Springer,
Nested Class Modularity in Squeak/Smalltalk

Nested Class Modularity in Squeak/Smalltalk

by

Matthias Springer

A thesis submitted to the
Hasso-Plattner-Institute
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in IT-Systems Engineering

Supervisors

Prof. Dr. Robert Hirschfeld

Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany

July 17, 2015

Abstract

The english abstract.

Zusammenfassung

Die Zusammenfassung auf deutsch.

Acknowledgments

I owe everything to my cat.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API application programming interface

1. Introduction

1.1. Modularity

1.2. The Squeak Programming Language

1.3. Outline of this Work

2. Modularity Problems in Squeak

2.1. Class Name Clashes

flat global namespace

2.2. Dependency Managment

two application require the same module but in different versions

2.3. Readability and Understandability

only one level of grouping: package

2.4. Code Reuse

share behavior among multiple classes

3. Related Work

3.1. Class Name Clashes

3.1.1. Namespaces/Packages

VisualWorks, Java, Ruby, Python

3.1.2. Squeak Environments

3.1.3. Newspeak Modules

3.2. Dependency Management

3.2.1. Java Class Loader

3.2.2. Separate Compilation

3.2.3. External Configuration in Newspeak

3.3. Readability and Understandability

3.3.1. Smalltalk Packages

3.3.2. Hierarchical Decomposition

Java, Python, Ruby, Newspeak, ...

3.3.3. Information Hiding with Interfaces

3.4. Code Reuse

3.4.1. Multiple Inheritance

3.4.2. Mixins

Ruby Modules, Python Multiple Inheritance, Newspeak, Jigsaw

3.4.3. Traits

Squeak implementation

4. Nested Class Modularity in Squeak

In this chapter, we describe the main concept of our work: classes as class members. Similar concepts are part of programming languages like Java, Ruby, Python, and Newspeak. Our concept follows closely the Newspeak notion of nested classes, but without making invasive changes to the Smalltalk programming language.

4.1. Nested Classes

In Smalltalk, every object is an instance of a class, defining the object's instance variables and the messages it understands. Consequently, a class is also an instance of its so-called meta class. Every meta class is an instance of `Metaclass`. In the remainder of this work, we denote the meta class of a class `C` by `C class`. Every Smalltalk image has a `globals` dictionary, mapping symbols to class objects, so that references to classes can be resolved at compile time. This implies that all references to classes are early bound.

Our system extends the Smalltalk class organization as follows: in addition to regular methods, we introduce the concept of *class generator methods*. Such a method generates a class and is associated with a set I of instance methods and a set C of class methods. Whenever the method is invoked, the system first executes the method body, then adds I to resulting class and C to resulting meta class, and finally returns the resulting class. For performance reasons, our system also caches the result, meaning that a class is not generated twice.

Details Class generator methods are only allowed as class-side methods. Instance-side class generator methods seem to provide neglectable benefits and make the implementation of our system more complicated. We provide an in-depth explanation of instance-side class generator methods in the Section 7.1.

A class generated by a class generator method is anonymous: it is not listed in the `globals` dictionary and can only be referenced using message sends to its enclosing class[1]. Consequently, its name is a concatenation of all class names on the path from the top-level class to class in question.

Notation and Example Figure 4.1 shows an example of nested classes in Squeak. `A` is a top-level class, i.e., it is part of the `globals` dictionary and known everywhere in the system; it can be referenced by just writing the identifier `A`. `A` has one instance

[1] It can also be references by sending the `class` method to one of its instances

method `m2` and two class methods `m1` and `B`. In accordance with UML notation, class-side method selectors are underlined.

`A class»B` is a class generator method that is associated with a set of instance methods {} and a set of class methods {`foo`, `B`}. The name of the class it generates is `A B`, which is in that case also a valid Smalltalk code expression that evaluates to the generated class. `A class»B class»C` is a class generator method that generates `A B C`. Note, that we use the » notation to not only reference methods but also the classes they generate, in case they are class generator methods.

