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 20, 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

Design Principles Behind Smalltalk

## 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 and Class Nesting

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. Some programming languages also support class nesting, where the enclosing class creates a namespace for all inner/nested classes.

**VisualWorks Namespaces** VisualWorks is a commerical Smalltalk implementation sold by Cincom and supports namespaces [0]. 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 [0]. 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 [0].

- *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.

Static member classes are similar to packages. By just looking at source code that references a static member class, it is not obvious whether the class is statically nested or contained in a package.

Java imposes certain restrictions on member classes. For example, nonstatic member classes are not allowed to have static member which are not final [0]. Furthermore, a subclass cannot override a member class definition [0]; it can just define its own member class. The difference is that overriding implies late binding, which is not the case in Java. With *Jx*, Nystrom et al. changed the Java language is such a way, that subclasses can enhance member classes [0]: the new member class overrides the original one and is always a subclass and a subtype of the member class in the superclass. Jx also allows changing the superclass of a member class in a subclass of the enclosing class, a form of mixin modularity.

**Ruby Modules** Ruby has the concept of classes and modules. Modules are classes which are not instantiable. They can be included in classes and be used as mixins.

Modules and classes can be nested in each other, defining a namespace. Classes and modules can be accessed using their fully qualified name, which is the concatenation of their names with two colons as separator. For example, if class `B` is nested in class `A`, `B`'s fully qualified named is `A::B`. Classes and modules can also be accessed using relative names. For example, when accessing `A::B`, Ruby first looks for `A` in the current class/module. If there no such member, it looks in the enclosing class/module.

In Ruby, a class can have methods, variables, and constants. An inner class or module is just a constant defined on the enclosing class. Constants are copied or shared during subclassing. Subclasses can replace inner classes with their own implementation. A nested class/module is always a class-side member of their enclosing class/module (nonstatic member class in Java).

In Ruby, classes and modules can be extended after they have been defined. In case of an accidential class/module name clash, the two (or more) classes/modules are effectively merged. In case of colliding methods, the method that was last seen (read from the file) overwrites all previous definitions. This process is often used deliberately in Ruby, in order change the behavior of a library or application, e.g., to fix a known bug (*monkey patching*) [0].

**Python Modules** In Python, every source code file is a module. Modules have to be imported, before they can be used within another module. Members defined in a module can be referenced by concatenating the module name and the name of the member (e.g., class or function) inside the module with a dot as a separator, if the module is imported. It is also possible to import single members from a module with their own name or an alias. These members can be accessed without writing down the module name.

In Python, every directory with a `__init__.py` source code file is a package. Packages can contain other packages and modules. Packages can be imported just like modules. The fully qualified name of a module is the concatenation of all package names and the module name, with a dot as a separator.

Modules in other packages can be imported by writing their fully qualified name or using a path relative to the current module [0].

Python supports inner classes, but only for readability and understandability reasons, and their usage is not wide-spread. Inner classes are class-side members of the enclosing class. In fact, for every inner class, Python creates an attribute on the enclosing class object with the inner class name as name and the inner class object as value. Since all nested class attributes are copied during subclassing, a subclass shares the same inner classes as the superclass. Redefining an inner class on the superclass simply replaces it. Inner classes do not affect the class lookup: for example, when two inner classes nested on the same level want to reference each other, both have to write their *full path* (i.e., sequence of attribute reads).

Whenever a top-level class is defined and there is already a class with that name in the same module, the new class replaces the existing one.

### 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.

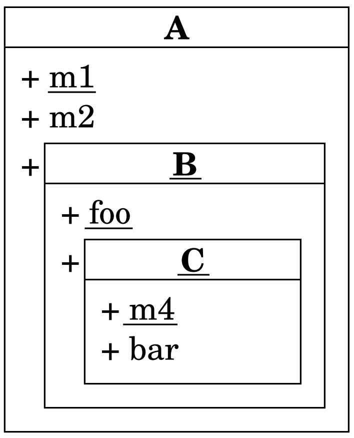


**Figure 4.1.:** Nested Classes Example

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 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.

## 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.

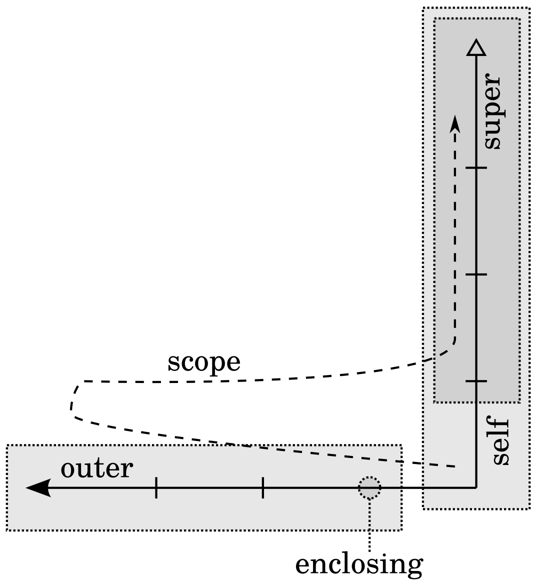


**Figure 4.2.:** Keywords for access to superclass and lexical scope.

**`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)

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

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 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`.

