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 19, 2015

Abstract

The english abstract.

Zusammenfassung

Die Zusammenfassung auf deutsch.

Acknowledgments

I owe everything to my cat.

Contents

1. Introduction **1**
1.1. Modularity . . . 1
1.2. The Squeak Programming Language . . . 1
1.3. Outline of this Work . . . 1

2. Modularity Problems in Squeak **3**
2.1. Class Name Clashes . . . 3
2.2. Dependency Managment . . . 3
2.3. Readability and Understandability . . . 3
2.4. Code Reuse . . . 3

3. Related Work **5**
3.1. Class Name Clashes . . . 5
3.1.1. Namespaces/Packages . . . 5
3.1.2. Squeak Environments . . . 7
3.1.3. Newspeak Modules . . . 7
3.2. Dependency Management . . . 7
3.2.1. Java Class Loader . . . 7
3.2.2. Separate Compilation . . . 7
3.2.3. External Configuration in Newspeak . . . 7
3.3. Readability and Understandability . . . 7
3.3.1. Smalltalk Packages . . . 7
3.3.2. Hierarchical Decomposition . . . 7
3.3.3. Information Hiding with Interfaces . . . 7
3.4. Code Reuse . . . 7
3.4.1. Multiple Inheritance . . . 7
3.4.2. Mixins . . . 7
3.4.3. Traits . . . 7

4. Nested Class Modularity in Squeak **9**
4.1. Nested Classes . . . 9
4.2. Accessing the Lexical Scope . . . 10
4.3. Parameterized Classes . . . 13

5. Implementation **17**
5.1. Meta Model and Instantiation . . . 17
5.2. Anonymous Classes and Subclass Generation . . . 21
5.3. Implementation of Keywords . . . 22

5.4. Class Caching 24
5.5. Class Updates 26
5.6. Integration in Squeak 26
5.6.1. Module Repository 26
5.6.2. IDE Support 26
5.6.3. Debugger 26

6. Use Cases **27**
6.1. Avoiding Clasds Name Clashes 27
6.2. Module Versioning 27
6.3. Dependency Management 27
6.4. Readability and Understandability 27
6.5. Mixin Modularity with Parameterized Classes 27
6.6. Class Generator Pattern 27
6.7. Extension Methods 27

7. Future Work **29**
7.1. Class as Instance-side Members 29
7.2. Bytecode Transformation instead of Recompilation 29
7.3. Adding Instance Variables 29
7.4. Squeak Integration 29

A. First Unimportant stuff. **33**

List of Figures

4.1. Nested Classes Example 10
4.2. Keywords for access to superclass and lexical scope. 11
4.3. Binding of `enclosing` to method's lexical scope. 12
4.4. Implementation of Mixins with Nested Classes 14

5.1. Squeak Class Model with Meta Classes 17
5.2. Meta Model for Nested Classes 19
5.3. Lazy class generation and initialization 20
5.4. Subclass notation in Squeak 21
5.5. Subclass notation with nested classes 22
5.6. `LexicalScope` for `outer` keyword example 23
5.7. `LexicalScope` for `scope` keyword example 24
5.8. Class Cache stored in `ClassSpecification` 25
5.9. Cached Mixin Application Example 25

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

Many programming languages have a concept of namespaces or packages. Classes are typically organized in a package, which is a set of classes. Classes within a package can usually reference each other directly. However, references to classes in other packages typically require imports, aliases, or a fully qualified name.

VisualWorks Namespaces VisualWorks is a commerical Smalltalk implementation sold by Cincom and supports namespaces [4]. A namespace is a container for other namespaces, classes, and shared variables. Since a namespace can be defined within another namespace, VisualWorks allows for a form of hierarchical decomposition. All namespace members (e.g., classes) in the same namespace can be referenced by just writing down their names. All namespace members in other namespaces can be referenced by writing down their fully qualified name, which is the concatenation of all nested namespace names and the name of the class with dots as separators. For example, the fully qualified name of a class `C1` in namespace `B` in namespace `A` is `A.B.C1`. Relatives name are also supported: for example, `A.B.C1` can be referenced as `B.C1` within `A`.

A namespace can import members from other namespaces by specifying a list of all imports when the namespace is defined [7]. Wildcard imports are possible, importing all members of a namespace. Imported members can be referenced within a namespace as if they were part of that namespace. A namespace member can also be defined as *private*; such a member cannot be imported, but always has to be referenced using its fully qualified name or using a relative name.

Namespaces are instances of the class `NameSpace`, which is a subclass of `Collection`. `NameSpace` defines a few helper methods to allow for meta programming, such as listing all classes or defining new namespaces or classes within a namespace.

Java Packages and Nested Classes The Java programming language has a concept of packages. A package is set of classes, interfaces, and pacckages, and corresponds to a directory on the file system. Classes and interfaces in the same package can be referenced directly using their name. Classes and interfaces in other packages can be referenced using their fully qualified name, which is generated exactly as in VisualWorks. They can also be imported explicitly, making it possible to reference them just using their name; wildcard imports are possible.

