A Sophisticated Programming Environment to Cope with Scoped Changes

Niklaus Haldimann
Supervised by: Alexandre Bergel
University of Berne, Switzerland
Software Composition Group

December 2005

Abstract

A class extension is a technique to evolve software in ways not foreseen when it was created; it's a method defined in a module whose class is defined elsewhere. The classbox model addresses the inherent problems of class extensions: It limits their impact to a well-defined scope. We present the classbox browser, a tool that assists programmers in working with classboxes in the Squeak Smalltalk environment. The browser enables the convenient modification of a class without affecting any of its existing clients.

1 Introduction

Software often evolves in unexpected ways. It is impossible to predict how exactly a software module will be used and extended by its clients. Object-oriented programming languages offer mechanisms that give the users of a software module a certain degree of flexibility. For example, an existing class can be subclassed to modify or extend its behaviour. But this is not always flexible enough. When an existing class is already being referred to in other modules, there is often a need to modify that specific class - creating a new class by subclassing wouldn't help here. Modifying the class might mean adding a new method to it. This we call a *class extension*, a method that is defined in a module but whose class is defined elsewhere.

Class extensions are a useful and accepted technique in Smalltalk environments. But they give rise to a number of problems: Their global impact affects all clients, and there can be conflicts if several class extensions are applied to the same class. There is a need to *control the scope* of class extensions. *Classboxes* propose a solution to this. Briefly, a classbox is a kind of module that limits the effect of class extension to its scope. We will discuss classboxes in more detail in the next section.

While the theory and implementation of classboxes in the Squeak environment is sound [BDN05], there was a lack of tools that integrate classboxes into everyday development. In this paper we introduce our implementation of a *classbox browser*. A browser in Smalltalk environments is a graphical tool to explore and manipulate objects in the system. As we will show, our classbox browser creates an intuitive programming environment to effectively work with classboxes and thereby easily cope with certain problems of software evolution.

The rest of this paper is structured as follows: Section 2 introduces the concept of classboxes, section 3 presents the design and implementation of the classbox browser and section 4 demonstrates the browser in action. Section 5 summarizes our results in a conclusion.

2 Classboxes

2.1 Issues with Class Extensions

Classboxes address the need to extend existing classes with new behaviour, focusing on a particular technique known as *class extensions.* Smalltalk [GR89], CLOS [Pae93], Objective-C [PW88], and more recently MultiJava [CLCM00] and AspectJ [KHH$^+$01] are examples of languages that support class extensions.

A *class extension* is a set of class members (*e.g.*, variables and methods) defined in a modular unit that is distinct from the one that defines the class to which these class members are related. A classical example of a class extension present in most of Smalltalk libraries is a method that converts a string into a url. One natural and concise way of defining such a method is on the class String. This is illustrated in figure 1. The package Text defines the class String which contains methods like substrings and indexOf:. The package Network defines network-related classes and extends the class String with a method asUrl. As a result, a url can be obtained from a string by directly sending a message asUrl to it, as in the Smalltalk expression 'http://www.iam.unibe.ch' asUrl.

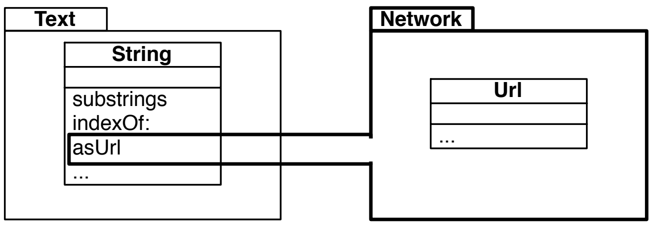


Figure 1: The package Network extends the class String defined in the package Text with a new method asUrl

Class extensions offer a good solution to the dilemma that arises when a developer would like to modify or extend the behaviour of an existing class. In such a case, subclassing is often inappropriate because that *specific* class is referred to by existing clients (and the source code of the class in question cannot be modified). But 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:

1. *Globality.* Most existing approaches apply a class extension either globally (visible to all clients) or purely local (only to specific clients named

explicitly in the class extension). In the first case, there may be unwanted effects on clients that do not require the class extension. In the second case, collaborating clients that are not explicitly named will not see the class extension, even though they should.