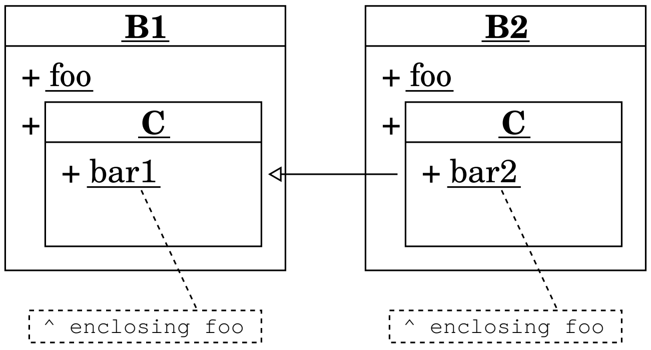


**Figure 4.3.:** Binding of `enclosing` to method's lexical scope.

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.

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 [0].

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.

**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.

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

**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.

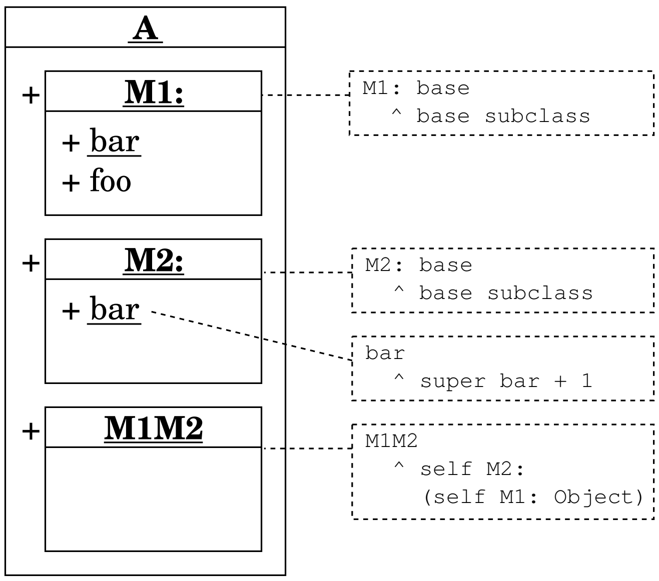


**Figure 4.4.:** Implementation of Mixins with Nested Classes

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 6.5.

# 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.

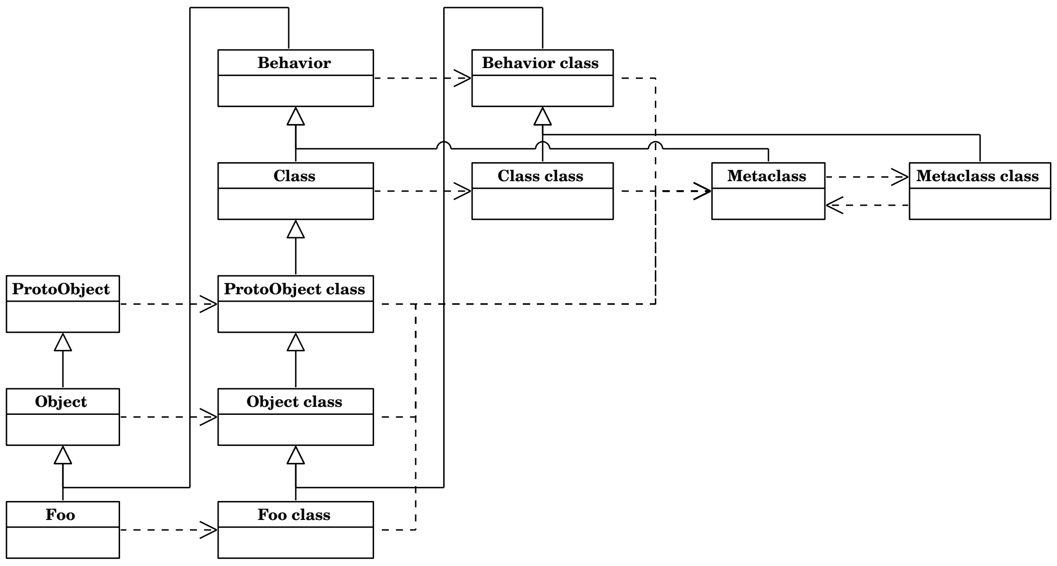


**Figure 5.1.:** Squeak Class Model with Meta Classes

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.

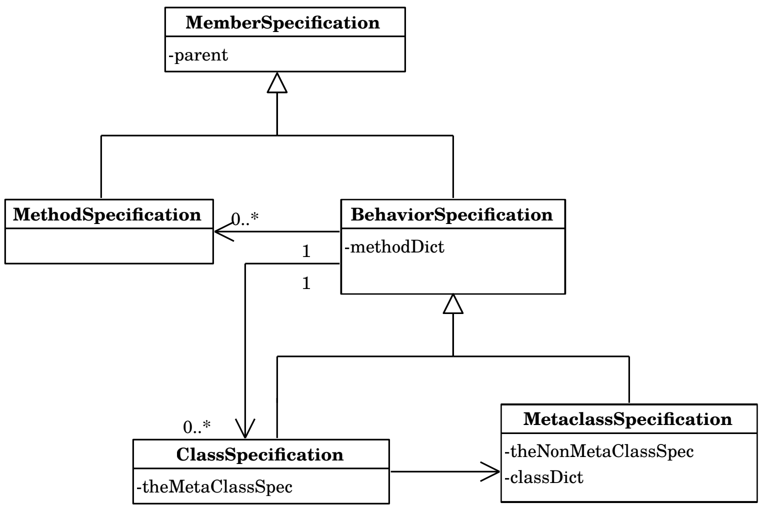


**Figure 5.2.:** Meta Model for Nested Classes

**Class Specification** 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.

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. 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).

[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.