Classes and interfaces can be defined as package-public or package-private. Only package-public members can be imported or referenced within members outside of the current package.

Java supports the concept of nested classes: a class can either be a top-level class or a class that is nested within another member. There are four different kinds of nested classes [1].

- *Static member class:* a class that belongs to another class, i.e., it is a static member of another class. It can be accessed like a static variable of the enclosing class. For example, if `B` is nested in `A`, it can be referenced with `A.B`. Messages sent from within the nested class are first looked up in the nested class and its superclass hierarchy, then on the class side of the enclosing class (static methods), and then in the enclosing class' enclosing class (if it is a nested class).
- *Nonstatic member class:* a class that belongs to an instance of another class, i.e., it is a nonstatic member of another class. It is similar to a static member class, but the method lookup happens on the instance side of the enclosing class. Every instance of a class has its own nonstatic member classes; however, all of these classes must inherit from a class that can be resolved at compile time. Effectively, all nonstatic member classes are the same, with the only exception that they are bound to different enclosing objects.
- *Anonymous class:* a class without a name. In older Java versions, it was frequently used as a substitute for missing block closures. Lambda expressions are available since Java 8, making anonymous classes obsolete in many use cases. Note, that since classes are not first-class objects in Java, it is difficult to pass anonymous classes around and to use them in a different context without using meta programming.
- *Local class:* a class that can be defined anywhere where a local variable can be defined. It is the least frequently used kind of classes.

Ruby Modules 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` (Figure 5.1). 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.

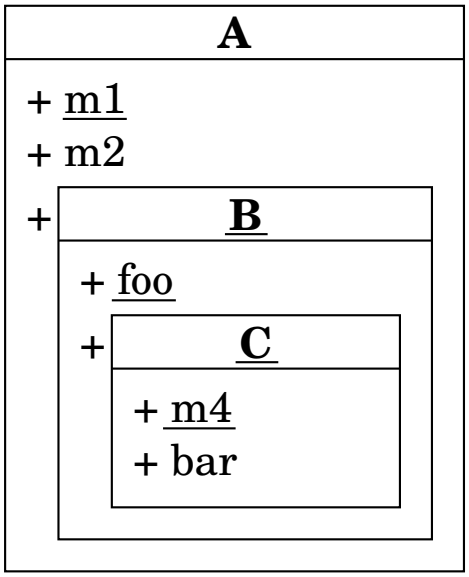

Figure 4.1.: Nested Classes Example

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

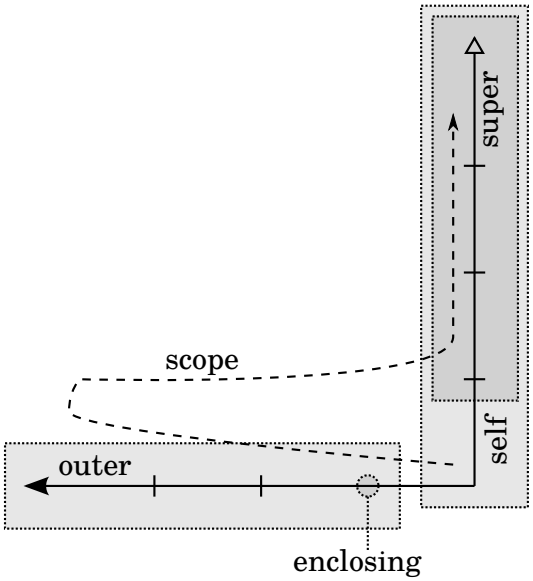

Figure 4.2.: Keywords for access to superclass and lexical scope.

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.

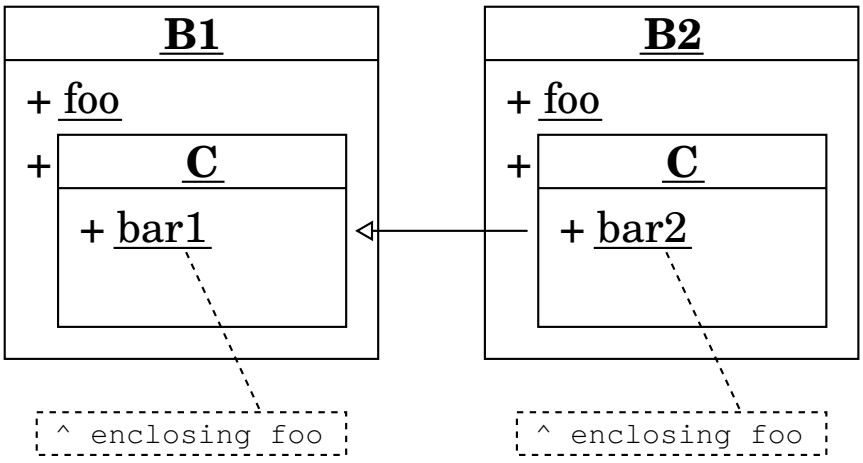

Figure 4.3.: Binding of `enclosing` to method's lexical scope.