2. *Conflicts.* Conflicts may occur when two class extensions are applied to the same class. To resolve them, conflicts might be forbidden or ignored by letting class extensions override each other. In either case, the utility of class extensions is severely impacted.

2.2 Controlling Visibility of Changes with Classboxes

The classbox model proposes a novel approach to class extensions that solves the problems discussed in the previous section. A *classbox* is a kind of module with three main characteristics:

- It is a *unit of scoping* in which classes 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.

As shown in previous work [BDN05], classboxes are a scalable mechanism to support software modification. They allow for changes to arbitrary classes (in the form of class extensions), but also control the scope of these changes in a useful way.

3 The Classbox Browser

3.1 The Need for a New Browser

A complete implementation of the classbox model for the Squeak environment exists [BDNW05]. But manipulating classboxes and applying class extensions was only possible by doing it by hand, *e.g.*, a small bit of code had to be evaluated to create a classbox. Other bits of code were necessary to define or import classes in the new classbox. There was no classbox-aware, intuitive programming environment, the utility of classboxes for the casual Squeak programmer was therefore limited.

Traditionally, *browsers* let programmers explore and manipulate a Smalltalk environment. A browser is an application that presents a view on objects in the system and provides a programmer with tools to explore and modify them. Squeak, for example, comes with several standard browsers, *e.g.*, for looking at a class hierarchy or for an inheritance based view on a single method.

To make classboxes within Squeak more accessible, a *classbox browser* is called for: a browser which displays classboxes and provides to a programmer all tools necessary to perform classbox-specific actions on the system, *e.g.*, creating classboxes and class extensions.

An obvious way to create a new browser is to extend Squeak's standard browsers. However, they were not designed for this. Basing a new browser with new concepts on them is a tedious task and results in an inflexible design.

For these reasons we decided to base our work on the OmniBrowser framework. OmniBrowser is a "ground up rewrite of the classical system browsers provided in Squeak Smalltalk" [Put]. It's explictly designed to be flexible and extensible, so as to encourage experimentation with new types of development tools, language extensions, and runtime environments. As we will see, OmniBrowser indeed enables us to design a new browser quite effortlessly.

The next subsection describes the OmniBrowser framework and then we go into the details of the classbox browser design and implementation.

3.2 OmniBrowser Concepts

In this section we show how a new browser can be implemented using the OmniBrowser framework. As an example, we will build a complete *file browser*, detailing all the steps and showing all needed code. In the end we will be able to browse directories and files from the Squeak environment. Figure 3 shows the completed file browser.

3.2.1 Nodes and Metanodes

OmniBrowser wants you to look at the domain a browser navigates as a *graph of objects*. Every object of the domain is represented by a *node* in this graph.

In our file browser, the domain is the filesystem with files and directories as domain objects. The object graph thus consists of file and directory nodes. Each directory node has edges connecting it to the files contained in it. Figure 2 (a) illustrates the case of a very simple file system with a root directory containing a file `pic1.jpg` and a subdirectory `/temp` containing two further files `pic2.jpg` and `pic3.jpg`.

In code, we need to model directory and file nodes as the subclasses DirectoryNode and FileNode of OBNode. Directories and files are both identified by their path. In our trivial implementation DirectoryNode is at this point a subclass of FileNode without any further methods. However, we will later add navigation methods to DirectoryNode. OmniBrowser requires us to implement the #name method for all subclasses of OBNode. The string returned from this method will be used in the browser when the reciever is displayed.

```
OBNode subclass: #FileNode
    instanceVariableNames: 'path' ...
```

