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 31, 2015

Abstract

We present the concept, the implementation, and an evaluation of Matriona, a module system for and written in Squeak/Smalltalk. Matriona is inspired by Newspeak and based on class nesting: classes are members of other classes, similarly to instance variables.

Top-level classes (modules) are globals and nested classes can be accessed using message sends to the corresponding enclosing class. Class nesting effectively establishes a global and hierarchical namespace, and allows for modular decomposition, resulting in better understandability, if applied properly.

Classes can be parameterized, allowing for external configuration of classes, a form of dependency management. Furthermore, parameterized classes go hand in hand with mixin modularity. Mixins are a form of inter-class code reuse and based on single inheritance.

We show how Matriona can be used to solve the problem of duplicate classes in different modules, to provide a versioning and dependency management mechanism, and to improve understandability through hierarchical decomposition.

Zusammenfassung

Diese Arbeit beschreibt das Konzept, die Implementierung und die Evaluierung von Matriona, einem Modulsystem für und entwickelt in Squeak/Smalltalk. Matriona ist an Newspeak angelehnt und basiert auf verschachtelten Klassen: Klassen, die, wie zum Beispiel auch Instanzvariablen, zu anderen Klassen gehören.

Klassen auf oberster Ebene (*top-level* Klassen) sind globale Objekte. Auf verschachtelte Klassen kann zugegriffen werden, indem eine Nachricht mit dem Namen der Klasse an die entsprechende äußere Klasse gesendet wird. Durch das Verschachteln von Klassen entsteht ein globaler, hierarchischer Namensraum, welcher es erlaubt, Programme modular aufzuteilen. Dadurch kann die Verständlichkeit der Programmstruktur verbessert werden.

Klassen können parametrisiert sein. Dadurch können Klassen von außen konfiguiert werden (eine Form von *dependency management*). Außerdem ergibt sich durch parametrisierte Klassen die Möglichkeit, Mixins zu implementieren. Mixins sind Ansammlungen von Methoden, die bei mehreren Klassen eingebettet werden können, und auf Einfachvererbung abgebildet werden.

Mit Matriona ist es möglich, Klassen mit gleichem Namen in verschiedenen Modulen zu haben. Außerdem stellt Matriona ein Versionierungssystem und ein Verfahren zur Verwaltung von Abhängigkeiten (Bibliotheken etc.) bereit. Darüber hinaus kann mit hierarchischer Dekomposition die Verständlichkeit von Programmtext und dessen Struktur verbessert werden.

Acknowledgments

I owe everything to my cat.

Contents

List of Figures

1. Introduction

This thesis describes the concept, the implementation, and concrete use cases of *Matriona*, a module system for and written in Squeak/Smalltalk. Matriona used to be a popular Russian name and is believed to be the origin of the name *Matryoshka* [19], also known as Russian doll. Matryoshka dolls are wodden dolls that can be nested in each other (Figure 1.1), and are a metaphor for class nesting, the most fundamental concept of Matriona.

Figure 1.1.: Matryoshka doll, also called Russian doll. It consists of multiple wodden pieces that can be nested in each other.

Before explaining the concept, we will elaborate what modularity is and why it is desirable. Then, we will go into more detail about the Smalltalk programming language and explain what modularity means in the context of Squeak/Smalltalk.

1.1. Modularity

What is modularity? According to Myers, "modularity is the single attribute of software that allows a program to be intellectually managable" [37]. This work describes a module system for the Squeak programming language, i.e., a system that should help the programmer in writing modular code. According to Meyer, there are five requirements that a method or system should satisfy to be "worthy of being called *modular*" [35].

Decomposability If a design method supports modular decomposability, it helps the programmer in breaking down big components into smaller one. These subcomponents should be less complex, serve a different purpose, and be mostly independent of each other. Meyer compares decomposability with *division of labor*: every subcomponent does a smaller, in itself less complex part of the job. Decomposability also supports independent development of subcomponents, if these are mostly independent of each other. An example of a method supporting decomposability is top-down design.

Composability If a design method supports modular composability, it helps the programmer in building more complex components out of smaller one. This encourages code reuse; subcomponents do not have to be implemented a second time. An example of composability are libraries. They fulfill a certain purpose, but cannot work on their own. Instead, they were designed to be used in another program, building complex functionality based on smaller pieces.

Understandability If a design method supports understandability, it helps programmers getting an overview and a broad understanding of an application more quickly. This goes hand in hand with decomposability: every subcomponent should be less complex and, therefore, easier to understand than the composed component. This is important to keep software maintainable and makes software development more time-efficient, as it reduces development time, because the programmer has to spend less time understanding the system.

Continuity If a design method supports continuity, it is easier to make changes to the program, since a single change should ideally only affect a single or at least a small number of modules. Every change should be confined to a very small number of modules. This is can be a side effect of decomposability, if done properly [44]. Continuity also makes it easier to extend the behavior of a program.

Protection If a design method supports protection, it helps the programmer writing code where program malfunctions are confined to a single or a small number of modules, instead of spreading across the entire program. For example, every subcomponent should have a well-specified interface and could check input parameters before running the actual implementation.

1.2. The Squeak Programming Language

Smalltalk is a dynamically-typed, object-oriented, class-based programming language and Squeak[1] [30] is a Smalltalk-80 dialect. It was originally developed by

[1] http://squeak.org

Alan Kay, Dan Ingalls, and Adele Goldberg. Dan Ingalls described Smalltalk-80 as a project whose purpose it is to "provide computer support for the creative spirit in everyone." In his article *Design Principles Behind Smalltalk* [31], which appeared in August 1981 in the BYTE Magazine, he mentions some of the most fundamental principles behind the Smalltalk project. Some of these go hand and hand with modularity and can be further supported by a good module system.