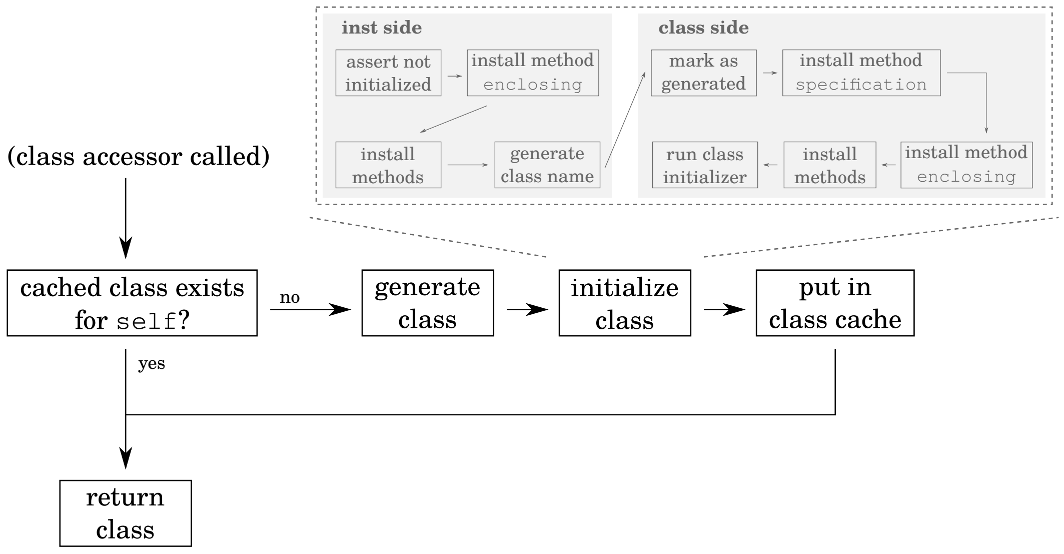


**Figure 5.3.:** Lazy class generation and initialization

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 [0, 0]. 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.

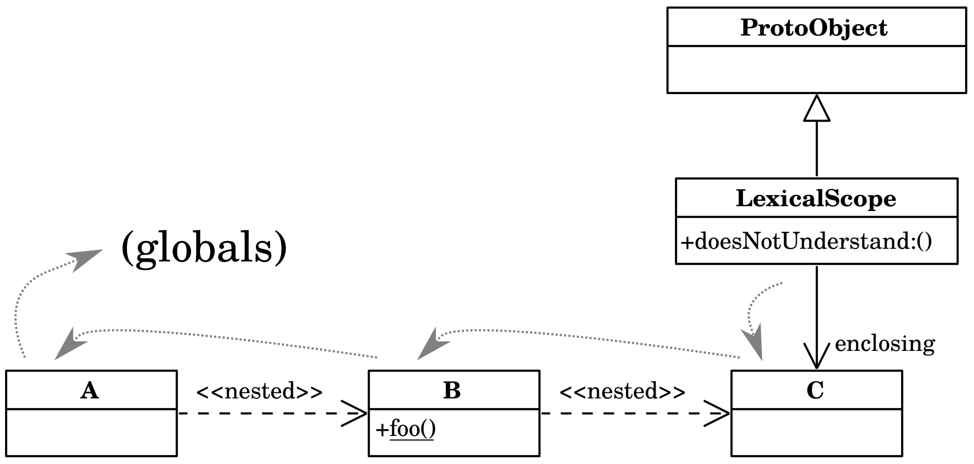


**Figure 5.6.:** `LexicalScope` for `outer` keyword example

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.
- `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`.

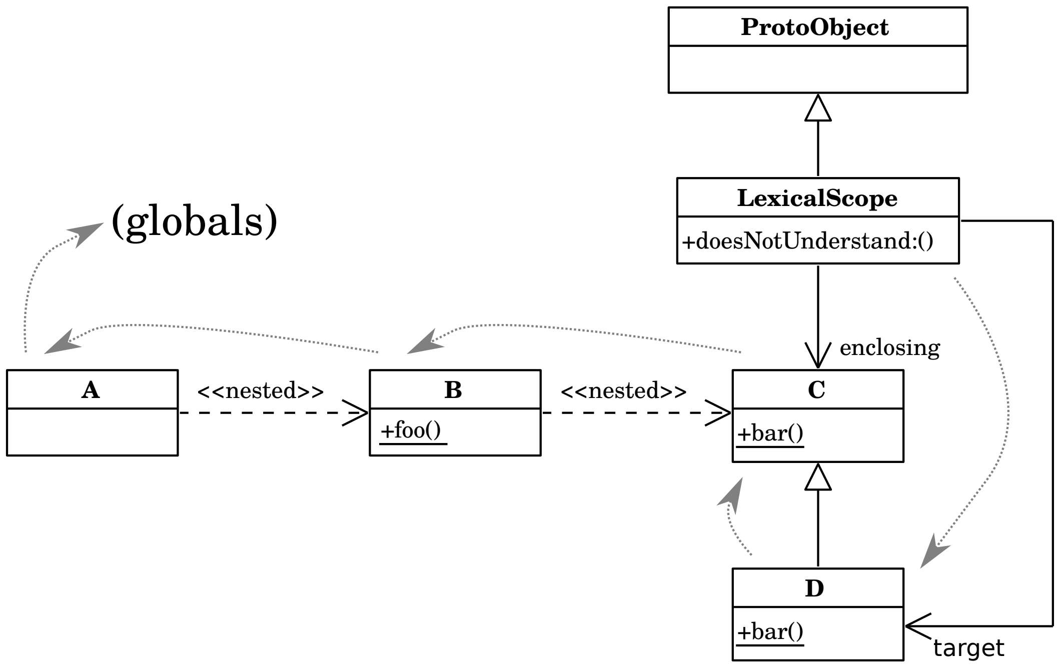


**Figure 5.7.:** `LexicalScope` for `scope` keyword example

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.

