The Classbox Module System

Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts

Software Composition Group, University of Bern,
Bern, Switzerland
{bergel, ducasse, wuyts}@iam.unibe.ch
A preliminary version of this paper was accepted at JMLC'03

Abstract. Classical modules systems support well the modular development of applications but lack the ability to add or replace a method in a class that is not defined in that module. But 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 allows 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 modularizing 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 definition 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. Even a lot of object-oriented programming languages, like Java, C++, or Eiffel [7] lack this facility. However, it is widely used in languages that support it, like 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 anedoctal

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

mechanism. Recent research is trying to introduce class extensions in Java (for example OpenClasses [2], Keris [18] or MixJuice [5]), again an 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]), the changes are always applied globally, for everyone to see after applying them.
To summarize, 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 webpage. 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 concretize the model by showing the implementation of an application to check for dead links on webpages. Section 5 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 exacerbating 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).
- 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 than 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 3.4.

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 unique in flattened C. This guarantees that class import cycles are not possible.

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

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

All these rules imply the following property that we name *local rebinding* . Imagine that a classbox cb1 defines a class C with two methods (m calling n) and that a classbox cb2 imports C from cb1 and replaces n. We say that cb2 is locally rebinding n into 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.

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 webpage. We use *Squeak*[6], an open source Smalltalk we used to implemented the Classbox model. The user specifies the webpage to be checked and the application returns the list of URLs that cannot be reached within a given timeout.
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' in one 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 webpage for which we want to check the dead links. Then this tree has to be walked, executing a check at each link to test whether it can be reached. While these checks can be hardcoded 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 a specialised visitor is written 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 of adding a new HTMLVisitor class, and adding one method to each existing

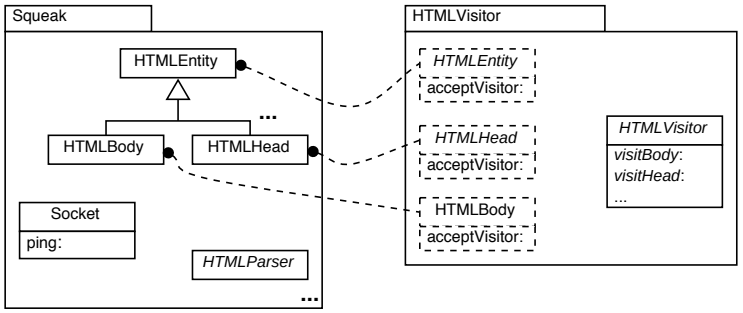

Fig. 1. The Squeak Classbox and the HTMLVisitor Classbox that extends Squeak with an HTML Visitor.

HTML parse tree node to call the visitor. Since the visitor methods and the visitor class itself logically belong together, we group them in one classbox called HTMLVisitor. Figure 1 shows a part of the Squeak classbox (that contains the whole unmodularized 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 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 visit method there. However this has two undesirable effects: first of all, there has to be a mechanism that uses the new subclasses and second, in the light of multiple extensions that all need to add subclasses to implement their desired behaviour, existing applications will not be aware of the new subclasses. Class extensions do not exhibit either problem.

Illustrated Model Properties: Local Rebinding. The addition of the method acceptVisitor: to the HTML tree classes are only visible for code executed in the context of the HTMLVisitor classbox. Code running in the Squeak classbox cannot see these method additions. 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.

