Hierarchical Layer-Based Class Extensions in Squeak/Smalltalk

Matthias Springer[‡] Hidehiko Masuhara[‡] Robert Hirschfeld[†,§]

[‡] Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan
[†] Hasso Plattner Institute, University of Potsdam, Germany
[§] Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA
matthias.springer@acm.org masuhara@acm.org robert.hirschfeld@hpi.de

Abstract

Class extensions are frequently used in programming languages such as Ruby and Smalltalk to add or change methods of a class that is defined in the same application or in a different one. They suffer from modularity issues if globally visible: Other applications using the same classes are then affected by the modifications.

This paper presents a hierarchical approach for dynamically scoping class extensions in dynamically-typed, class-based programming languages supporting class nesting. Our mechanism allows programmers to define the scope of class extensions and to reuse class extensions in other programs. Class extensions can be scoped according to a nested class hierarchy or based on whether programmers regard a potentially affected class as a black box or not. Class extensions are organized in layers, where multiple layers targeting the same class can be active at the same time.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Class extension, context-oriented programming, mixins

1. Introduction

Class extensions are in-place modifications of existing classes: instance methods are added or modified in an already existing class (*target class*) that is typically defined somewhere else in the same program or in an external library. We call the former case a *class addition* and the latter one a *class refinement*.

There are a variety of use cases for class additions. The most common use case in mainstream programming languages such as Ruby is adding convenience methods to organize helper methods in an object-oriented way. For example, the Ruby library *ActiveSupport* provides methods like `Fixnum.minutes` and `Fixnum.hours` to make it easy to perform time calculations such as `4.hours + 2.minutes`. Another use case is multi-dimensional separation of concerns [12]. The idea is that a class or group of classes may exhibit a number of different concerns which programmers may want to group together for understandability reasons. While object-oriented design without class extensions allows only for a "single, dominant dimension of separation", class extensions can be used to group methods of a class belonging to the same concern.

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...$15.00
http://dx.doi.org/10.1145/2892664.2892682

Class refinements are typically used for bug fixing or modularization of behavioral variations. Multiple behavioral variations can target the same classes and methods. In that case, there must be a way to specify which variation should be used and possibly combined. Context-oriented programming [7] (COP) is a mechanism for modularizing context-dependent behavioral variations. The mechanism presented in this paper is similar to context-oriented layer activation and focuses on dynamically-scoped class extensions, but other mechanisms will be discussed towards the end of this paper.

1.1 Background

This paper proposes a concept for class extensions in the Matriona module system [10] for Squeak/Smalltalk, but the main ideas are amenable to other class-based programming languages. Matriona runs on top of Squeak and provides class nesting and class parameterization. It is implemented without modifying the underlying virtual machine and is based on metaprogramming.

In Matriona, every class can have variables, methods, and nested classes. Nested classes are always class-side members and their purpose is typically to serve their enclosing classes [2]. A new hierarchical name lookup mechanism featuring the `scope` keyword looks up class names and methods by traversing the class nesting hierarchy, i.e., the lookup starts in the current class and continues in enclosing classes until the single top-level class `Smalltalk` is reached (see Section 3.3).

1.2 Requirements

Our motivation for supporting class extensions in Matriona is threefold: First, Matriona should support class extensions for backward compatibility to Smalltalk, because Matriona aims to be a superset of Smalltalk. Second, we would like to promote modular understandability through multi-dimensional separation of concerns. Third, we would like to provide an easy way to implement behavioral variations and to add new operations to an existing class. At the same time, class extensions should be confined to a local scope to avoid the problem of destructive class extensions.

Backward Compatibility Class extensions (star categories in Monticello [8]) are a widely-used feature of Smalltalk. For backward compatibility reasons, Matriona should support this kind of class extensions. The main question is how to organize class extensions in Matriona, where classes are no longer organized in *flat* packages but in a hierarchical nesting structure.