## 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 [0, 0]. 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 might want to add additional methods to the mixin application, in which case the mixin application must be subclassed or aliased as described, anyway.

## 5.5. Class Updates

Squeak is a live programming environment with immediate feedback. When the programmers changes a class, these changes should immediately affect all instances of the class in the system, i.e., exisiting instances must be migrated to the new class [0]. In that sense, Squeak and many other Smalltalk implementations [0] are

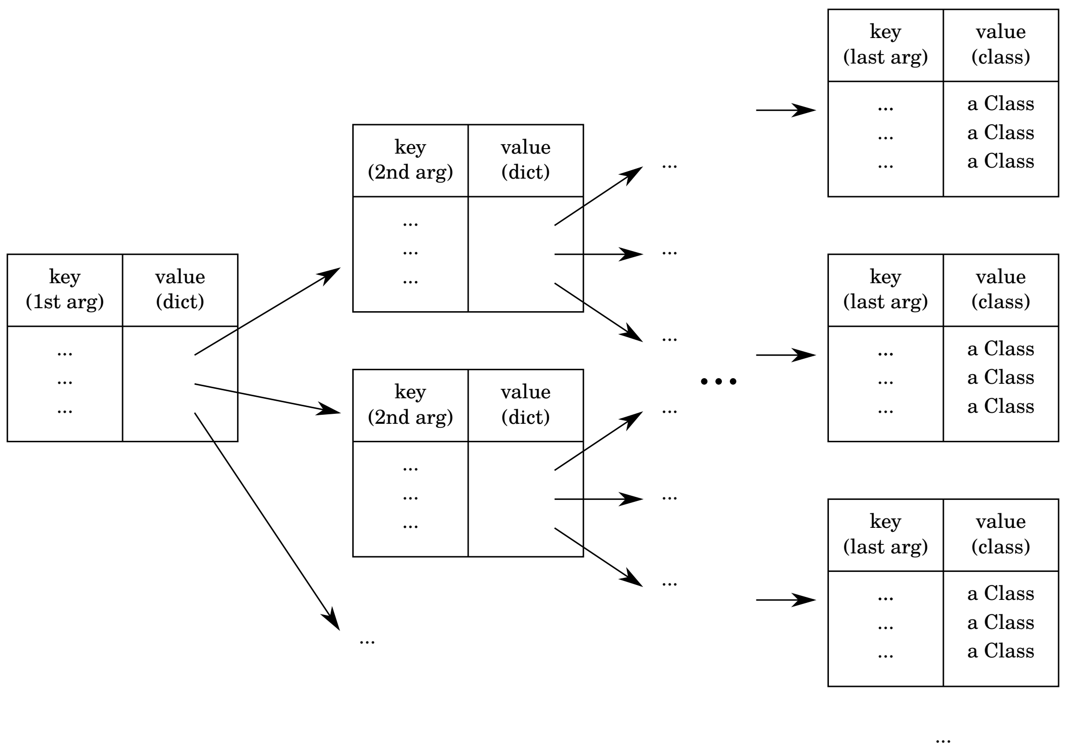


**Figure 5.8.:** Class Cache stored in `ClassSpecification`

```
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

different from other programming languages with an “edit/compile/run cycle” [0]: the programmer has the feeling that there is no difference between compile time and runtime.

For this reason, our system has to ensure that changes to the source code are immediately applied to all living objects in the image. It is important to understand, that changes to parameterized class specifications can affect multiple classes (model instantiations) at runtime. Therefore, every class specification stores a weak collection of all its instantiations. When a class specification is changed, all of its instantiations can be looked up easily and adapted one by one.

**Changing Instance Methods** Whenever an instance method is added, removed, or changed, the system retrieves the collection of all instantiations, and performs the corresponding change on the class object. This does not require creating a new class object, but merely changing the method dictionary using Squeak's meta object protocol [0, 0].

**Changing Class Methods** Changing class methods is equivalent to changing instance methods, with the only difference being that the meta class object is changed instead of the class object.

**Changing Instance Variables** This kind of class change is more difficult to handle than method changes. Whenever an instance variable is add or removed, some methods might have to be recompiled, because instance variables are referenced with indices in the bytecode (see also Section 5.1).

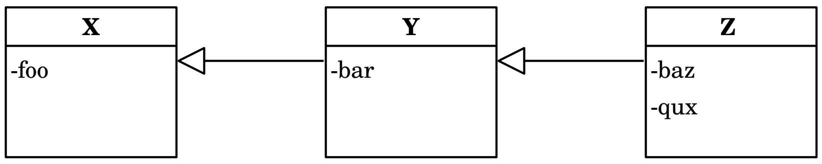


**Figure 5.10.:** Instance Variable Indexing

In Figure 5.10, class `X` has instance variable `foo`, class `Y` has instance variables `foo` and `bar`, and class `Z` has instance variables `foo`, `bar`, `baz`, and `qux`. Instance variables are indexed according to the superclass hierarchy. Therefore, `foo` has index 0, `bar` has index 1, `baz` has index 2, and `qux` has index 3. These indices are used in the bytecode instead of string literals or symbols. Therefore, when instance variables are changed in `X`, all classes (methods in these classes) `X`, `Y`, and `Z` have to be recompiled. If instance variables in `Z` are changed, only `Z` has to be recompiled.