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

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 [3].

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

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.

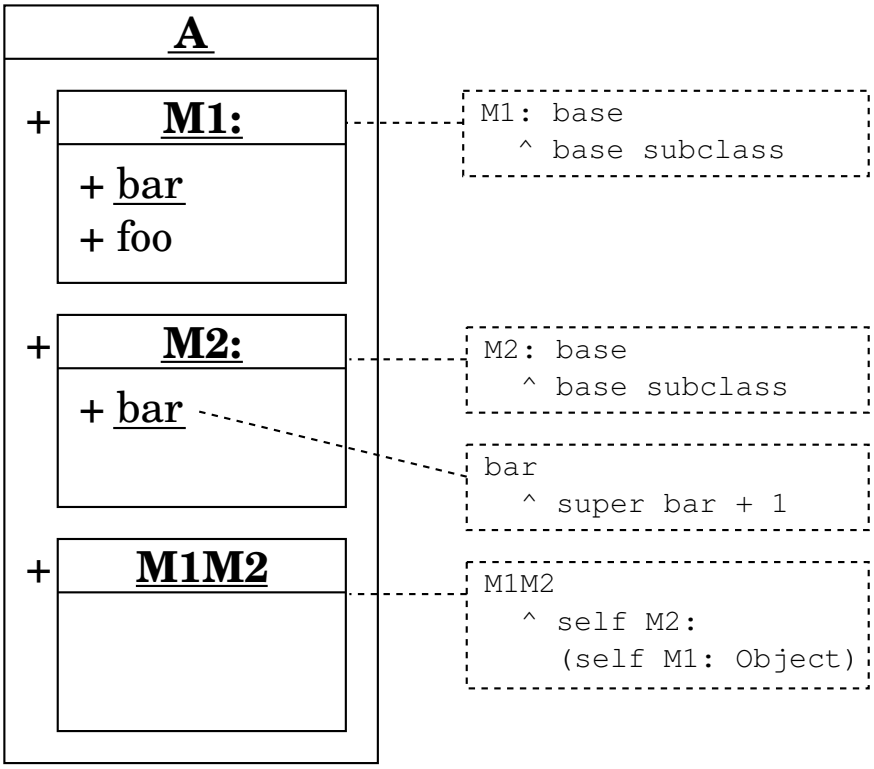

Figure 4.4.: Implementation of Mixins with Nested Classes

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

Our system has a simple meta model for describing (nested) classes and their methods. The graphical user interface operates exclusively on the meta model and makes changes to it. The meta model can then be instantiated to generate the actual classes. When changes to the meta model are made, these changes can also be applied to already existing instantiations of the model, allowing giving programmers the feeling of working with a live system.

Smalltalk-80 Class/Meta Model Squeak already comes with a meta model: objects are instances of a classes, consequently, classes are also instances of a class. In Smalltalk, every class is an instance of its own meta class, which is in turn instance of `Metaclass` (Figure 5.1).

Our system allows class generation at runtime: class generator methods generate classes along with their respective meta classes. Therefore, we need a specification/blueprint that describes how a class generator method should construct a class. At first glance, it might seem logical to use meta classes; after all, a meta class is the class of a regular (non-meta) class and classes are instance generators. However, meta classes cannot be used as class object generators in a way required by our system for two reasons.

