Matriona: Class Nesting with Parameterization in Squeak/Smalltalk

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

[†] Hasso Plattner Institute, University of Potsdam, Germany
[‡] Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan
[§] Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA

matthias.springer@acm.org fniephaus@acm.org robert.hirschfeld@hpi.de masuhara@acm.org

Abstract

We present Matriona, a module system for Squeak, a Smalltalk dialect. It supports class nesting and parameterization and is based on a hierarchical name lookup mechanism. Matriona solves a range of modularity issues in Squeak. Instead of a flat class organization, it provides a hierarchical namespace, that avoids name clashes and allows for shorter local names. Furthermore, it provides a way to share behavior among classes and modules using mixins and class hierarchy inheritance (a form of inheritance that subclasses an entire class family), respectively. Finally, it allows modules to be externally configurable, which is a form of dependency management decoupling a module from the actual implementation of its dependencies.

Matriona is implemented on top of Squeak by introducing a new keyword for run-time name lookups through a reflective mechanism, without modifying the underlying virtual machine. We evaluate Matriona with a series of small applications and will demonstrate how its features can benefit modularity when porting a simple application written in plain Squeak to Matriona.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Inheritance, Modules, Packages

General Terms Languages

Keywords Class nesting, class parameterization, mixin modularity, class hierarchy inheritance

MODULARITY'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-3995-7/16/03...
http://dx.doi.org/10.1145/2889443.2889457

1. Introduction

A popular description of *modularity* claims that a design method should satisfy five requirements [29] if we want to call it *modular*: decomposablity, composability, understandability, continuity, and protection. In this paper, we present the *Matriona*[1] module system for Squeak/Smalltalk, which supports class nesting and class parameterization and aims for supporting the first three modularity requirements.

1.1 Modularity Requirements

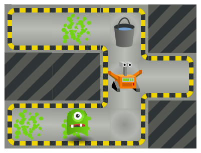
In this work, we focus on a selection of common modularity issues. We decided to demonstrate our approach using Smalltalk, because it is a language suitable for prototyping. In Section 5, we show how we solved these problems with class nesting and parameterized classes.

Running Example SpaceCleanup[2] is a Bomberman clone, implemented in Squeak using the Morphic framework. This game will serve as a running example in the remainder of this paper. Figure 1 illustrates the architecture of SpaceCleanup. The game consists of a *level*, which is a matrix arrangement of *tiles*. A tile is a game field and can contain multiple *items*, such as the player, a monster (enemy), slime or a wall (blocking items), a street (walkable terrain), or a bucket used to *wash away* slime. The goal of the game is to remove all slime and monsters by placing exploding buckets.

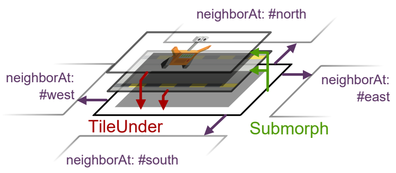
SpaceCleanup uses the class `Morph` provided by Squeak's Morphic framework [21, 28]. For example, the classes `Level`, `Item`, and `Tile` are subclasses of `Morph`. A morph is a user interface element that can have a certain shape, background color, image, and multiple submorphs which are contained in and rendered on top of the morph. Morphs can *step*; the Morphic framework invokes every morph's `step` method repeatedly and the return value of the method `stepTime` determines the duration between these invocations. The method

[1] The word "Matriona" has its origin in the Russian word "Matryoshka", which is the name for a set of wooden dolls that can be nested within each other [16].

[2] `https://github.com/HPI-SWA-Lab/mod16-matriona`



(a) Screenshot



(b) Architecture with `Tile` and `Item`

Figure 1: Screenshot and Architecture of SpaceCleanup

`Level»stepTime` essentially determines the speed of the entire game.

Hierarchical Namespaces In plain Smalltalk, all classes are organized in a single global dictionary mapping identifiers to class objects. Therefore, it is not possible to install two applications which provide classes with the same name. For example, another game providing a `Level` class cannot be installed side by side with SpaceCleanup in a Squeak image. Even though namespaces (*environments*) were recently introduced in Squeak and aim at solving this issue, it is still common to use unique class name prefixes as a workaround because of missing tool support and conceptual difficulties (Section 6.4).

Namespaces benefit modular composability by removing restrictions on which software components can be installed side by side. Hierarchical namespaces allow for a class organization that is easy to understand. The basic idea is to “group together what belongs together” [8]. For example, in SpaceCleanup, class `Item` can only be used as part of a tile. Making these kinds of relationships obvious can benefit modular understandability. If programmers new to the code base are interested in how `Item`-related functionality works, it is probably sufficient to take a look at class `Item` and all classes that “belong” to `Item`. This is particularly useful for large applications with hundreds or thousands of classes.

Application Customization It is sometimes necessary to change an application while keeping the original code around. This is needed for experiments at development time or in the light of software product lines [6]. Consider, for example, that *Speedy SpaceCleanup*, a variant of SpaceCleanup, should be designed, where the game runs twice as fast as before. To achieve this, the method `Level»stepTime` can be modified. As another example, imagine that *Damage SpaceCleanup* should be designed, where items have a health value and some items can cause damage on others. To achieve this, classes `Item` and its subclasses (e.g., `Player` and `Monster`) should be modified.

Programmers could make a copy of the entire application and change the required parts. The downside of this approach is that both copies can easily get out of sync. Every change made to the original application must be manually applied to the copy. A good implementation approach should not require code duplication and changes to the original game should be effective immediately in the modified game.

Sharing Behavior Among Classes There are cases where a group of methods is shared among multiple classes. The most obvious approach is to duplicate the methods in all of these classes. A good module system should provide a better a way to share common behavior among multiple classes which does not require code duplication and applies changes to shared code to respective classes automatically.

For example, consider *tiles* in SpaceCleanup. A tile is a container for items and should provide methods like `allSatisfy:` to ensure that there is no item on top of a tile which would prohibit the player from entering. Such methods are required frequently, possibly multiple times in SpaceCleanup. The programming language should provide a mechanism to modularize such functionality as an extension “without pre-determining what exactly it can extend” [36]. For example, basic collection functionality should be provided by the execution environment in such a way that it can be reused at arbitrary points in an application.

External Configuration In Smalltalk, application dependencies (libraries etc.) are usually bound to specific implementations in the application source code by referencing a class from a certain implementation, i.e., dependencies are bound at compile time (specific versions cannot be specified; Smalltalk uses whatever version is currently present in the image). A more modular approach binds dependencies at run-time such that an application can be instantiated with different implementations, without modifying the application. Smalltalk applications can already be developed in such a way that dependencies are passed in as part of the run-time parameters; however, this approach reaches its limits if the class hierarchy should depend on those parameters.