- "Personal mastery: If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual." A module system can support understandability of a system by breaking up big components into smaller ones (*hierarchical decomposition*) and hiding irrelevant implementation details.
- "Factoring: Each independent component in a system would appear in only one place." A module system can encourage code reuse by making it easy to share behavior and reuse it in other modules, eliminating code duplication.
- "Modularity: No component in a complex system should depend on the internal details of another component." Through information hiding, a module system can encourage programmers not to rely on implementation-specific behavior. A notion of what is considered a public interface can help keeping modules exchangable and increases understandability, since only the public interface should be sufficient to understand what a module is doing.
- "Good Design: A system should be built with a minimum set of unchangable parts; those parts should be as generic as possible." Consequently, if we are to create a module system for Smalltalk, that system should build on top of a single fundamental concept, and all features and use cases should evolve out of this concept in a natural way, without any special corner cases.

1.3. Outline of this Thesis

The remainder of this thesis is structured as follows. Section 2 gives an overview of modular programming in Squeak/Smalltalk, shows what is possible already, and describes concrete points where we see room for improvement. Section 3 describes the concept of our module system in an abstract way, without diving into notation or implementation details. Section 4 describes the implementation of our module system in Squeak/Smalltalk, as well as corner cases and pitfalls. Section 5 describes concrete use cases and provides examples, implemented in our module system, based on the shortcomings motivated in Section 2. Sections 6 and 7 compare our implementation with other existing systems, and give an overview of the next steps, respectively. Finally, we give a short summary of our concept and implementation in Section 8.

2. Modularity Problems in Squeak

In this section, we describe and evaluate how Squeak can be used to write modular programs at the moment. Based on our observations and programming experience with Squeak, there are three areas where we see room for improvement. For every area, we will describe what the problem is and how it is currently solved in Squeak.

Class-based Modularity In pure Smalltalk, classes are the highest level of modular units. Classes are first-class objects and can be passed around. This functionality can be used to make behavior interchangable and promotes loose coupling. Classes are Smalltalk's way of sharing behavior with a number of objects, i.e., it is a form of code reuse. Squeak also supports Traits, a design method for composing classes out of pieces of behavior (see Section 6.4.3).

Smalltalk is, as most object-oriented and class-based programming languages, amenable to well-established software design patterns [25], making it easier to write maintainable and understandable code.

2.1. Duplicate Class Names

In Squeak, there can only be only one class with a certain name. Whenever the programmer tries to add another class with the same name, a conflict occurs. When source code is loaded into the system with the Monticello source control system or manually, the system asks the programmer if the already existing class should be replaced. As a workaround, it is good practice to add unique namespace prefixes to all class names within an application.

Squeak has packages [39], but these are not used as namespaces. Their purpose is to make it easier to find existing classes (like method protocols). They are also used as deployment units. The programmer does usually not load single classes into the system. Instead, packages (groups of classes) are loaded.

Squeak environments provide a way to have multiple classes with the same name in one image. However, they suffer from poor tool support and do not integrate well with some of the other goals for our system. See Section 6.1.2 for a detailed discussion of Squeak environments and why we did not use them in Matriona.

Example Consider the game Breakout (Figure 2.1, see also Section 5.3). This application uses `Bro` as a prefix for all classes. If we would not use namespace prefixes, generic class names like `Block` or `Ball` would be likely to collide with

```
BroBreakout
 |__ BroBall
 |__ BroBlock
 |__ BroBoundary
 |__ BroBreakout
 |__ BroExplosion
 |__ BroLevelBuilder
 |__ BroLevelStatistics
 |__ BroLevelStatisticsItem
 |__ BroLevelView
 |__ BroLevelWorld
 |__ BroMenuLabel
 |__ BroMenuView
 |__ BroPowerup
 |__ BroPowerupAccelerate
 |__ BroPowerupBall
 |__ BroPowerupDecelerate
 |__ BroPowerupEnlarge
 |__ BroPowerupShrink
 |__ BroRacket
 |__ BroView
 |__ BroWelcomeView
```

Figure 2.1.: Breakout class structure. All classes have the `Bro` namespace prefix and are contained in the package `BroBreakout`.

other classes. On the other hand, if all application and library developers adhere to this convention, it is unlikely that class name clashes occur.

2.2. Dependency Managment

Dependency management describes the task of keeping track of dependencies and ensuring that required dependencies are available within the application in question. We distinguish between two cases of dependency management: internal dependency management, i.e., the application specifies all dependencies, and external dependency managment (*external configuration*), i.e., user of the application specifies dependencies. But before managing dependencies, we need a versioning concept that allows us to represent library versions in an image.

Versioning There are situations when it is useful to have multiple versions of the same library in one image; for example, if there are two different applications installed and both require the same library, but in different versions. Old versions of a library might have bugs that an application has to work around. An application might then not work with a newer library, where the bug is fixed. Furthermore, the public API of a library might have changed with new versions, especially if it is a new major version.

Therefore, we need a versioning mechanism in Matriona, that helps us storing and referencing different versions of the same application or library in one image. Part of this mechanism must be a way to develop new library versions, and a mechanism to reference a certain version.

Internal Dependency Management In this case, every application or library specifies itself which dependencies (and their versions) it depends on. The application effectively maintains the list of dependencies itself. Consequently, the application is coupled to its dependencies and cannot be used with different versions or implementations without changing its source code.

A form of internal dependency managment is *dependency injection*, a mechanism that is heavily used in the Java world [49]. What a class specifies is that it requires some dependency implementing a certain interface, but not what exact dependency it is or in what version. Dependency injection is also known as *inversion of control* [34], because it inverts the control of dependencies: it is shifted from the classes using a dependency to the *injector*, a component that is usually part of the application and knows about all dependencies. The benefit of this approach is that all dependencies are managed at a central position in the application.

External Configuration In this case, the dependency management is delegated to the client/user of an application or library. What the application specifies is that is requires some dependency implementing a certain interface, but not what exact dependency it is or in what version. Conrete dependencies are provided by the client. External configuration is useful for dependencies with variation points, i.e., modules that can be used with different dependencies, based on the use case. For example, an application might want to use a graph library with an adjacency list instead of an adjacency matrix data structure, if it operates on sparse graphs; both implement the same interface. Another example is an image editing library that needs some dependency for exporting images to the file system, but it is up to the client to decide which file format to use.