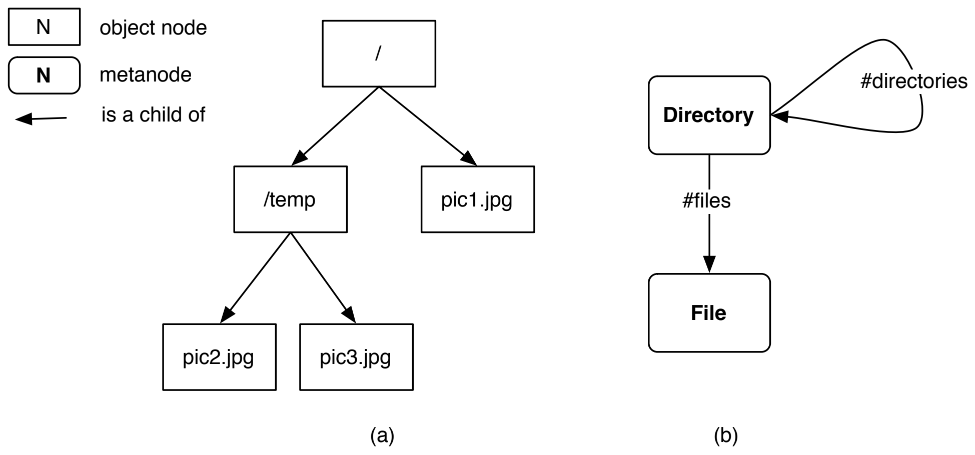


Figure 2: A graph describing a filesystem (a) and its corresponding metagraph (b).

```
FileNode>>setPath: newPath
    path := newPath

FileNode>>path
    ^ path

FileNode class>>on: path
    ^ self new setPath: path

FileNode>>name
    ^ (FileDirectory directoryEntryFor: self path) name.

FileNode subclass: #DirectoryNode
    instanceVariableNames: '' ...
```

The abstract relationships between different types of nodes within the browser are defined using another type of graph, a *metagraph.* In a metagraph each type of node of an object graph is represented by a *metanode.* As a whole, the metagraph describes the structure of the object graph navigated by the browser. Edges between metanodes define which kinds of nodes may be connected to which other kinds of nodes. "Kind of node" is a notion that is formalized by object node classes; typically, there is a metanode for each node class, as is the case in our file browser.

The metagraph for the file browser is shown in Figure 2 (b). There are two metanodes for the two types of nodes in the object graph. A directory may contain other directories and files, the two edges in the metagraph represent these two relationships.

OmniBrowser opts to store metagraphs in a central place. New metagraphs should be returned from class methods of the class OBMetagraph. Thus we construct the file browser metagraph in the method #fileBrowser:

```
OBMetagraph class>>fileBrowser
    | directory |
    directory := OBMetaNode named: 'Directory'.
    directory
        childAt: #directories put: directory;
        childAt: #files put: (OBMetaNode named: 'File').
    ^ directory
```

3.2.2 Browsing Graph and Metagraph

The file browser metagraph in Figure 2 (b) has its edges annotated with messages which also occur in the metagraph code above. These are the messages the framework will use to navigate the graph and retrieve children of an object node. This is the implementation of the two edges #directories and #files leading away from the directory metanode:

```
DirectoryNode>>files
    | dir |
    dir := FileDirectory on: self path.
    ^ dir fileNames collect: [:each |
        FileNode on: (dir fullNameFor: each)]

DirectoryNode>>directories
    | dir |
    dir := FileDirectory on: self path.
    ^ dir directoryNames collect: [:each |
        DirectoryNode on: (dir fullNameFor: each)]
```

The details of how files and directories are handled in the Squeak library do not interest us here. The crucial point is that these navigation methods return *collections of object nodes*, the children of some concrete directory node.