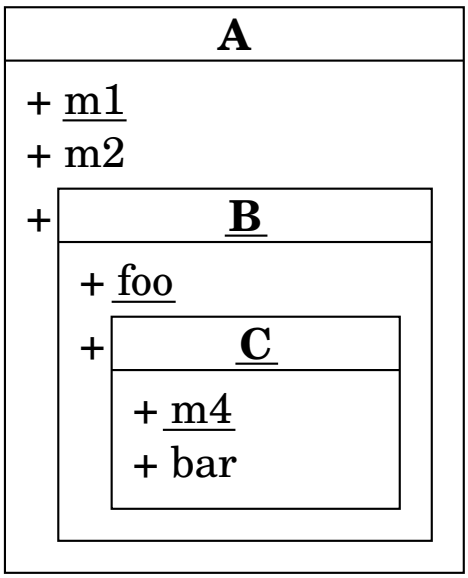

Figure 4.1.: Nested Classes Example

4.2. Accessing the Lexical Scope

It is sometimes necessary to access a method's lexical scope (i.e., the enclosing classes), in order to send messages to enclosing classes. For this reason, our system introduces new keywords, in addition to `self` and `super`, which are already present in every Smalltalk dialect. Figure 4.2 gives an overview of all method lookup-related keywords in the system.

`self` Keyword This keyword is used make a message send within an object. The receiver is the same object as the sender and the lookup starts at the (polymorphic) class of the receiver. If that class does not provide a corresponding method, the lookup continues in the superclass hierarchy. If no class in the superclass has a corresponding method, a `MethodNotUnderstood` error is raised.

`super` Keyword This keyword is also used to make a message send within an object. Again, the receiver is the same object as the sender, but the lookup starts at

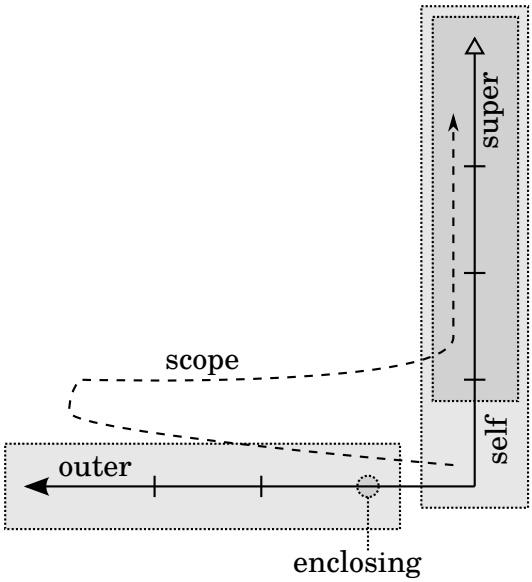

Figure 4.2.: Keywords for access to superclass and lexical scope.

superclass of the sender's method class. Note, that `super` is bound to the superclass of the method class, not the superclass of the receiver's class.

`enclosing` Keyword This keyword is used to make a message send to the class that contains the current current. Consider, for example, that we want to send a message `foo` to class `A B` within `A class»B class»C»m5` in Figure 4.1. Either one of the following two statements works in this case.

- `A B foo.`
- `enclosing foo.`

`enclosing` is a keyword that evaluates to the method owner's enclosing class upon method compilation. Note, that `enclosing` is bound to the method's lexical scope, not the receiver's lexical scope.

Figure 4.3 illustrates how `enclosing` is bound. In `B1 class»C class»bar1`, `enclosing` is bound to `B1`. In contrast, `B2 class»C class»bar2` binds `enclosing` to `B2`. Consequently, `B1 C bar1` calls `B1 foo` and so does `B2 C bar1`, even though the receiver of `bar1` is an instance of `B2 C class` and not `B1 C class` in the latter case. Note, that `B2 C bar2` calls `B2 foo`, because `bar2`'s lexically enclosing class is `B2`.