External configuration is beneficial for modularity, because it supports loose coupling of application and dependencies. This, in turn, promotes understandability, maintainablity, and exchangability (code reuse), because an application cannot rely on implementation details of a loosely bound dependency.

Dependency Management in Squeak In Squeak, there can currently only be one version of a library or application installed at a time. Monticello is used as a source code management system and loads new versions of the source code into an image. Metacello is a package management system (see Section 6.2.4), similar to Maven in Java. Every Metacello package has a configuration class containing a list of external dependencies and internal packages to load for every version, along with the location of an external repository where the packages should be loaded from [48].

External configuration can be simulated in Smalltalk by writing class constructors that accept other dependencies as parameters. These dependencies should then be stored in instance variables and only be accessed using these instance variables. However, this technique has two pitfalls. Firstly, dependencies have to be forwarded to all other classes, resulting in boilerplate code. Secondly, only instance methods can benefit from external configuration, because class methods are shared among instances (configurations) of the class and do not have access to instance variables.

Matriona needs a structured way to reference dependencies. The source code should not be filled with references to external dependencies. It should be easy to replace one dependency with another one or to change the version number of a dependent module.

2.3. Hierarchical Decomposition

Smalltalk packages allow the programmer to group together what belongs together [21]. This is especially useful in big projects with many classes and allows for a form of modular decomposition. Different criterias for modular decomposition have been proposed: e.g., functional decompositon (making every step in the *flowchart* a module) or information hiding [44]. The following list shows some benefits of good modular decomposition.

- Changability (continuity): only few classes are affected when changing a detail.
- Independent development: classes can be developed in parallel.
- Understandability: in order to understand the behavior of a class, it is sufficient to read code within that class.

What we want to achieve is hierarchical decomposition [5], which is in a basic form realized in Java packages, Ruby namespace modules, or Python modules. It can increase comprehensibility of the overall system when it acts as some kind of decision tree that helps the programmer finding a submodule corresponding to a certain functionality in an unknown application.

If the source code is functionally decomposed in a hierarchical way [57], it is also easier to understand single submodules of the system. The reader of the source code might only be interested in a certain level of detail (e.g., no low-level functionality), and then skip deeply nested submodules [61] (information hiding or abstraction). Since in functional decomposition, the purpose of nested modules is usually only to serve their enclosing modules, readers can start off with a high-level idea of what the module is doing by going through the first few levels of nesting, and dive in deeper as needed.

Therefore, one of the requirements for our system is to provide a mechanism for hierarchical code decomposition that is more than just one level deep (Smalltalk packages).

Example Consider the game SpaceCleanup, which is a simple bomberman clone (Figure 2.2, see also Section 5.3). The source code for this game is organized in multiple packages. For example, all items in the game are grouped in the package `SpaceCleanup-Items`. Besides this obvious single-level decomposition, the game is actually already functionally decomposed in a hierarchical way. For example, `ScuLevel` represents a level in the game. A level consists of multiple tiles (`ScuTile`). A tile cannot exist without a level; its sole purpose is to serve `ScuLevel`. Similarly, items always belong to a tile and cannot be used without a tile. All in all, SpaceCleanup is already functionally decomposed, but this decomposition is not fully reflected in the class organization.

- **SpaceCleanup-Core**
  - ScuEventDispatcher
  - ScuGame
  - ScuGameBuildState
  - ScuGameConfigState
  - ScuGameOverState
  - ScuGamePausedState
  - ScuGameRunningState
  - *ScuGameState*
  - ScuGameWonState
  - *ScuMonsterStrategy*
  - ScuMonsterRandomStrategy
  - ScuMonsterToPlayerStrategy
- **SpaceCleanup-Items**
  - ScuBucket
  - *ScuDestructibleItem*
  - ScuFloor
  - *ScuItem*
  - ScuMonster
  - *ScuMovingItem*
  - ScuPickUpItem
  - ScuPlayer
  - ScuPortal
  - ScuSlime
  - ScuWall
  - ScuWater
- **SpaceCleanup-Level**
  - ScuLevel
  - *ScuLevelBuilder*
  - ScuGridPatternLevelBuilder
  - ScuRandomLevelBuilder
  - ScuTile
- **SpaceCleanup-Resources**
  - ScuResourceManager
- **SpaceCleanup-UI**
  - ScuCheatWindow
  - ScuConfigurationWindow
  - ScuControls
  - ScuGameInformation

Figure 2.2.: SpaceCleanup class organization. All classes have the `Scu` namespace prefix and are grouped in five packages, according to their responsibilities.

3. Nested Class Modularity in Squeak

In this chapter, we describe the main concept of this 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 or the underlying virtual machine.

3.1. 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 4.1). 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[1], 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.

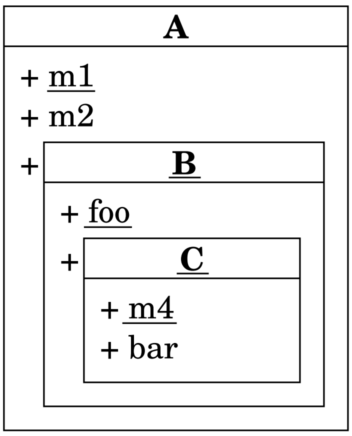


Figure 3.1.: Example: nested classes. A class can have class-side member classes.

Matriona 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 the resulting class and C to the resulting meta class, and finally returns the resulting class. For performance reasons, Matriona also caches the result, meaning that a class is only generated once[2].

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

[1] Squeak also supports *environments*, effectively making it possible to compile methods in the context of another `globals` dictionary. See Section 6.1.2 for more details.
[2] Parameterized classes are an exception.

complicated. We discuss instance-side class generator methods in more detail 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[3]. Consequently, its name is a concatenation of all class names on the path from the top-level class to the class in question.

Notation and Example Figure 3.1 shows an example of nested classes in Matriona. `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`, `C`}. 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.

Top-level classes are called *modules*. All other classes are called *nested classes*. The class in which an other class is nested is called the *enclosing class*.