The code shown up to this point already implements a minimal, functional browser. We can now open a browser on the #fileBrowser metagraph and an arbitrary DirectoryNode instance by executing this code in the Squeak environment:

```
b := OBBrowser
        graph: #fileBrowser
        root: (DirectoryNode on: 'Macintosh HD:tmp:').
b open
```

Figure 3 shows the minimal file browser in action. A *browser* within the OmniBrowser framework uses the traditional column-based view, e.g., there are one or more columns which list the domain objects, hierarchically from left to right. A text panel below the columns may display additional properties of the currently selected object and provide means to manipulate these properties.

As shown, we only had to give OmniBrowser a metagraph and a root node to start with, that's enough for it to infer the intended behaviour of the file browser. We will now look a bit more into how it does that. The browser takes a node of the metagraph and a node of the domain graph as input, the root metanode and the root node, respectively. It looks at the edges leading away

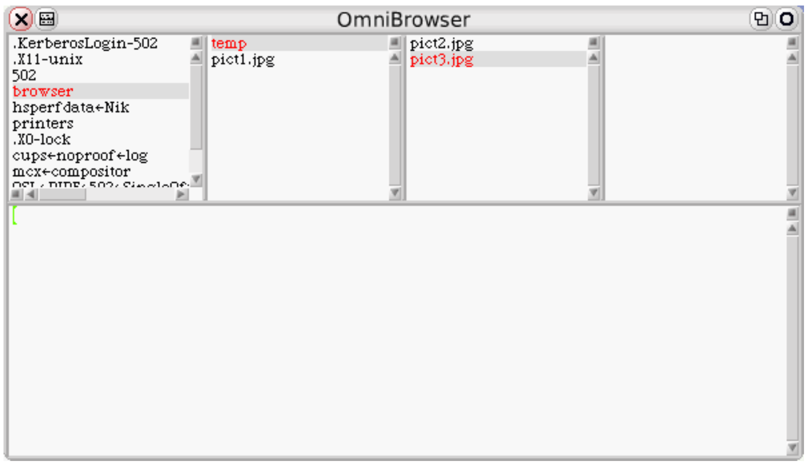


Figure 3: A minimal file browser based on OmniBrowser.

from the root metanode and sends the messages it finds there to the root node. These methods should return lists of further object nodes which are collapsed into one single list and displayed in the first column of the browser.

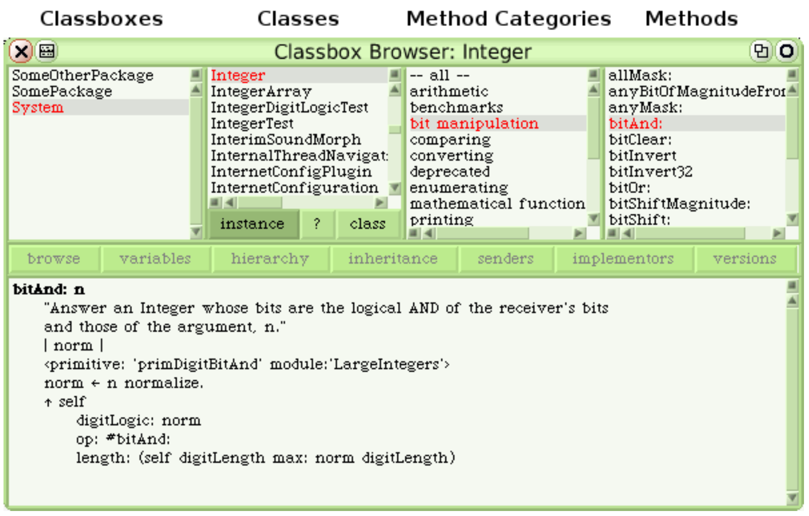
In our example, the file browser finds the #files and #directories edges leading away from the root metanode and thus sends these messages to the root node, a DirectoryNode on the path 'Macintosh HD:tmp:'. This results in a list of directory and file nodes, the children of the tmp directory, which are displayed in the first column of the browser.

When a user selects an object from a column of the browser, this process repeats itself. The metanode associated with the selected object is examined for edges, the messages are sent to the object node and the returned nodes are displayed in the column next to the current one. So, if the user clicks on browser in the first column of the file browser, the #files and #directories are sent to the directory node on that path and the results build up the second column as shown in Figure 3.