What is more interesting is how instance variables are defined in our system: they are part of the class generator method (Figure 5.5). Therefore, the system has to execute that method a second time whenever it is changed. The method returns a new class object which must be initialized again, i.e., all methods are recompiled. Note, Squeak has the same behavior: whenever an instance variable is changed, methods in the current class and all subclasses are recompiled.

Class generator methods for unparameterized classes can just be invoked without any parameters. However, in order to update parameterized classes, the system has to cache the arguments provided to the class generator method when the class was generated. Therefore, in addition to the class cache, every class specification maintains an argument cache, mapping instantiations (classes) to an array of arguments. This argument cache is a `WeakIdentityKeyDictionary` and different from the dictionary data structure shown in Figure 5.8. That class cache would map arguments to instantiations, but this cache maps instantiations to arguments. Whenever there is no reference to an instantiation in the image anymore, the array of arguments can be garbage collected, because nobody can access the class anymore; therefore, this class does not have to be updated.

Note, that invoking the class generator method a second time might generate a new class. Therefore, all references to the old class have to replaced with references to the new class using the `becomeForward:` method. Also, all instances of the old class have to be migrated to the new class. This is no different from what Squeak does when an instance variable is added or removed, and not described in any more detail in this work. We encourage the reader to consult the *Smalltalk Blue Book* [0] for more information.

**Changing Class Instance Variables** Changing class instance variables is equivalent to changing instance variables, with the only difference being that the meta class object is changed instead of the class object.

## 5.6. Integration in Squeak

In this section, we describe how our system is integrated in Squeak.

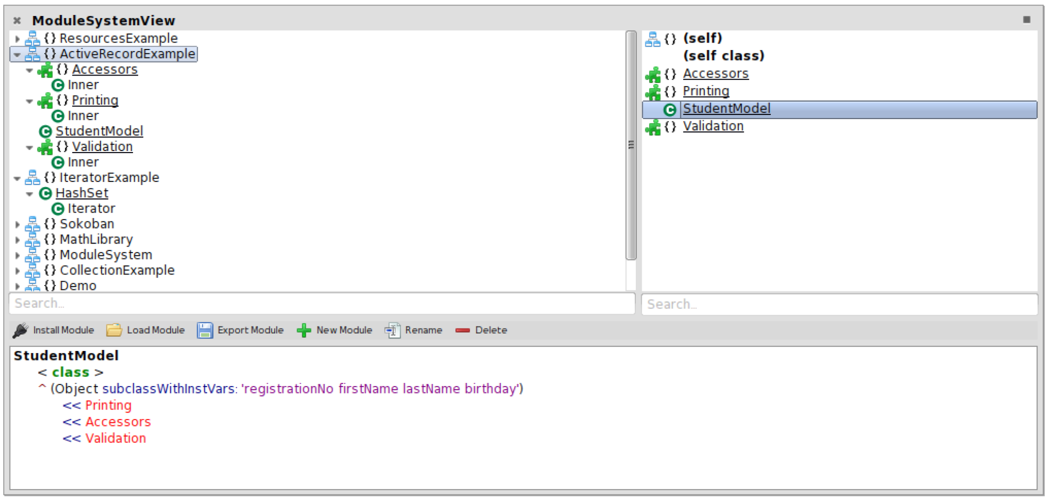
### 5.6.1. Module Repository

At the moment, there is a separate *module repository* for our system. This is a singleton class with a collection all top-level class specifications and a collection of instantiated top-level class specifications. This is useful for development purposes, because basic Squeak classes can be migrated to our system without the risk of damaging the base system. References to classes are first looked up in the module repository, then in the Smalltalk `globals` dictionary.

Eventually, all classes from our system should be listed in the Smalltalk `globals` dictionary, replacing system classes with their counterparts written in our system, which would also make the module repository obsolete.

### 5.6.2. IDE Support

Our system comes with a proof-of-concept implementation of a class browser. The existing system browser cannot be used, because it cannot handle class nesting. Our class browser is written in Vivide [0], a framework for dataflow-driven tool development, and shown in Figure 5.11. It supports creating and deleting methods and nested classes, but basic refactoring functionality and functionality such as browsing senders and receivers is still missing.



