Classboxes: A Minimal Module Model Supporting Local Rebinding

Alexandre Bergel, Stéphane Ducasse, Roel Wuyts

Classboxes: A Minimal Module Model Supporting Local Rebinding

Published in JMLC'03
Best Award Paper

Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts

Software Composition Group, University of Bern,
Bern, Switzerland
{bergel, ducasse, wuyts}@iam.unibe.ch

Abstract. Classical module systems support well the modular development of applications but do not offer the ability to add or replace a method in a class that is not defined in that module. On the other hand, languages that support method addition and replacement do not provide a modular view of applications, and their changes have a global impact. The result is a gap between module systems for object-oriented languages on one hand, and the very desirable feature of method addition and replacement on the other hand. To solve these problems we present *classboxes*, a module system for object-oriented languages that provides 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 experiments modularising code.

Keywords: Language Design, Method Lookup, Modules, Smalltalk, Class Extension, Selector Namespace

1 Modules in the Presence of Extensibility

The term *module* is overloaded. We follow the definitions of Modular Smalltalk [16] and Szyperski [12].

> *Modules are program units that manage the visibility and accessibility of names. A module defines a set of constant bindings between names and objects [16]. A module is a capsule containing (definitions of) items. The module draws a strong boundary between items defined inside it and items defined outside other modules [12].*

A *class extension* is a method that is defined in another source packaging entity (for example, a Java package or an Envy application [9]) than the class it is defined for. There exist two kinds of class extension: a *method addition adds* a new method, while a *method replacement replaces* an existing method.

Classical module systems, like those of Modula-2[17], Modula-3 [1], Oberon-2 [8], Ada [13], or MzScheme's [4] do not support class extensions. Numerous object-oriented programming languages, such as Java, C++, and Eiffel [7] lack this facility. However, it is widely used in those languages that support it, such as Smalltalk[16] and GBeta [3]. In "Capsules and Types in Fresco" A. Wills reports that in the goody library[1] goodies-lib@cs.man.ac.uk 73% of the files modify existing classes, and 44% define no new classes at all [14]. Even if these figures should be tempered due to the fact that goodies are not industrial applications, these numbers reflect that class extensions are not an anecdotical mechanism. There is some ongoing research that explores the introduction of class extensions to Java (for example OpenClasses [2], Keris [18] or MixJuice [5]), which is another indication that this is quite an important concept.

Languages supporting class extensions such as Smalltalk or Flavors do not offer the notion of modules. In these languages the changes are globally visible and impact the whole system. Even in module systems that support class extensions (Modular Smalltalk [16]), changes are visible to everyone after they have been applied.

To summarise, module systems exist for languages that do not support class extensions on the one hand, and languages exist that support class extensions but not modules on the other hand. The Classbox model provides modules that fully support class extensions, and these extensions are only visible to the classbox that defined them. Outside the classbox the system runs unchanged. This is accomplished by redefining the method lookup mechanism to take classboxes into account, so that the desired method is executed.

For validation we implemented this system in Squeak, an open-source Smalltalk environment, and implemented some small applications. Section 3 describes one of these examples, an application to check dead links on a web page. Classboxes are used to extend an existing system with a visitor and to replace existing system code.

The rest of the paper is structured as follows. Section 2 presents an overview of the Classbox model. In Section 3 we illustrate the model by showing the implementation of an application to check for dead links on web pages. Section 6 concludes the paper.

2 Overview of the Classbox Model

This section describes the semantics of the Classbox model. The next section illustrates the semantics and usage on a concrete case-study highlighting its unique features.

Classbox contents. A classbox consists of *imports* and *definitions*:

- An import is either a *class import* (stating explicitly from which classbox the class is imported, called the *parentbox*) or a *classbox import* (*i.e.,* that imports every class from the imported classbox).

[1] A goody is a small application provided without warranty or support.

– A definition can be a class definition or a method definition. A method definition declares the class that a method belongs to, the name of the method, and the implementation of the method.