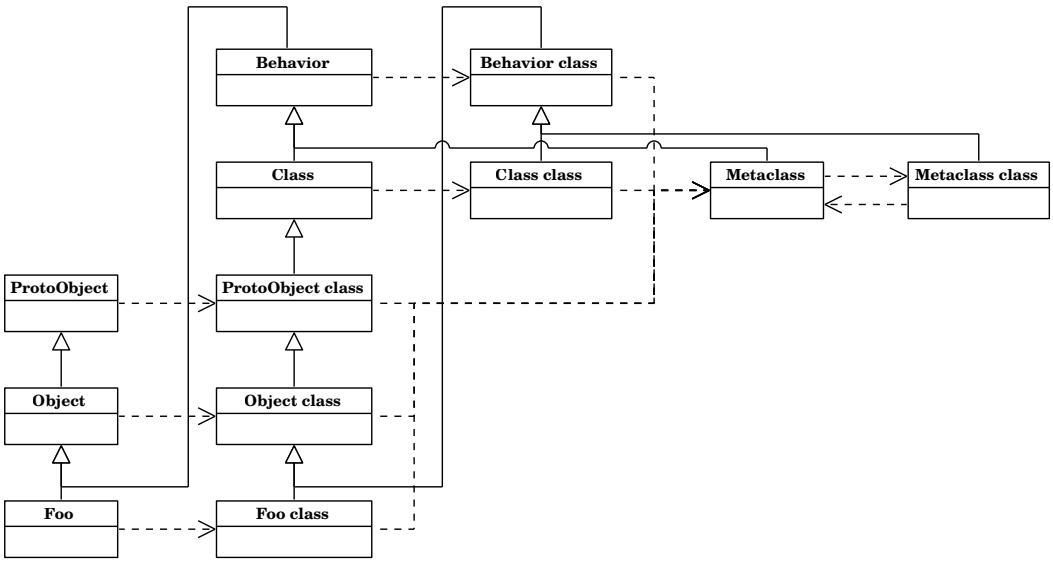

Figure 5.1.: Squeak Class Model with Meta Classes

Firstly, meta classes do not have any information about their non-meta class counterpart: for example, they do not know anything about their instance methods or their instance variables. Instantiating a meta class would not generate a functional class object, which is why Smalltalk prohibits generating new instances of a meta class. In fact, the class `ClassBuilder` is used to create new classes and it always creates class objects alongs with their meta class objects.

Secondly, our system supports defining methods on the instance side and on the class side. Consequently, we do not only need to generate class object but also meta class objects. All meta classes are an instance of `Metaclass`. But if we wanted to generate different meta classes, we would need a different `Metaclass` class, each of which generates its corresponding meta class. In some programming languages, the instance-of chain carries on infinitely; Ruby is an example. However, in Smalltalk, every meta class is an instance of `Metaclass` and this is where the instance-of chain recurses: `Metaclass` is an instance of `Metaclass class`, which is an instance of `Metaclass`.

For this reason, we cannot use the Smalltalk-80 meta model to generate new classes on the fly and use our own simple meta model instead.

Nested Classes Meta Model Figure 5.2 shows the meta model in our system. The meta model is built around specifications: there are specifications for classes, meta classes, and methods. A specification describes how its corresponding object is built. `ClassSpecifications` generate classes, `MetaclassSpecifications` generate meta classes, and `MethodSpecifications` generate methods. Since classes cannot exist without their respective meta classes, a class specification is always linked with its meta class specification and vice-versa. When a class specification is instantiated, the system generates both the class and the meta class. Meta class specifications cannot be instantiated.

Class Specifications A class specification describes classes. It has a collection of `MethodSpecifications`, representing instance methods of the class. Upon instantiation, all method specifications are instantiated within the target class. For every class specification, there is a corresponding method specification containing the source code of the class generator method in the parent's method dictionary. This method specification determines (when executed in the running system) to which class the methods will be added (*target class*). Top-level classes are an exception: they are always a new subclass of the class `Module`.

Meta Class Specification A meta class specification describes meta classes. It has a collection of `MethodSpecifications`, representing class methods of the class (i.e., instance methods of the meta class). Upon instantiation, all method specifications are instantiated within the targer class' meta class. Consequently, meta classes do not method specifications associated with.

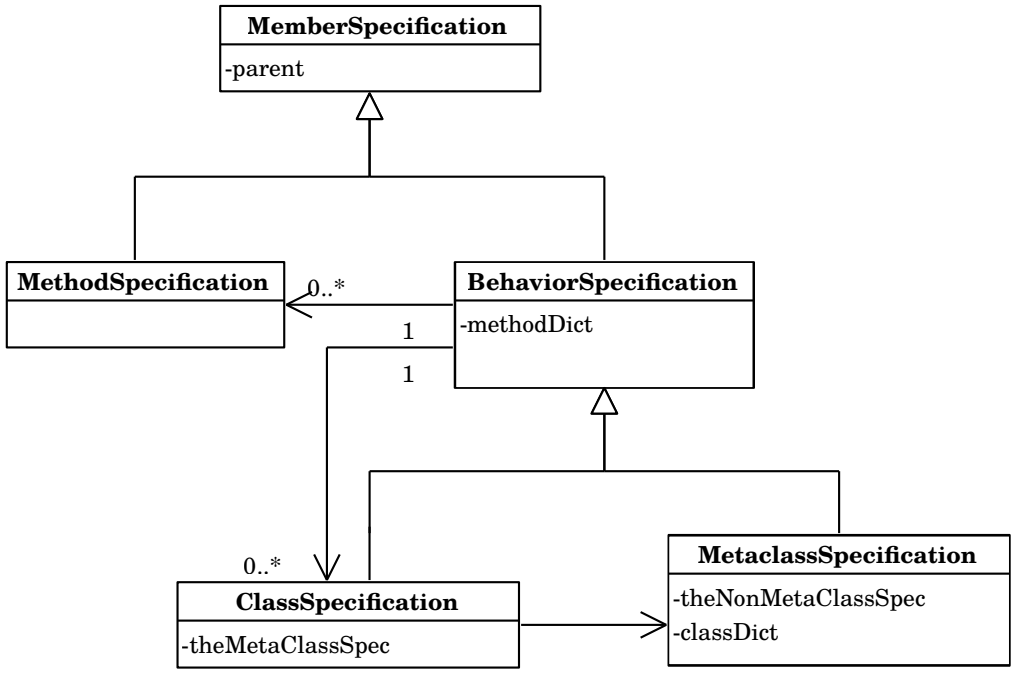

Figure 5.2.: Meta Model for Nested Classes

However, meta classes can have nested classes of their own. For every class defined in a meta class, there is a corresponding method specification present in the method dictionary (see previous paragraph).