3.3 Designing the Classbox Browser

As we have seen in the file browser example, at its core a browser is a column-based, hierarchical view on a system. For our classbox browser we must now ask what exactly that hierarchy should be and what kind of objects are to be displayed in the columns. In a system using classboxes, they are the top-level unit of scoping, therefore they must be at the root of our hierarchy. At the next level, contained within classboxes, we have *classes*, and defined in a class are its *methods*. Smalltalk systems use one more intermediary organizational unit, *method categories*. These have no semantical meaning in the language, but are used to organize the methods of a class for easier access.

Thus we end up with a hierarchy of objects that looks like this: Classboxes ⇒ Classes ⇒ Method Categories ⇒ Methods. This is also exactly the order in which we want to display these objects in the columns of our browser. Figure 4 shows the classbox browser according to this design. Not quite by coincidence this superficially looks identical to a standard Smalltalk system browser. The difference is that in the first column we see classboxes whereas in a system browser we see class categories, an organizational unit for classes in Smalltalk.



```
bitAnd: n
    "Answer an Integer whose bits are the logical AND of the receiver's bits
    and those of the argument, n."
    | norm |
    <primitive: 'primDigitBitAnd' module:'LargeIntegers'>
    norm ← n normalize.
    ↑ self
        digitLogic: norm
        op: #bitAnd:
        length: (self digitLength max: norm digitLength)
```

Figure 4: The Classbox Browser.

There are a few more things to note about Figure 4. There is a row of buttons between the columns and the properties window. In Squeak, these are normally used to open different kinds of browsers on currently selected objects. We haven't yet implemented any alternative browsers to be used in conjunction with classboxes, so these are disabled and we will not talk about them in this paper.

Furthermore, in the second column we see three buttons labeled "instance", "?" and "class". The question mark button is easily explained: When clicked, it shows the documentation of the selected class. The "instance" and "class" buttons differentiate between instance and class methods; depending on which of those is clicked, the third and fourth column show instance or class methods. We will talk about these buttons in more detail later, when we discuss their implementation.

3.4 Classbox Browser Implementation

The classbox browser implementation is of course much more complex than the simple file browser example we have seen, and it uses some OmniBrowser concepts we haven't introduced. We therefore don't discuss the whole implementation but focus on some of its interesting aspects, the metagraph and the object nodes.

3.4.1 Classbox Metagraph

The design described in the previous section leads us directly to the metagraph for our classbox browser, shown in Figure 5. At the root we have an Environment metanode that represents the whole Smalltalk environment. It contains classboxes, represented by the Classbox metanode. Next are the three metadnodes Class, Metaclass and Comment, this part of the graph will be explained in a moment. From classes and metaclasses we can get to their method categories, represented by the MethodCategory metanode. Finally, the Method metanode represents the methods that belong to these categories.

This metagraph has one peculiarity that warrants further discussion: Three metanodes, Class, Metaclass and Comment are connected to the Classbox metanode. When OmniBrowser has to compute the children of a classbox, it will follow all of these links. This means, it sends the three messages #classes, #metaclasses and #comments to the classbox object node. Each of these will return a list of object nodes for *every* class in the classbox. So without additional measures, every class appears three times as a child of the classbox in the second browser column; as a class node, a metaclass node and a comment node.

