



# Classboxes: Controlling Visibility of Class Extensions

Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, Roel Wuyts

► **To cite this version:**

Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. Computer Languages, Systems and Structures, Elsevier, 2005. <inria-00533458>

**HAL Id: inria-00533458**

**<https://hal.inria.fr/inria-00533458>**

Submitted on 14 Jan 2011

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Classboxes: Controlling Visibility of Class Extensions<sup>c</sup>

Alexandre Bergel<sup>a</sup> Stéphane Ducasse<sup>a</sup> Oscar Nierstrasz<sup>a</sup>  
Roel Wuyts<sup>b</sup>

<sup>a</sup>*Software Composition Group–IAM, Universität Bern, Switzerland*

<sup>b</sup>*Lab for Software Composition and Decomposition, Brussels, Belgium*

<sup>c</sup>*Preprint of Journal of Computer Languages, Systems and Structures (special issue for International Smalltalk Conference). Full reference available at <http://stephane.ducasse.free.fr/Publications.html>*

---

## Abstract

A *class extension* is a method that is defined in a module, but whose class is defined elsewhere. Class extensions offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. Unfortunately existing approaches suffer from various limitations. Either class extensions have a global impact, with possibly negative effects for unexpected clients, or they have a purely local impact, with negative results for collaborating clients. Furthermore, conflicting class extensions are either disallowed, or resolved by linearization, with consequent negative effects. To solve these problems we present *classboxes*, a module system for object-oriented languages that provides for method addition and replacement. Moreover, the changes made by a classbox are only visible to that classbox (or classboxes that import it), a feature we call *local rebinding*. To validate the model we have implemented it in the Squeak Smalltalk environment, and performed benchmarks.

---

\* 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).

*Email addresses:* [bergel@iam.unibe.ch](mailto:bergel@iam.unibe.ch) (Alexandre Bergel),  
[ducasse@iam.unibe.ch](mailto:ducasse@iam.unibe.ch) (Stéphane Ducasse), [nierstrasz@iam.unibe.ch](mailto:nierstrasz@iam.unibe.ch) (Oscar Nierstrasz), [wuyts@ulb.ac.be](mailto:wuyts@ulb.ac.be) (Roel Wuyts).

*Preprint of Journal of Computer Languages, Systems and Structures*

## 1 Introduction

It is well-established that object-oriented programming languages gain a great deal of their power and expressiveness from their support for the *open/closed principle* [1]: classes are *closed* in the sense that they can be instantiated, but they are also *open* to incremental modification by inheritance.

Nevertheless, classes and inheritance alone are not adequate for expressing many useful forms of incremental change. For example, most modern object-oriented languages introduce *modules* as a complementary mechanism to structure classes and control visibility of names. Reflection is another example of an increasingly mainstream technique used to modify and adapt behaviour at run-time. Aspect-oriented programming, on the other hand, is a technique to adapt sets of related classes by introducing code that addresses cross-cutting aspects.

In this paper we focus on a particular technique, known as *class extensions*, which addresses the need to extend existing classes with new behaviour. Smalltalk [2], CLOS [3], Objective-C [4], and more recently MultiJava [5] and AspectJ [6] are examples of languages that support class extensions. Class extensions offer a good solution to the dilemma that arises when one would like to modify or extend the behaviour of an existing class, and subclassing is inappropriate because that *specific* class is referred to, but, one cannot modify the source code of the class in question. A class extension can then be applied to that specific class.

Despite the demonstrated utility of class extensions, a number of open problems have limited their widespread acceptance. Briefly, these problems are:

- (1) *Globality*. In existing approaches, the effects of a class extension are either global (*i.e.*, visible to all clients), or purely local (*i.e.*, only to specific clients named in the application of the class extension). In the first case, clients that do not require the class extension may be adversely affected. In the second case, *collaborating clients* that are not explicitly named will not see the class extension, even though they should.
- (2) *Conflicts*. If two or more class extensions attempt to extend the same class, this may lead to a conflict. In existing approaches, conflicts are either forbidden, or extensions are linearized, possibly leading to unexpected behaviour. In either case, the utility of class extensions is severely impacted.

We propose a modular approach to class extensions that largely solves these two problems by defining an implicit context in which class extensions are visible. A *classbox* is a kind of module with three main characteristics:

- It is a *unit of scoping* in which classes, global variables and methods are defined. Each entity belongs to precisely one classbox, namely the one in which it is first *defined*, but an entity can be made visible to other classboxes by *importing* it. Methods can be defined for any class visible within a classbox, independently of whether that class is defined or imported. Methods defined (or redefined) for imported classes are called *class extensions*.
- A class extension is *locally visible* to the classbox in which it is defined. This means that the extension is only visible to (i) the extending classbox, and (ii) other classboxes that directly or indirectly import the extended class.
- A class extension supports *local rebinding*. This means that, although extensions are locally visible, their effect extends to all collaborating classes. A classbox thereby determines a namespace *within* which local class extensions behave *as though they were global*. From the perspective of a classbox, the world is flattened.

We have previously introduced classboxes by means of a specialized method lookup algorithm [7] reproduced in Section 5. The main contributions of this paper are: (i) A set-theoretic account of the semantics of classboxes that does not require a special method lookup algorithm and (ii) a detailed description of the prototype implementation in the Squeak Smalltalk system, including performance benchmarks.

The rest of the paper is structured as follows. Section 2 presents the problems in supporting unanticipated changes by giving a motivating example and Section 3 outlines the classbox model. Then a set-theoretic account of classboxes is given in Section 4. We discuss implementation issues arising in our prototype implementation in Section 5. Section 6 contrasts classboxes with related work. Finally, in Section 7 we conclude with some remarks concerning future work.

## 2 Motivation: Supporting Unanticipated Change

Class extensions provide a mechanism to support *unanticipated changes* in a static setting. Let us first consider a typical scenario, which will enable us to establish some key requirements for class extensions, while highlighting the main problems to be overcome.

A Link-Checker is an application whose purpose is to report a list of the dead links on a web-page at a given URL. One natural way to implement a Link-Checker, depicted in Figure 1, is to download the HTML page from the remote website, and parse it to get an abstract syntax tree of the page composed of various elements representing the HTML tags. Then using a recursive call over the hierarchy, get the list of the links referenced in the page. The liveness of

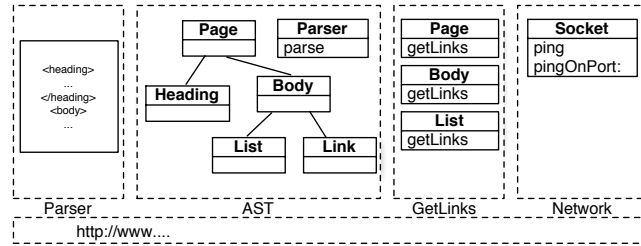


Fig. 1. The conceptual decomposition of the deadlink checker: an HTML parser, an abstract syntax tree for HTML documents, facility to get links from a page, and a network library.

each these links elements is checked by pinging the associated host and trying to obtain the status of the linked page. When a timeout is issued or if the HTTP reply corresponds to an error the link is declared dead.

Based on this example we can identify four properties that a packaging system for an object-oriented programming language should support: *class extensions* allowing *redefinition*, *locality* of changes, *propagation* of changes to collaborating clients, and resolution of *conflicts*.