```
StudentModel
    < class >
    ^ (Object subclassWithInstVars: 'registrationNo firstName lastName birthday')
        << Printing
        << Accessors
        << Validation
```

**Figure 5.11.:** Class Browser for Nested Classes

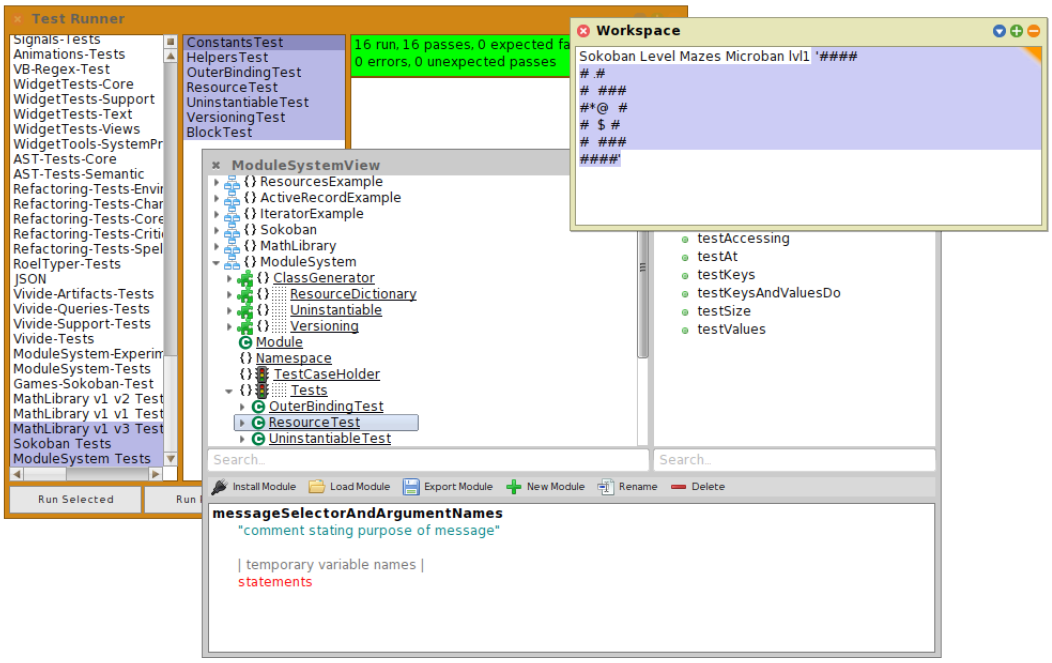
Our system is also integrated with the Squeak workspace and the test runner (Figure 5.12). Unit tests can be written and will show up in the test runner, as long as test classes are defined in a nested class called `Tests` within a top-level class. Later versions might traverse the entire nested classes graph to look for subclasses of `TestCase`, but this basic functionality allows us already to test parts of our system with code written in the system itself.

### 5.6.3. Debugger

The Squeak debugger can be used to step through the source code. Parts of the source code can be selected and being evaluated. This also works keywords that were introduced with our system, such as `outer` and `enclosing`, because they are bound in the Squeak environment of the class.

What is still an issue is that the debugger shows slightly different source code from what the programmer wrote. For example, class references are prepended with the `scope` keyword. In addition, whenever the scope keyword is used, code must be inserted that generates a new instance of `LexicalScope`, because `scope` cannot be bound at compile time (see Section 5.3). When stepping through the source code, the programmer will see additional stack frames for the class generator method and the class accessor method. The class accessor method is merely generated code, which is why it might be hidden in future versions of our system.

Whenever the source code is changed in the debugger, the corresponding method specification is changed, causing all instantiations to be updated.



```
Signals-Tests
Animations-Tests
VB-Regex-Test
WidgetTests-Core
WidgetTests-Support
WidgetTests-Text
WidgetTests-Views
WidgetTools-SystemPr
AST-Tests-Core
AST-Tests-Semantic
Refactoring-Tests-Envi
Refactoring-Tests-Char
Refactoring-Tests-Core
Refactoring-Tests-Criti
Refactoring-Tests-Spel
RoelTyper-Tests
JSON
Vivide-Artifacts-Tests
Vivide-Queries-Tests
Vivide-Support-Tests
Vivide-Tests
ModuleSystem-Experim
ModuleSystem-Tests
Games-Sokoban-Test
MathLibrary v1 v2 Test
MathLibrary v1 v1 Test
MathLibrary v1 v3 Test
Sokoban Tests
ModuleSystem Tests
```

```
ConstantsTest
HelpersTest
OuterBindingTest
ResourceTest
UninstantiableTest
VersioningTest
BlockTest
```

```
16 run, 16 passes, 0 expected fa
0 errors, 0 unexpected passes
```

```
{} ResourcesExample
{} ActiveRecordExample
{} IteratorExample
{} Sokoban
{} MathLibrary
{} ModuleSystem
   {} ClassGenerator
   {} ResourceDictionary
   {} Uninstantiable
   {} Versioning
   Module
   {} Namespace
   {} TestCaseHolder
   {} Tests
      OuterBindingTest
      ResourceTest
      UninstantiableTest
```

```
testAccessing
testAt
testKeys
testKeysAndValuesDo
testSize
testValues
```

```
Sokoban Level Mazes Microban lvl1 '####
# #
# ###
#*@ #
# $ #
# ###
####'
```

```
messageSelectorAndArgumentNames
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

**Figure 5.12.:** Integration in Squeak

# 6. Use Cases

## 6.1. Avoiding Class Name Clashes

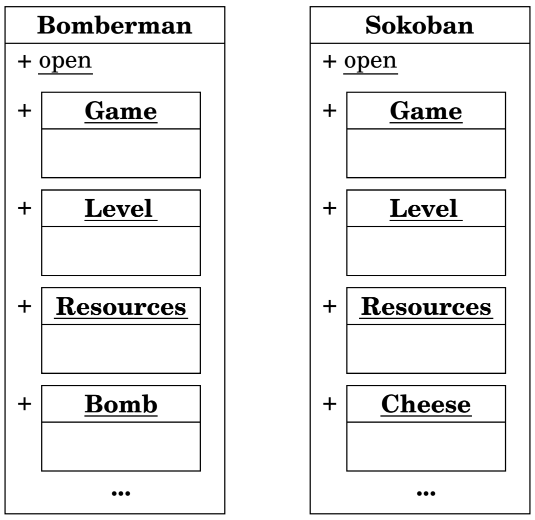


**Figure 6.1.:** Resolving class name clashes with class nesting

In this example, class nesting is used to avoid class name clashes and to give every class a unique fully qualified name. Consider, that we want two load two computer games in a single Squeak image. The first game is a bomberman game, providing classes `Game`, `Level`, `Resources` among others. The second game is a Sokoban game, and has three classes with the same name. Without our system, this would be a problem: as soon as another class with the same is installed, the old one is overwritten with the new name.