Static completeness. Methods can only be defined on classes that are known within the classbox (*e.g.,* defined or imported). Furthermore, the implementation of a method can only refer to classes known in the classbox.

Extension. Extending a class C with one method m has the following semantics: if the class C has a method with the same signature as m, m replaces that method, otherwise m is added to C.

Flattened class. A flattened class describes what methods a class in a certain classbox contains, taking imports into account:

– The flattened definition of a class C *defined* in a classbox cb1 consists of C and all the method definitions for C in cb1.
– The flattened definition of a class C *imported* in a classbox cb1 is the *flattened* definition of C in its parentbox extended by the method definitions for C in cb1.

Note that this implies that for the method lookup, importing takes precedence over inheritance (first the import chain is used, and then the inheritance chain). This is explained in Section 4.

Flattened classbox. A flattened classbox consists of the flattened definitions of all the classes (defined or imported) of that classbox.

Class name uniqueness. When defining or importing a class C in a classbox cb1, the name of C has to be uniquely defined in flattened C. This guarantees that class import cycles are not possible.

Method addition. Method m is a method *addition* for class C if m is a method defined for C, and the *flattened* definition of C in its parentbox does not define a method with the same signature as m.

Method replacement. Method m is a method *replacement* for class C if m is a method defined for C, and the *flattened* definition of C in its parentbox contains a method with the same signature as m. Following the definition of flattening, the method replacement takes precedence in the flattened version of C.

These rules ensure the following property, which we call *local rebinding*. Suppose a classbox cb1 defines a class C with two methods, m calling n, and classbox cb2 imports C from cb1 and replaces n. We say that cb2 locally rebinds n in cb1 to represent the fact that calling m in the context of cb2 invokes the method n as defined in cb2 while calling m in the context of cb1 invokes the method n as it is defined in cb1. The runtime semantics are illustrated in the next section and explained in Section 4.

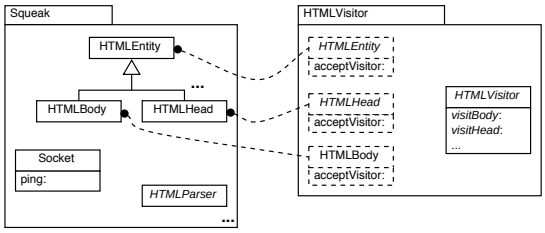

Fig. 1. The Squeak Classbox and the HTMLVisitor Classbox that extends Squeak with an HTML Visitor.

3 The Running Example

To illustrate the key properties of the Classbox model we develop an application that allows one to check dead links on a web page. We use *Squeak*[6], an open source Smalltalk, to implement the Classbox model. The user specifies the web page to be checked and the application returns the list of URLs that cannot be reached within a given time-out.

Out of the box, the Squeak environment comes with a sophisticated development environment and a rich class library. All this code is contained in a single "image" within a single global namespace and consists of about 1800 classes. Squeak contains a HTML parser, a hierarchy of HTML nodes that are built by the HTMLParser and several network protocols.

To write our application we use the existing HTML parser to create a HTML tree of the web page for which we want to check the dead links. Then we walk this tree, checking whether each link can be reached. While these checks can be hard-coded as methods in the HTML parse tree itself, it is a common practice to write a visitor for the HTML parse tree which can be reused by other applications. Then we only need to write a specialised visitor that checks for dead links. Therefore, the implementation of our application consists of the definitions of two classboxes: one extending Squeak with a visitor for the HTML parse tree and one customising that generic visitor with one that checks for dead links. The resulting system is shown in Figure 2, and is explained in the next sections.