**Class Extensions with Redefinition.** First, the different elements composing the solution should be packaged so that they can be used in further applications. We can identify the following modules: an HTML scanner and parser, an abstract syntax tree for the HTML elements, a recursive call over these elements to get links contained in a page and some network facilities. One key point is that we have to be able to group together the definitions of the `getLinks` methods in a module that is different from that of the AST. This means that the `GetLinks` module has to be able to extend the class definitions of the tree node elements.

Although languages such as CLOS, Smalltalk, MultiJava, and AspectJ offer some solutions, most other languages (including Java), do not allow a class to be extended by a different module or package than the one defining the class. Note that subclassing the tree node elements is not a general solution, since clients that explicitly name the original class will not see the subclass extension.

In our development environment, the default Squeak distribution, the `ping` method used by the environment does not raise an exception but opens a dialog box when a target host cannot be reached. We therefore not only need the ability to add methods (for packaging the `GetLinks` module), but also to *redefine* them (to patch existing methods). *We therefore require a module system that supports class extensions with redefinition.*

**Locality of Changes.** The second key aspect concerns the visibility of changes, *i.e.*, which modules see the extensions made by other modules. In most ap-

proaches that support them, class extensions have a global visibility. All clients have a common view of any given class, and any extensions are also seen by all clients. This may lead to unexpected behaviour for some clients.

In the case of the `ping` method, we only want our redefined version to be visible within the scope of our application. Other applications may actually rely on the *ad-hoc* behavior provided by `ping`. Therefore the extensions and changes to the system made by one module should not impact the system as a whole, but only the module introducing the changes and its client modules. *Class extensions should be confined to the module that introduces them.*

**Local Rebindings.** Even though class extensions should be visible only to the module that introduces them, the actual effect *from the perspective of that module* should be as if the extension were global.

The `pingOnPort:` method first adjusts the port (value kept in a variable) and then call the `ping` method. We want that any call to `ping` made by `pingOnPort:` triggers the definition brought by our LinkChecker application, even if `pingOnPort:` is defined in a scope that also contains a previous definition of `ping`. *Class extensions visible within a module should propagate to collaborating clients.*

**Conflicts.** Class extensions are useful when, for instance, a library needs to add a particular method to a class provided by the system. Conflicts arise when an application relies on two modules that extend the same method of the same class in different ways.

The `ping` method provided by Squeak is useful for pinging a remote host. Its default behavior is to display the result in a popup window. The Link-Checker application redefines this method to make it yield a value and to raise an exception if the host is not reachable. Conflicts can arise with other modules that make changes to this method. As a concrete example, Squeak has a `SocketICMP` module that implements the ICMP network protocol. Amongst other things, this implementation redefines the `ping` method with an ICMP-based implementation. Using both the Link-Checker and the `SocketICMP` module therefore leads to a conflict because both redefine the method `ping`.

There are several ways to handle this conflict: (1) the definition in Link-Checker overrides the definition in `SocketICMP`'s, (2) `SocketICMP`'s definition overrides Link-Checker's, (3) a conflict is detected at composition time and needs to be resolved, or (4) each extension is defined in a different namespace than that of the class.

With Smalltalk, CLOS and Objective-C the result depends on which module is loaded/initialized last which effectively impacts the system. On the other hand, Multijava and Hyper/J detect conflicting situations at compile time. *Selector namespaces*, Smallsript [8] and ModularSmalltalk [9] define the ex-

tension in a particular namespace: conflicts are avoided and both extensions are applied to the system within different scopes. *Resolution of conflicting class extensions should take the context of affected clients into account.*

### 3 Classboxes in a Nutshell

A *classbox* is a module containing *scoped definitions* and *import statements*. Classboxes define classes, methods and variables. Imported declarations may be *extended*, possibly redefining imported methods. When a classbox is instantiated, it yields a *namespace* in which the directly defined, imported and extended entities co-exist with the implicitly imported entities.

**Scoped Definitions.** A classbox defines *classes*, *methods*, or *variables*. Each class, method or variable *belongs* to precisely one classbox, namely the one in which it is originally defined. Classes and variables defined in a classbox are globally accessible to all methods in the scope of that classbox.

**Imports.** A classbox may import classes and variables from other classboxes. Imported entities thus become available within the scope of the importing classbox. An imported class may be *extended* with new methods, or methods that redefine existing methods. The extended class is then visible within the scope of the extending classbox, but not in the defining classbox of the extended class.

#### 3.1 Scope of Methods

A method defined on a class in a classbox  $CB$  is visible within that classbox, and within other classboxes that import this class from  $CB$ . In a given classbox all the methods defined along the chain of import are visible within this classbox.

If several classboxes extend a class with a method with the same name but with different implementations, the implementation chosen during an invocation is the one that is reachable according to the import chain.

A classbox  $CB$  that defines a method that already exists in the import chain *hides* its former definition from this classbox  $CB$  and other classboxes that may import the extended class from  $CB$ .

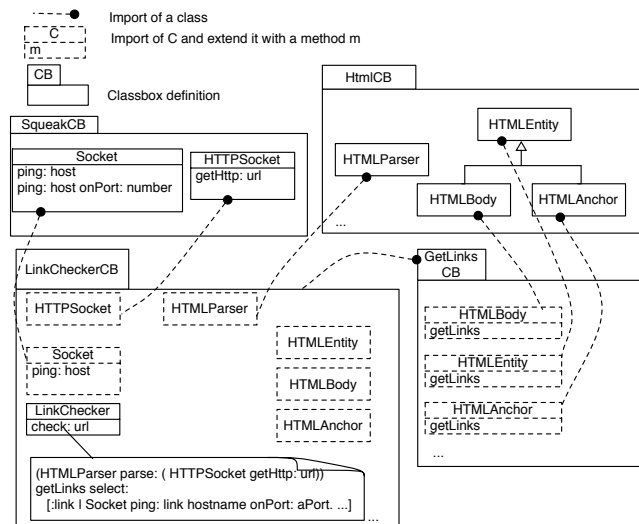


Fig. 2. The dead-link checker modularized with classboxes.

### 3.2 The Link-Checker with Classboxes

This section shows how to use classboxes to modularize the Link-Checker example. Because classboxes have been fully implemented in the Squeak [10] environment, code fragments are presented in Smalltalk.

The architecture of the Link-Checker application is depicted in Figure 2. The classbox `SqueakCB` contains the network facility for checking the existence of a remote host (class `Socket` with class method `ping: host`) and for fetching the content associated to a given URL (class `HTTPSocket` with class method `getHttp: url`).

The classbox `HtmlCB` defines the HTML framework facilities. The class `HTMLParser` is used to parse a text, yielding an abstract syntax tree (AST) composed of nodes such as `HTMLEntity` (the root of the structure), `HTMLBody`, `HTMLAnchor` (representing a link), ...

The classbox `GetLinksCB` implements the recursive algorithm intended to produce a collection of all the links contained over the AST elements. It imports the relevant nodes from the classbox `HtmlCB` and *extends* each of the classes representing HTML tag elements by defining the corresponding `getLinks` methods.