Multi-dimensional Separation of Concerns Matriona is a module system that aims to support modular composability, modular decomposability, and modular understandability. The latter point can be further supported by grouping methods describing a common concern together, even beyond class boundaries. The basic idea is to define a set of classes according to one "dominant" dimension

and to group class additions to "encapsulate concerns in dimensions other than the dominant one" [12].

Behavioral Variations and Additional Operations Changing existing or adding new methods to an existing class is difficult without class extensions. One approach is to create a subclass and perform changes in the subclass, but this approach fails if the programmer is not in control of instance creation. Another approach for tree-based data structures is to use the Visitor design pattern [5], but this approach results in more overhead and infrastructural complexity due to additonal classes and double dispatch.

Destructive Class Extensions Class extensions are known to suffer from modularity issues. First, whenever two different modules extend the same class, conflicts can occur when both extensions define methods with the same name. It is then unclear which extension to use. Such class extensions are called *destructive* [11]. Second, class extensions might only be suitable in the context of their defining modules. For example, if one module refines a class, it might be desirable to run the original implementation whenever the class is used outside of the scope of the refining module.

1.3 Contributions

This paper makes the following main contributions.

- Hierarchically and dynamically-scoped class extensions
- A layer-based approach for handling destructive class extensions

In the remainder of this paper, we motivate our design using three examples, describe the concept of our approach, and give an overview of related work.

2. Examples

In this section, we present three examples to justify the requirements and give a high-level overview of the class extension mechanism, that we will formally define in the following section.

2.1 Locality of Changes

This example is taken from Method Shells [11], so it might be familiar to some readers. We would like to develop an embedded web browser and an audited viewer for web pages, represented by classes `Browser` and `Viewer`, respectively. Both applications use the same `WebPage` library, which contains functionality for showing popups.

```
Smalltalk class»WebPage                                    < class >
   ↑ Smalltalk Kernel Object subclass
Smalltalk WebPage»popup: URL
   " Show popup dialog "
Smalltalk WebPage»open: URL
   " ... "
   popupRequested ifTrue: [ self popup: '...' ]
```

The browser application should not display popup windows, whereas the audited viewer application should show a popup window whenever a confidential file is accessed.

```
Smalltalk class»Viewer                                     < class >
   ↑ Smalltalk Kernel Object subclass
Smalltalk Viewer»check: file
   | page |
   page := Smalltalk WebPage new.
   (self isConfidential: file) ifTrue: [
      page popup: '<b>confidential</b>' ].
   " ... "
```

In this design, the embedded web browser defines a class refinement for `WebPage.popup` to disable popup windows (Figure 1). The class extension mechanism should allow for the following behavior.

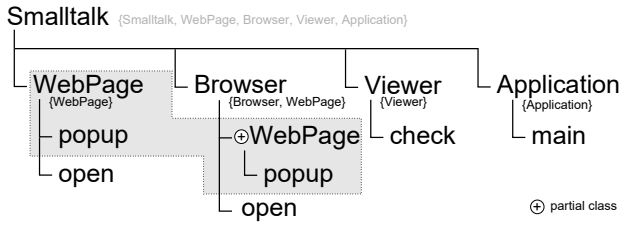

Figure 1: Example: Locality of Changes. Gray boxes indicate classes and their extensions, sets enclosed in curly braces indicate the scope of a class.

First, when another application uses both the embedded web browser and the audited viewer, the viewer should still show popup windows. In other words, the *scope* of the browser's class extensions should be confined to `Browser`.

Second, in another design, where the embedded web browser uses the audited viewer as a dependency, programmers should be able to choose whether the browser's class extensions should affect the audited viewer or not. In other words, programmers should be able to specify whether the scope of the browser's class extensions includes `Viewer` or not.