As an example, consider that programmers or users would like to run SpaceCleanup with different versions of the Morphic framework or with a Morphic alternative that implements the Morphic interface but uses the host operating system for rendering. If SpaceCleanup is externally configurable, its users can provide a Morphic implementation as a run-time argument (similar to command line arguments). Most classes in SpaceCleanup are subclasses of `Morph`; therefore, the class hierarchy (superclasses) would depend on the parameters for external configuration.

1.2 Contributions

We propose a new class organization for Smalltalk based on a hierarchical namespace, where classes can be parameterized and nested within other classes. Matriona is our first prototype for Squeak/Smalltalk. This paper makes the following contributions.

- A backward-compatible module system for Smalltalk supporting class nesting and class parameterization

- A nested class and parameter lookup mechanism supporting application customization and external configuration

Section 3 and 4 will describe the abstract concept and its implementation in Matriona. Section 5 shows how the previously mentioned problems can be solved in Matriona. Section 6 and 7 compare Matriona with related work and give an overview of future work. Section 8 gives a brief conclusion of our work.

2. Background

Squeak is an open-source Smalltalk programming system. Three fundamental concepts of Smalltalk are the programming language with its libraries and tools, and the image [20].

A Smalltalk image[3] is a snapshot of an object space, i.e., it is the collection of all objects managed by the virtual machine at a time. Classes are accessible through an image-wide dictionary, which represents Smalltalk's global namespace.

Matriona is module system for Squeak; Smalltalk's architecture and meta object protocol have proven to be a suitable platform for our language design experiments. Nevertheless, the findings of this paper are amenable to other class-based, object-oriented programming languages with single inheritance if they support dynamic class generation and installing methods at runtime. In the following, we gibe a brief overview of Smalltalk-specific notation.

The name of a method is called *selector*. The number of colons in a selector is equal to the number of parameters of the corresponding method. When calling a method, the ith argument is written after the ith colon, similar to Objective-C syntax. The term *message send* is used as a synonym for *method call*. The first line of every method listing has the form `A»foo`, which means that `foo` is defined as an instance method of `A`. `A class»foo` means that `foo` is defined as a class method (*static method*). Class methods are instance methods of the meta class (class's class) which is created automatically whenever a new class is defined [20]. Both instance-side and class-side methods can be overridden. Inside a method body, the upward arrow (↑) denotes a return statement.

Control flow constructs such as *if* branches and loops are message sends with a block closure (anonymous function) containing the conditional control flow (e.g., loop body) as an argument. For example, `do:` is Smalltalk's equivalent of a for-each loop.

3. Concept

In Matriona, classes are organized in a hierarchical namespace based on class nesting and they can be parameterized.

3.1 Nested Class Definition

Matriona supports class nesting as a means of establishing a global hierarchical namespace. Classes can have multiple nested classes as *class-side* members. The parent of a nested class is called the *enclosing class*. A nested class can be accessed via a message send to its enclosing class. The single root of the namespace is the top-level class `Smalltalk`. Nested classes are virtual, i.e., they can be overridden in subclasses of their respective enclosing classes.

Nested classes are inherited in subclasses, similarly to methods. However, in contrast to method inheritance, the original class and the inherited class are not equal with respect to object identity. In the examples presented in this section, inherited classes do not differ from their counterparts in the superclass, e.g., they have the same methods, the same instance variables, and the same superclass. However, they are different classes with different enclosing classes. Therefore, such a class is called an *inherited class copy*. Superclass statements are virtual, i.e., they can evaluate to different results depending on the runtime enclosing class[4]; the name lookup for the superclass is part of the class initialization.

Notation For readability reasons, we use the following abbreviations for class names in the context of this paper: `St` for `Smalltalk`, `C` for `Collection`, `QC` for `QuickCollection`, and `Scu` for `SpaceCleanup`. If `C1` is an inherited class copy of `C2`, we write `C1[C2]`.

Class Lookup without Inheritance Classes can be referenced by either specifying their fully qualified name (a chain of message sends) or using a relative name. The fully qualified name of a class is defined as the fully qualified name of its enclosing class concatenated with the name of the nested class. The fully qualified name of top-level class `Smalltalk` is simply `Smalltalk`.

The lookup for a relative class name starts in the class containing the method referencing the class (*lexical scope lookup*). The lookup continues in enclosing classes, until the top-level class `St` is reached. If no nested class is found in `St`, an error is raised. As described in the next paragraph, the class lookup becomes more complicated in the light of inheritance.

Class Lookup with Inheritance A subclass of an enclosing class inherits all its nested classes, similarly to its methods. Furthermore, nested classes are virtual members, i.e., they can be overridden in subclasses. Consider the example in Figure 2(b). Class `St.C` contains two nested classes `St.C.Array` and `St.C.Algorithm`. The latter class is only used internally by collection classes such as `St.C.Array`, for example to perform sorting operations. The method `St.C.Array.sort` uses the method `St.C.Algorithm. Sorting.sort` (via a relative lookup of class `Algorithm`).

In Figure 2(a), an optimized collection library is defined. The only difference compared to the library in Figure 2(b) is that it uses a different sorting algorithm. Class `St.QC` is a subclass of `St.C`, therefore, class `St.QC.Array[St.C.Array]`

[3] Smalltalk systems without an image exist, e.g., GNU Smalltalk.

[4] Consequently, an inherited class copy can have a superclass different from its original class's superclass. This fact will be particularly important for single inheritance-based class hierarchy linearization (Section 5.3).

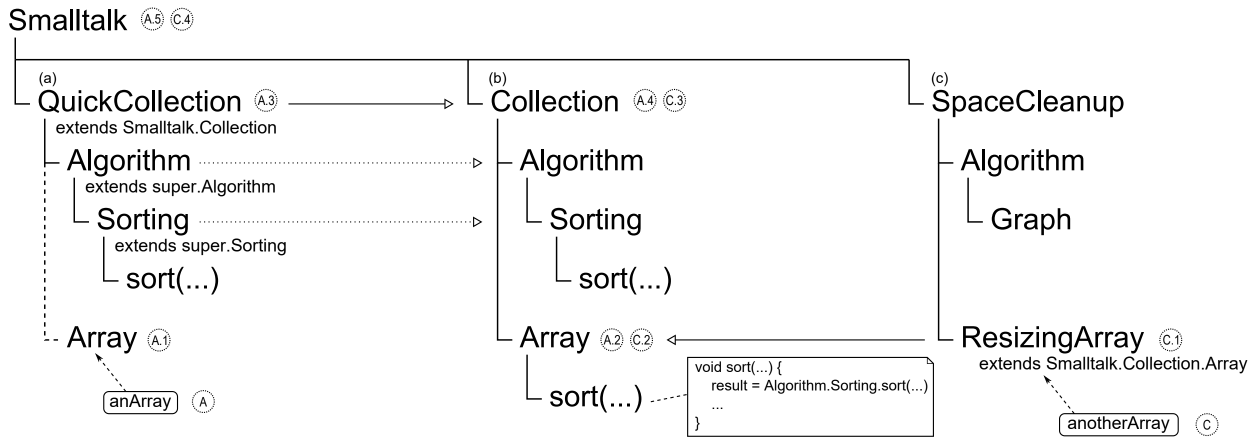


Figure 2: Array Class Organization. Dashed lines indicate inherited class copies. Dotted inheritance arrows indicate that the superclass is an inherited class copy. For example, `St.QC.Algorithm` is a subclass of `St.QC.Algorithm[St.C.Algorithm]`.