The classbox `LinkCheckerCB` contains the actual link checker application. It defines the class `LinkChecker`, containing one method (`check: url`) which is the entry point of the application. This method first gets the raw content of a page designated by `url` using the class `HTTPSocket`). It then parses the page using the class `HTMLParser`, obtaining an AST of the page. Then it invokes the



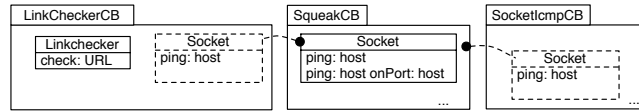


Fig. 3. The method `ping` is extended by two different classboxes. Conflict is avoided because extensions are confined to their respective classboxes.

method `getLinks` on the root of that AST, obtaining a collection of all the links on the page. Finally it checks the liveness of these links by pinging the hosts mentioned in each link. `LinkCheckerCB` imports the complete classbox `GetLinksCB`, so all the extended classes (HTML nodes) are visible within it. As a consequence, within the classbox `LinkCheckerCB` the AST generated by `HTML-Parser` (class imported from `HtmlCB`) understands the extensions brought by `GetLinksCB`. To solve the problem that the method `ping: host` in the classbox `SqueakCB` displays its results in a dialog box, the classbox `LinkCheckerCB` redefines it to raise an exception instead.

### 3.3 Discussion

**Locality of Changes.** Although the method `ping` of class `Socket` is redefined, its visibility is confined to the `LinkCheckerCB` classbox. Unrelated code in the system relying on the original definition of this method is not affected. This illustrates both *class extensions with redefinition* and *locality of changes*.

**Local Rebinding.** The classbox `SqueakCB` defines the class `Socket` with two methods: `ping: host onPort: number` and `ping: host`. The first one calls the second one, and the latter posts a popup menu to display the result of pinging a host. This implementation is not suitable for our application. The classbox `LinkCheckerCB` imports the class `Socket` from `SqueakCB` and extends it by redefining the method `ping: host` with an implementation that throws an exception when a host is not reachable. Calling `ping: host onPort: number` within `LinkCheckerCB` triggers the new implementation of `ping: host`. This illustrates the *local rebinding* property.

**Conflict.** The classbox `LinkCheckerCB` extends the class `Socket` by redefining the method `ping`. This extension is local to the classbox. Figure 3 shows another classbox `SocketIcmpCB` that also imports the class `Socket` and redefines the same method `ping`. This class extension is local to `SocketIcmpCB`. Conflict is avoided because each extension is confined to the classbox that defines it.

## 4 The Classbox Model

This section presents a set-theoretic model that precisely defines the semantics of classboxes. We abstract away from the operational details of statements and expressions of a given object-oriented language, and instead focus on the key features that interact with classboxes. We start by introducing a basic model of *classes*, *objects* and *namespaces*, where we capture *instantiation*, *message sending*, and *self-* and *super-calls*.

On top of this basic model, we then show how classboxes are defined as a mechanism for introducing class extensions, and for controlling the visibility of class extensions in different namespaces. We show how *locality of changes* and *local rebinding* arise as a consequence of the way that classboxes are composed.

### 4.1 Environments

We use the basic concept of an extensible *environment* as a mechanism for modeling classes, objects and classboxes.

**Definition 1** An *environment*  $\epsilon : D \rightarrow R^*$ , is a mapping from some domain  $D$  to an extended range  $R^* = R \cup \{\perp\}$ , such that the inverse image  $\epsilon^{-1}(R)$  is finite.

We represent environments as finite sets of bindings, for example:  $\epsilon_1 = \{a \mapsto x, b \mapsto y\}$  is an environment that maps  $a$  to  $x$  and  $b$  to  $y$ . All other values in the domain of this environment (for example,  $c$ ) are mapped to  $\perp$ .

We normally leave out unessential parentheses. Since an environment is a function, we simply invoke it to look up a binding. In this case,  $\epsilon_1 a = x$ ,  $\epsilon_1 b = y$  and  $\epsilon_1 c = \perp$ .

**Definition 2** An environment  $\epsilon : D \rightarrow R^*$  may *override* another environment  $\epsilon'$ . We define  $\epsilon \triangleright \epsilon' : D \rightarrow R^*$  as follows:

$$(\epsilon \triangleright \epsilon')x \stackrel{\text{def}}{=} \begin{cases} \epsilon'x & \text{if } \epsilon x = \perp \\ \epsilon x & \text{otherwise} \end{cases}$$

For example, if  $\epsilon_2 = \{b \mapsto z, c \mapsto w\}$ , then  $(\epsilon_1 \triangleright \epsilon_2)a = x$ ,  $(\epsilon_1 \triangleright \epsilon_2)b = y$ , and  $(\epsilon_1 \triangleright \epsilon_2)c = w$ . We employ overriding both for method dictionaries and class namespaces.

## 4.2 Classes, Namespaces and Objects

The primitive elements of our model are the following disjoint sets:  $\mathcal{C}$ , a countable set of *class names*,  $\mathcal{M}$ , a countable set of *messages*, and  $\mathcal{B}$ , a countable set of *method bodies*.

**Definition 3** A *method dictionary*,  $\delta \in \mathcal{D}$  is an environment,  $\delta : \mathcal{M} \rightarrow \mathcal{B}^*$  that maps a finite set of messages to bodies.

For example,  $\delta = \{m_1 \mapsto b_1, m_2 \mapsto b_2\}$  defines a dictionary  $d$  that maps message  $m_1$  to body  $b_1$  and  $m_2$  to  $b_2$ , and all other messages to  $\perp$ .

Note that, for the purpose of this paper, we are not concerned with the implementation details of the method bodies. We only consider which kinds of messages are sent in the bodies.

**Definition 4** A *class*,  $c\langle\delta, B, \epsilon\rangle$  consists of a method dictionary  $\delta$ , a superclass name  $B \in \mathcal{C} \cup \{\text{nil}\}$ , and an environment  $\epsilon$ , called a *class namespace*, that binds class names to classes.

`nil` represents an empty class, from which the root of a class hierarchy inherits. By convention, every class namespace is assumed to contain the binding `nil`  $\mapsto$   $c\langle\emptyset, \text{nil}, \emptyset\rangle$ , which we therefore do not list explicitly.

**Definition 5** An *object*  $o\langle c, \phi\rangle$  consists of a class  $c$  and an environment  $\phi$ , which is a class namespace (obtained from  $c$ ) extended with a binding for `self`.

Note that, for the present purposes, we do not model attributes (instance variables) of objects, aside from the pseudo-variables `self` and `super`.

We can send messages to classes and to objects. We use the notation  $x[m]$  to send the message  $m$  to the class or object  $x$ .

**Definition 6** We can *instantiate* an object by sending the message `new` to a class  $c = c\langle\delta, B, \epsilon\rangle$ :

$$c[\text{new}] \stackrel{\text{def}}{=} \mu\sigma.o\langle c, \{\text{self} \mapsto \sigma\} \triangleright \epsilon\rangle$$

At this point we recursively bind `self` to the value of the object itself.

As usual,  $\mu x.E$  binds free occurrences of  $x$  in  $E$  to the value of the recursive expression itself, *i.e.*,  $\mu x.E \stackrel{\text{def}}{=} E\{\mu x.E/x\}$ , where  $E\{y/x\}$  is the usual substitution operation, replacing free occurrences of  $x$  in  $E$  by  $y$  while avoiding name clashes.

Although we do not model the internal details of method bodies here, we must take care to be precise about the environment within which methods are evaluated. As we shall see when we define classboxes, it is precisely the way in which these environments are composed that determines the scope within which class extensions are visible.