Representing Class Extensions In Matriona, class extensions are represented by *partial methods*. Every partial method is contained in one *partial class*. Every partial class belongs to exactly one *target class*, which is the class that is extended, and is contained (nested) in exactly one class. For example, `Browser` is a class defining a partial class `WebPage` targeting the class `St.WebPage` and containing a partial method `popup`.

```
Smalltalk class»Browser                                    < class >
   ↑ Smalltalk Kernel Object subclass
Smalltalk Browser class»WebPage                            < partial >
   ↑ Smalltalk WebPage
Smalltalk Browser WebPage»popup: text
   " Do nothing "
Smalltalk Browser»open: URL
   Smalltalk WebPage new open: URL
```

Scope of Class Extensions To avoid destructive class extensions, Matriona has rules to activate and deactivate class extensions. Matriona supports (de)activation on a per-class level, i.e., partial classes cannot be (de)activated separately. We use the term *class (de)activation* to denote that all class extensions defined in a class are (de)activated. The rule for class activation is simple: A class is activated if one of its methods is executed. For example, if `Browser.open` is executed, class `Browser` is activated.

Class deactivation depends on the *scope* of a class. The scope of a class determines how long a class should remain activated if activated before, i.e., the scope of a class affects only deactivation but not activation. It always contains the class itself (*reflexivity*). For example, when calling `WebPage.popup` from `Browser`, the method lookup will select the class refinement.

The scope of a class also contains the target classes of all partial classes. For example, if `WebPage.open` calls `popup`, the method lookup will use the class refinement if `open` was called from `Browser`, because `WebPage` $\in$ *scope*(`Browser`). This behavior is known as *local rebinding* [1].

Full Example We now define an `Application` using both `Browser` and `Viewer`. The browser application will not generate popup windows, whereas the audited viewer application will, because `Browser` is deactivated when `Browser.open` returns to `Application.main`, since `Application` $\notin$ *scope*(`Browser`).

```
Smalltalk class»Application                    < class >
    ↑ Smalltalk Kernel Object subclass
Smalltalk Application»main
    Smalltalk Browser new open: 'http://...'.
    Smalltalk Viewer new check: 'secret.html'.
```

Consider a slightly different case now where `Browser` uses `Viewer` internally, while both of them still use `WebPage` for rendering purposes (Figure 2). Once `Browser` calls a method in `Viewer`, class `Browser` is deactivated, because `Viewer` $\notin$ *scope*(`Browser`). `Viewer` still works properly by showing a popup window.

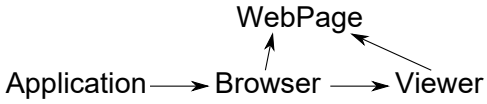

Figure 2: Example: `Browser` uses `Viewer`

The programmer could apply `Browser`'s class extensions to `Viewer` by adding a partial class targeting `Viewer` to `Browser`.

2.2 Hierarchical Scoping

Consider a networking library that consists of a class `Networking` where functionality such as sockets and DNS name resolution is organized in a class nesting structure within `Networking`. Network endpoints are represented by instances of class `Address`, which is nested inside `Networking`. In this design, the networking library defines a class addition `String»asAddress` to make it easy to convert string representations of DNS names and IP addresses to instances of `Networking` (Figure 3).

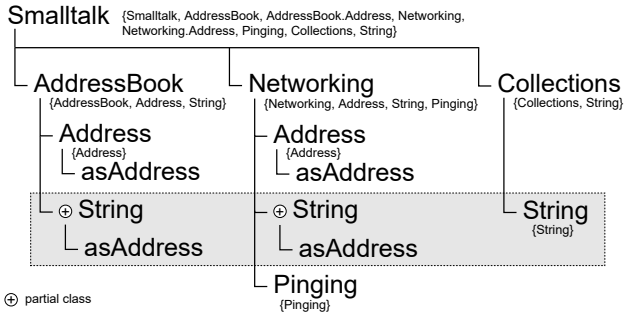

Figure 3: Example: Duplicate Convenience Methods

The class extension mechanism should ensure that the class addition `String»asAddress` is visible in the entire networking library. In other words, the scope of that class extension should contain all nested classes of `Networking`. Moreover, in accordance with the mechanism described in the previous section, the class extension should not be visible in other applications. For example, consider that an address book application contains a class `Address` representing mail addresses along with a converter method `String»asAddress`. The scope of each class extension should be confined to its respective enclosing class.