3.2. Accessing the Lexical Scope

Within a method, it might be necessary to access the lexical scope (i.e., the enclosing classes), in order to send messages to enclosing classes. For example, a method might want to reference a class defined in an enclosing class. For this reason, Matriona introduces new keywords, in addition to `self` and `super`, which are already present in every Smalltalk dialect. This is a point where we extended the programming language. Figure 3.2 gives an overview of all method lookup-related keywords in the system.

3.2.1. `self` Keyword

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

3.2.2. `super` Keyword

[3] It can also be referenced by sending the `class` message to one of its instances

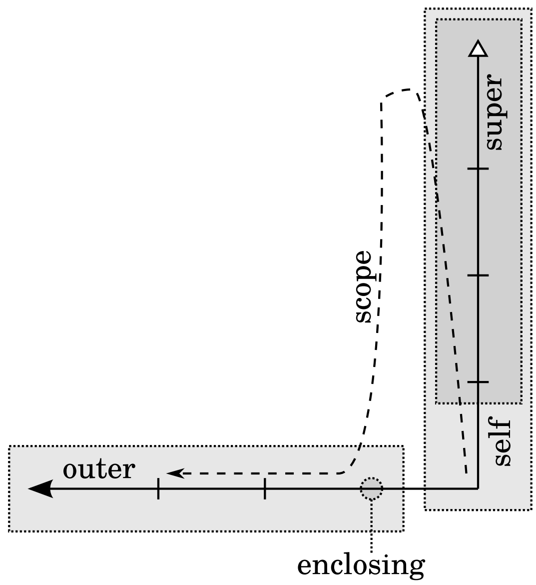


Figure 3.2.: Keywords for superclass and lexical scope access. The lookup starts at `self`, and continues with the lexical scope.

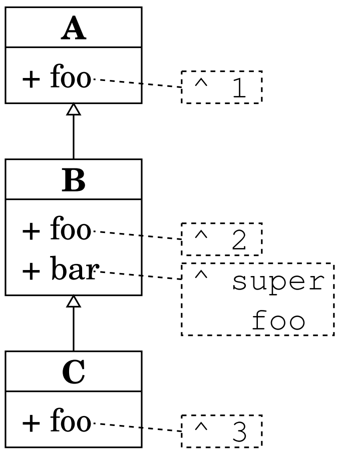


Figure 3.3.: Example: Binding of `super`. The method lookup starts at the superclass of the calling method's class.

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 the 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. For example, in Figure 3.3, `C new bar` returns `1`, because, in `B»bar`, `super` is bound to `A`, even though the receiver `C new` is an instance of `C`.

3.2.3. `enclosing` Keyword

This keyword is an implementation artifact. It can be used for meta programming purposes, but should be avoided in general. It is used to make a message send to the class that contains the current class. Consider, for example, that we want to send a message `foo` to class `A B` within `A class»B class»C class»m4` in Figure 3.1. Either one of the following two statements works in this case[4].

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

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

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 class' lexical scope.

Figure 3.4 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.

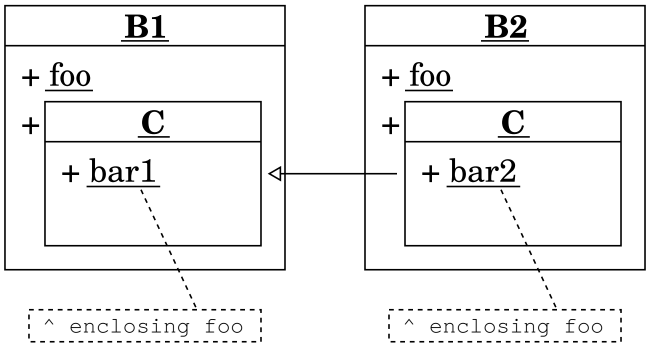


Figure 3.4.: Example: Binding of enclosing. The keyword is bound to enclosing class of the class where the method containing the keyword is contained.

Note, that enclosing can be used for meta programming purposes; however, it should be avoided in general, because it can lead to fragile code that makes too many assumptions about the structure of the class nesting. A later refactoring could then lead to broken code. Probably for the same reason, Smalltalk does not have a super keyword that does the lookup only in the superclass[5] (single-level super). Matriona provides a scope keyword that should be used instead.

3.2.4. enclosing Method

In addition to enclosing, every class in the system has a method enclosing that returns the enclosing class (*owner*) of the receiver, making it possible to send messages to enclosing classes which are more than one level away. If, for example, in Figure 3.1, 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 and for meta programming. The statement enclosing enclosing would be somewhat similar to a super super statement. Arguably, this can result in verbose and complicated code, and is at the very least questionable with regards to the law of demeter.

[5]However, there is a method Class»superclass.

Note, that, in contrast to the `outer` keyword, the message send of `enclosing` in `enclosing` is no longer bound to the lexical scope of the method.

3.2.5. `outer` Keyword

This keyword is used to make a message send to classes in the 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 the second-level enclosing class in the current lexical scope. Eventually, the message is sent to a top-level class, if no other class 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`.

Example Figure 3.5a illustrates how message sends to `outer` are looked up. Consider, for example, that the method `foo` in `A1 B C` calls `outer bar`. Both `A1 B C foo` and `A2 B C foo` call `A1 class»bar` in this case, because `outer` is bound to the lexical scope of the method.

Figure 3.5b shows why it is important that `outer` is bound to the lexical scope. In this example, `D B2` is a subclass of `A B1`. If the `outer` lookup simply traversed the chain of enclosing classes of the (late bound) receiver class, i.e., first lookup in `self enclosing`, then `self enclosing enclosing`, etc., the message send of `bar` would fail in `D B2 C foo`. As another consequence, a nested class might then no longer be able to access a class defined in an enclosing class if the nested class is subclassed and nested in a different enclosing class, since classes are accessed using message sends.