**Definition 7** A *method closure*  $m\langle b, \phi \rangle$  consists of a method body  $b$  and a class namespace  $\phi$  that additionally binds both **self** and **super**.

Note that **super** is bound by methods, not objects, since **super**-calls are relative to the class in which a method is defined, not the class from which the object is instantiated.

**Definition 8** We can *send a message*  $m$  to an object  $o\langle c, \phi \rangle$ , where  $c = c\langle \delta, B, \epsilon \rangle$  obtaining a method closure:

$$o\langle c, \phi \rangle[m] \stackrel{\text{def}}{=} \begin{cases} m\langle \delta m, \{\mathbf{super} \mapsto o\langle \epsilon B, \phi \rangle\} \triangleright \phi \rangle & \text{if } \delta m \neq \perp \\ o\langle \epsilon B, \phi \rangle[m] & \text{else if } B \neq \text{nil} \\ \perp & \text{otherwise} \end{cases}$$

This definition captures the basic method lookup algorithm of object-oriented programming languages. If the message sent does not correspond to a method defined in the class of the object, the lookup continues in the parent class, and so on. If the method is not found, the message is reported as not being understood ( $\perp$ ). If a suitable method is found, it is evaluated in a context where **super** is bound to the current object, but from the perspective of the method's superclass. As we can clearly see, **super** is an object, not a class. Note that according to Definition 4 the superclass  $B$  can be nil.

**Definition 9** A closure may be evaluated, in which case it may send various messages. Here we are interested in **self**- and **super**-sends, and static class references.

$$\begin{aligned} m\langle b, \phi \rangle[\mathbf{self} \ m] &\stackrel{\text{def}}{=} (\phi \ \mathbf{self})[m] \\ m\langle b, \phi \rangle[\mathbf{super} \ m] &\stackrel{\text{def}}{=} (\phi \ \mathbf{super})[m] \\ m\langle b, \phi \rangle[\mathbf{C} \ \mathbf{new}] &\stackrel{\text{def}}{=} (\phi \ C_\phi)[\mathbf{new}] \end{aligned}$$

### 4.3 Classboxes

A classbox is an *open* entity that provides a number of classes, and which can be extended. When a classbox is *closed*, it yields an ordinary class namespace (Definition 4).

The key point in modeling classboxes is that multiple versions of the same class may be implicitly present within the same classbox. Suppose that we import the class `LinkChecker` from the classbox `LinkCheckerCB`, and we locally define a class `Socket`. Even though `LinkChecker` collaborates with `Socket`, ours is a *different* socket class that has nothing to do with the `Socket` class known to `LinkChecker`. To capture this aspect we must refine the notion of class names to express the *originating classbox* to which a class belongs:

- $\mathcal{C}$  is the countable set of *raw class names*,
- $\mathcal{X}$  is the set of classbox names,
- $\mathcal{C}^+ = \{C^n \mid C \in \mathcal{C}, n \in \mathcal{X}\}$  is the set of *decorated class names*.

The *decorated class name* simply encodes the classbox to which the class belongs, *i.e.*, where it was first defined. We call the superscript  $n$  of a decorated class name  $C^n$  its *origin*.

**Definition 10** A raw class name  $C$  *matches* a decorated class name  $B^n$  if  $C = B$ :

$$C \sim B^n \text{ iff } C = B$$

For example, when we use the raw class name `Socket`, it may not be clear *which* `Socket` class we are referring to. However the decorated class name `SocketSqueakCB` unambiguously identifies the `Socket` class first introduced in the `SqueakCB` classbox.

Note that it is this *same* class that is extended in `LinkCheckerCB`, since there is no `Socket` class defined there. There is no `SocketLinkCheckerCB`.

**Definition 11** A *classbox*  $\mathbf{b}\langle n, \alpha \rangle$  consists of an identifier  $n \in \mathcal{X}$  (*i.e.* classbox names) and a function  $\alpha$  from class namespaces to class namespaces.

The intuition here is that a classbox is *open* because it can always be extended with new class definitions, imports and extensions. As a consequence, we do not yet know the class namespace of the classes it provides. However we can *close* a classbox, thereby fixing the class namespace of all the provided classes.

**Definition 12** A *classbox*  $\mathbf{b}\langle n, \alpha \rangle$  can be *closed* by sending it the `close` message, generating a fixpoint:  $\mathbf{b}\langle n, \alpha \rangle[\text{close}] \stackrel{\text{def}}{=} \mu\epsilon.\alpha\epsilon$

The resulting class namespace must be closed, *i.e.*, all used class names must be defined. Since  $\alpha$  is a function from class namespaces to class namespaces,  $\mu\epsilon.\alpha\epsilon$  represents a fixpoint in which all the classes provided by the classbox are made visible to each other.

**Definition 13** We may *lookup* the decorated class name  $C^n$  corresponding to a raw class name  $C$  in a classbox  $\mathbf{b}\langle n, \alpha \rangle$ :

$$C_\alpha \stackrel{\text{def}}{=} \begin{cases} C^n & \text{if } \exists! n \in \mathcal{X}, (\mathbf{b}\langle n, \alpha \rangle[\text{close}])C^n \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

Suppose the `LinkCheckerCB` classbox is represented by  $\mathbf{b}\langle \text{LinkCheckerCB}, \alpha \rangle$ . Then `Socketα` yields `SocketSqueakCB`, since `SqueakCB` is the origin of `Socket` in the `LinkCheckerCB` classbox.

**Definition 14** An *empty classbox* with identifier  $n$  is:  $\text{empty}(n) \stackrel{\text{def}}{=} \mathbf{b}\langle n, \lambda\epsilon \rightarrow \emptyset \rangle$ . Note that  $\text{empty}(n)[\text{close}] = \emptyset$ , *i.e.*, closing an empty classbox yields an empty class namespace.

**Definition 15** We can *introduce* to a classbox  $\mathbf{b}\langle n, \alpha \rangle$  a new class  $C$  that subclasses  $B$  (defined in a classbox  $\mathbf{b}\langle m, \beta \rangle$ ) with  $\delta$  as method dictionary by sending it the message `def subclasses with`.

$$\begin{aligned} & \mathbf{b}\langle n, \alpha \rangle[\text{def } C \text{ subclasses } B^m \text{ with } \delta] \\ & \stackrel{\text{def}}{=} \begin{cases} \mathbf{b}\langle n, \lambda\epsilon. \{C^n \mapsto \mathbf{c}\langle \delta, B^m, \epsilon \rangle\} \triangleright \alpha\epsilon \rangle, & \text{if } C_\alpha = \perp \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

Note that the formal parameter  $\epsilon$  represents the fixpoint we obtain when the classbox is finally closed. We must therefore extend  $\alpha\epsilon$  with the new subclass definition, obtaining  $\{C^n \mapsto \dots\} \triangleright \alpha\epsilon$ . We retain  $\epsilon$  as a formal parameter so that the classbox remains open (*i.e.*,  $\lambda\epsilon. \dots$ ). The side condition states that it is an error to introduce a class that is already defined in the classbox. Within a classbox, only decorated class names occur. The newly introduced class has the origin  $n$ . We also explicitly identify the origin  $m$  of the superclass.