3.1 Class Import and Class Extensions

As shown in Figure 1, extending the HTML parse tree with a visitor consists in adding a new HTMLVisitor class, and adding one method to each existing HTML parse tree node to call the visitor. Since the visitor methods and the visitor class itself logically belong together, we group them in the same classbox called HTMLVisitor. Figure 1 shows a part of the Squeak classbox (that contains the whole unmodularised library of around 1800 classes of the Squeak environment) and the HTML visitor classbox. This classbox imports every HTML parse tree node class (only three are shown in the picture) and extends each of these classes

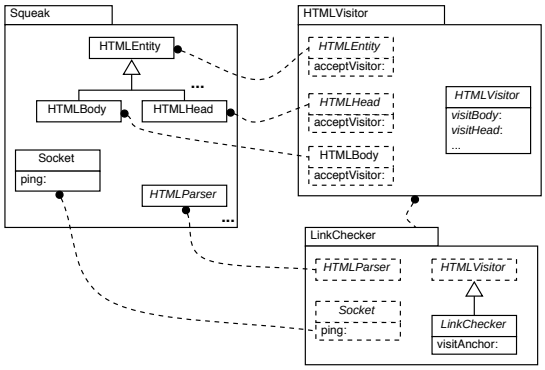

Fig. 2. The LinkChecker classbox, that defines a HTMLVisitor subclass that checks for dead links and that replaces the method ping: in the class Socket to throw exceptions instead of opening dialogue boxes.

with a single method to visit them (called acceptVisitor:). It also contains the HTMLVisitor class, that implements the abstract visitor class.

Illustrated Model Properties: Method Additions. The example shows that classboxes can be used not only to define whole classes (like the HTMLVisitor class); they can also define methods on classes that are imported (all the acceptVisitor: methods), *i.e.*, classboxes support method additions [16] [2].

Without this feature it is very difficult to factor out the visitor in a separate classbox from the tree it operates on. In languages that do not support method additions, the solution would be to create visitable subclasses for every HTML node and add the visiting methods there. However this has two undesirable effects: first of all, the existing code that used the classes has to be changed to use the new subclasses, and second, other applications that need to make similar extensions would have to independently add the same subclasses. Class extensions do not exhibit either problem.

Illustrated Model Properties: Local Rebinding. The addition of the method acceptVisitor: to the HTML tree classes is only visible for code executed in the context of the HTMLVisitor classbox. Code running in the Squeak classbox cannot see this method addition. The next sections elaborate on this point.

3.2 Classbox Import and Method Replacement

With the HTMLVisitor classbox defined it becomes easy to implement the Link Checker application. It basically boils down to adding a subclass of HTMLVisitor that overrides the visitAnchor: method to implement the checking of dead links. Checking a link means opening a connection to the specified URL within a certain amount of time.

In practice it turns out that the class that actually builds the connection (the class Socket) does not throw exceptions when links are not reachable. Instead it directly opens a dialogue box explaining the error that was encountered!

This makes it suddenly quite hard to implement the Link Checker. The solution would be to change the method that opens a dialogue box and let it throw exceptions instead. However, this also means that all the applications that use this method and rely on dialogue boxes to be opened have to be changed as well. While this may be a worthwhile endeavour that would result in a cleaner Squeak system, it is too much work when just writing a Link Checker. The solution is to change this method in such a way that it will throw exceptions only in the places where it is needed. All other places should still use the unchanged method. This is what is done by the LinkChecker classbox. How this works is explained in detail in Section 4.

Figure 2 shows the classbox LinkChecker. It imports the classbox HTMLVisitor, *i.e.*, all classes defined in HTMLVisitor are imported and it imports class Socket from the Squeak classbox. The classbox LinkChecker defines a class LinkChecker, a subclass of HTMLVisitor, and a method visitAnchor: that implements the actual checking of the links contained in a HTML document. It also defines a method ping: on Socket that replaces the existing implementation that opens dialogue boxes with an implementation that throws exceptions.

Illustrated Model Properties: Local rebinding. This example shows that a classbox allows one to replace methods for existing classes. Moreover, these changes are again *local* to LinkChecker: it is only in the LinkChecker classbox that exceptions are raised when network locations are not reachable. The rest of the system is unaffected by this change, and still gets dialogue boxes when time-outs occur.

3.3 Local rebinding and Flattening

This section elaborates how local rebinding in the presence of the flattening property allows a classbox to change the behaviour of methods in the system in such a way that these changes are *local* to the classbox.