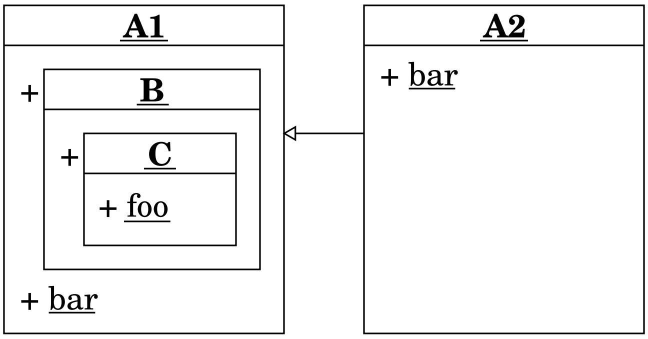
3.2.6. `scope` Keyword

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.

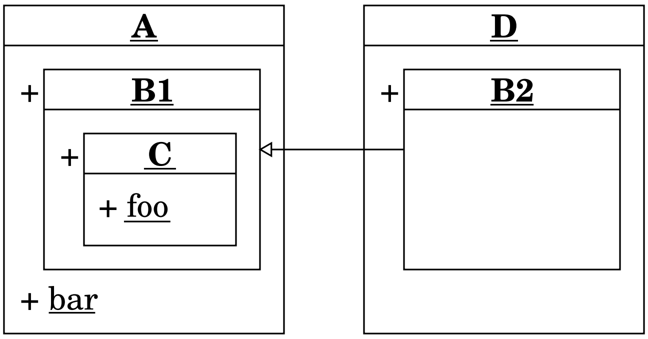
Matriona essentially first looks up the methods in `self`, then in the superclass hierarchy, and then in the lexical scope. This is how the method lookup in Java works, also known as *comb semantics* [10]. Newspeak uses a different lookup: it first looks for a method in the receiver's class, then in the lexical scope, and finally in superclass hierarchy [12].

3.2.7. Implicit `scope` Receiver

In Matriona, references to globals are in fact message sends with `scope` as implicit receiver. This should make it easier for Smalltalk programmers to write code in



(a) Subclassed top-level class



(b) Subclassed nested class with different enclosing class

Figure 3.5.: Example: `outer` keyword. Message sends to `outer` are looked up with respect to the lexical scope of the method, instead of following the chain of enclosing classes (owner hierarchy).

Matriona, 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/keyword[6], 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 3.1. 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 unary method in `self`, the lexical

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

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. Note, that only unary messages can have an implicit `scope` receiver, since we would have to change the Smalltalk syntax, otherwise.

3.3. Parameterized Classes

In Matriona, classes are accessed using message sends. Since messages can have parameters, it seems natural to have parameterized class accessor methods, and, therefore, parameterized classes. All examples shown in the previous sections use unparameterized classes, i.e., class generator methods are always unary. Class generator methods can, however, also have binary selectors or selectors with a higher arity. For memory conservation reasons, these classes are then no longer cached.

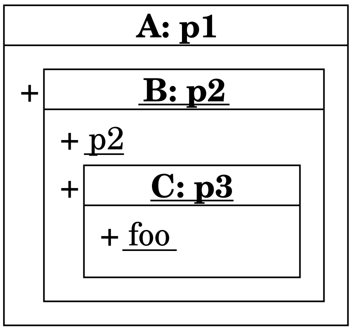


Figure 3.6.: Example: Parameterized classes. A class can have parameters accessible with message sends to `enclosing`.

Parameterized classes can be used to make modules externally configurable or to implement mixins. We will present some conrete use cases in Section 5.

The arguments passed to a parameterized class generator method are considered when a message is sent to `enclosing`. At first, the system tries to send the message to the enclosing class. If that fails, Matriona checks if the selector corresponds to one of the parameter names in the enclosing class' class generator method.

Consider, for example, that method `A: class»B: class»C: class»foo` in Figure 3.6 contains the following statements.

- `scope p3`: method lookup succeeds in `A: class»B: class»C:` and returns the class parameter `p3`.
- `scope p2`: method lookup succeeds in `A: class»B:` and calls the method `p2`, which shadows the class parameter `p2`.
- `scope p1`: method lookup succeeds in `A:` and returns the class parameter `p1`.

3.4. Inheriting Nested Classes

Nested classes are accessed using methods returning the generated class. They are similar to class instance variables in a sense that nested classes belong to the enclosing class object. Therefore, a subclass of the enclosing class has its own nested class, i.e., the nested classes might have the same methods and variables declared, but they are different objects. Nested classes can be overridden in subclasses of enclosing classes, just as regular methods can be overridden. The following paragraphs give an overview of how a subclass of an enclosing class can customize the nested class.

Override with Nested Class A subclass of an enclosing class can define a new nested class. The programmer simply adds a new class generator method with the same selector to the subclass. The superclass will keep using the old nested class, whereas the subclass will use the new one, because the method lookup ends in the subclass when the corresponding class accessor method is found. The new nested class will only have the methods defined for the subclass' nested class and not inherit or copy any methods from the superclass' nested class.

Override with Regular Method A subclass of an enclosing class can replace (override) a nested class with a regular method. The programmer simply adds a new method which is not a class generator method to the subclass.

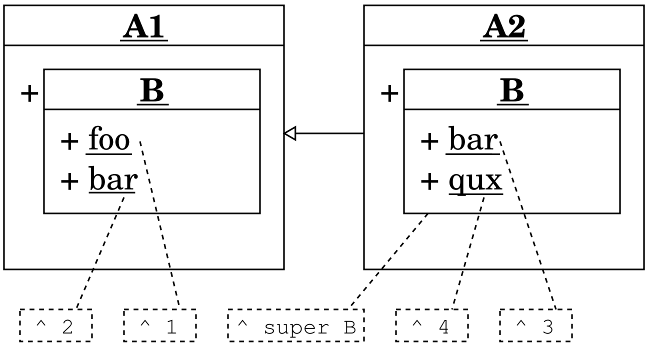
Extend Inherited Nested Class without Subclassing A subclass can extend the inherited nested class, i.e., the nested class in the subclass will have the same superclass as the nested class in the superclass. However, the nested class in the subclass will have all methods defined for the nested class in the superclass and additionally all methods defined for the nested class in the subclass. Duplicate methods will be replaced, similarly to extension methods in Squeak.