#### 4.4 Importing Classes

**Definition 16** A classbox  $\mathbf{b}\langle n, \alpha \rangle$  may *import* a raw named class from another, classbox  $\mathbf{b}\langle m, \beta \rangle$ , by sending it the message `import`.

$$\begin{aligned} & \mathbf{b}\langle n, \alpha \rangle[\text{import } C \text{ from } \mathbf{b}\langle m, \beta \rangle] \\ & \stackrel{\text{def}}{=} \begin{cases} \mathbf{b}\langle n, \lambda\epsilon. \{C_\beta \mapsto (\mu\phi. \beta(\epsilon \triangleright \phi))C_\beta\} \triangleright \alpha\epsilon \rangle, & \text{if } C_\alpha = \perp \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

Let us call the new classbox we obtain  $\mathbf{b}\langle n, \alpha' \rangle$ .  $\alpha'$  extends  $\alpha$  with the imported definition, but we must also take care that the environment of the imported class is properly extended with any pertinent definitions that occur in  $\alpha'$ . As before,  $\epsilon$  represents the class namespace that we obtain when we take the fixpoint of  $\alpha'$ . We therefore pass  $\epsilon$  to  $\alpha$  so it is available to all the existing class definitions in  $\alpha$ . We must also look up the correct decorated class name  $C_\beta$ . Finally, we must bind this to the correct definition from  $\beta$ , *extended* with any new definitions from  $\alpha'$ .

Suppose we would simply use  $C_\beta \mapsto (\mu\phi.\beta\phi)C_\beta$ , this would clearly be wrong, because the class we obtain would only see other class definitions from  $\beta$ , and not any definitions that may have already been extended in  $\alpha$ . Instead, we create an *intermediate namespace*  $\mu\phi.\beta(\epsilon \triangleright \phi)$ .  $\epsilon \triangleright \phi$  represents the environment of  $\beta$  *extended with any new definitions from  $\alpha'$* . We then pass this into  $\beta$  to make it available to *all* class definitions in  $\beta$ . Finally we extract this definition, bind it to  $C_\beta$  and use it to extend  $\alpha\epsilon$ .

Consider, for example, the import relationships in Figure 2. The classbox `LinkCheckerCB` imports `HTMLParser` from `HtmlCB` and `HTMLEntity` and its subclasses from `GetLinksCB`. If `HTMLParser` were naively imported from `HtmlCB`, it would not see the extensions imported from `GetLinksCB`. Instead, the import operation is defined so that when `HTMLParser` is imported, its environment (*i.e.*,  $\phi$ ) is extended by all definitions in `LinkCheckerCB` (*i.e.*,  $\epsilon \triangleright \phi$ ). So when `HTMLParser` is imported, it sees the extended versions of `HTMLEntity` and its subclasses. This is the *local rebinding* mechanism of classboxes.

Note that it is critical that `HTMLEntity` imported from `GetLinksCB` has the same origin as that expected by `HTMLParser`. If `LinkCheckerCB` or `GetLinksCB` were to define a *new* class `HTMLEntity`, then this would have a different decorated class name from the `HTMLEntity` originally defined in `HtmlCB`, and would therefore be invisible to `HTMLParser`.

#### 4.5 Extending Imported Classes

**Definition 17** A classbox  $\mathbf{b}\langle n, \alpha \rangle$  may *extend* a raw class named class from another classbox  $\mathbf{b}\langle m, \beta \rangle$ , by sending it the message `extend with`.

$$\mathbf{b}\langle n, \alpha \rangle[\text{extend } C \text{ with } \delta' \text{ from } \mathbf{b}\langle m, \beta \rangle]$$

$$\stackrel{\text{def}}{=} \begin{cases} \mathbf{b}\langle n, \lambda\epsilon. \{C_\beta \mapsto \delta' \triangleright (\mu\phi.\beta(\epsilon \triangleright \phi))C_\beta\} \triangleright \alpha\epsilon \rangle & \text{if } C_\alpha = \perp \\ \perp & \text{otherwise} \end{cases}$$

where

$$\delta' \triangleright c\langle\delta, B, \epsilon\rangle \stackrel{\text{def}}{=} c\langle\delta \triangleright \delta, B, \epsilon\rangle$$

Extend works just like import, except that the imported class definition is extended with  $\delta'$ .

As a consequence, importing a class is the same as extending it with a nil extension:

$$b\langle n, \alpha \rangle[\text{import } C \text{ from } b\langle m, \beta \rangle] \equiv b\langle n, \alpha \rangle[\text{extend } C \text{ with } \emptyset \text{ from } b\langle m, \beta \rangle]$$

As should be clear from the definition, class extensions are purely local to the classbox making the extension. This guarantees *locality of changes*. Extensions become visible to other classboxes only when they are explicitly imported, or implicitly made visible by the mechanism of local rebinding (as seen in the `HTMLParser` example discussed above).

*Method redefinition* is supported since the  $\delta'$  introduced by a class extension can redefine methods existing in the class being extended. For example, not only can the `GetLinksCB` classbox extend the `HTMLEntity` and related classes with a new `getLinks` method, but the `LinkCheckerCB` classbox can import `Socket` from the `SqueakCB` classbox and *redefine* the `ping` method.

#### 4.6 Proving Classbox Properties

**Proposition 1** A method defined in a classbox is visible within this classbox.

**Proof.** Because a method is defined either when a class is (i) defined or (ii) imported, this Proof is divided in two parts.

(i) Methods defined at the same time than the class they refer to are visible within the classbox where they are effectively defined. This first part of the proof consists in showing that defining a class  $C$  with a method  $m$  bound to a compiled method  $CM$  makes this method visible within the classbox (*i.e.* invoking  $m$  on an instance of  $C$  triggers the expected method  $CM$ ).

Without loss of generality, assume that  $C$  has no superclass (*i.e.* it inherits from `nil`).

$$\begin{aligned} & b\langle n, \alpha \rangle[\text{def } C \text{ subclasses nil with } \{m \mapsto CM\}] \\ &= b\langle n, \lambda\epsilon. \{C^m \mapsto c\langle\{m \mapsto CM\}, \text{nil}, \epsilon\rangle\} \triangleright \alpha\epsilon \rangle = b\langle n, \alpha' \rangle \end{aligned}$$



Closing this classbox yields:

$$\mathbf{b}\langle n, \alpha' \rangle[\text{close}] = \mu\epsilon.\alpha'\epsilon = \varphi = \{C^n \mapsto \mathbf{c}\langle \{\mathbf{m} \mapsto CM\}, \text{nil}, \varphi \rangle\} \triangleright \alpha\varphi$$

Now the instance of this class  $C^n$  is  $\text{obj} = \mathbf{o}\langle \varphi C^n, \{\text{self} \mapsto \text{obj}\} \triangleright \varphi \rangle$ . Sending a message  $\mathbf{m}$  to it yields:

$$\text{obj}[\mathbf{m}] = \mathbf{m}\langle \{\mathbf{m} \mapsto CM\} \mathbf{m}, \{\text{super} \mapsto \mathbf{c}\langle \emptyset, \text{nil}, \emptyset \rangle\} \triangleright \varphi \rangle$$

The implementation identified for the method  $\mathbf{m}$  is the result of  $\{\mathbf{m} \mapsto CM\} \mathbf{m} = CM$ .