is inherited from `St.C`. The class lookup should be designed such that `St.QC.Array.sort` uses `St.QC.Algorithm.Sorting` instead of `St.C.Algorithm.Sorting`.

In Figure 2(c), the application SpaceCleanup extends class `St.C.Array` using subclassing. SpaceCleanup provides a class `St.Scu.Algorithm` of its own, but this class is specific to SpaceCleanup and does not contain algorithm functionality required by `St.C.Array`. The class lookup should be designed such that `St.Scu.ResizingArray.sort` uses `St.C.Algorithm. Sorting` and does not attempt to use `St.Scu.Algorithm.Sorting`.

On the one hand, class `St.QC.Algorithm` should be looked up by following the class nesting structure in Figure 2(a), i.e., the enclosing class `St.C` with its nested class `Algorithm` should be late bound (*enclosing class lookup*). On the other hand, `St.C.Algorithm` should be looked up according to the lexical scope of method `St.C.Array.sort` in Figure 2(c) (*lexical scope lookup*).

Name Lookup Mechanism We need a name lookup mechanism that lets programmers refine class hierarchies and control the scope of their refinements. Our mechanism uses inheritance relationships to indicate that the name lookup should traverse a new class nesting hierarchy and can be seen as a generalization of polymorphic method lookup to class nesting hierarchies: Class c' can override methods defined in class c if $c' \rhd c$ (c' is a subclass of c). An enclosing class e' of a subclass c' can *override* a name defined in the enclosing class e of superclass c only if $e' \rhd e$ (and $c' \rhd c$).

Class hierarchy extensions and refinements like `St.QC` follow that subclassing pattern. Note that, in Figure 2(a), class `St.C` is being refined and not `St.C.Array` because sorting logic is part of `St.C` and not `St.C.Array`. In Figure 2(c), `St.C.Array` is being extended but not `St.C`, which is why `St.Scu` $\not\rhd$ `St.C`.

Definition. *We denote the* run-time class nesting hierarchy *(used for enclosing class lookup) by R and the* lexical class nesting hierarchy *(used for lexical scope lookup) by L. Traverse R and L in parallel. Let r be the current class from R and l be the corresponding class on the same level from L. First try to lookup the name in r and its superclasses, then in l and its superclasses. Ensure that one of the following conditions holds true.*

- *$r = l$, i.e., r and l are the same classes*
- *$r \rhd l$, i.e., r is a subclass of l*
- *$r \rightsquigarrow l$, i.e., r is an inherited class copy of l*
- *$r \rhd_{\rightsquigarrow} l$, i.e., r is a subclass of an inherited class copy of l*

If, at some point, none of these conditions is satisfied or the end of R is reached, continue the lookup in L only. If the end of L is reached, raise a lookup error.

As an example, observe how the lookup of `Algorithm` in `St.QC.Array[St.C.Array].sort` in Figure 2(a) traverses the following classes, with $R = ($`St.QC.Array[St.C.Array]`, `St.QC`, `St`$)$ and $L = ($`St.C.Array`, `St.C`, `St`$)$.

1. `St.QC.Array[St.C.Array]` and superclasses, because `St.QC.Array[St.C.Array]` $\rightsquigarrow$ `St.C.Array` (failure)
2. `St.QC` and superclasses, because `St.QC` $\rhd$ `St.C` (success)
3. `St`, because `St` = `St` (however, lookup already ended)

The lookup of `Algorithm` in `St.Scu.ResizingArray.sort` in Figure 2(c) traverses these classes, with $R = ($`St.Scu.ResizingArray`, `St.Scu`, `St`$)$ and $L = ($`St.C.Array`, `St.C`, `St`$)$.

1. `St.Scu.ResizingArray` and superclasses, because `St.Scu.ResizingArray` $\rhd$ `St.C.Array` (failure)
2. `St.C`, because `St.Scu` is not equal to/subclass of/inherited copy of/subclass of inherited copy of `St.C` (success)
3. `St`, only L from now on (however, lookup already ended)

Algorithm 1 describes the relative class lookup as pseudo code. The parameters `name`, `cls`, and `method` denote the name of the class to be looked up, the run-time class (polymorphic

Algorithm 1 Relative Name Lookup

```
procedure RelativeLookup(name, cls, method)
    r ← cls
    l ← method.class
    checkRuntime ← true
    repeat
        if checkRuntime ∧ name ∈ r
            return r[name]
        else if name ∈ l
            return l[name]
        end if
        l ← enclosing(l)
        if checkRuntime
            r ← enclosing(r)
            if r = null ∨ !($r = l \vee r \rhd l \vee r \rightsquigarrow l \vee r \rhd_{\rightsquigarrow} l$)
                checkRuntime ← false
            end if
        end if
    until r = null
    raise lookup failed
end procedure
```

class) of the executing method, and the executing method, respectively. The function *enclosing(class)* returns the enclosing class of *class*. In Figure 2(a), *method* = `St.C.Array.sort` and *cls* = `St.QC.Array[St.C.Array]`.

3.2 Parameterized Classes

A parameterized class is a class template which must be instantiated with arguments before use. Parameters can be accessed from within the class and all of its nested classes. They are first-class members and can be used during class definition, e.g., as a superclass.

The lookup mechanism for parameters is identical to that for nested classes; parameters are class-side members just as nested classes are class-side members. A subclass can override the parameter of a superclass if it has the same name. Otherwise, the parameter will be treated as a different one. Note that a subclass can define parameters which are entirely different from the superclass's parameters. However, the parameters of the superclass must be specified when referencing the superclass during subclass definition.