Figure 3.7a shows an example of a nested class extension. Class `A2` is a subclass of `A1`, which defines a nested class `B`. Therefore, both classes `A1` and `A2` have a nested class `B`. `A2` extends `B` by perfoming a super call. The following list gives an overview of how the classes `B` behave.

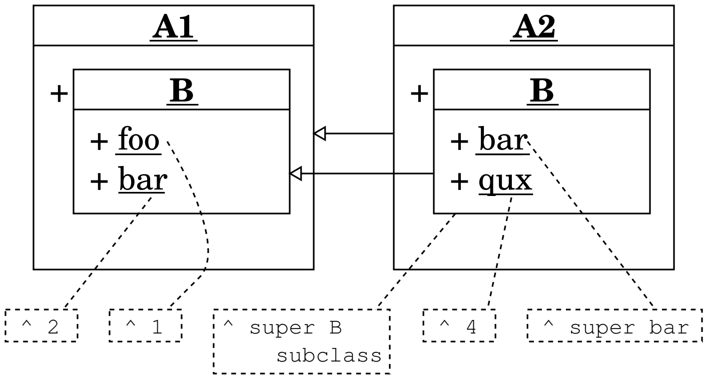
- `A1 B foo`: returns 1.
- `A1 B bar`: returns 2.
- `A1 B qux`: raises `MessageNotUnderstood`, because `qux` is not defined on `A1 B`.
- `A2 B foo`: returns 1, because `A2 B` has all methods defined for `A1 B`.
- `A2 B bar`: returns 3, because that method was replaced in `A2 B`.
- `A2 B qux`: returns 4.

Note, that `A1 B` and `A2 B` have the same superclass, but are different class objects. `A2 B` is *not* a subclass of `A1 B`. When `A2 B` is invoked for the first time, Matriona first generates the class `A1 B` (because of the `super` call) and caches it for `A2 B`[7]. That class is then *reinitialized* according to `A2 B` (without making a subclass), i.e., all methods defined for `A2 B` are added. A subsequent call to `A1 B` will not return the previous generated and extended class for `A2`, because the class cache works on a per-receiver basis.

[7] Caches are receiver-specific.



(a) Extending inherited nested classing



(b) Subclassing inherited nested classing

Figure 3.7.: Example: Extending and subclassing nested classes. Subclassing inherited nested classes leads to parallel class hierarchies.

Also note, that if we actually wanted to extend `A1 B` and alias it as `A2 B`, which is technically similar to an extension method in Smalltalk (see Section 5.8), then `A2 B` should be defined as `^ A1 B`, because the receiver of the message `B` will then be `A1` instead of `A2`.

At the moment, there is no way to add additional instance variables or class variables to an extended nested class, because the class definition (containing the definition of variables) is done in the `super` call.

Subclass Inherited Nested Class A subclass can subclass the inherited nested class, i.e., the nested class in the subclass is a subclass of the nested class in the superclass. Effectively, this results in a parallel class hierarchy. The nested subclass can override methods and use `super` to call methods in the nested superclass.

Figure 3.7b shows an example for subclassing a nested class, which is similar to Figure 3.7a. Note, that `A2 B` is now a subclass of `A1 B` and `super` calls in `A2 B` now start their lookup in `A1 B`. The new subclass `A2 B` behaves like the class in the previous example, except for `A2 B bar`. That statement returns 2, because the `super` call invokes `A1 class»B class»foo`.

4. Implementation

In this chapter, we present the implementation of Matriona and explain briefly how the system is used. Larger examples and concrete use cases will follow in the next chapter.

4.1. Meta Model for Nested Classes

Matriona 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 be applied to already existing instantiations of the model, 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 an instance of `Metaclass` (Figure 4.1).

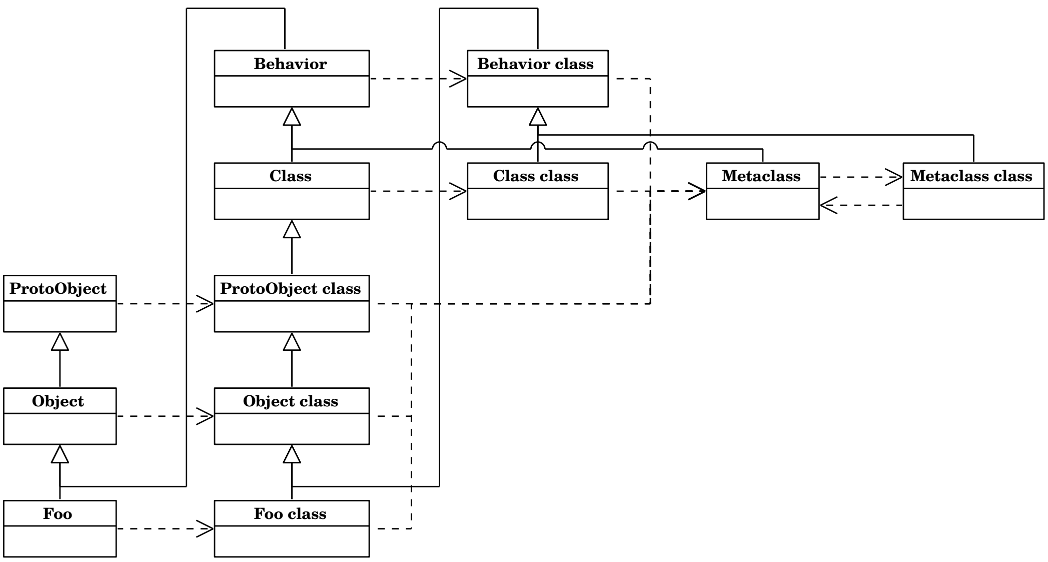


Figure 4.1.: Squeak class model. Every class is an instance of its meta class. Meta classes are instances of `Metaclass`. Meta classes and non-meta classes form a helix [17], connecting the meta class hierarchy with the non-meta class hierarchy.