Method Specification A method specification describes methods. It contains the source code of the method and stores information necessary for class caching and UI metadata. Whenever a method specification is instantiated, the method source code is compiled in the target class.

Note, that different byte code must be generated for different target classes: for example, instance variable reads and write are compiled to parameterized[1] `pushRcvr:` and `popIntoRcvr:` bytecodes, where instance variables are referenced with their index[2]. In addition, the `outer` and the `enclosing` keyword must be bound to different method literals, depending on the lexical scope of the class.

Class Initialization Figure 5.3 illustrates how the system generates and initializes a nested class (class specification instantiation).

Whenever a class accessor method is invoked, the method first checks if the class is already cached. If that is the case, it is returned. Otherwise, the class generator method called, returning an empty uninitialized class, i.e., all instance methods are still missing and only the superclass and the instance and class variables are set up correctly. The following list gives an overview of the steps necessary for initializing a class.

[1] There are separate bytecodes for reading the first or second instance variable etc.
[2] The first instance variable has index 0, second index variable has index 1, etc.

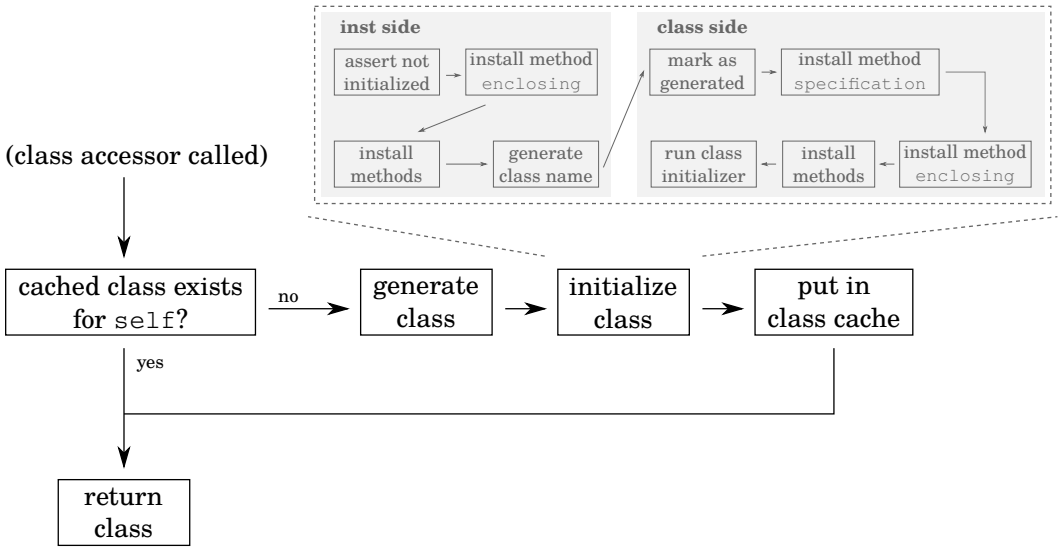

Figure 5.3.: Lazy class generation and initialization

1. Install `enclosing` instance method. This method returns the enclosing class.
2. Install/compile all instance methods listed in the class specification.
3. Generate the class name. The class name is a concatenation of the enclosing class' name and the selector of this class' accessor method. It is stored as an instance variable at `Class`. Note, that every class object is an instance of its meta class, which is a subclass of `Class` (Figure 5.1).
4. Add a marker method to the meta class to mark it as generated. This makes is easy to check if a class is an ordinary (legacy) Smalltalk class or was generated within our system.
5. Install `specification` class method. This method returns the class specification, which is useful for meta programming purposes.
6. Install `enclosing` class method. This method is identical to the instance method.
7. Install/compile all class methods listed in the meta class specification.
8. Send `initialize` to the class object.

Note, that class initialization is lazy. A class is only generated and initialized if the corresponding accessor method was called. All references to classes in the source code actually call the accessor method, making sure that the class is available when it is needed.

Class generator methods can return subclasses of other classes; the superclass is referenced by calling the accessor method. Compared to the default package-loading process in Squeak, this makes class creation easier. In Squeak, the system has to analyze which classes are subclasses of each other, in order to create classes in the correct order (superclass has to exist before subclass is created). In our system, classes are created when their accessor method is called, and if these classes depend on another superclass, that superclass is created when the class generator method calls its accessor method (if it does not already exist).

Class Accessor Methods and Class Generator Methods For a nested class, two methods are installed on the meta class object: a class generator method, returning the class to which methods should be added (usually a newly-created subclass), and a class accessor method, checking whether the class was already created and is in the cache or calling the class generator method, otherwise.