Scope of Class Extensions We extend the notion of the scope of a class such that it also includes all nested classes of the class. Furthermore, we activate a class not only if one of its method is executing, but also if a method contained in one of its nested classes is executing. For example, when calling a method in `Pinging`, the classes `Pinging`, `Networking`, and `Smalltalk` are activated. The class `Networking` remains active even inside `Pinging`, because `Pinging` $\in$ *scope*(`Networking`).

Moreover, according to the rule described in the previous section, both classes `AddressBook` and `Networking` can define class additions `String»asAddress` and work side by side, because class extensions defined in `AddressBook` are not active in `Networking` and class extensions defined in `Networking` are not active in `AddressBook`. The reason for that is that `Networking` $\notin$ *scope*(`AddressBook`) and `AddressBook` $\notin$ *scope*(`Networking`).

2.3 Importing Class Extensions

Consider a libary representing abstract syntax trees (AST) that defines a tree-based data structure of nodes, along with various evaluation strategies. In this design, every strategy is represented as a set of class extensions for AST node classes providing an implementation of `evaluate`. `Evaluating` is the *default* evaluation strategy. Notice how the concerns *evaluating* and *printing* are grouped into their own separate enclosing classes (Figure 4).

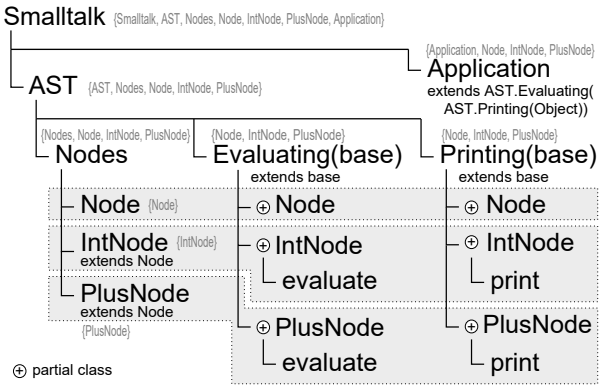

Figure 4: Example: AST Library

```
Smalltalk AST Nodes class»Node                 < class >
    ↑ Smalltalk Kernel Object subclass
Smalltalk AST Nodes class»IntNode              < class >
    ↑ Smalltalk Kernel Object
        subclassWithInstVars: 'value'
Smalltalk AST Nodes class»PlusNode             < class >
    ↑ Smalltalk Kernel Object
        subclassWithInstVars: 'left right'
Smalltalk AST Evaluating: class»IntNode  < partial >
    ↑ scope Nodes IntNode   "→ St AST Nodes IntNode"
Smalltalk AST Evaluating: IntNode»evaluate
    ↑ value
Smalltalk AST Evaluating: class»PlusNode < partial >
    ↑ scope Nodes PlusNode "→ St AST Nodes PlusNode"
Smalltalk AST Evaluating: PlusNode»evaluate
    ↑ left evaluate + right evaluate
```

The class extension mechanism should allow programmers to choose an evaluation strategy for their application, or possibly combine multiple strategies. For example, strategy `Mod10Evaluating` uses the *default* evaluation strategy and takes the result modulo 10. In other words, programmers should be able to import class extensions into their application as if they were part of their application.

Mixin-based Class Extensions Matriona does not provide a dedicated language construct to import classes, but support for mixins [3]. A mixin is a class with a `base` parameter that serves as the superclass. Therefore, it can be applied (imported) to many classes.

```
Smalltalk AST class»Evaluating: base          < class >
   ↑ base subclass
```

An application can apply the mixin `Evaluating:` during class definition, making it part of the resulting class's superclass hierarchy. Matriona's method lookup takes into account superclasses of classes: It first activates class extensions from superclasses and then class extensions from the actual class (similar to layer activation in COP).

```
Smalltalk class»Application
   ↑ (Smalltalk AST Evaluating:
      Smalltalk AST Printing: (
         Smalltalk Kernel Object)) subclass
```