Note, that `enclosing` can be used for meta programming purposes; however, it should be avoided in general. Our system also provides a `scope` keyword that should be used instead.

`enclosing` Method In addition to `enclosing`, every class in the system has a method `enclosing` that returns the enclosing class of the receiver[2], making it possible to send messages to enclosing classes which are more than one level away.

[2]The enclosing class of an object that is not a class is its class' enclosing class.

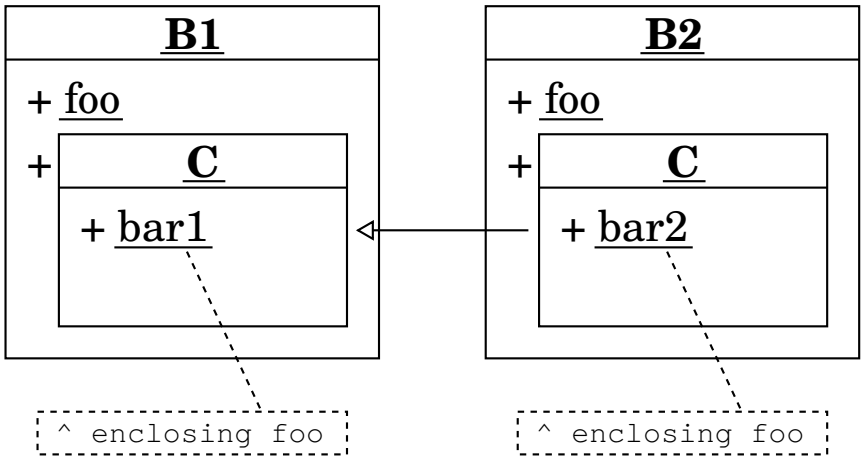

Figure 4.3.: Binding of `enclosing` to method's lexical scope.

If, for example, in Figure 4.1, `A class»B class»C»bar` wants to send the message `m1` to `A`, either one of the following two statements works.

- `A m1.`
- `enclosing enclosing m1.`

Again, the method `enclosing` should be avoided in general, but is useful to implement parts of our system with code written in the system itself.

`outer` Keyword The method `enclosing` can be used to traverse the the lexical scope of a class. Arbitrarily many `enclosing` sends can be chained, as long as the respective receiver still has an enclosing class and is, therefore, not a top-level class. Arguably, this can result in verbose and complicated code, and is at the very least questionable with regards to the law of demeter.

In addition to `enclosing`, the system provides the `outer` keyword, bound to the method's lexical scope. Whenever a message is sent to `outer`, the message is first interpreted as a send to `enclosing`. If that message send fails, the message is sent to `enclosing enclosing`, and, eventually, to the top-level class, if no other class in the lexical scope understands the message. If even that message send is not understood, the selector is looked up in the `globals` dictionary. If the selector is absent, a `MessageNotUnderstood` error is raised.

`outer` is similar to `super`, with the difference that `outer` does a horizontal lookup (lexical scope) and `super` does a vertical lookup (superclass chain). Note, that messages sent to `outer` are sent to an object different from `self`.

`scope` Keyword This keyword combines `super` and `outer`: a message sent to `scope` is first treated as a `self` send. If the message is not understood, it is treated as an `outer` send.

Our system essentially first looks up the methods in `self`, then in the superclass hierarchy, and then in the lexical scope. This is also how the method lookup in

erence:
W.R.: New-
t: Pro-
on the
p. 109 –
ingerVer-
9), in
pe-Based
mming:
ts, Lan-
and Ap-
ns, Noble,
ari and
editors

Java works, also known as *comb semantics*. Newspeak uses a different lookup: it first looks for a method in the receiver's class, then in the lexical scope, and finally in superclass hierarchy [1].

The statement `enclosing enclosing m1` in the previous example can also be written as `scope m1`. If the method `m1` would now be moved to its enclosing class (if it had one), the lookup would still succeed. However, `scope` exposes the risk of accidentially capturing method names in superclasses or the lexical chain.