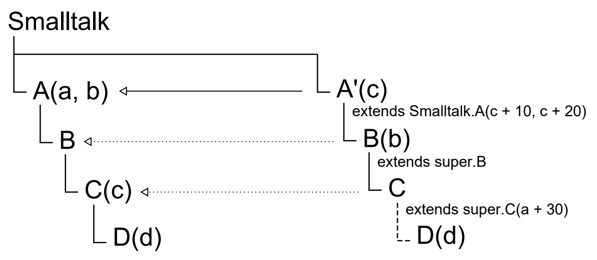


Figure 3: Parameterized Classes

Consider Figure 3 as an example. Let us assume that a method `foo` is defined in `St.A.B.C.D` which returns the tuple (a, b, c, d). An invocation of `St.A(1, 2).B.C(3).D(4).foo` returns a tuple with the following values.

(a) `St.A.a` = 1

(b) `St.A.b` = 2

(c) `St.A.B.C.c` = 3

(d) `St.A.B.C.D.d` = 4

An invocation of `St.A'(1).B(2).C.D(4).foo` returns a tuple with the following values. This example resembles a fictitious case meant to demonstrate intricate details of the lookup mechanism. Real use cases are typically simpler.

(a) `St.A'.a[St.A.a]` = `St.A'.c` + 10 = 11

(b) `St.A'.B.b` = 2

(c) `St.A'.B.C[St.A.B.C].c` = `St.A'.a[St.A.a]` + 30 = 41

(d) `St.A'.B.C.D[St.A.B.C.D].d = 4`

Algorithm 1 is used for looking up both nested classes and class parameters. In case there are a nested class and a class parameter with the same name defined in the same class, the nested class will take precedence in the lookup.

Note that a newly-introduced name in a subclass of a nested class can also *shadow* a name in its superclass, since the lookup ends as soon as the first occurrence of a name is found. In the previous example, `St.A'.B.b` shadows `St.A.b`.

4. Implementation

In this section we describe Matriona's implementation on top of Squeak. Matriona leaves existing Squeak classes mostly unchanged; therefore, it can run next to the original Squeak toolchain and other Squeak applications. Minor changes were made to the Smalltalk compiler, but not to the virtual machine. The Matriona user interface is based on Vivide, a framework for dataflow-driven tool development [39].

4.1 Syntax

In Squeak/Smalltalk, classes are created by sending a `subclass:` message to the class's superclass. The newly-created class is then added to the *globals* dictionary, a collection of all globally accessible objects. For example, the following listing shows how class `ScuGame` is defined as a subclass of `Morph`.

```
Morph subclass: #ScuGame
   instanceVariableNames: 'state'
   classVariableNames: ''
   category: 'SpaceCleanup-Core'
```

In Matriona, `Smalltalk` is the only global object and top-level class. Nested classes are defined as a special kind of method with a `<class>` pragma, which is similar to a method annotation in Java. Such a method is called a *class generator method* and is expected to return the new class object. Class generator methods can return different results in subclasses, which is why superclasses of nested classes are virtual. The following listing shows how class `SpaceCleanup` is defined as a subclass of `Object` and nested within `Smalltalk`. This new class acts as a container/namespace for all classes that belong to SpaceCleanup. Class `Game` is defined as a nested class within `SpaceCleanup`.

```
Smalltalk class»SpaceCleanup                    < class >
   ↑ Smalltalk Kernel Object subclass

Smalltalk SpaceCleanup class»Game               < class >
   ↑ Smalltalk Morphic Morph
      subclassWithInstVars: 'state'
      classVars: ''
```

Classes in Squeak/Smalltalk are defined by interacting with its superclass (executing code in the system browser or workspace). In contrast, nested classes in Matriona are defined by adding a method to a class. The method `Class»subclassWithInstVars:` is provided by Matriona and does not require the class name as a parameter, because the class name is defined as the name of the enclosing class concatenated with the selector of the method. The method `Class»subclass` is a shortcut in case no additional instance or class variables are needed.

4.2 Class Accessor Methods

A nested class can be accessed with a message send to its enclosing class. Whenever the compiler encounters a `<class>` pragma during method compilation, two methods are generated: a class generator method with a name-mangled selector (and the code that programmers specify) and a *class accessor method* with the selector specified by programmers. The class accessor method invokes its corresponding class generator method and performs an additional *class initialization*. The return value of the class generator method is called the *target class*. The following list gives an overview of the steps that are performed when a class accessor method is executed.

1. Return the nested class if it is cached[5].
2. Invoke the class generator method.
3. Set the target class name.
4. Install/compile all instance methods.
5. Install/compile all class methods.
6. Invoke the class initializer.
7. Add the nested class to the cache.
8. Return the nested class.

Two classes a and b are inherited class copies if they were initialized using the same class specification.

[5] Caching is important to ensure class object identity of subsequent class accessor method invocations and substantially improves performance. Class caches are parameter-specific.

4.3 Accessing the Lexical Scope

Matriona adds a new `scope` keyword to the Smalltalk language for looking up both classes and parameters. Messages sent to `scope` are treated as names to be looked up according to Algorithm 1. The following listing shows how `scope` can be used in methods and class definitions. In the following example (Figure 2(b)), `scope` is used to lookup `Algorithm`.

```
Smalltalk Collection Array»sort
   | sorted |
   sorted := scope Algorithm Sorting sort: self.
   " ... "
```

In the next example, `scope` is used to reference the superclass `Item`, nested in `Tile`, during class definition. Values for defining instance variables and class variables do not have to be known at compile time of the class generator method.

```
Smalltalk SpaceCleanup Level Tile class»Monster
   < class >
   ↑ scope Item subclassWithInstVars: '...'
```

The nested lookup must necessarily be performed dynamically. Otherwise, class references could not be overridden, when a method is executed in the context of a subclass. `scope` is currently implemented as an object with no methods except for a handler for all unsuccessful message sends. That handler performs the lookup and invokes the correct method.

The object referred to by `scope` is a first-class object and can be passed around without any restrictions. In that sense, it is similar to `thisContext` for accessing stack frames. However, references to `thisContext` are compiled to a special byte code instruction, whereas `scope` is generated during compilation and bound as a method literal, which avoids changes to the virtual machine, but has its pitfalls (Section 7).

`outer` is a keyword similar to `scope`, but the lookup starts in the innermost enclosing class instead of the run-time class of the executing method. `scope` is to `outer` as `self` is to `super`.

Nested classes and methods can be overridden with each other in a subclass because, from Smalltalk's point of view, referencing a nested class and invoking a method is the same as sending a message to the enclosing class object. Consequently, `scope` and `outer` do not only look up nested classes and class parameters, but also regular class-side methods.

5. Use Cases

This section gives an overview of how Matriona can be used to solve the problems described in Section 1.