(ii) Methods defined when importing a class are visible within the importing classbox.

$$\begin{aligned} & \mathbf{b}\langle n, \alpha \rangle[\text{extend } C \text{ with } \{\mathbf{m} \mapsto CM\} \text{ from } \mathbf{b}\langle m, \beta \rangle] \\ &= \mathbf{b}\langle n, \lambda\epsilon.\{C^n \mapsto \mathbf{c}\langle \{\mathbf{m} \mapsto CM\}, \text{nil}, \epsilon \rangle\} \triangleright (\mu\phi.\beta(\epsilon \triangleright \phi))C_\beta \rangle \triangleright \alpha\epsilon \end{aligned}$$

Assuming that  $C_\beta = C^p$  closing this classbox yields:

$$\mathbf{b}\langle n, \dots \rangle[\text{close}] = \varphi = \{C^p \mapsto \mathbf{c}\langle \{\mathbf{m} \mapsto CM\}, \text{nil}, \varphi \rangle\}$$

The rest of the proof follows what is already shown in (i).

**Proposition 2** Importing a class makes its methods previously defined visible in the importing classbox.

**Proof.** If  $\mathbf{b}\langle m, \beta \rangle C_\beta = \mathbf{c}\langle \{\mathbf{m} \mapsto CM, \}, B \rangle \epsilon$  then

$$\begin{aligned} & \mathbf{b}\langle n, \alpha \rangle[\text{import } C \text{ from } \mathbf{b}\langle m, \beta \rangle] \\ &= \mathbf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto (\mu\phi.\beta(\epsilon \triangleright \phi))C_\beta\} \triangleright \alpha\epsilon \rangle \end{aligned}$$

Assuming that  $C_\beta = C^p$  closing the resulting classbox yields:

$$\mathbf{b}\langle n, \dots \rangle[\text{close}] = \varphi = \{C^p \mapsto \beta C^p\} = \{C^p \mapsto \mathbf{c}\langle \{\mathbf{m} \mapsto CM\}, B, \varphi \rangle\} \triangleright \alpha\varphi$$

Then as already shown in the first proof, sending a message  $\mathbf{m}$  to an instance of  $\varphi C^p$  triggers the execution of  $CM$ .

**Proposition 3** Within a classbox, a method redefinition takes precedence over its former implementation.

**Proof.** Within a classbox  $b\langle m, \beta \rangle$  a class  $C$  has in its method dictionary an entry  $m$  bound to a first implementation  $CM1$ . This proof consists in showing that importing  $C$  in another classbox and redefining  $m$  bound to  $CM2$  hides the former implementation.

If  $b\langle m, \beta \rangle C_\beta = c\langle \{m \mapsto CM1\}, B, \epsilon \rangle$  then

$$\begin{aligned} & b\langle n, \alpha \rangle [\text{extend } C \text{ with } \{m \mapsto CM2\} \text{ from } b\langle m, \beta \rangle] \\ &= b\langle n, \lambda \epsilon. \{C_\beta \mapsto \{m \mapsto CM2\} \triangleright (\mu \phi. \beta(\epsilon \triangleright \phi)) C_\beta \} \triangleright \alpha \epsilon \rangle \end{aligned}$$

Assuming that  $C_\beta = C^p$  closing the resulting classbox yields:

$$\begin{aligned} b\langle n, \dots \rangle [\text{close}] &= \varphi = \{C^p \mapsto \{m \mapsto CM2\} \triangleright \beta C^p\} = \\ & \{C^p \mapsto \{m \mapsto CM2\} \triangleright b\langle m \mapsto CM1, B \rangle \varphi\} = \\ & \{C^p \mapsto c\langle \{m \mapsto CM2\}, B, \varphi \rangle\} \end{aligned}$$

The conclusion of this proof follows the end of the very first proof. Instantiating  $C^p$  and sending the message  $m$  executes the new implementation  $CM2$ .

#### 4.7 Resolving Diamond Conflicts

*Conflicts* are largely avoided. Classes that coincidentally have the same name but are introduced in different classboxes do not conflict because they have separate origins. Contradictions arising from attempts to import the same class from different classboxes of course cannot be resolved automatically. However, an important class of *indirect* conflicts is automatically resolved by the nature of the local rebinding mechanism.

Figure 4 illustrates a diamond pattern arising from two import chains with a common ancestor class. Classbox **CB1** defines a class **A** which provides a method **foo** returning the value 1. This class is imported by **CB2** where the method **foo** is redefined to return 2. **CB2** also defines a subclass of **A** named **B**. In a similar way, classbox **CB3** imports **A** from **CB1** and redefines **foo** to return 3. A subclass of **A** named **C** is also defined. A fourth classbox **CB4** imports **B** from **CB2** and **C** from **CB3**. **CB4** does not explicitly import class **A**.

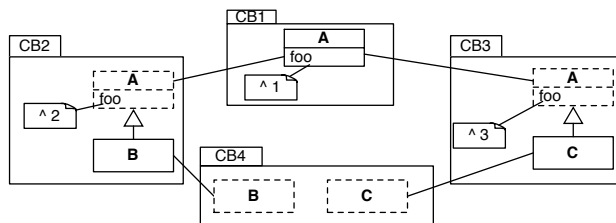


Fig. 4. Resolving Diamond conflicts

In the context of CB4 invoking `foo` on an instance of `B` yields the value 2, whereas invoking `foo` on an instance of `C` yields 3. However, if CB4 would explicitly import `A` from any one of CB1, CB2 or CB3, then that version of `A` would be visible to both `B` and `C`. For example, if CB4 would import `A` from CB1 and redefine `foo` to return 4, then both instances of `B` and `C` would return 4 when `foo` is invoked.

## 5 Implementation Issues

Classboxes can be implemented by changing the method lookup algorithm in the virtual machine. This requires a virtual machine that is available for changing, which is why we performed our experiments in Squeak, a Smalltalk environment of which the virtual machine is open source [10,11]. We adapted the method lookup and compiled a new virtual machine that is classbox-aware. In this section we evaluate the impact of this extended method lookup algorithm on performance.

### 5.1 Method Lookup Description

Encoding the classbox with the method signature makes it possible for different implementations a method to coexist. However, to take advantage of this, the method lookup mechanism has to be changed as well. Figure 5 describes the lookup algorithm we implemented that ensures the local rebinding property.

The algorithm first checks whether the class in the current classbox implements the selector we are looking for (lines 5 to 9). If it is found, the lookup is successful and we return the found method (line 9). If it is not found, we recurse. The algorithm favours imports over inheritance, meaning that first the import chain is traversed (in lines 12 to 18) before considering the inheritance chain (in lines 19 to 30). This last part is the difficult part of the algorithm, since we need to find the classbox where the superclass is defined

```

1 lookup: selector class: cls
2     startBox: startbox currentBox: currentbox classboxPath: path
3
4     | parentBox theSuper togoBox newPath |
5     self
6         lookup: selector
7         ofClass: cls
8         inClassbox: currentbox
9         ifPresentDo: [:method | ^ method].
10    parentBox := currentbox providerOf: cls name.
11    ^ parentBox
12    ifNotNil: [path addLast: parentBox.
13              self
14                  lookup: selector
15                  class: cls
16                  startBox: startbox
17                  currentBox: parentBox
18                  classboxPath: path]
19    ifNil: [theSuper := cls superclass.
20           theSuper ifNil: [^ cls method: selector notFoundIn: cls].
21           togoBox := path detect: [:box | box scopeContains: theSuper].
22           newPath := togoBox = startbox
23                   ifTrue: [OrderedCollection with: startbox]
24                   ifFalse: [path].
25           self
26               lookup: selector
27               class: theSuper
28               startBox: startbox
29               currentBox: togoBox
30               classboxPath: newPath]

```

Fig. 5. The lookup algorithm that provides the local rebinding.

that is closest to the classbox we started the lookup from. Therefore the algorithm remembers the path while traversing the import chain (line 12), and uses this when determining the classbox for the superclass (line 21).