With our system, two classes with the same name can coexist in the same image, as long as they are nested within different classes (Figure 6.1).

Note, that, for example, `Bomberman Game` and `Sokoban Game` are different classes. Whenever a class inside `Bomberman` references `Game` using the source code statement `scope Game` or `Game` (equivalent statement), the method lookup recurses in the enclosing classes, until `Game` is found in the `Bomberman` class.

## 6.2. Module Versioning and Dependency Management

In this example, class nested is used to keep multiple different versions of the same library in one image. This is necessary if two applications require different versions of the same library. In the best case, the API of a library should not change within one major version, such that a newer library version should work with an application that was developed with an older library version. However, sometimes, application developers have to work around known bugs or rely on implementation-specific classes which are not designed to be used by library users and subject to change. In that case, application code can break when suddenly a different version of the library is used.

### 6.2.1. Representing Module Versions

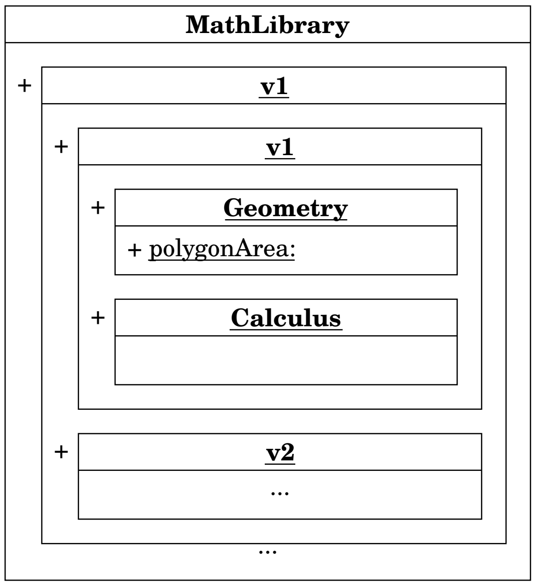


**Figure 6.2.:** Module versioning

Figure 6.2 shows how nested classes can be used for module versioning. In this example, we are developing a library for mathematical operations. The top-level class contains nested classes for every major version. Every major version can again have nested classes for minor versioning. In fact, this scheme can be used to have any kind of versioning system, as long as it is based in numbers.

Two versions of `MathLibrary` are installed in this example: version 1.1 and version 1.2. These versions can be referenced by writing `MathLibrary v1 v1` and `MathLibrary v1 v2`. Note, that even though all versions define classes with the same name, no class clashes occur. If a class in `MathLibrary` references another class in `MathLibrary`, the method lookup will look for classes in the same version of `MathLibrary`.

### 6.2.2. Aliasing Module Versions

Whenever an application requires a class from a library in a certain version, the application can either write down the fully qualified name of the class or create an alias. For example, the fully qualified name of the class `Calculus` in `MathLibrary` version 1.2 is `MathLibrary v1 v2 Calculus`. However, it is very likely that an application requires more than just one class from a library. In this case, an alias should be defined, because it keeps the required version number at a single point in the code (making it easy to change the version) and results in less verbose code.

```
MyApplication»MathLibrary
   ^ Repository MathLibrary v1 v2

MyApplication»rectArea: origin extent: extent
   ^ MathLibrary polygonArea: {
      origin x @ origin y.
      (origin x + extent x) @ y.
      (origin x + extent x) @ (origin y + extent y).
      origin x @ (origin y + extent y) }
```

**Figure 6.3.:** Defining class aliases

Figure 6.3 shows how class alias can be used to specify module versions at a single point in the code. The programmers defines a method`MathLibrary` returning the module in the required version. In `MyApplication»rectArea:extent:`, the reference to `MathLibrary` will be replaced with `scope MathLibrary`, which will call the aliased method. Note, that in `MyApplication»MathLibrary`, we have to reference the library with `Repository MathLibrary`, forcing the lookup to start at the root of our system. Otherwise, the method `MathLibrary` would call itself.

**Helper Methods** In Figure 6.2 the top-level class and major version should be a subclass of the class `Versioning`, a class provided by our system. This class contains convenience methods making it easier to work with version containers. The following list gives an overview of the helper methods `Versioning` provides.

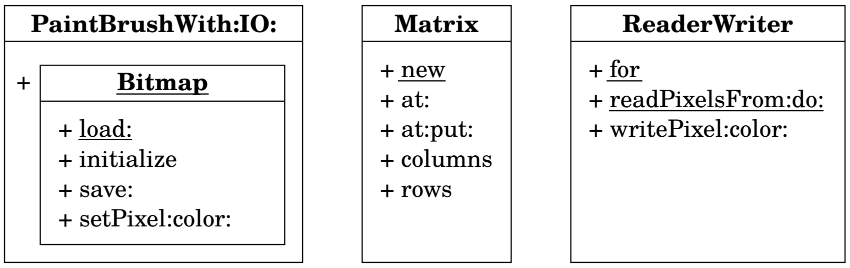
- `Versioning»myLatest`: returns the latest version contained as a nested class in the receiver. For example, `MathLibrary myLatest` returns `MathLibrary v1`.
- `Versioning»latest`: returns the lastest version in the receiver recursively. For example, `MathLibrary latest` returns `MathLibrary v1 v2`.
- `Versioning»atLeast:`: returns the latest version recursively and asserts that its version number is greater than the parameter. For example, `MathLibrary atLeast: '1.1'` returns `MathLibrary v1 v2`, and `MathLibrary atLeast: '1.3'` throws an error.