Implicit `scope` Receiver In our system, references to globals are in fact message sends with `scope` as implicit receiver. This makes it easier for Smalltalk programmers to write code in our system, even if they do not know about `enclosing` and `scope`. It also makes the code less verbose and easier to read.

Whenever code references an identifier that is not a temporary variable, not an instance variable, and not a *special* object [3], the compiler replaces that identifier with a message send to `scope`.

Consider, for example, that we want to reference class `A B` within `A class»B class»C»bar` in Figure 4.1. Either one of the following two statements works in this case.

- `A B.`
- `enclosing.`
- `enclosing enclosing B.`
- `outer B.`
- `scope B.`
- `B.`

In this example, we used the implicit `scope` receiver for class lookup, which is in our opinion the most useful case. However, any method in `self`, the lexical scope, or the superclass hierarchy can in fact be looked up this way. One can argue that this is bad practice and should be forbidden for methods that are not class generator methods. However, it is allowed in Newspeak and other programming languages like Java, and seems to work well, as long as the programmer is aware of how the method lookup works. Note, that only unary messages can have an implicit `scope` receiver, since we would have to change the Smalltalk syntax otherwise.

4.3. Parameterized Classes

All examples shown in the previous section use unparameterized classes, i.e., class generator methods are always unary. Class generator method can, however, also have binary selectors or selectors with a higher arity. For memory conservation reasons, these classes are then no longer cached.

[3] `self`, `super`, `thisContext`, `scope`, `outer`, `enclosing`

Mixins Parameterized classes can be used to build mixins. Mixins are not a special feature of this system: they are an application of our system and come for free by just having class nesting as described in the previous sections; they are an immediate consequence of parameterized classes. A mixin is a function that takes as an input a class and outputs a subclass with additional behavior, i.e., it is a class transformator.

A mixin can be implemented by writing a class generator method with one parameter which is the input class. The method creates a subclass of that input class and returns it. Associated with that parameterized class generator methods is a set of instance-side methods and a set of class-side methods. These are the methods that will be added when applying the mixin.

Recursive Mixin Application A mixin can make sure another mixin is applied upon its application. This is done creating a subclass of a mixin application in the class generator method. Consequently, the system first creates a subclass of the base class, adds the methods of the inner mixin, then creates a subclass of the resulting class, and finally adds the methods of the outer mixin.

Example Figure 4.4 shows an example of parameterized classes and how they can be used to build mixins.