We want only class nodes, metaclass nodes or comment nodes to be displayed, depending on which radio button the user selected at the bottom of the second column. This is achieved with the OmniBrowser concept of a *filter*. A

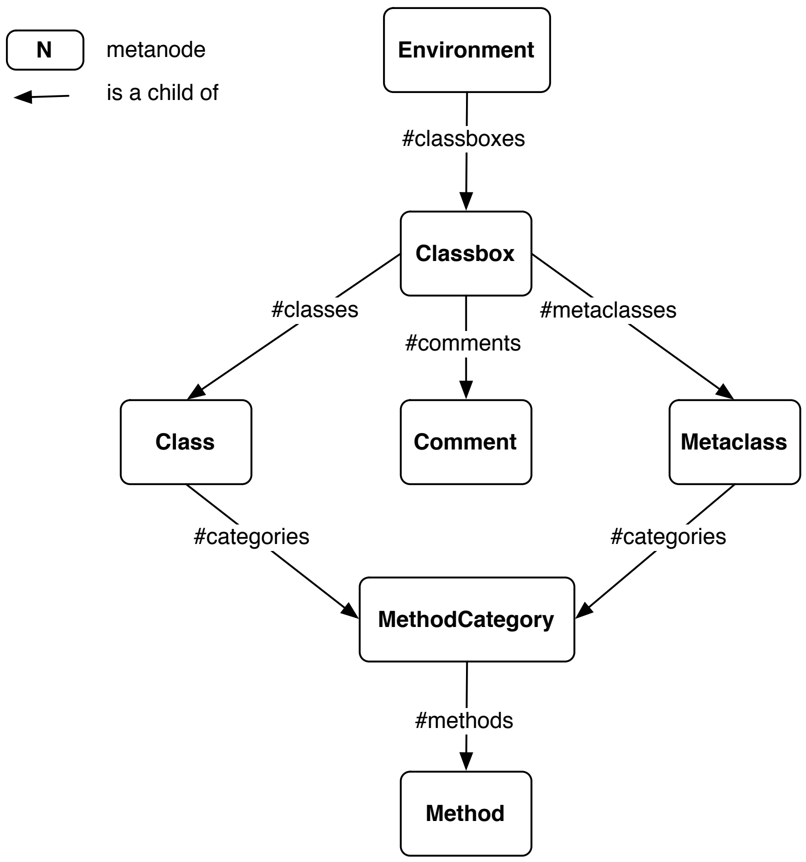


Figure 5: Classboxes metagraph.

filter can selectively hide nodes from a view. In the classboxes metagraph, a *modal filter* is used. This is turned into the set of radio boxes by OmniBrowser, building up the familiar three buttons in the second browser column.

3.4.2 Object Node Classes

We must implement object node classes to build up concrete object graphs according to the blueprint the metagraph provides. In general, every metanode needs at least one correlating node class derived from OBNode. Node classes must obviously implement the navigation methods as defined by the edges leading away from their correlating metanode. For example, we need a class ClassboxMethodCategoryNode that implements the #methods method specified in the metagraph.

Apart from that, the main responsibility of a node class is to regulate the *actions* on them provided through the browser. Examples of such actions are creating and removing a classbox. OmniBrowser provides a rather elaborate framework for actions; they can be initiated through the context menu, buttons or drag-and-drop movements. But we will not go into the details of how these actions are implemented here. Instead here's an overview of the node classes implemented for the classbox browser with high-level descriptions of their behaviour:

ClassboxNode provides access to the classes under the scope of a given classbox. It implements actions for creating and removing classboxes as well as a find dialog that searches a classbox for classes. In addition, it handles the classbox-specific feature of importing a class from another classbox.

ClassboxClassNode provides access to the method categories reachable from a class in a given classbox. It supports the creation and removal of classes, the former through the help of ClassboxClassDefinition. This class is responsible for showing the correct class creation template in the bottom panel of the browser and also determines what happens when the template is modified and accepted.

Some actions on classes do make a distinction between imported classes and classes defined in the current classbox. For example removing an imported class only deletes the import relationship, whereas removing a defined class actually deletes it from the system. However, these slight differences turned out to not warrant spawning subclasses to cleanly separate imported from defined classes.

ClassboxMetaclassNode inherits from ClassboxClassNode and only overrides a very small number of methods to provide analogous functionality for metaclasses. Smalltalk's object model makes it possible to treat classes and metaclasses uniformly here.