Scope of Class Extensions We extend the notion of the scope of a class such that it also includes all classes contained in the scope of its superclass. For example, `Application`'s superclass is an application of mixin `Evaluating:`, whose scope includes all AST node classes. The only class being activated is `Application` but not its superclass. However, the method lookup does not only take into account partial classes defined in an activated class L, but also partial classes defined in its superclass L', starting with L and then L' (see Section 3.3). The `super` keyword can be used in a partial method of L to call a partial method defined in L' or one of its superclasses (similar to `proceed` in COP).

Consequently, when executing a method in `Application`, class extensions from `Evaluating:` are active and remain active as long as methods from `Application` or any AST node class are executed.

Layered Class Extensions We now want to modify our AST evaluator in such a way that all results and partial results are calculated modulo 10. For that reason, we define a set of class extensions `Mod10Evaluating:` that runs on top of `Evaluating:`, i.e., whenever the mixin `Mod10Evaluating:` is applied, the mixin `Evaluating:` is applied first, automatically.

```
Smalltalk AST class»Mod10Evaluating: base  < class >
   ↑ (self Evaluating: base) subclass
Smalltalk AST Mod10Evaluating: class»IntNode
   < partial >
   ↑ scope Nodes IntNode
Smalltalk AST Mod10Evaluating: class»PlusNode
   < partial >
   ↑ scope Nodes PlusNode
Smalltalk AST Mod10Evaluating: IntNode»evaluate
   ↑ super evaluate \\ 10
Smalltalk AST Mod10Evaluating: PlusNode»evaluate
   ↑ super evaluate \\ 10
```

A mixin application of `Mod10Evaluating:` contains two partial methods for `IntNode.evaluate` and `PlusNode.evaluate`. Since a mixin application of `Evaluating:` is a subclass of a mixin application of `Mod10Evaluating:`, method execution will start with methods from the latter one. The `super` keyword in the `evaluate` methods corresponds to `proceed` calls in COP and executes the `evaluate` implementation defined in `Evaluating:`.

3. Concept

Matriona provides a variant of context-oriented programming [7] (COP) to support class extensions. Every class can not only contain variables, methods, and nested classes, but also *partial classes*, i.e., every class with its partial classes can act as a layer. A partial class is a special form of a nested class which does not define a new class via subclassing but extends a specific existing *target class*. Every partial class can contain a number of partial methods. Such a method can be a class addition or a class refinement.

3.1 Rationale

Class extensions can be beneficial for their defining classes, but they can break other classes or libraries if they are global. Programmers typically treat external libraries as black boxes [11], i.e., it is hard to anticipate the effect of a class extension. Therefore, our class extension mechanism should allow programmers to define the scope of a class extension, i.e., where it is active. As a rule of thumb, we propose that a class extension should be deactivated if the control flow is passed to another class or library that the programmer regards as a black box. Defining a partial class (class extension) for a target class C within class L essentially means that C is no longer being regarded as a black box from L's point of view.

Class nesting can be used as an orthogonal scoping mechanism to indicate that a class extension defined for an enclosing class should also be active for all of its nested classes. If an enclosing class defines a class extension for C , then C is not being regarded as a black box from the perspective of any nested class.

3.2 Scope of a Class

Matriona maintains a global stack for active classes. A class is active only as long as the control flow stays within the class's *scope*.

Definition. *The scope of a class L is defined as the set containing L, all target classes (and their reachable nested classes[1]) corresponding to partial classes of L, all classes in the scope of all nested classes of L[2], and all classes in the scope of the superclass of L.*

$$
\begin{aligned}
&scope(L) = \{L\} && \textit{(reflexivity)}\\
&\quad \cup \{C \mid C \in nested^*(target(P)) \wedge P \in partials(L)\} && \textit{(local rebinding)}\\
&\quad \cup \{C \mid C \in scope(N) \wedge N \in nested(L)\} && \textit{(hierarch. scoping)}\\
&\quad \cup scope(superclass(L)) && \textit{(importing class extensions)}
\end{aligned}
$$

Note that class extensions remain active even if the control flow changes from one target class A to another one B. The rationale is that programmers do not regard A and B as black boxes, since they are changing both of their behavior. Consequently, programmers should also be aware of the interaction between A and B.