5.1 Hierarchical Namespaces

In plain Smalltalk, the class namespace is flat. Therefore, two classes with the same name from two different applications cannot coexist in the same Smalltalk image. With nested classes, every application or library can be represented by a dedicated class nested in `Smalltalk`. All classes that belong to the application are then nested within that class, regardless of the names of classes that belong to other applications. In Matriona, such a class is called a *module*.

5.2 External Configuration

A module is externally configurable if its users (clients) can control its configuration from the outside. When talking about configuration, we refer to application input parameters and module dependencies. In case the configuration does not affect the class structure, the same class structure can be reused for multiple configurations (cases in which class-side/static state is used may be an exception). In case the configuration affects the class structure (e.g., `Level` is a subclass of `Morphic Morph`), a separate class structure is necessary, because a class generator method is only executed once and its result is cached. Consequently, passing the configuration as part of the arguments for a factory method does not work. We focus on the latter case in this section.

In the following listing, the Morphic dependency can be configured externally. The parameter of class `SpaceCleanupWithMorphic:` must be a class or object implementing the interface of the Morphic framework (e.g., there must be a class `Morph` accessible via a message send to the parameter).

```
Smalltalk class»SpaceCleanupWithMorphic: Morphic
   < class >
   ↑ Smalltalk Kernel Object subclass

Smalltalk SpaceCleanupWithMorphic: class»Level
   < class >
   ↑ scope Morphic Morph subclass
```

```
(Smalltalk SpaceCleanupWithMorphic: Smalltalk ↩
    Morphic) open.
(Smalltalk SpaceCleanupWithMorphic: Smalltalk ↩
    NativeRenderingFramework) open
```

The parameter `Morphic` does not necessarily have to be a version of the Morphic framework. It can be any graphics rendering framework (such as a fictious native rendering framework), as long as it implements the same interface.

5.3 Application Customization

With class hierarchy inheritance [10, 12, 14], it is possible to inherit from an entire application and change it in a way that was not foreseen by the application developer. The basic idea is to define a subclass of the module. Methods and nested classes are inherited, but can be overridden. Methods and nested classes are lately bound (virtual), i.e., the exact method/nested class is determined at runtime.

Overriding Single Methods In this example, a modified version `SpaceCleanup` should be defined which supports changing the speed of the game by modifying the method `Level»stepTime`. The following listing shows how new instances of `Level` are created in `SpaceCleanup`. At some point, `scope Level` (relative lookup) is used to reference class `Level` and create a new instance of it.

```
Smalltalk SpaceCleanup»loadLevel: levelId
   level := scope Level new
      loadFromString: (levels at: levelId);
      yourself
```

In the following listing, `SpeedySpaceCleanup` is defined as a subclass of `SpaceCleanup` and does not only inherit methods and variables, but also nested classes. Class `Level` is overridden with a subclass of its inherited class copy, where the step time is variable.

```
Smalltalk class»SpeedySpaceCleanup          < class >
   ↑ Smalltalk SpaceCleanup subclass

Smalltalk SpeedySpaceCleanup class»Level    < class >
   ↑ super Level subclassWithInstVars: 'stepTime'

Smalltalk SpeedySpaceCleanup Level»stepTime
   ↑ stepTime

Smalltalk SpeedSpaceCleanup Level»speed: anInteger
   stepTime := anInteger

Smalltalk SpeedySpaceCleanup»speed: anInteger
   level speed: anInteger
```

Whenever a new instance of `SpeedySpaceCleanup` is started, the method `SpaceCleanup»loadLevel:` will be executed in the context class of `SpeedySpaceCleanup` and the `scope` lookup will return `SpeedySpaceCleanup Level`, because `SpeedySpaceCleanup` ▷ `SpaceCleanup`. From now on, the speed of the game can be adjusted as follows.

```
| game |
game := Smalltalk SpeedySpaceCleanup new.
game speed: 500
```

Class Hierarchy Linearization In this example, SpaceCleanup should be modified such that items cause damage on other items. Every item should have a `damage` method returning a dictionary mapping item symbols to damage values. For example, monsters cause a damage value of `0.25` if they meet with the player. Additional methods should be defined on `Item`, but are not described here for brevity reasons.

This example is interesting, because both `Item` and some subclasses of `Item` should be extended (Figure 4), resulting in a case where multiple inheritance is desired. Class `St.DScu.Tile.Monster` should inherit from both `St.Scu.Tile.Item` and `St.DScu.Tile.Monster`[6].

`St.Scu.Tile.Monster` is a subclass of `scope Item`, i.e., `Item` is referenced relatively and subject to the lookup mechanism in Algorithm 1. `St.DScu.Tile.Monster` a subclass of `super.Monster`, i.e., a subclass of the inherited class copy `St.DScu.Tile.Monster[St.Scu.Tile.Monster]`. This inherited class copy is a subclass of `Item`, whose lookup results in `St.DScu.Tile.Item`, because `St.DScu.Tile` $\triangleright_{\rightsquigarrow}$ `St.Scu.Tile`. Similarly, `St.DScu.Tile.Item` is a subclass of the inherited

[6] For readability reasons, we use `DScu` as a shortcut for `DamageSpaceCleanup`.

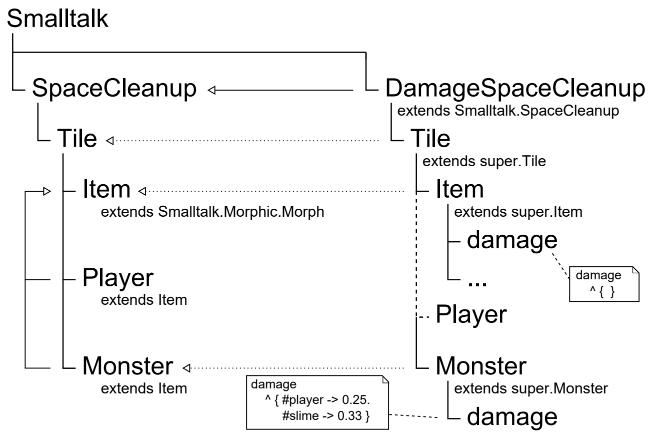


Figure 4: Linearization of Multiple Inheritance

class copy `St.DScu.Tile.Item[St.Scu.Tile.Item]`. Consequently, `St.DScu.Tile.Monster` has the following superclass hierarchy.

1. `St.DScu.Tile.Monster[St.Scu.Tile.Monster]`
2. `St.DScu.Tile.Item`
3. `St.DScu.Tile.Item[St.Scu.Tile.Item]`
4. `St.Morphic.Morph`

The inheritance hierarchy of `St.DScu.Tile.Monster` is a linearization of multiple inheritance from `St.DScu.Tile.Item` and `St.Scu.Tile.Monster`. This mechanism can be generalized to more than two hierarchies and results in a linearization that traverses hierarchies in a zigzag way (Figure 5).