The selector for the class accessor method is the name of the class. The selector for the class generator method is the same selector, but with a dollar sign prefix. This ensures that the method can only be called by using meta programming from our system, and also avoids accidential name clashes with other methods. For example, if a class is named `Foo`, the class accessor method has the selector `Foo` and the class generator method has the selector `$Foo`.

5.2. Anonymous Classes and Subclass Generation

In Smalltalk, new classes are created by subclassing an already existing class. Squeak has special class, the `ClassBuilder`, containing all the functionality for creating the class object, the meta class object, giving the class a name, possibly migrating the old class and its instances (if an existing class was changed), and registering it in the `globals` dictionary.

Our system reuses the class builder and adds functionality for creating anonymous subclasses. Anonymous subclasses do not have a name and certain checks are omitted (e.g., if the class name starts with a capital letter). Also, anonymous subclasses are not added to the `globals` dictionary.

Subclass Notation Figure 5.4 shows how subclasses are created in Squeak. The first statement is a message send to `Object` which not only creates the subclass but also adds it to the `globals` dictionary. The second statement is also executable code that adds an instance variable to the meta class object. The difference between class variables and class instance variables is that class variables are shared among all subclasses, whereas class instance variables have different values for every class object [6, 5]. For example, if `A` has a class variable `Bar` and `B` is a subclass of `A`, then both `A` and `B` share one variable `Bar`.

```
Object subclass: #NewClass
   instanceVariableNames: 'foo bar'
   classVariableNames: 'Bar'
   poolDictionaries: ''
   category: 'Demo-Experiments'.

NewClass class
  instanceVariableNames: 'Foo'.
```

Figure 5.4.: Subclass notation in Squeak

Figure 5.5 shows how subclasses are created in our system. `NewClass` is a class generator method and also the name of the new class. Therefore, it is no longer necessary to pass a symbol with the name of the new class to the `subclass:` method. Note, that the `<class>` pragma is necessary to distinguish between class generator methods and regular methods, which might accidentially return a class. Only in the former case, a class specification object is created.

```
NewClass
   < class >
   ^ Object
      subclassWithInstVars: 'foo bar'
      classVars: 'Bar'
      classInstVars: 'Foo'
```

Figure 5.5.: Subclass notation with nested classes

5.3. Implementation of Keywords

In this section, we explain how the keywords `enclosing`, `outer`, and `scope` are implemented. All message sends to `enclosing` are forwarded to the enclosing class. All message sends to `outer` are forwarded all enclosing classes consecutively, whenever a class does not understand the message. All message sends to `scope` are first treated as `self` sends, then as sends to `outer`.

Implementation of `enclosing` During compilation, all references to `enclosing` bound to the enclosing classes, which is known during class initialization. Technically, every class has its own Squeak environment which binds `enclosing` to the enclosing class. Therefore, it is also possible to evaluate `enclosing` in the debugger, for example.

Implementation of `outer` During compilation, all references to `outer` are bound to an instance of `LexicalScope`. This class is a subclass of `ProtoObject`, holds a reference to the enclosing class, and contains a `doesNotUnderstand:` handler, that forwards messages to the enclosing class. If the enclosing class does not understand the message, the message is forwarded to that class' enclosing class. If at some point, a top-level class without an enclosing class is reached, the handler looks for an entry in the `globals` dictionary with the message's selector.

As an example, let us assume that we have classes nested as shown in Figure 4.1 and that all following message sends to `outer` happen in `A class»B class»C class»m4`. See Figure 5.6 for a visualization of the lookup.

- `outer foo`: lookup in `enclosing` (class `A B`) succeeds.
- `outer B`: lookup in `enclosing` fails, but lookup in `enclosing enclosing` (class `A`) succeeds.

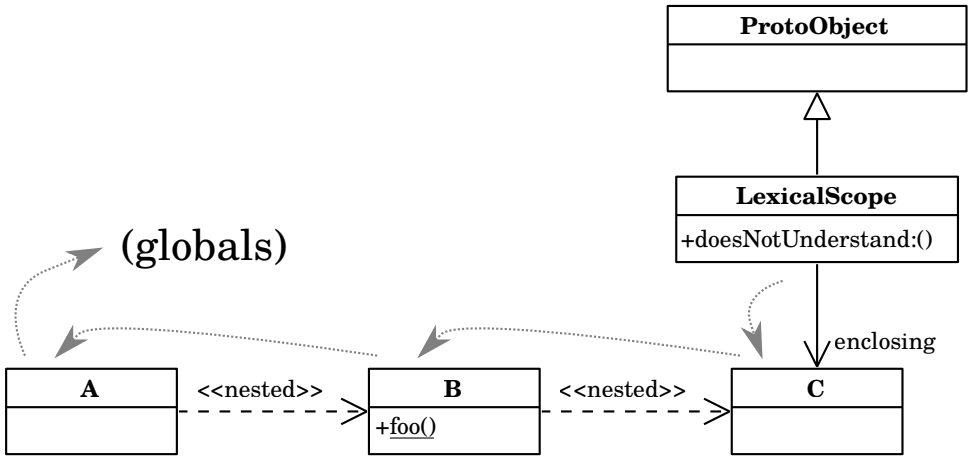

Figure 5.6.: `LexicalScope` for `outer` keyword example

- `outer A`: lookup in `enclosing` and `enclosing enclosing` fails, but `A` is present in the `globals` dictionary.
- `outer Object`: same as before. All classes outside of our system are also present in the `globals` dictionary.
- `outer D`: lookup fails and raises a `MessageNotUnderstood` error.