Whenever a method `C.foo` is invoked or the control flow returns to that method, Matriona performs the following steps before execution.

1. For all active classes L, if $C \notin scope(L)$, deactivate L.
2. Push `C` and all of its enclosing classes onto the the layer (class) composition stack (starting with the outermost class).

Classes are activated implicitly only through method invocation. Programmers cannot activate classes manually. Furthermore, classes are never activated multiple times. If an already activated class is activated once more, the position of that class will be changed, such that it is on top of the stack. The only purpose of *scope* is to decide when to deactivate a class.

3.3 Method Lookup

Whenever a message is sent to an object, Matriona first determines the receiver's class C. Instead of using C's superclass hierarchy for

[1] $nested^*(C) = \{C\} \cup \{D \mid D \in nested^*(N) \wedge N \in nested(C)\}$, i.e., C and all nested classes of C and their nested classes etc.

[2] Therefore, $nested^*(L) \subseteq scope(L)$.

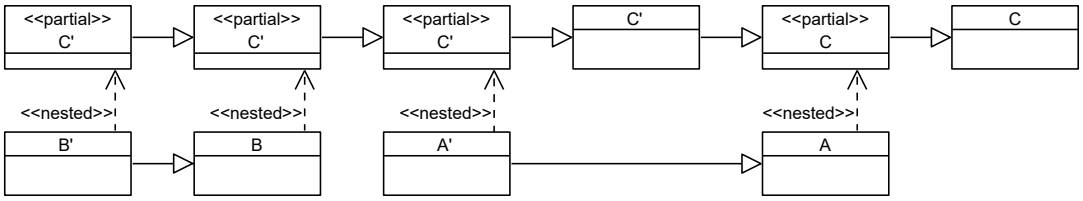

Figure 5: Example: Effective Superclass Hierarchy

the method lookup, Matriona uses its *effective superclass hierarchy*, a combination of C's superclass hierarchy and partial classes for C and its superclasses nested in classes on the layer composition stack.

Effective Superclass Hierarchy In contrast to many COP frameworks, Matriona does not have a `proceed` statement for calling a partial method defined in the next class on the layer composition stack. Matriona uses the `super` keyword instead. This keyword is used for both calling overridden methods defined higher in the superclass hierarchy and for calling partial methods defined in a class lower on the layer composition stack. From a method lookup point of view, Matriona merges the *actual superclass hierarchy* and the layer composition stack into a combined *effective superclass hierarchy*. The rule for merging both hierarchies is simple: For every class C in the actual superclass hierarchy, first look up methods in partial classes of C on the layer composition stack, then look up methods in C. Partial classes are effectively subclasses which are applied dynamically depending on the layer composition.

Algorithm 1 shows the mechanism for generating the effective super class hierarchy as pseudo code, i.e., it returns a list of classes which will be used for looking up a method of (late-bound) receiver class *class* during subsequent super calls.

Algorithm 1 Effective Superclass Hierarchy

```
 1: procedure EFFECTIVE(class, layers)
 2:     h ← []
 3:     for l ∈ layers.reversed() do      (starting with top of stack)
 4:         s ← l
 5:         repeat
 6:             if s.hasPartial(class) then
 7:                 h.add(s.getPartial(class))
 8:             end if
 9:             s ← superclass(s)
10:         until s = null
11:     end for
12:     return h + [class] + EFFECTIVE(superclass(class), layers)
13: end procedure
```

Figure 5 illustrates the effective superclass hierarchy in an example. Let us assume that the layer composition stack contains `A'` and `B'` on top. Classes `A'`, `B`, and `B'` have partial classes for `C'` and class `A` has a partial class for its superclass `C`. Consequently, the effective class hierarchy for `C'` (EFFECTIVE(`C'`, (`A'`, `B'`))) starts with partial classes for `C'`, continues with `C'` itself, followed by a partial class for `C` and class `C` itself.

