamic unanticipated changes using class wrappers. Lava's approach is similar to classboxes but it is finer grained: Lava brings extensions to objects, whereas classboxes operate on classes. Contrary to classboxes, Lava cannot be used to define cross-cutting changes over several classes.

The Self prototype-based language [AGE 95] allows slots to be dynamically added: a prototype can be freely extended at run-time with new variables or new methods. However, these slot additions are global so they may override already existing slots which might affect any clients.

Dynamically Extension and Replacement of Classes. In MultiJava [CLI 00] an *open class* is a class that can have its behavior extended with new methods. However, redefining methods and adding new instance variables are not permitted.

IguanaJ [RED 02] supports dynamic adaptation of the behavior of arbitrary classes and objects in unanticipated ways. Internal behavior of classes can be changed but it does not support modifications to their external interfaces. Neither a preprocessor nor source code of the base application are required. However, it is not clear if a compiler is required or not.

Dynamic Classes [MAL 00] and HotSwap [DMI 01] allow Java programs to change class definitions during their execution. Instances are updated according to the new class definition following the tradition of Smalltalk and Lisp which have been using such a technique for decades to support interactive and incremental programming. However, Dynamic Classes and HotSwap do not provide higher level mechanisms for applying cross-cutting changes to several classes.

6. Conclusion

While aspect-oriented programming supports the definition and application of cross-cutting concerns, few solutions exist to support dynamic adaptation and dynamic application of static aspects. Classboxes allow a module to add or refine methods, and to add state to classes defined in other modules. Classboxes can dynamically be replaced by a new classbox or a set of classboxes.

A classbox offers a uniform and powerful mechanism to support a simple and structural kind of aspect. Classboxes do not offer the possibility to define advice annotation like before or after. However, classboxes unify the notion of modules and aspects. In addition classbox dynamicity supports dynamic adaptation by allowing aspects to be loaded or unloaded dynamically. As such, classboxes represent a important step in our quest for the *minimal* mechanisms that support aspect-oriented programming.

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). We also like to thank Noury Bouraqadi and Oscar Nierstrasz for their valuable comments and Andrew P. Black for his feedback on the paper.

7. References

[AGE 95] AGESEN O., BAK L., CHAMBERS C., CHAN B.-W., HÖLZLE U., MALONEY J., SMITH R. B., UNGAR D., WOLCZKO M., "The SELF 4.0 Programmer's Reference Manual", Sun Microsystems, 1995.

[Asp] "AspectR Home Page".

[BER 03] BERGEL A., DUCASSE S., WUYTS R., "Classboxes: A Minimal Module Model Supporting Local Rebinding", *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, vol. 2789 of *LNCS*, Springer-Verlag, 2003, p. 122–131, Best paper award.

[BER 05] BERGEL A., DUCASSE S., NIERSTRASZ O., WUYTS R., "Classboxes: Controlling Visibility of Class Extensions", *Computer Languages, Systems and Structures* , 2005, Elsevier, To appear.

[BOU 00] BOURAQADI N., "Concern Oriented Programming using Reflection", *Workshop on Advanced Separation of Concerns – OOSPLA 2000*, 2000.

[CLI 00] CLIFTON C., LEAVENS G. T., CHAMBERS C., MILLSTEIN T., "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java", *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, p. 130–145.

[DMI 01] DMITRIEV M., "Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications", *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, Oct. 2001.

[DOU 02] DOUENCE R., SÜDHOLT M., "A model and a tool for Event-based Aspect-Oriented Programming (EAOP)", Technical report, Dec. 2002, Ecole des Mines de Nantes.

[EID 02] EIDE E., REID A., FLATT M., LEPREAU J., "Aspect Weaving as Component Knitting: Separating Concerns with Knit", *Workshop on Advanced Separation of Concerns in Software Engineering*, 2002.

[FIN 98] FINDLER R. B., FLATT M., "Modular object-oriented programming with units and mixins", *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ACM Press, 1998, p. 94–104.