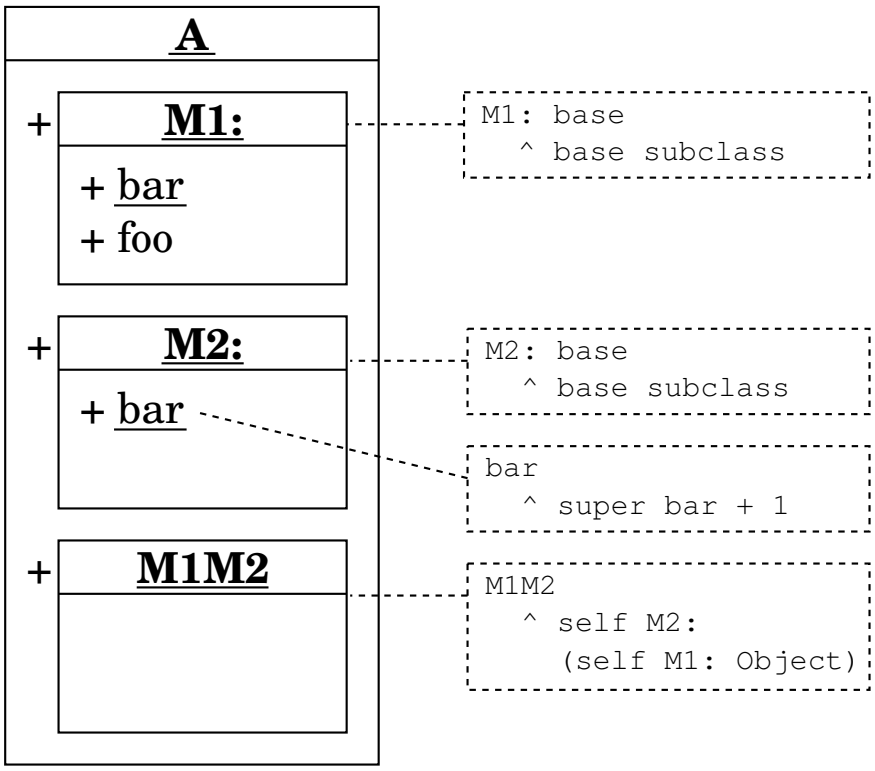

Figure 4.4.: Implementation of Mixins with Nested Classes

Two class generator methods `A M1:` and `A M2:` are defined, which take as input a base class and output a subclass with additional behavior. `A M1M2` is an application of both both mixins. `A M1M2`'s superclass is *some* `A M2:`, whose superclass is *some* `A M1:`, whose superclass is `Object`. Note, that `A M1:` and `A M2:` are not specific classes: we use this notation as a name for *some* application of `A class»M1:` and `A class»M2:`, respectively. Therefore, even if two classes have the same name, they are not necessarily the same class if they names contain a colon.

Note, that evaluating `A M1: Object` multiple times returns different class object, since parameterized classes are not cached. However, `A M1M2` is cached, because it is a unary method. Therefore, calling `A M1M2` multiple times always returns the same class object.

The notation used in `A class»M1M2` can be a bit confusing at first. That method first applies `A M1:` to `Object`, and then `A M2:`; however, in the source code, `A M2:` appears before `A M1:`. For readability reasons, and to support more features like pre-include hooks and post-include hooks, we present the Class Generator Pattern in Section 6.6.

5. Implementation

5.1. Meta Model and Instantiation

accessor methods and generator methods, lazy initialization

5.2. Anonymous Classes and Subclass Generation

5.3. `thisOuter` and `thisScope`

5.4. Class Updates

using instances weak array on specification

5.5. Integration in Squeak

5.5.1. Module Repository

replacement for Smalltalk dict

5.5.2. IDE Support

works in workspace, test runner. How to write tests? New system browser

5.5.3. Debugger

shows slightly different code (thisContext automatically inserted, generator methods)

6. Use Cases

6.1. Avoiding Clasds Name Clashes

example: multiple games, all providing game class

6.2. Module Versioning

example: multiple versions of the same module in the same image

6.3. Dependency Management

example: app that requires a specific version (defining local alias), then all access goes through that method. example: external configuration of modules with parameterized classes (alternative to dependency injection)

6.4. Readability and Understandability

example: large project, where parts of the code can now be understood, given that we have hierarchical nesting. could be an example where grouping according to multiple criteria is needed (would result in n x m packages)

6.5. Mixin Modularity with Parameterized Classes

6.6. Class Generator Pattern

better syntax for class generators

6.7. Extension Methods

better way is needed (e.g., class boxes, refinements, COP). return already existing class in generator method

7. Future Work

7.1. Class as Instance-side Members

7.2. Bytecode Transformation instead of Recompilation

7.3. Squeak Integration

Bibliography

[1] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. "Modules as Objects in Newspeak". English. In: *ECOOP 2010 – Object-Oriented Programming*. Ed. by Theo D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 405–428. ISBN: 978-3-642-14106-5. DOI: 10.1007/978-3-642-14107-2_20.

Appendix A.

First Unimportant stuff.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Potsdam, den 17. Juli 2015

Matthias Springer

Todo list

- Add reference: Smith, W.R.: NewtonScript: Prototypes on the Palm, pp. 109 – 139. SpringerVerlag (1999), in Prototype-Based Programming: Concepts, Languages and Applications, Noble, Taivalsaari and Moore, editors 11