Implementation of `scope` References to `scope` cannot be replaced by a constant literal during compile time. This is because the lookup involves a lookup in `self` send. Looking up methods in the class of the method under compilation is not sufficient, because that method might be overridden in a subclass. Therefore, we have to construct a `LexicalScope` object at runtime (instead of compile time) and pass it two objects: the enclosing class and `self`.

Figure 5.7 shows how the `scope` lookup works in a slightly modified example. Just as in the previous example, we assume that all message sends happen in `A class»B class»C class»m4`. However, `m4` is invoked on class `D`, which is a subclass of class `A B C`. Therefore, `self` is bound to `D`.

- `scope bar`: lookup in `self` succeeds: method `D class»bar`.
- `scope foo`: lookup in `self` fails, but lookup in `enclosing` (class `A B`) succeeds.
- The lookup for the all examples listed for `outer` (previous paragraph) yields the same result here.

Note, that the reference to `self` (`target`) cannot be established at compile time, because it is unclear what the polymorphic receiver class is. Therefore, references to the keyword `scope` have to be replaced by a message send: `LexicalScope for: self in: enclosing`. This has the side effect that the decompiled source code (and the code shown in the debugger) looks slightly different from the code written by the programmer.

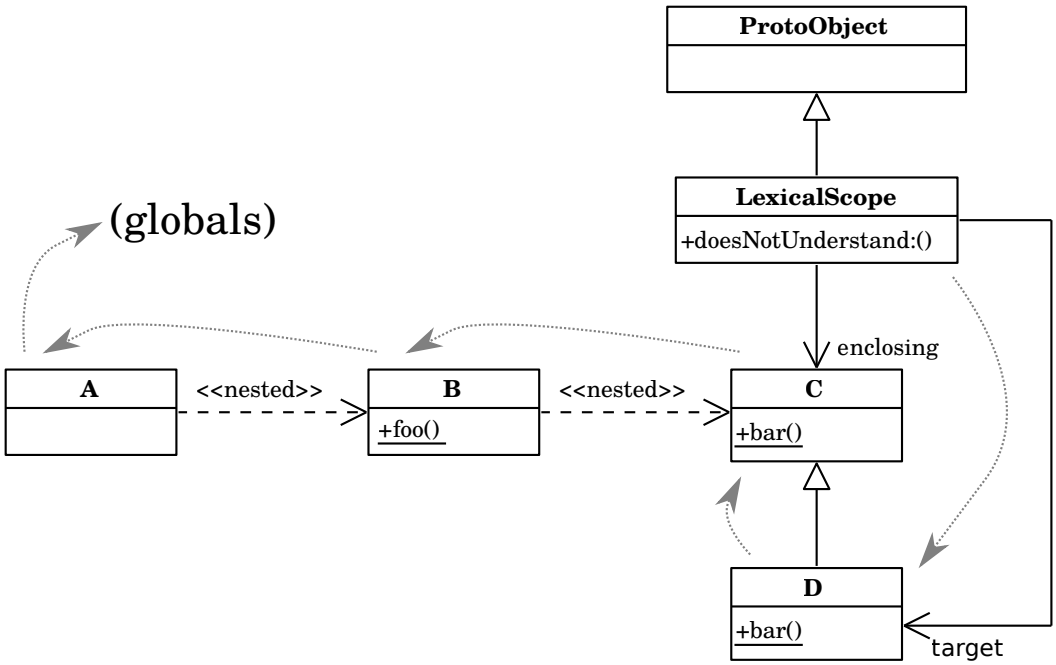

Figure 5.7.: `LexicalScope` for `scope` keyword example

5.4. Class Caching

Whenever a nested class is accessed, the class accessor method checks if the class was already generated. If that is the case, the cached version if the class is returned. For this reason, every class specification with a unary selector (unparameterized class) has an instance variable `classCache`, which contains the cached class object. The system does not cache parameterized classes, as this could result in an excessive number of classes being kept around.

One can argue, that a nested weak identity key dictionary data structure could solve this problem: `classCache` is a `WeakIdentityKeyDictionary`, whose keys are the first argument. The values are again `WeakIdentityKeyDictionarys`, mapping the second argument to `WeakIdentityKeyDictionarys`. Eventually, the last argument is mapped to class objects instead of dictionaries (Figure 5.8).