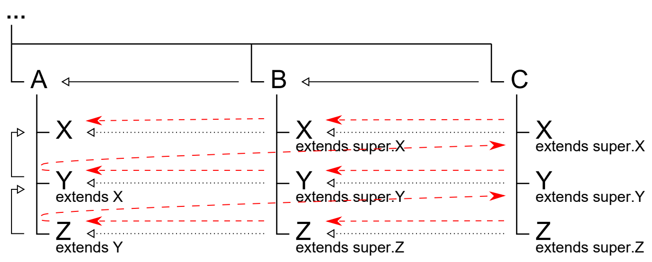


Figure 5: Zigzag Linearization with Three Hierarchies. Dashed arrows indicate the effective class hierarchy for `C.Z`.

5.4 Mixins

Mixins can be used as a form of inter-class code reuse and be seen as class transformers. Given a base class, a mixin application generates a subclass with additional or changed behavior [11]. They are also called *abstract subclasses* [13].

In Matriona, parameterized classes can be used to implement mixins. The parameter of the parameterized class is used as the superclass of the new class in the class generator method. In the following example, two mixins `LogicMixin` and `FilterMixin` should be defined. The methods of these mixins rely on an implementation of `do:` in a subclass of the mixin application (Figure 6).

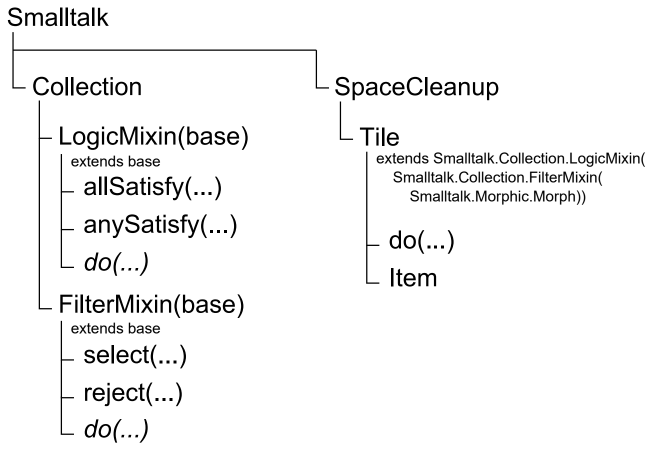


Figure 6: Collection Mixins

The following listing shows how to define the mixin `St.C.LogicMixin:`, which provides the helper methods `allSatisfy:` and `anySatisfy:`, given that the method `do:` for iterating over all elements of the collection is defined.

```
Smalltalk Collection class»LogicMixin: base
   < class >
   ↑ base subclass
```

```
Smalltalk Collection LogicMixin:»allSatisfy: aBlock
   self do: [ :el |
      (aBlock value: el) ifFalse: [ ↑ false ] ].
   ↑ true
```

```
Smalltalk Collection LogicMixin:»do: aBlock
   self subclassResponsibility
```

The following listing shows how `Tile` can be defined as a subclass of `Morph` with the `LogicMixin:` and another mixin mixed in. `Tile` must implement the `do:` method to iterate over all submorphs. In fact, `Tile»do:` overrides the method `LogicMixin:»do:`. Based on `LogicMixin`, the method `canPlayerEnter` ensures that a player cannot enter a tile if it contains a wall as an item.

```
Smalltalk SpaceCleanup Level class»Tile      < class >
   ↑ (Smalltalk Collection LogicMixin:
        (Smalltalk Collection: FilterMixin:
            scope Morphic Morph)) subclass
```

```
Smalltalk SpaceCleanup Level Tile»do: aBlock
   " Iterate over all items on top of the tile. "
   self submorphsDo: aBlock
```

```
Smalltalk SpaceCleanup Level Tile»canPlayerEnter
   " Example usage of mixed in method. "
   ↑ self allSatisfy: [ :item | item isWall not ]
```

The resulting class `Tile` contains two mixin applications and has the following inheritance hierarchy (superclasses).

1. An instantiation of `St.C.LogicMixin:`
2. An instantiation of `St.C.FilterMixin:`

3. `St.Morphic.Morph`

Mixin applications as shown before have two downsides. Firstly, the source code does not reflect the correct order of mixin applications. For example, in `Smalltalk Collection LogicMixin: (Smalltalk Collection FilterMixin: scope Morphic Morph)`, `FilterMixin:` is applied before `LogicMixin:`, but their selectors appear in an inversed order in the source code. Secondly, parameterized classes cannot be easily passed around, because they are not objects or classes, but only exist as "methods" (class accessor methods).

To overcome these shortcomings, a parameterized mixin class can be wrapped in an unparameterized class, as shown in the following listing. `LogicMixin` is now a class and first-class object. Matriona provides a convenience method << for mixin application.

```
Smalltalk Kernel Class»<< aMixin
   ↑ aMixin Mixin: self

Smalltalk Collections class»LogicMixin       < class >
   ↑ Smalltalk Matriona Mixin subclass

Smalltalk Collections LogicMixin class»Mixin: base
   < class >
   ↑ base subclass

Smalltalk SpaceCleanup Level class»Tile      < class >
   ↑ (Smalltalk Morphic Morph
      << Smalltalk Collection FilterMixin
      << Smalltalk Collection LogicMixin)
         subclass
```

Mixins should be subclasses of class `Mixin`. That class provides functionality for pre-mixin and post-mixin hooks, as well as an associative << operation (method) for combining two mixins. For example, pre-mixin hooks can ensure that the base class satisfies certain conditions required by the mixin. Post-mixin hooks can mimic trait composition conflicts [35]. Pre/post-mixin hooks can apply mixins transitively.

6. Related Work

Class nesting and parameterized classes are well-established ideas and have been subject to research in the past years. They are supported by several programming languages. However, these concepts differ conceptually and in their implementation when looking at individual languages (Figure 7). In this section we give an overview of different implementation approaches and compare them with Matriona.

6.1 Class Nesting

Class nesting is supported by a variety of mainstream programming languages such as Java, C++, Python, and Ruby. A nested class always results in a new (nested) namespace. How members from other namespaces are looked up differs among programming languages. In many languages, message sends have implicit receivers, i.e., programmers do not have to specify the receiver, which is similar to Matriona's `scope` keyword. Lookup mechanisms for members differ among languages. For example, Java starts the lookup in the current class and its superclass hierarchy, and then progresses with the lexical scope (enclosing classes). In Newspeak, the lookup starts in the current class, continues with the lexical scope, and checks the superclass hierarchy afterwards [14]. In Python, the lookup always starts at the top level, i.e., members defined in enclosing classes cannot be accessed relatively [12].