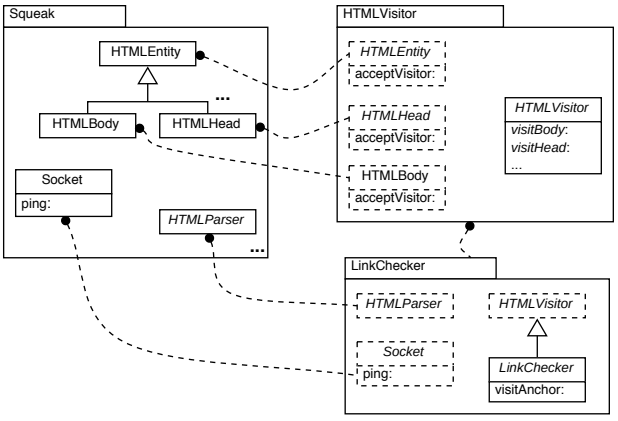

Fig. 2. The LinkChecker classbox, that defines a HTMLVisitor subclass that checks for dead links and that replaces the method ping: in the class Socket to throw exceptions instead of opening dialog boxes.

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 dialog 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 dialogs and let it throw exceptions instead. However, this also means that all the applications that use this method and rely on dialog boxes to be opened have to be changed as well. While this may be a worthwhile endeavor that would result in a cleaner Squeak system, it is too much work when just writing a Link Checker. The solution is to be able to change this method to throw exceptions in such a way that this changed method is only visible 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 Sections 3.4.
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, 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 dialog boxes with an implementation that throws exceptions.

Illustrated Model Properties: Local rebinding. This example shows that a classbox allows one to replace methods on 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 dialog boxes when timouts occur. Section 3.4 presents how this is accomplished.

3.3 Local rebinding and Flattening

This section elaborates on 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.

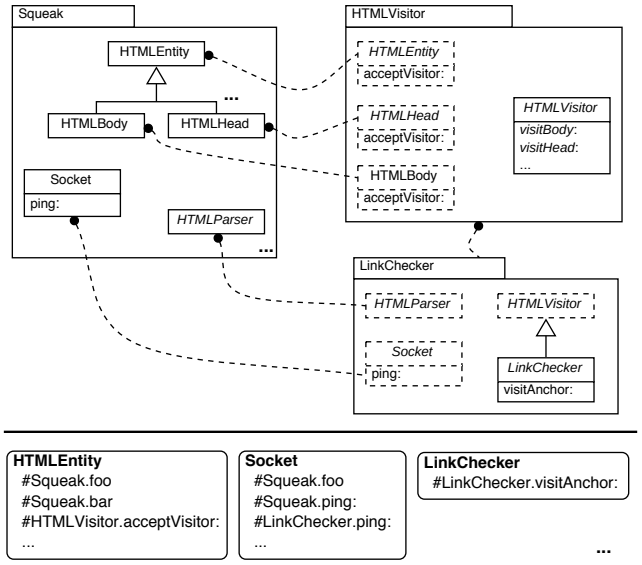

Fig. 3. The example showing the method dictionaries for some of the classes below the divider line. The method dictionaries for the other classes are not given.

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
     retrieveContents contentStream)
```

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).

3.4 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 holds for other object-oriented languages. Figure 3 illustrates the explanation. It shows the example that was used before,