[GAL 01] GAL A., SCHRÖDER-PREIKSCHAT W., SPINCZYK O., "AspectC++: Language Proposal and Prototype Implementation", *Workshop on Advanced Separation of Concerns in Object-Oriented Systems – OOPSLA 2001*, Oct. 2001.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Mass., 1995.

[ING 97] INGALLS D., KAEHLER T., MALONEY J., WALLACE S., KAY A., "Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself", *Proceedings OOPSLA '97*, ACM Press, Nov. 1997, p. 318–326.

[KIC 96] KICZALES G., "Aspect-Oriented Programming: A Position Paper From the Xerox PARC Aspect-Oriented Programming Project", MUEHLHAUSER M., Ed., *Special Issues in Object-Oriented Programming*, 1996.

[KIC 01] KICZALES G., HILSDALE E., HUGUNIN J., KERSTEN M., PALM J., GRISWOLD W. G., "An overview of AspectJ", *Proceeding ECOOP 2001*, num. 2072 LNCS, Springer Verlag, 2001.

[KIM 02] KIM H., "AspectC#: An AOSD implementation for C#", report , 2002, Department of Computer Science, Trinity College, Dublin.

[KNI 00] KNIESEL G., "Darwin – Dynamic Object-Based Inheritance with Subtyping", PhD thesis, CS Dept. III, University of Bonn, Germany, 2000.

[LIE 99] LIEBERHERR K., LORENZ D. H., MEZINI M., "Programming with Aspectual Components", report num. NU-CCS-99-01, March 1999, College of Computer Science, Northeastern University, Boston, MA 02115.

[MAL 00] MALABARBA S., PANDEY R., GRAGG J., BARR E., BARNES J. F., "Runtime Support for Type-Safe Dynamic Java Classes", *Proceedings of the 14th European Conference on Object-Oriented Programming*, Springer-Verlag, 2000, p. 337–361.

[OLI 99] OLIVA A., BUZATO L. E., "The Design and Implementation of Guarana", *USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, 1999.

[OSS 00] OSSHER H., TARR P., "Hyper/J: multi-dimensional separation of concerns for Java", *Proceedings of the 22nd international conference on Software engineering*, ACM Press, 2000, p. 734–737.

[OVL 02] OVLINGER J., LIEBERHERR K., LORENZ D., "Aspects and Modules Combined", report num. NU-CCS-02-03, March 2002, College of Computer Science, Northeastern University, Boston, MA, http://www.ccs.neu.edu/research/demeter/papers/ac-aspectj-hyperj

[POP 02] POPOVICI A., GROSS T., ALONSO G., "Dynamic weaving for aspect-oriented programming", *Proceedings of the 1st international conference on Aspect-oriented software development*, ACM Press, 2002, p. 141–147.

[POP 03] POPOVICI A., ALONSO G., GROSS T., "Just-in-time aspects: efficient dynamic weaving for Java", *Proceedings of the 2nd international conference on Aspect-oriented software development*, ACM Press, 2003, p. 100–109.

[RED 02] REDMOND B., CAHILL V., "Supporting Unanticipated Dynamic Adaptation of Application Behaviour", *Proceedings of European Conference on Object-Oriented Programming*, vol. 2374, Springer-Verlag, 2002, p. 205-230.

[RIV 97] RIVARD F., "Évolution du comportement des objets dans les langages à classes réflexifs", PhD thesis, Ecole des Mines de Nantes, Université de Nantes, France, 1997.

[SAR 04] SARIDAKIS T., "Managing Unsolicited Events in Component-Based Software", *Workshop on Component Models for Dependable Systems*, Aug. 2004, To appear.

[SER 99] SERRANO M., "Wide classes", GUERRAOUI R., Ed., *Proceedings ECOOP '99*, vol. 1628 of *LNCS*, Lisbon, Portugal, June 1999, Springer-Verlag, p. 391-415.

[Squ] "Squeak Home Page", http://www.squeak.org/