In many module systems, including Matriona, classes are only supported as class-side members. Class-side nested classes can be generated during compile time. Matriona is based on Squeak and does not have a edit/compile/run cycle [30]; nested classes are generated when they are accessed for the first time. Instance-side nested classes are supported in Java, but all instances of an enclosing class share the same nested classes. As a consequence, all nested classes have the same superclass, which makes it impossible to use class nesting to implement mixins (Section 6.3). Another use case for instance-side nested classes is the *Adapter* design pattern [19]: a class can expose different interfaces by providing one nested class per interface [8].

Virtual classes [25, 27] can be overridden in subclasses and form the basis for class hierarchy inheritance [17] (also known as *nested inheritance* in the context of Jx [32]). Extensions of the Java programming language have been proposed in the last years to add support for them [4, 41]. In Jx, an overriding class is always a subclass of the overridden class (*class enhancement*).

6.2 Parameterized Classes

C++ class templates [38] are generators for classes and a form of compile-time polymorphism [34] which allows for partial evaluation [42]. Templates must be *instantiated* before they can be used. Template instantiations for different parameters result in different classes. In Java, generic classes are used for type checking reasons. Generic type information is *erased* during compilation and all generic instantiations share the same class [31]. Matriona uses parameterized classes for mixins and external configuration.

In Matriona, class parameters are first-class objects and can be used during class definition. An arbitrary object can serve as a class parameter. Only type names/classes are allowed as class parameters in Java, which is why class parameterization cannot be used for external configuration as described in this paper.

In Beta [26, 27], Matriona, and Newspeak [14], virtual classes can be overridden/specialized in subclasses and be used as a form of class parameterization. In addition, a Newspeak class's "factory method" can take arbitrary arguments, which are only visible inside a class if they are stored in instance variables.

6.3 Mixins

Mixins [13] are a way of sharing behavior among multiple classes. Mixins are "abstract subclasses" [13] and typically

	Smalltalk					Java					
	Matriona	Squeak	VisualWorks	Newspeak	Beta	Java	Jx	MixGen	C++	Ruby	Python
class nesting											
instance-side nesting	(✗)	✗	✗	✓	✓	(✓)	(✓)	(✓)	✗	✗	✗
class-side nesting	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
virtual nested classes	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗
virtual superclass	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗
parameterization											
parameterized classes	✓	✗	✗	✓	(✓)	✓	✓	✓	✓	✗	✗
non-type parameters	✓	n/a	n/a	✓	✗	✗	✗	✗	(✓)	n/a	n/a
parameter as superclass	✓	n/a	n/a	✗	✗	✗	✗	✓	✓	n/a	n/a
namespace											
global	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
hierarchical	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
mixin support	✓	✗	✗	✓	✓	✗	✗	✓	✓	✓	(✓)
traits support	✗	✓	(✓)	✗	✗	✗	✗	✗	✗	(✗)	(✗)
method/class visibility	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓	(✗)

Figure 7: Overview of Programming Language Features

based on single inheritance. Mixin layers have been proposed for collaboration-based designs [7, 37] and implemented and evaluated in C++ [36]. Similarly to the C++ implementation, in Matriona a collaboration (*role*) can be represented as nested class (mixin layer) containing mixins. The single parameter of the mixin layer is either the application of another mixin layer or a class containing all base classes.

Squeak and Pharo support *traits* [35], which have been proposed as an alternative to mixins. Traits are collections of methods and can be applied during class definition. Conflicts between colliding methods from different traits must be resolved manually. External libraries exist for Ruby and Python, which implement traits using metaprogramming. Traits do not integrate as well with Matriona's class parameterization mechanism as mixins, which can be seen "classes parameterized over superclasses" [9].

In some programming languages, mixins are supported through a special syntactical construct [3, 18, 24, 40]. In principal, mixins can also be implemented with an instance-side nested class whose superclass is passed as an argument to its enclosing class constructor. Mixins can also be seen as a linearization of multiple inheritance [5, 11, 33]. Matriona, C++ [36], MixGen [2], and other extensions of Java [1] use parameterized classes.

6.4 Namespaces

Squeak supports namespaces through *environments*. An environment is a dictionary mapping identifiers to objects. Every class has a reference to an environment which is used during compilation time (of methods) to resolve references to global objects. An environment can be imported into other environments, making environments hierarchical. *Name policies* can be used to solve class name clashes. For example, a name policy could add a unique prefix to all imported identifiers. Early ideas for Matriona used Squeak environments instead of message sends for accessing nested classes. However, nested classes would be non-virtual and could not be overridden in subclasses. Furthermore, it is unclear how to implement parameterized classes using Squeak environments.

Newspeak is based on class nesting without a global namespace [14]. Every *module definition* is represented by a top-level class. External dependencies can be loaded from the file system using object graph deserialization; the file system acts as a replacement for the global namespace. Name collisions between classes from two modules cannot occur.

Many languages with support for class nesting have a second namespacing concept for organizing top-level classes and functions. It is called *package* in Java, *module/package* in Python, and *namespace* in C++. VisualWorks is a Smalltalk implementation with nested namespaces. Classes can be referenced using their fully qualified name or imported from another namespace, unless they are declared as "private" [15].

7. Future Work

The implementation of the lexical scope keywords `outer` and `scope` are mostly based on metaprogramming. No changes to the virtual machine were necessary to implement the relative class lookup. The downside of this approach is reduced run-time performance and some special cases in which a message selector collides with a method defined on `ProtoObject`. For example, the message `isNil` will not be picked up by the `doesNotUnderstand:` handler, because `ProtoObject` defines such a method. An alternative implementation approach could introduce a new bytecode for a nested message sends, similarly to Newspeak message sends.

In Matriona, all methods and classes are public. Early ideas for Matriona provided for a visibility control mechanism, where members could be declared as `private` (only accessible from methods within the class), `public` (accessible from every method), or `import-public`. Methods annotated with the latter keyword are accessible from methods defined within a class where one of its enclosing classes or the class itself imports the class (or one of its enclosing classes) containing the method of interest. However, this approach turned out to be unintuitive to Smalltalk programmers.

In Smalltalk, extension methods are methods which belong to a class that was defined in another package. Extension methods are used to add functionality to already existing classes. Extension methods suffer from modularity issues: Method collisions are not handled properly, i.e., the lastly defined extension method overwrites previously defined extension methods or even regular methods with the same name. Context-oriented programming [22] seems to be a promising alternative. Every module could act as a layer which is automatically activated for message sends originating from the module or as long as a method defined within the module is executing. Extension methods would be defined as partial methods in the module. As soon as the control flow leaves the module, the original module behavior would be restored.