To illustrate this we execute some expressions in the example described in previous sections. We execute an expression that creates an HTML parse tree for a certain url in two different contexts: first in the Squeak classbox then in the LinkChecker classbox.

```
HtmlParser parse: ('http://www.iam.unibe.ch/~scg/' asUrl contents)
```

In the Squeak classbox, the result of this expression is a parse tree consisting of instances of HTMLEntity that cannot be visited. This is exactly the intended behaviour, as the expression is performed in the context of the Squeak classbox, and that classbox does not know anything about visitors for its parse tree.

In the LinkChecker classbox the same expression the result is a HTML parse tree that can be visited. Again, this is exactly what is intended since we imported the HTML parse tree nodes from HTMLVisitor (indicating that we want those classes to be used).

```
lookup: selector class: cls
            startBox: startbox currentBox: currentbox classboxPath: path

    | parentBox theSuper togoBox newPath |
    self
        lookup: selector
        ofClass: cls
        inClassbox: currentbox
        ifPresentDo: [:method | ^ method].
    parentBox := currentbox providerOf: cls name.
    ^ parentBox
        ifNotNil: [path addLast: parentBox.
                   self
                      lookup: selector
                      class: cls
                      startBox: startbox
                      currentBox: parentBox
                      classboxPath: path]
        ifNil: [theSuper := cls superclass.
                theSuper ifNil: [^ cls method: selector notFoundIn: cls].
                togoBox := path  detect: [:box | box scopeContains: theSuper].
                newPath := togoBox = startbox
                               ifTrue: [OrderedCollection with: startbox]
                               ifFalse: [path].
                self
                    lookup: selector
                    class: theSuper
                    startBox: startbox
                    currentBox: togoBox
                    classboxPath: newPath]
```

Fig. 3. The lookup algorithm that provides local rebinding.

4 Runtime Semantics of the Model

Depending on the classbox an expression is executed in, objects can understand different messages or have methods with different behaviour. For this to work, a classbox-aware lookup mechanism for methods and a change in the structure of method dictionaries are needed. We focus on Smalltalk method dictionaries here, but the same approach holds for other object-oriented languages.

Normally, method dictionaries are used to lookup a key consisting of the signature of the method (in Smalltalk this is only the name, as there are no static types), and return a value corresponding to the method body. To support classboxes we encode the classbox where the method is defined in the method signature (*i.e.*, the key). For example the method dictionary for HTMLEntity has entries prefixed with "#Squeak." for the methods defined in Squeak, and entries with "#HTMLVisitor." for the method additions defined in that classbox. The method dictionary for class Socket now has two entries for the ping: method: one for the Squeak classbox and one for the LinkChecker classbox. Class LinkChecker has only a single entry for the visitAnchor method.

Encoding the classbox with the method signature makes it possible to let different implementations for a method live alongside each other. However, to take advantage of this, the method lookup mechanism has to be changed as well.

Benchmark	Regular lookup	Classbox lookup	Overhead
direct call	5439	6824	25%
looked up call	5453	6940	27%
opening and closing a web browser	332	548	65%
opening and closing a mailreader	536	760	41%
call through 3 classboxes	-	10554	-
call through 6 classboxes	-	10654	-

Table 1. Benchmarks results from Squeak comparing the regular method lookup mechanism with the classbox-aware lookup mechanism (units are in milliseconds).

Figure 3 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 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).

4.1 Runtime Performance

As can be expected, introducing the classbox aware method lookup mechanism introduces some runtime overhead, especially since our current implementation is currently not optimised. Table 1 shows the results for some benchmarks that we performed to compare the regular method lookup performance vs. the classbox-aware lookup performance:

1. sending a message defined in the class of the instance (10 millions times), and sending a message defined in a super class hierarchy (10 millions times).
2. measuring launching and closing of two applications implemented in Squeak (a web browser and an e-mail client) within the same classbox (average over 10 times).
3. performing a method call through a chain formed by classboxes extending a class.