In this case, class objects are garbage collected once there is no reference to at least one of the arguments in the system anymore. However, it depends on how exactly parameterized classes are used. If parameterized classes are used heavily, for example with `SmallIntegers` as parameters, no class would ever be garbage collected, because `SmallIntegers` are represented as tagged objects in Squeak [2, 8]. If parameterized classes are used as mixins, this is arguably less of a problem, because the the number of base classes to which a mixin is applied is usually not excessively large. However, note, that mixin applications can easily be cached by aliasing them as an unparameterized class (Figure 5.9). We argue that mixins will most of the time be used in such a way, because writing the mixin application explicitly is more verbose and hinders readability; in addition, the programmer

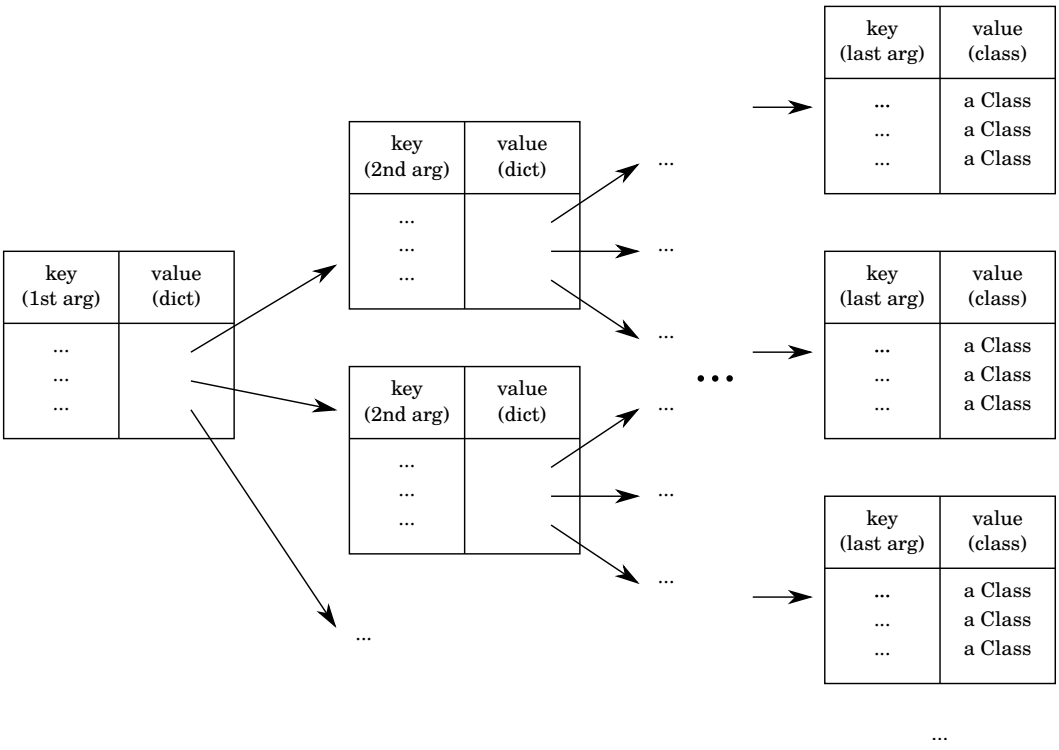

Figure 5.8.: Class Cache stored in `ClassSpecification`

might want to add additional methods to the mixin application, in which case the mixin application must be subclassed or aliased as described, anyway.

```
MyLibrary class>>BaseClass
  < class >
  " This is the class that serves as
    an input for the mixin in this example. "


MyLibrary class>>CollectionMixin: base
   < class >
   " This class is uncached because it is parameterized "

   ^ base subclass


MyLibrary class>>MyCollection
   < class >
   " This is the cached mixin application. "

   ^ self CollectionMixin: self BaseClass
```

Figure 5.9.: Cached Mixin Application Example

5.5. Class Updates

using instances weak array on specification

5.6. Integration in Squeak

5.6.1. Module Repository

replacement for Smalltalk dict

5.6.2. IDE Support

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

5.6.3. 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. Adding Instance Variables

7.4. Squeak Integration

Bibliography

[1] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0321356683, 9780321356680.

[2] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. "Self-Sustaining Systems". In: ed. by Robert Hirschfeld and Kim Rose. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Back to the Future in One Week – Implementing a Smalltalk VM in PyPy, pp. 123–139. ISBN: 978-3-540-89274-8. DOI: 10.1007/978-3-540-89275-5_7.

[3] 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.

[4] Johannes Brauer. *Programming Smalltalk–Object-Orientation from the Beginning: An introduction to the principles of programming*. Springer, 2015.

[5] Juanita J. Ewing. *Class Instance Variables for Smalltalk/V*. 1994.

[6] Juanita J. Ewing. *How to Use Class Variables and Class Instance Variables*. 1994.

[7] Cincom Systems Inc. *Cincom Smalltalk – Application Developer's Guide*. 2009.

[8] Tobias Pape, Arian Treffer, Robert Hirschfeld, and Michael Haupt. *Extending a Java Virtual Machine to Dynamic Object-oriented Languages*. Tech. rep. 2013.

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 19. 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 13