## 5.2 Import Takes Precedence Over Inheritance

Figure 5, lines 11-12 shows that if a class is imported (`parentBox` is not nil) then the lookup pursues in the provider classbox. If this class is not imported (`parentBox` is nil), as shown at the line 19, then the lookup continues in the superclass.

The lookup in a superclass is done only if it is stated that a class does not provide any implementation for a given message. Within the classbox model this implies that we have to run over the chain of imports to make sure that a classbox does not extend this class with the corresponding method.

Figure 6 illustrates this property of the algorithm by depicting an example. It shows four classboxes: `GraphicCB`, `RoundedWindowCB`, `DoubleBufferCB` and `DoubleBufferAndRoundedCB`. Each of these defines extensions or simply imports classes to combine some of the extensions.

`GraphicCB` defines a hierarchy composed of three classes: `Component` provides

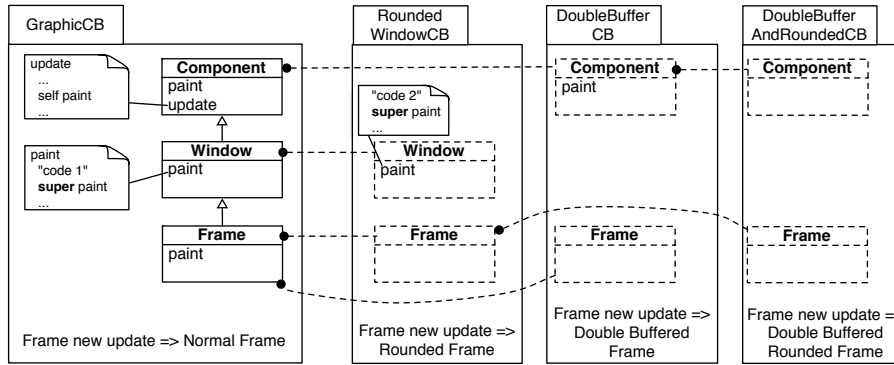


Fig. 6. Import takes precedence over inheritance

the methods `update` and `paint`, and `Window` and `Frame` both override the method `paint`. `Window` and `Frame` are imported in `RoundedWindowCB`. This first class is extended with a new implementation of `paint` to make corners of windows smooth by rounding them. `DoubleBufferCB` extends `Component`, which is imported from `GraphicCB`, and simply imports `Frame` from this same classbox. `Component` is extended with a redefinition of `paint` to take double buffering facility into account. Finally, `DoubleBufferAndRoundedCB` combines the two characteristics by importing `Component` from `DoubleBufferAndRoundedCB` and by importing `Frame` from `RoundedWindowCB`.

In `RoundedWindowCB` the new implementation of `paint` does a `super paint` which executes the `paint` method in `GraphicCB`. Evaluating `Frame new update` in `RoundedWindowCB` triggers the `update` method contained in `Component` and the local definition of `paint` is executed, the one provided by `RoundedWindowCB`.

`DoubleBufferAndRoundedCB` combines the double buffer and the rounded facilities by importing `Component` from `DoubleBufferCB` and `Frame` from `RoundedWindowCB`. Evaluating `Frame new update` in `DoubleBufferAndRoundedCB` triggers `update` defined in `GraphicCB` which send the message `paint`. The implementation taken is the one provided by `RoundedWindowCB` because `Frame` is imported from it. This implementation does a `super paint`, which execute the `paint` method defined in `DoubleBufferCB`.

### 5.3 Method Lookup Performance

Making the overhead related to our new method lookup as low as possible was one of our major concerns. Compared to the description given in [7], our implementation of the model is greatly enhanced: there is no need to modify the VM (due to the message passing control mechanism [12] offered by Squeak) and the cost of the new method lookup greatly reduced (thanks to a cache mechanism).

Classboxes allow you to have several versions of a method to coexist simultaneously. Depending on where this method is called from (*i.e.* from which classbox) the right method implementation is selected according to the method lookup algorithm described previously. When a classbox extends a class it can either be a method addition or a method redefinition. With our current implementation, calling a method that has been simply added by a classbox does not impose any overhead. However calling a method that has been redefined has an extra cost: the lookup algorithm previously presented is performed. However, this result is cached. Our cache mechanism is based on the following basic assumption: *a redefined method is often called by the same object within the same classbox*. The byte-code of an extended method is transformed to include 5 byte-codes that check if the caller for this method is the one that has been previously cached. For method addition there is no need to use a cache because there is only one version of the method present in the system.

The following table illustrates the cost of the lookup of a redefined method compared with traditional lookup.

| 6,000,000 calls   | Classbox lookup (ms) |
|-------------------|----------------------|
| Over 1 Classbox   | 5176                 |
| Over 2 Classboxes | 5126                 |
| Over 3 Classboxes | 5145                 |
| Normal method     | 1477                 |

The experiment consists in calling 6 millions times a method that is redefined. It shows that there is a constant overhead that does not depend on the graph of import. This overhead is due to the extra few byte-codes added at the beginning of the method. The method used for the benchmarks is composed of one byte-code (simply return a numerical value). The same method that checks if the cache is valid is about 2.5 times  $((5176 - 1477) / 1477)$  slower.

## 6 Related Work

**Selector Namespaces.** Languages like ModularSmalltalk [9], Subsystems [13] and Smallsript [8] provide a scoping mechanism called *Selector Namespaces*, in which methods are inserted. As a result, class extension conflicts are avoided, and several applications can bring the same class extension referring to the same class and method without interfering with each other. As a result, class extensions are not globally visible, but confined to a bounded scope. However selector namespaces do not support the *local rebinding* property, since a new definition does not take precedence when original code is called.

**Multijava.** Multijava [5] is an extension of Java that supports *open classes*

and *multiple method dispatch*. An *open class* is a class whose methods are extensible. New methods can be added to an open class. These new methods are visible within the package that provides them and in the packages importing that package. Method redefinitions are not allowed: an open class cannot have one of its existing methods redefined. On the other hand, two class extensions can define a method on the same class with the same signature. In that case the extensions are scoped separately.

**Unit.** MZScheme [14] offers an advanced module system where a *unit* is the basic building block. A unit is a packaging entity composed of requirements, definitions and exports. Units have to be instantiated and composed with each other to form a program. The key point of this model is that connections between modules or classes are specified separately from their definitions. This principle allows a module to be instantiated at link time. Reusability and extensibility are expressed by recombining units. An application, made of units, can be recomposed and by aliasing new units can be inserted. Units differ from classboxes since a unit acts as a black box: a class within a unit cannot be extended. Instead a new unit has to be provided and included in a recomposition.