As another example, Figure 6 shows the effective superclass hierarchy for class `IntNode` defined in Section 2.3. If the mixin `Mod10Evaluating:` is applied, the hierarchy is prepended with the corresponding partial class defined in that mixin.

Class Nesting The keyword `scope` exists in Matriona to send messages to enclosing classes one by one, until a class understand the message[3]. With respect to partial classes, the lookup mechanism uses the effective superclass hierarchy instead of the actual superclass hierarchy when trying to send messages to enclosing classes.

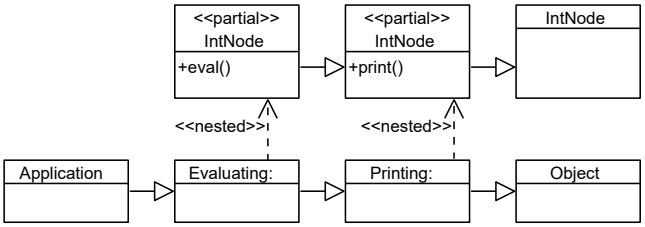

Figure 6: Example: Effective Superclass Hierarchy for `IntNode`

3.4 Class Activation

If a class extension defined in class L should be active when executing a method defined in class A, one of the following two designs can be applied.

Scope-based Activation In this design, programmers have to ensure that the control flow reaches A via a sequence of methods defined in classes $L \to C_1 \to \ldots \to C_n \to A$ with $C_i \in scope(L)$ (for all $i = 1 \ldots n$) and $A \in scope(L)$. Programmers can enforce that a class $C_i \in scope(L)$ by adding a partial class targeting C_i to L (partial classes can be empty). For example, in Section 2.1, $n = 1$, $L =$ `Browser`, $C_1 =$ `WebPage`, and $A =$ `WebPage` when calling `WebPage.popup` via `WebPage.open` from `Browser.open`.

Mixin-based Activation The previous design is hard to accomplish if a class extension should be shared among a variety of classes. The following design encapsulates class extensions in mixins and activates them using mixin application. A mixin is an abstract subclass that can be applied to a number of superclasses. When partial classes are nested inside a mixin M that is applied to a superclass C, all of M's class extensions are active when the control flow passes through a method in the context of the resulting class C' (i.e., the polymorphic receiver class is C').

Consider Figure 6 as an example. `Evaluating:` and `Printing:` are mixins with partial classes for `IntNode`. Class `Application` is defined as a subclass of the application of both mixins. When a method is executed in the context of `Application`, then that class is pushed onto the layer composition stack. The effective superclass hierarchy of class `IntNode` contains the partial classes of both mixins (see Algorithm 1).

4. Related Work

In this section, we compare our approach for class extensions with other approaches, focusing on *invasiveness* and *scoping*.

[3] In the light of subclassing, the lookup is more complex, but the details do not matter in this paper.

Invasiveness A mechanism is invasive if it affects other components of an application. The least invasive mechanisms for adding operations to classes are the Visitor design pattern [5] and subclassing, but they either result in an overhead of classes and complexity or fail if the programmer is not in control of instance creation. A variety of extensions to programming languages have been proposed supporting locality of changes, i.e., class extensions are active only in a certain scope.

Scoping A Classbox [1] is a container and namespace for classes. Classes can be imported from other classboxes. Changes to imported classes are visible only in the extending classbox or in classboxes importing extended classes. Similar to Matriona, including a class C from another classbox into a classbox B extends the scope of B onto C (*local rebinding*). An application can be represented by a classbox defining its classes, importing external classes, and extending them locally. In Matriona, an application is a single (enclosing) class, whose class extensions are visible in all nested classes.