### 6.2.3. External Configuration

Parameterized classes can not only be used to build mixins, but also externally configuarable modules. The basic idea is taken from Newspeak, where all module dependencies are encapsulated in a `platform` object. This platform object is installed along with the application source code and contains all libraries that the application depends on in the correct version [**bracha2008newspeak**]. This has the advantage that there is no need for a global namespace and all references to external classes are resolved using the platform object, effectively making import statements obsolete. A configurable module does not need to know anything about concrete implementations of external libraries, as long as the implementations provided in the platform implement the expected interfaces.

In our system, methods inside parameterized classes can reference arguments provided to the class accessor method. The idea is that, instead of referencing classes in the global namespace, the programmer references these arguments. The user of the module can then decide which exact implementation he wants to use.

**Example** Figure 6.4 shows part of the implementation of simple drawing application. `PaintbrushWith:IO:` is a parameterized top-level class which takes as arguments a matrix implementation and a file IO library. The matrix implementation is used for storing the pixels inside the application. In the simplest case, this is could be the class `Matrix` from the Squeak standard library. It could, however, also be a class which stores pixels in a compressed form (e.g., using run-length encoding), but has `at:`, `at:put:`, `rows`, and `columns` as public API methods. `ReaderWriter` must be a class or object that supports reading and writing files on a pixel-by-pixel basis. Depending on which IO class the user of the library provides to `PaintbrushWith:IO:`, the application might for example generate JPEG files or PNG files.



```
PaintbrushWith: Matrix IO: ReaderWriter
   < class >
   ^ Object subclass

(PaintbrushWith: Matrix IO: ReaderWriter) class»Bitmap
   < class >
   ^ Object subclassWithInstVars: 'pixels'

(PaintbrushWith: Matrix IO: ReaderWriter) class»Bitmap»initialize
   pixels := Matrix new.

(PaintbrushWith: Matrix IO: ReaderWriter) class»Bitmap»
      setPixel: aPoint color: aColor
   pixels at: aPoint put: aColor.

(PaintbrushWith: Matrix IO: ReaderWriter) class»Bitmap class»
      load: aFile
   | instance |
   instance := self new.
   ReaderWriter
      readPixelsFrom: aFile
      do: [ :point :color | instance setPixel: point color: color ].
   ^ instance

(PaintbrushWith: Matrix IO: ReaderWriter) class»Bitmap»save: aFile
   | writer |
   writer := ReaderWriter BitmapWriter for: aFile.
   1 to: pixels columns do: [ :x |
      1 to: pixels rows do: [ :y |
         writer writePixel: x@y color: (pixels at: x@y) ] ].
   writer close.
```

**Figure 6.4.:** Parameterized classes for external module configuration

It is important to understand that the implementation of `PaintbrushWith:IO:` is entirely decoupled from the pixel data structure representation and the import/-export functionality. It is up to the user of `PaintbrushWith:IO:` to configure the class as needed.

#### 6.2.4. Dependency Injection

## 6.3. 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)

## 6.4. Mixin Modularity with Parameterized Classes

## 6.5. Class Generator Pattern

better syntax for class generators

## 6.6. Extension Methods

better way is needed (e.g., class boxes, refinements, COP, world (paper viewpoints), monkey patching). 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

[0] Edward Benson. *The Art of Rails (Programmer to Programmer)*. Birmingham, UK, UK: Wrox Press Ltd., 2008. ISBN: 0470189487, 9780470189481.

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

[0] 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.

[0] 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.

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

[0] Gwenael Casaccio, Stéphane Ducasse, Luc Fabresse, Jean-Baptiste Arnaud, and Benjamin Van Ryseghem. "Bootstrapping a smalltalk". In: *Smalltalks*. 2011.

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

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

[0] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6.

[0] Atsushi Igarashi and Benjamin C Pierce. "On inner classes". In: *Information and Computation* 177.1 (2002), pp. 56–89.

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

[0] *Java 7 Language Specification*. http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.1.3. Accessed: 2015-07-19.

[0] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0262111586.

[0] Oscar Nierstrasz and Tudor Gîrba. “Lessons in Software Evolution Learned by Listening to Smalltalk.” In: *SOFSEM*. Ed. by Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe. Vol. 5901. Lecture Notes in Computer Science. Springer, Dec. 15, 2009, pp. 77–95. ISBN: 978-3-642-11265-2.

[0] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. “Scalable Extensibility via Nested Inheritance”. In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’04. Vancouver, BC, Canada: ACM, 2004, pp. 99–115. ISBN: 1-58113-831-8. DOI: 10.1145/1028976.1028986.

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

[0] D. Jason Penney and Jacob Stein. “Class Modification in the GemStone Object-oriented DBMS”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA ’87. Orlando, Florida, USA: ACM, 1987, pp. 111–117. ISBN: 0-89791-247-0. DOI: 10.1145/38765.38817.

[0] Marcel Taeumel, Bastian Steinert, and Robert Hirschfeld. “The VIVIDE Programming Environment: Connecting Run-time Information with Programmers’ System Knowledge”. In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2012. Tucson, Arizona, USA: ACM, 2012, pp. 117–126. ISBN: 978-1-4503-1562-3. DOI: 10.1145/2384592.2384604.

[0] *The Python Tutorial, Modules*. https://docs.python.org/2/tutorial/modules.html#intra-package-references. Accessed: 2015-07-19.

# 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 20. 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 12