Matriona 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 in our system for two reasons.

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 along with their meta class objects.

Secondly, Smalltalk supports defining methods on the instance side and on the class side. Consequently, we do not only need to generate class objects 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 different `Metaclass` classes, generating their corresponding meta classes. In some programming languages, the instance-of chain carries on infinitely; Ruby is an example [46]. 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 store information about nested classes and to generate new classes on the fly. We use our own simple meta model instead.

Nested Classes Meta Model Figure 4.2 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 to 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 on their own.

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[1] method dictionary. This method specification determines (when executed in the running system) to which class the methods will be added (*target class*).

[1] The parent of a class specification is the class specification of the enclosing class.

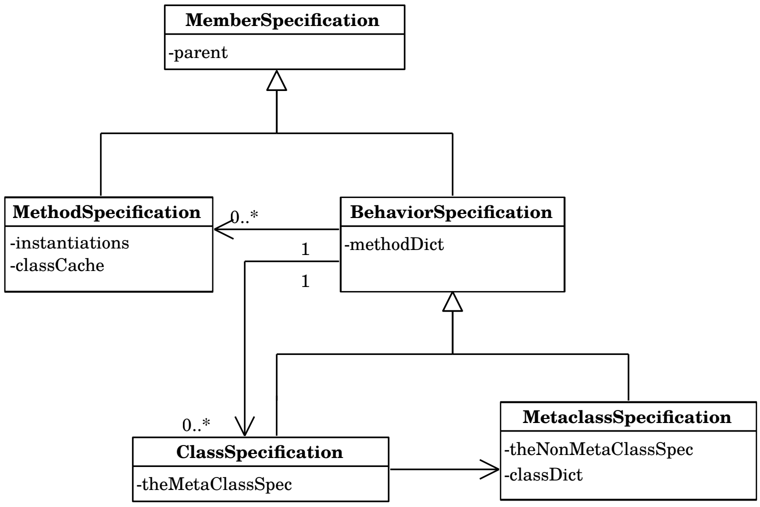


Figure 4.2.: Meta model for nested classes in Matriona. Class specifications are containers for method specifications. Meta class specifications can have additional nested class specifications.

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 target class' meta class. Consequently, there is no method specification in the parent for a meta class.

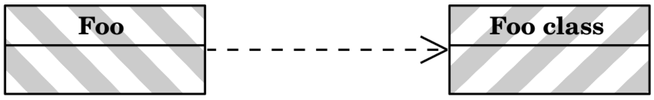
Meta classes can have nested classes of their own. For every class defined in a meta class, there is a corresponding method specification present in the method dictionary (class generator method). The class dictionary contains class specifications for nested classes.

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 bytecode must be generated for different target classes: for example, instance variable reads and writes are compiled to parameterized[2] `pushRcvr:` and `popIntoRcvr:` bytecodes, where instance variables are referenced with their index[3]. In addition, the `outer` and the `enclosing` keyword must be bound to different method literals, depending on the lexical scope of the method.

Example Figure 4.3 shows an example of a class specification for a class `Foo`. There is a class specification for `Foo` and a meta class specification for `Foo class`. The enclosing class of `Foo` is not shown in the UML class diagram part. It is, however, shown in the meta model, because the enclosing class specification has a method specification corresponding to the class specification of `Foo`.