**Hyper/J.** Hyper/J [15] is based on the notion of *hyperspaces*, and promotes compositions 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 on classes that are not necessarily defined in that hyperslice. Such methods define a class extension, and classes intended to be extended are known at integration time. However this kind of extension does not allow redefinition and consequently does not help in supporting unanticipated evolution.

**Virtual Classes.** The specification of a *virtual class* [16] [17] is completely analogous to the specification of a virtual procedure. By introducing a dynamic lookup of a class name in a hierarchy of encapsulating entities (module for Keris [18], collaboration interfaces for Caesar [19] [20], classes for gbeta [21], or teams for Objectteams [22]) it is possible to refine a class within a sub-entity. One limitation with virtual classes is that the “virtuality” is scoped to a hierarchy: outside this hierarchy a class is not virtual anymore. For instance let us assume C to be a virtual class attribute in a hierarchy H1. In an unrelated hierarchy H2, class C is not virtual anymore and cannot be redefined.

**Object-Based Inheritance.** By providing *true delegation*, Lava [23] supports dynamic unanticipated changes using class wrappers. By introducing a new language construct, an object *a* (instance of *A*) can delegate all non-understood messages it receives to a *delegatee* object *b* (instance of *B*). Lava provides a *true-delegation* mechanism whereas the self reference used in the

class  $B$  refers to the delegating object  $a$ . Methods defined in  $b$  that are unknown to  $a$  are the extensions brought on  $a$ . So redefined or new methods are attached to a particular object rather than a class. True delegation provides a way for adding or redefining methods for a particular object whereas classboxes extend classes.

## 7 Conclusion and Future Work

*Classboxes* address the problem that classical module systems do not offer the ability to add or replace a method in a class that is not defined in that module. Classboxes offer a minimal module system for object-oriented languages in which extensions (method addition and replacement) to imported classes are *locally visible*. Essentially, a classbox defines a scope within which certain entities, *i.e.*, classes, methods and variables, are defined. A classbox may *import* entities from other classboxes, and optionally extend them *without impacting the originating* classbox. Concretely, classes may be imported, and methods may be added or redefined, without affecting clients of that class in other classboxes. Local rebinding strictly limits the impact of changes to clients of the extending classbox, leading to better control over changes, while giving the illusion from a local perspective that changes are global.

To see the impact of classboxes on a real-world example we remodularized an existing application (the seaside web server application [24] built upon a web server [25]) with classboxes. The goal is to show the usefulness of class extensions by measuring the proportion of class extension among the defined methods (for more details see the technical report [26]).

We have implemented a proof-of-concept prototype of classboxes in Squeak. In our implementation, the method lookup mechanism in the Squeak virtual machine has been modified to take classboxes into account. This prototype exhibits an overall 10% slowdown in performance for real-world applications.

In the future we will analyze some very large applications developed without any local rebinding facilities in order to identify places where programmers simulated local rebinding.

Currently classboxes function purely as a packaging and scoping mechanism. We intend to investigate various extensions of classboxes. We expect that an integration with traits will be fruitful, as this will enable packaging of collaborating traits [27] (and their associated tests). Presently classboxes lack any notion of a *component model*. We expect that explicit interfaces and composition mechanisms for classboxes will increase their usefulness. In particular, we intend to investigate the application of encapsulation policies [28] to class-



boxes.

**Acknowledgment.** We also like to thank Curtis Clifton, Erik Ernst, Günter Kniesel and Peri Tarr for their valuable comments and discussions.

## References

- [1] B. Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.
- [2] A. Goldberg, D. Robson, *Smalltalk-80: The Language*, Addison Wesley, 1989, book scglib.
- [3] A. Paepcke, User-level language crafting, in: *Object-Oriented Programming : the CLOS perspective*, MIT Press, 1993, pp. 66–99.
- [4] L. J. Pinson, R. S. Wiener, *Objective-C*, Addison Wesley, 1988.
- [5] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, in: *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 130–145.
- [6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: *Proceeding ECOOP 2001*, 2001.
- [7] A. Bergel, S. Ducasse, R. Wuyts, Classboxes: A minimal module model supporting local rebinding, in: *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, Vol. 2789 of LNCS, Springer-Verlag, 2003, pp. 122–131, best paper award.
- [8] D. Simmons, *Smallscript*, <http://www.smallscript.com> (2002).
- [9] A. Wirfs-Brock, B. Wilkerson, An overview of modular Smalltalk, in: *Proceedings OOPSLA '88*, 1988, pp. 123–134.
- [10] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: *Proceedings OOPSLA '97*, ACM Press, 1997, pp. 318–326.
- [11] Squeak home page, <http://www.squeak.org/>.
- [12] S. Ducasse, Evaluating message passing control techniques in Smalltalk, *Journal of Object-Oriented Programming (JOOP)* 12 (6) (1999) 39–44.
- [13] A. Wirfs-Brock, Subsystems — proposal, *OOPSLA 1996 Extending Smalltalk Workshop* (Oct. 1996).
- [14] M. Flatt, M. Felleisen, Units: Cool modules for hot languages, in: *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, ACM Press, 1998, pp. 236–248.

- [15] H. Ossher, P. Tarr, Hyper/J: multi-dimensional separation of concerns for java, in: Proceedings of the 22nd international conference on Software engineering, ACM Press, 2000, pp. 734–737.
- [16] E. Ernst, Family polymorphism, in: J. L. Knudsen (Ed.), ECOOP 2001, LNCS, Springer Verlag, 2001, pp. 303–326.
- [17] O. L. Madsen, B. Moller-Pedersen, Virtual classes: A powerful mechanism in object-oriented programming, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 397–406.
- [18] M. Zenger, Evolving software with extensible modules, in: International Workshop on Unanticipated Software Evolution, Malaga, Spain, 2002.
- [19] M. Mezini, K. Ostermann, Conquering aspects with caesar, in: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press, 2003, pp. 90–99.
- [20] M. Mezini, K. Ostermann, Modules for crosscutting models, in: 8th International Conference on Reliable Software Technologies (Ada-Europe '03), svlns, 2003.
- [21] E. Ernst, gbeta – a language with virtual attributes, block structure, and propagating, dynamic inheritance, Ph.D. thesis, Department of Computer Science, University of Aarhus, Århus, Denmark (1999).
- [22] S. Herrmann, Object confinement in Object Teams – reconciling encapsulation and flexible integration, in: 3rd German Workshop on Aspect-Oriented Software Development, SIG Object-Oriented Software Development, German Informatics Society, 2003.
- [23] G. Kniesel, Darwin – dynamic object-based inheritance with subtyping, PhD thesis, CS Dept. III, University of Bonn, Germany (2000).
- [24] Seaside: Squeak enterprise aubergines server, <http://www.beta4.com/seaside2/>.
- [25] Comanche: a full featured web serving environment for Smalltalk, <http://squeaklab.org/comanche>.
- [26] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Classboxes: Controlling visibility of class extensions, Technical Report IAM-04-003, Institut für Informatik, Universität Bern, Switzerland (Jun. 2004).
- [27] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming), Vol. 2743 of LNCS, Springer Verlag, 2003, pp. 248–274.
- [28] N. Schärli, S. Ducasse, O. Nierstrasz, R. Wuyts, Composable encapsulation policies, in: Proceedings ECOOP 2004 (European Conference on Object-Oriented Programming), LNCS 3086, Springer Verlag, 2004, pp. 248–274.