ClassboxMethodCategoryNode provides access to a category's methods defined under the scope of a classbox. New method categories can be created, but removal is not supported at this point.

ClassboxAllMethodCategoryNode is a special kind of method category node, in fact a specialization of ClassboxMethodCategoryNode. It represents the "– all –" category always listed first in the method category column of the browser. This is an artificial category that contains all methods of a given class.

ClassboxMethodNode supports the creation, extension and redefinition of methods. Extended and redefined methods are marked with an asterisk in front of the name in the browser. Method removal is currently not supported.

It's noteworthy that the classbox superficially looks and acts almost identical to the system browser already present in OmniBrowser. It was therefore most convenient to derive parts of the classbox browser from OmniBrowser's system browser implementation. Most new node classes inherit from existing nodes of the system browser (ClassboxClassNode from OBClassNode, ClassboxMethodNode from OBMethodNode, and so on).

4 Using the Browser

Some of the more subtle design decisions in the browser haven't been discussed, yet. We would like to do this in this section by putting the browser to work.

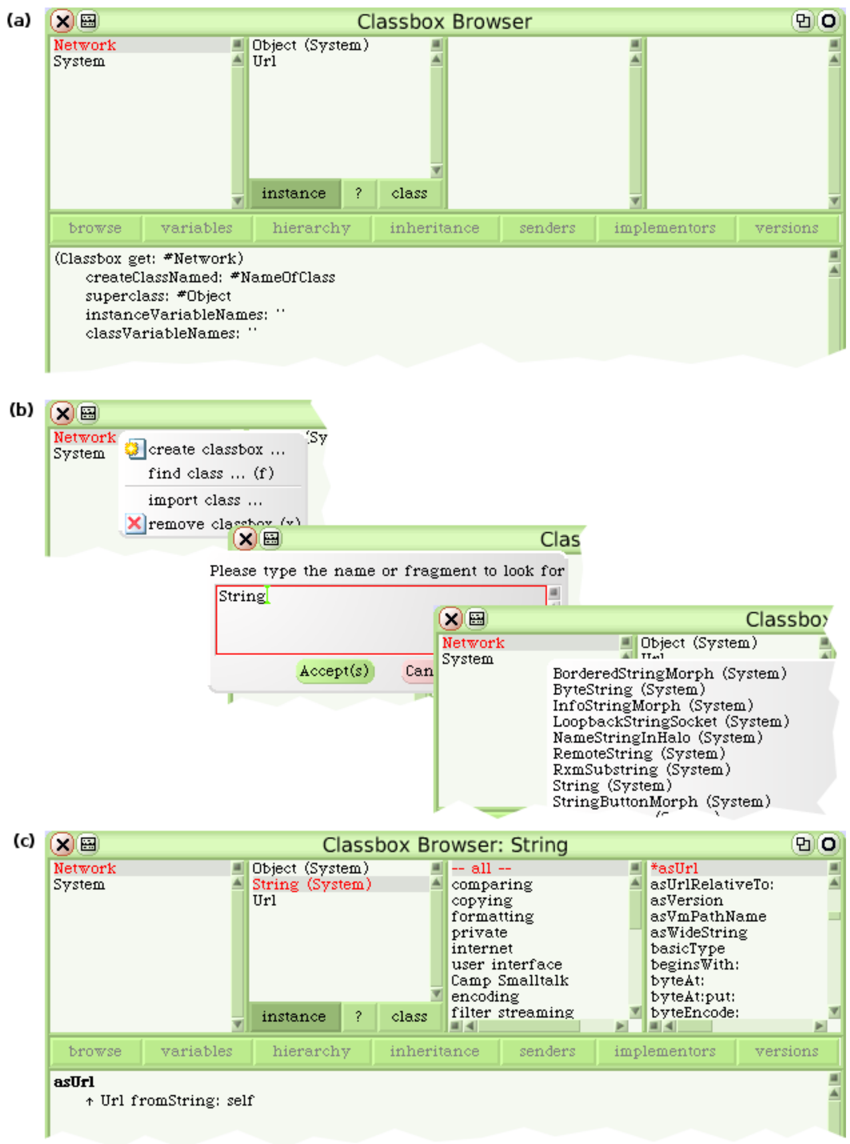
We reuse the previously discussed class extension example shown in Figure 1. We assume that we have a classbox Network that contains a class Url. There is another classbox named System that contains all the other classes in the environment; all classes not explicitly created in a classbox are place in this default classbox. Figure 6 (a) depicts this scenario in the browser.