```
1  (define (lookup classbox class selector explicitClass classAhead startbox)
2    (if (class-under-scope? classbox class)
3        (if (class-has-method? classbox class selector)
4            (retrieve-method-from-classbox classbox class selector)
5            (let ((parentBox (getParentClassbox classbox class)))
6              (if (not parentBox)
7                  (if (null? (superclassOf class))
8                      (throw-exception)
9                      (lookup startbox (superclassOf class) selector
10                             explicitClass explicitClass startbox))
11                 (if (eq? classbox startbox)
12                     (lookup parentBox class selector class class startbox)
13                     (lookup parentBox class selector explicitClass
14                             explicitClass startbox)))))
15     (let* ((r (findNextClassboxAndClass classAhead classbox))
16              (classAhead (car r))
17             (parentBox (cdr r)))
18            (lookup parentBox class selector explicitClass
19                    classAhead startBox))))
20
21 (define (findNextClassboxAndClass class classbox)
22   (if (class-defined? classbox class)
23       (findNextClassboxAndClass (superclassOf class) classbox)
24       (cons class (getParentClassbox classbox class))))
```

Fig. 4. The lookup algorithm that allows for local rebinding.

but also shows the method dictionaries for two of the classes involved in the example.

Normally, method dictionaries are dictionaries in which the key consists of the signature of the method (in Smalltalk this is only the name, as there are no static types), and the value is the method body. For classboxes we slightly change the method dictionaries by encoding the classbox where the method is defined in the signature. For example, and as shown in the Figure 3, 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 has to be changed as well. Figure 4 describes a lookup mechanism we implemented that allows the local rebinding property as explained before. The algorithm favors imports over inheritance (meaning that first the import chain is traversed before looking in the inheritance chain). The complexity of the algorithm stems from the identification of the correct classbox for the superclass.

When looking for a method, the lookup mechanism tries to locate the method in the flattened class. Line 2 checks whether the class is defined in the classbox. If it is, then line 3 checks whether it implements the method. If it does (line 4), then we have found the method and can stop. If it does not, then we look for it in the *import chain* for that class. Line 5 looks where the class was imported from, and keeps this information in the variable parentBox. If this parentBox happens to be the classbox we started from (line 11), then we do the lookup, remembering the class we are currently investigating (line 12). If not, we do

the same lookup in the new classbox (lines 13 and 14). When there is no parentbox to be found for the class, and we still have not found the method we are looking for, we have to go find it in the superclass (line 6). If there is no superclass, (line 7), the method we are looking for is not understood, and we throw an exception (line 8). If there is a superclass, we start the lookup in the superclass (lines 9 and 10). This recursively does the lookup for that new class as explained before.

Previous paragraph describes whap happens when the class is defined in the classbox. But when looking for a superclass, it is often the case that the superclass is not known in the classbox we started from. For example, when we imported HTMLParser, this class inherits methods from Scanner. For looking up an inherited method we have to look in Scanner, but Scanner is not known in LinkChecker. To know in what classbox we have to look for Scanner, we follow the import chain of the *closest subclass* defined in the *closest classbox* (lines 15 to 19 and the auxiliary function).

With the extended method dictionaries and lookup mechanism, the behaviour of the examples is obtained. Depending on the classbox the original expression is executed in, different method implementations are chosen in the method dictionary. Where the instance is created has no effect. This means that this scheme guarantees that any object that is created in some classbox and gets passed to another classbox will respond to the messages from that classbox.

4 Related Work

None of the existing languages supports at the same time class extensions, modules, and local rebinding. Classical module systems, like those of Modula-2[17], Modula-3 [1], Oberon-2 [8], or Ada [13], do not support class extensions. Java's import packages statement acts as name shortcut. Java does not support class extensions.

Keris introduces extensible modules which are composed hierarchically and wired implicitly. Keris does not support class extension [18]. MzScheme's units are modules system with external connections facilities [4], they do not support class extensions and act as component abstraction while classboxes are source code management abstractions.

OpenClasses [2] supports a modular definition of class extensions but they support only method additions 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 methods addition which are globally visible [16]. In the Subsystems proposal [15], modules (subsystems) support selector namespaces, as in SmallScript [10]. Selector namespaces structure the lookup of method selector in a tree structure similar to the ones of variable in Pascal-like language. 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 nested namespace may redefine method defined in their surrounding namespaces. However selector namespaces do not support local rebinding. Us, a subject-oriented programming version of Self [11], allows object extensions and method invocations in the context of *perspectives*, but Us does not provide modules.

5 Conclusion

This paper introduces the Classbox Model, a module model for object-oriented systems that supports local rebinding. Hence it allows method additions and replacements that are only visible in the module that defined them. Classboxes enhance both existing object-oriented languages that have method additions and replacements, and module systems. For the former it localizes 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 on using classboxes. In the paper we describe an example of how classboxes allow one to extend an existing parsetree 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.

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. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the PLDI '98 Conference on Programming Language Design and Implementation*, 236–248, 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, 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 15–19 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*, jun 2002.