Versioning conflicts are a problem particularly in environments where one process/execution environment is used to run a variety of applications. If these applications require the same libraries in different versions, there must be a way to represent multiple versions in one namespace (to avoid class name clashes) and to specify which version to use. One promising idea makes the version number part of the class nesting hierarchy and will be investigated in future work.

8. Conclusion

We presented the Matriona module system for Squeak/Smalltalk. It is based on a hierarchical name lookup mechanism and supports class nesting and parameterized classes. Its implementation does not change the virtual machine, but is based on extensions to the Smalltalk compiler. Matriona promotes modular source code with respect to composability, decomposability, and understandability [29].

Although Matriona is a module system for Smalltalk, we think that other programming languages can benefit from our findings. Our name lookup mechanism can be seen as a generalization of polymorphic method lookup to nested classes, allowing programmers to specify the scope of class refinements. Furthermore, it can be applied to other programming languages supporting dynamic class generation such as Ruby and Python. Enclosing class lookup is the foundation of application inheritance. In a broader sense, we see application inheritance as a future mechanism for customizing the middleware layer of a programming language (e.g., the standard library) that is shared by multiple applications running in the same execution environment or for using a framework. For example, the Eclipse IDE could provide a more direct way for customization using application inheritance as opposed to the current plugin system to support other programming languages. Together with a future versioning concept, application inheritance is also a first step towards an environment where multiple applications are executed in a constantly-running virtual machine.

Matriona is also a module system in the Smalltalk tradition and backward-compatible to Squeak/Smalltalk. Except for class definitions, the notation and syntax of Matriona is a superset of Smalltalk. The Matriona user interface is similar to existing Squeak development tools, but provides functionality for traversing the class nesting tree. One of the design principles behind Smalltalk is to provide a "system [that] should be built with a minimum set of unchangeable parts; these parts should be as general as possible" [23]. Matriona introduces a small number of new features: class nesting, class parameterization, and a new name lookup mechanism. Class nesting and class parameterization use the notion and syntax of methods and message sends. In a sense, this makes the concept of methods in Smalltalk more general.

Acknowledgments

We would like to thank Gilad Bracha for comments on earlier versions of Matriona and Marcel Taeumel for his help in the development of the Matriona user interface.

References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding Type Parameterization to the Java Language. OOPSLA '97, pages 49–65. ACM, 1997.

[2] E. Allen, J. Bannet, and R. Cartwright. A First-class Approach to Genericity. OOPSLA '03, pages 96–114. ACM, 2003.

[3] D. Ancona, G. Lagorio, and E. Zucca. Jam - A Smooth Extension of Java with Mixins. In *ECOOP '00*, volume 1850 of *LNCS*, pages 154–178. Springer, 2000.

[4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173. Springer, 2006.

[5] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A Monotonic Superclass Linearization for Dylan. OOPSLA '96, pages 69–82. ACM, 1996.

[6] D. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented Framework and Product Lines. SPLC1, pages 227–247. Kluwer Academic Publishers, 2000.

[7] K. Beck and W. Cunningham. A Laboratory for Teaching Object Oriented Thinking. OOPSLA '89, pages 1–6. ACM.

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

[9] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *ECOOP '99*, volume 1628 of *LNCS*, pages 43–66. Springer, 1999.

[10] G. Bracha. Inheriting Class. `http://gbracha.blogspot.jp/2013/01/inheriting-class.html`. Accessed: 2015-08-09.

[11] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, 1992.

[12] G. Bracha. On the Interaction of Method Lookup and Scope with Inheritance and Nesting. DYLA '07, 2007.

[13] G. Bracha and W. Cook. Mixin-based Inheritance. OOPSLA/ECOOP '90, pages 303–311. ACM, 1990.

[14] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as Objects in Newspeak. In *ECOOP '10*, volume 6183 of *LNCS*, pages 405–428. Springer, 2010.

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

[16] M. Dixon-Kennedy. *Encyclopedia of Russian and Slavic Myth and Legend*. ABC-CLIO, 1998.

[17] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. POPL '06, pages 270–282. ACM, 2006.

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. POPL '98, pages 171–183. ACM, 1998.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[21] M. J. Guzdial and K. M. Rose. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

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

[23] D. H. Ingalls. Design Principles Behind Smalltalk. *BYTE Magazine*, 6(8):286–298, Aug. 1981.

[24] T. Kamina and T. Tamai. McJava – A Design and Implementation of Java with Mixin-Types. In *Programming Languages and Systems*, volume 3302 of *LNCS*, pages 398–414. Springer, 2004.

[25] O. L. Madsen. Semantic Analysis of Virtual Classes and Nested Classes. OOPSLA '99, pages 114–131. ACM, 1999.

[26] O. L. Madsen, B. Mø-Pedersen, and K. Nygaard. *Object-oriented Programming in the BETA Programming Language*. ACM Press/ Addison-Wesley Publishing Co., 1993.

[27] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-oriented Programming. OOPSLA '89, pages 397–406. ACM, 1989.

[28] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. UIST '95, pages 21–28. ACM, 1995.

[29] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[30] O. Nierstrasz and T. Gîrba. Lessons in Software Evolution Learned by Listening to Smalltalk. In *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *LNCS*, pages 77–95. Springer, 2010.

[31] J. Niño. The Cost of Erasure in Java Generics Type System. *Journal of Computing Sciences in Colleges*, 22(5):2–11, 2007.

[32] N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheritance. OOPSLA '04, pages 99–115. ACM.

[33] M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[34] P. Pirkelbauer, S. Parent, M. Marcus, and B. Stroustrup. Runtime Concepts for the C++ Standard Template Library. SAC '08, pages 171–177. ACM, 2008.

[35] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behaviour. In *ECOOP '03*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.

[36] Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *Generative and Component-Based Software Engineering*, volume 2177 of *LNCS*, pages 164–178. Springer, 2001.

[37] Y. Smaragdakis and D. S. Batory. Implementing Layered Designs with Mixin Layers. ECOOP '98, pages 550–570. Springer, 1998.

[38] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[39] M. Taeumel, B. Steinert, and R. Hirschfeld. The VIVIDE Programming Environment: Connecting Run-time Information with Programmers' System Knowledge. Onward! 2012, pages 117–126. ACM, 2012.

[40] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 3rd edition, 2009.

[41] K. Thorup. Genericity in Java with Virtual Types. In *ECOOP '97*, volume 1241 of *LNCS*, pages 444–471. Springer, 1997.

[42] T. L. Veldhuizen. C++ Templates as Partial Evaluation. PEPM '98, pages 13–18. ACM, 1999.