Note that the Network classbox also contains the Object class, the base class for user-defined classes in Squeak. This class is not actually defined here but *imported* from another classbox, namely System. The browser makes this obvious to the user by appending the name of the classbox it has been imported from in parentheses after the class name. Object is in fact by default imported into all new classboxes created. This is just a small convenience for the user. All new classes defined in a classbox must be subclassed from an existing class in its scope; very often this will be Object from System.

Following our example, we want to extend the String class with an asURL method that returns an instance of our own custom Url class. Actually, String already does have this method, so we will in fact *redefine* it in our classbox. Also note that an Url class already exists in the System classbox. The classbox model allowed us to define a completely independent Url class in the scope of the Network classbox.

To extend String we first have to import it into our classbox. We do this by right-clicking on the classbox in the browser and choosing "Import class ..." from the context menu. We are presented with a dialog where we can enter the name or a fragment of the name of the class to import. If there are several classes matching the entered name, the browser shows a list of them for the user to confirm. Figure 6 (b) shows this sequence of actions for the import of String.

For the redefinition of String>>asUrl we simply find the method in the browser, modify the source of the existing implementation and save it. Seeing that the method was defined in another classbox, the browser will correctly recognize this as a redefinition. The redefined method is shown in Figure 6 (c). The name of the method is now prepended with an asterisk in the browser. All



```
(Classbox get: #Network)
    createClassNamed: #NameOfClass
    superclass: #Object
    instanceVariableNames: ''
    classVariableNames: ''
```

```
BorderedStringMorph (System)
ByteString (System)
InfoStringMorph (System)
LoopbackStringSocket (System)
NameStringInHalo (System)
RemoteString (System)
RxmSubstring (System)
String (System)
StringButtonMorph (System)
```

```
asUrl
    ↑ Url fromString: self
```

Figure 6: Steps for a method redefinition in the classbox browser.

extended or redefined methods are marked like this to clearly distinguish them from unmodified methods of imported classes.

Our example is now complete. When we now use String>>asUrl from anywhere within the Network classbox, our new implementation with our own class will be used. All other existing code in the environment is not affected by our modification. Here, classboxes succeed in controlling changes to existing classes by putting them under a scope.

5 Conclusion

A class extension is a useful technique to evolve software in ways not foreseen when it was created. The classbox model addresses the problems of class extensions: It limits their impact to a well-defined scope.

With the classbox browser we created a tool that assists programmers in working with classboxes in the Squeak environment. Using the browser, a programmer can create classboxes, define classes in the scope of a classbox and import classes from other classboxes. The browser can also apply class extensions and method redefinitions within a classbox. This enables the convenient modification of a class without affecting any of its existing clients.

We created the classbox browser using the OmniBrowser framework. In this paper we also introduced OmniBrowser's core concepts. This can be read as a tutorial for newcomers to OmniBrowser.

References

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, New York, NY, USA, 2005. ACM Press. To appear.

[BDNW05] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

[GR89] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language.* Addison Wesley, 1989.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[Pae93] Andreas Paepcke. User-level language crafting. In *Object-Oriented Programming : the CLOS perspective*, pages 66–99. MIT Press, 1993.

[Put] Colin Putney. OmniBrowser, an extensible browser framework for Smalltalk. http://www.wiresong.ca/OmniBrowser/.

[PW88] Lewis J. Pinson and Richard S. Wiener. *Objective-C.* Addison Wesley, 1988.