The table shows that the penalty for the new lookup scheme by itself is roughly 25 percent, where the real-world applications run about 60 percent slower. We think that this difference is due to the fact that we did not adapt the method cache in the virtual machine.

Note that our current implementation is straightforward and does not incorporate any optimisations yet. For example we are thinking of changing the structure of the method cache in order to take classboxes into account.

5 Related Work

No existing mainstream language supports class extensions, modules, and local rebinding. Classical module systems, like those of Modula-2[17], Modula-3 [1], Oberon-2 [8], Ada [13] or Java, do not support class extensions.

Keris introduces extensible modules which are composed hierarchically and linked implicitly. Keris does not support class extension [18]. MzScheme's units are modules system with external connection facilities [4], and act as components that can be instantiated and linked together. They do not support class extensions. Classboxes on the other hand, are source code management abstractions.

OpenClasses [2] supports a modular definition of class extensions but only supports method addition and not method replacement. MixJuice [5] offers modules based on a form of inheritance which combines module members and class extensions but not local rebinding.

Modular Smalltalk only supports method additions that are globally visible [16]. In the Subsystems proposal [15], modules (subsystems) support selector namespaces, as in SmallScript [10]. A selector namespace contains method definitions (of possibly different classes). Selector namespaces are nested and this structure is used for the method lookup.A local selector takes precedence over the same selector defined in a surrounding namespace. With selector namespaces, class extensions can be defined as layers where methods defined in a nested namespace may redefine methods defined in their enclosing namespaces. Selector namespaces, however, do not support local rebinding.

Us, a subject-oriented programming extension of Self [11], provides for object extensions and method invocations in the context of *perspectives*, but Us does not provide modules.

6 Conclusion

This paper introduces the Classbox Model, a module model for object-oriented systems that supports local rebinding. Hence it provides method additions and replacements that are only visible in the module that defines them. Classboxes enhance both existing object-oriented languages that support method addition and replacement, and those that provide module systems. For the former it localises method additions and replacements. It extends the latter with a mechanism that supports unanticipated evolution. To apply local rebinding to an object-oriented language efficiently, the method lookup mechanism has to be changed, and a slightly different method dictionary has to be introduced.

We have implemented the model in the Squeak Smalltalk environment, and performed experiments using classboxes. In the paper we describe an example of how classboxes allow one to extend an existing parse tree with a visitor (making use of class extensions), and replacing a badly implemented method in a system class without affecting the whole system (using method replacement). As far as we know, no other module system is able to achieve this separation.

Acknowledgments

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

References

1. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, Aug. 1992.
2. C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 130–145, 2000.
3. E. Ernst. Propagating class and method combination. In *Proceedings ECOOP '99*, volume 1628 of *LNCS*, 67–91, June 1999. Springer-Verlag.
4. R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, 94–104. ACM Press, 1998.
5. Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, June 2002. Springer Verlag.
6. D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, 318–326. ACM Press, Nov. 1997.
7. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
8. H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.
9. J. Pelrine and A. Knight. *Mastering ENVY/Developer*. Cambridge University Press, 2001.
10. D. Simmons. Smallscript, 2002. http://www.smallscript.com.
11. R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.
12. C. A. Szyperski. Import is not inheritance — why we need both: Modules and classes. In *Proceedings ECOOP '92*, volume 615 of *LNCS*, 19–32, June 1992. Springer-Verlag.
13. S. T. Taft. Ada 9x: From abstraction-oriented to object-oriented. In *Proceedings OOPSLA '93*, volume 28, 127–143, Oct. 1993.
14. A. Wills. Capsules and types in fresco. In *Proceedings ECOOP '91*, volume 512 of *LNCS*, 59–76, July 1991. Springer-Verlag.
15. A. Wirfs-Brock. Subsystems — proposal. OOPSLA 1996 Extending Smalltalk Workshop, Oct. 1996.
16. A. Wirfs-Brock and B. Wilkerson. An overview of modular Smalltalk. In *Proceedings OOPSLA '88*, 123–134, Nov. 1988.
17. N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, 1983.
18. M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, June 2002.