Method Shells [11] are a similar mechanism. Classes and *revisers* (containers for class extensions) are contained in a method shell. Class extensions are visible only within the extending method shell or when the method shell is imported into another one. Classes from other method shells can also be imported with the `link` keyword, which will not include revisers in the current method shell and switch the *context* (active method shell) to the method shell of the included class when a method from that class is executing. Matriona cannot `link` other classes; however, the scope of a class allows programmers to deactivate class extensions, which is similar to linking in Method Shells and sufficient to implement the examples shown in that paper [11]. One important difference is that local rebinding is not transitive in Matriona, i.e., the scope of a class does not include the scope of all target classes, but only the target classes themselves. In Method Shells, `include` is transitive.

MultiJava [4] and Expanders [13] support statically-scoped class additions. The scope of class additions is confined to the source code file where they were defined, unless they are imported. MultiJava and Expanders take into account class additions during type checking at compile time, making it possible to detect and prohibit destructive class extensions. In contrast to Matriona, there is no scoping mechanism extending class extensions to collaborating classes, because class refinements are forbidden and class additions can only be referred to in a type-safe way if they were imported explicitly.

ContextS is a framework for context-oriented programming (COP) in Squeak/Smalltalk [6]. Partial methods can be used to encapsulate class extensions in COP layers. However, class extensions are not deactivated if the control flow reaches a class that programmers regard as a black box. For example, if a layer disabling popups is activated when running the application in Figure 2, `Viewer` will be broken, because it will not show popups. Programmers have to either know about `Viewer`'s internals to ensure that a class extension is not destructive or deactivate the layer manually before calling a method in `Viewer`. Matriona applies layer deactivation implicitly.

5. Summary

We proposed a hierarchical and layer-based approach for organizing class extensions in Squeak/Smalltalk based on the Matriona module system. This approach is similar to context-oriented programming, but class (layer) (de)activation is performed implicitly. A class in Matriona can be compared to a classbox or a method shell, but it is scoped hierarchically and does not require additional syntactical elements except for the definition of partial classes. Scoping and locality of changes are important to avoid destructive class extensions. Matriona lets programmers control the scope of class extensions by specifying whether a class should be regarded as a black box or not. We also showed how our mechanism can be used for multi-dimensional separation of concerns: Every concern is encapsulated in a mixin and can be activated during class definition, which is similar to `include` statements in Method Shells.

Future work might focus on a formal definition of the semantics of our mechanism and consider performance optimizations. Our current implementation approach is based on an image-side reimplementation of the method lookup using metaprogramming. Two particular problems are implicit class (de)activation, which takes place not only when a method is executed but also when the method returns, and `super` calls: Both are expensive in Matriona, but performance is explicitly not a goal at this time. Future versions of Matriona might contain optimizations like partial method inlining or layer composition caching [9].

References

[1] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Computer Languages, Systems and Structures*, November 2005.

[2] Joshua Bloch. *Effective Java (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[3] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM.

[4] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An Introduction to Context-Oriented Programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer Berlin Heidelberg, 2008.

[7] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[8] Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Squeak by Example*. Square Bracket Associates, 2009.

[9] Matthias Springer, Jens Lincke, and Robert Hirschfeld. Efficient Layered Method Execution in ContextAmber. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, COP'15, pages 5:1–5:6, New York, NY, USA, 2015. ACM.

[10] Matthias Springer, Fabio Niephaus, Robert Hirschfeld, and Hidehiko Masuhara. Matriona: Class Nesting with Parameterization in Squeak/Smalltalk. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 129–141, New York, NY, USA, 2016. ACM.

[11] Wakana Takeshita and Shigeru Chiba. Method Shells: Avoiding Conflicts on Destructive Class Extensions by Implicit Context Switches. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Software Composition*, volume 8088 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin Heidelberg, 2013.

[12] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119. ACM, 1999.

[13] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically Scoped Object Adaptation with Expanders. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 37–56, New York, NY, USA, 2006. ACM.