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



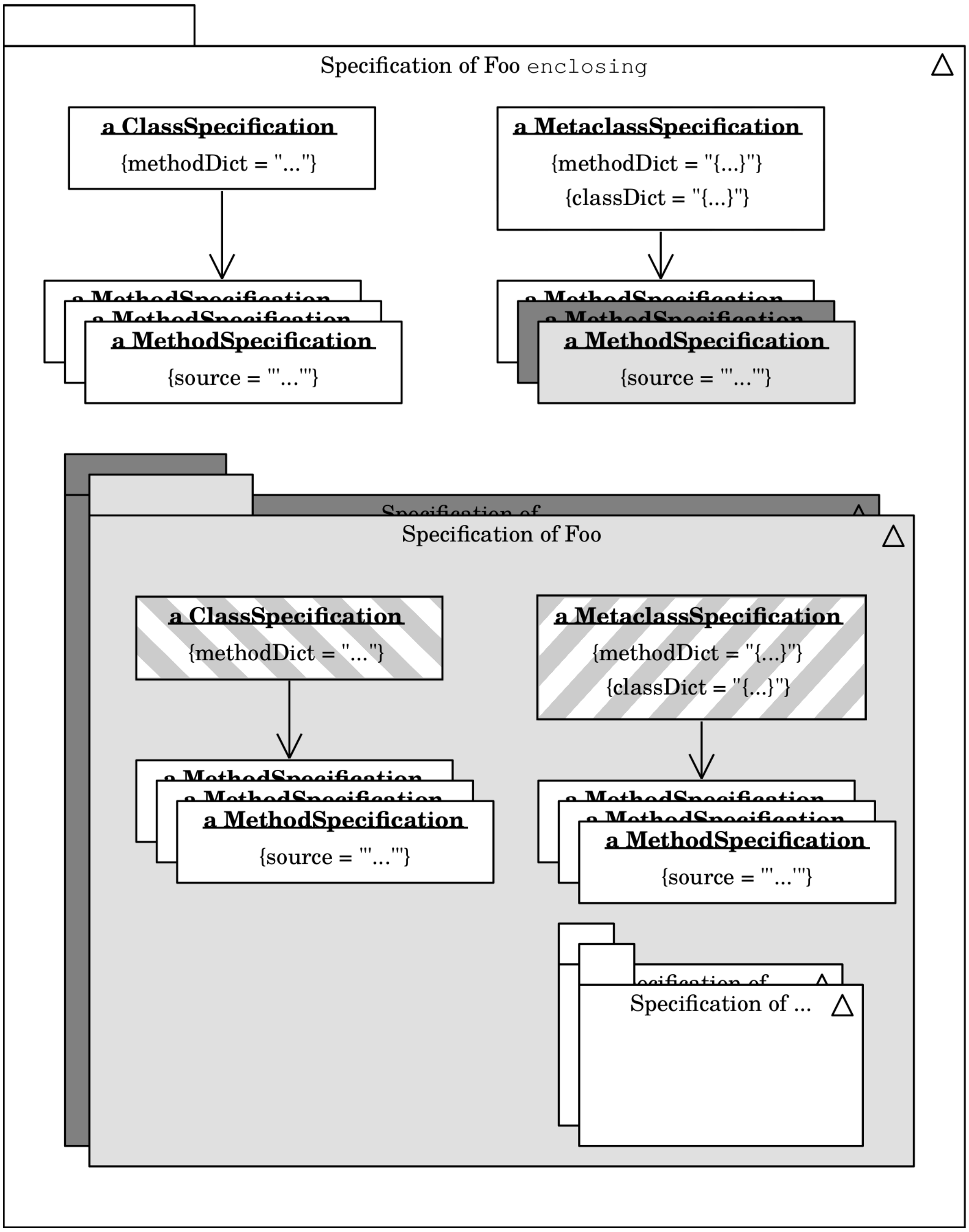


Figure 4.3.: Example: Meta model. There is a class specification for `Foo` and a meta class specification for `Foo class` (same pattern). `Foo`'s enclosing class has a method specification (light gray color) defining the class to which the methods are added (target class), in addition to the corresponding class specification.

4.2. Meta Model Instantiation

The class specifications and meta class specifications described in Section 4.1 are not class objects or meta class objects. These specifications can be instantiated, producing class objects and meta class objects. This section gives an overview of how Matriona generates Smalltalk classes based on class specifications. We distinguish between *class definitions* and *class extensions*. In the former case, the nested class is a newly-generated subclass. In the latter case, an already existing class is extended (extension methods) or aliased.

4.2.1. Class Definitions

In this subsection, we consider the most likely case that a brand new class is defined as a nested class, i.e., the nested class is a *class definition*. Figure 4.4 illustrates how the system generates and initializes a class (class specification instantiation).

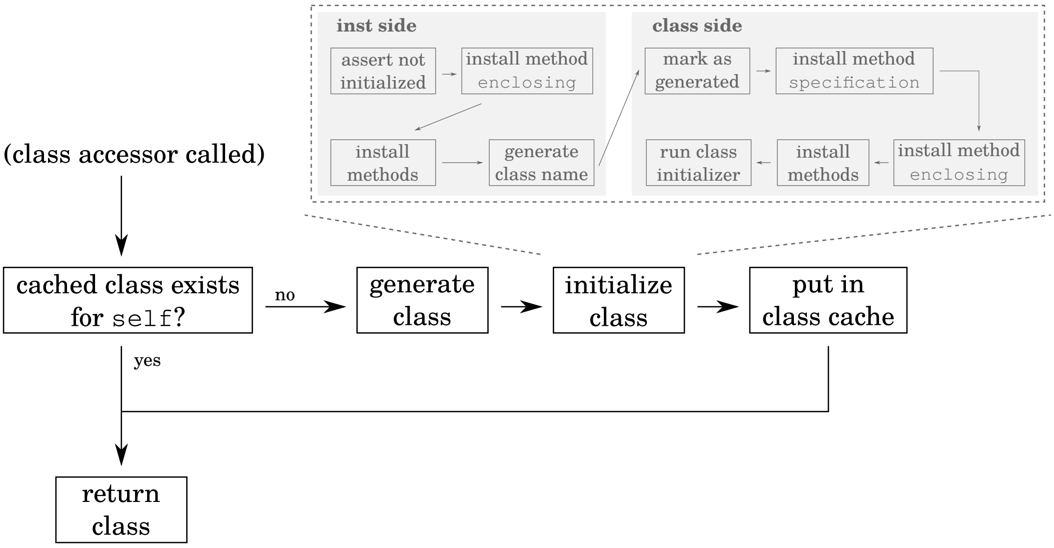


Figure 4.4.: Nested class definition initialization. Classes are generated lazily and initialized using a class specification and a meta class specification.

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 is 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[4]. 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 (bound as a method literal). Note, that the enclosing class cannot be stored in an instance variable of the nested class, because `enclosing` should be early bound

[4]The class generator method can return any class object, but we consider only class definitions.

and `super enclosing` should return a class different from `self enclosing`, in case the superclass of the nested class has a different 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 on `Class`. Note, that every class object is an instance of its meta class, which is a subclass of `Class` (Figure 4.1).
4. Add a marker method to the meta class to mark it as generated. This makes it easy to check if a class is an ordinary (legacy) Smalltalk class or was generated within Matriona.
5. Install `specification` class method. This method returns the class specification (bound as a method literal), which is useful for meta programming purposes. Note, that the specification cannot be stored in an instance variable of the class, because `specification` should be early bound and `self specification` should return a specification different from `super specification`.
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 call the corresponding accessor method, making sure that the class is available when it is needed.

In class definitions, class generator methods always return new subclasses of other classes[5]; the superclass is referenced by calling its 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 methods are called, and if these classes depend on other superclasses, these superclasses are created when the class generator methods call their accessor methods (if they do 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 avoids accidential name clashes with other methods. For example,

[5]See Section 4.3 for syntax details.

if a class is named `Foo`, the class accessor method has the selector `Foo` and the class generator method has the selector `$Foo`.

4.2.2. Class Extension

Whenever the class generator method for a nested class returns an already existing class that is already initialized, we call the nested class a *class extension*. Class extensions are useful to extend inherited nested classes without subclassing (see Section 3.4) and to declare extensions methods (see Section 5.8).

The process of class initialization for class extensions (Figure 4.5) is easier than the process for class definitions: there are no `enclosing` methods installed and no new class name is generated. The already existing `specification` methods are modified, such that they return an array of specifications. If a class is extended multiple times, this array contains all corresponding class specifications. The first specification in the array is always the specification where the class was defined.

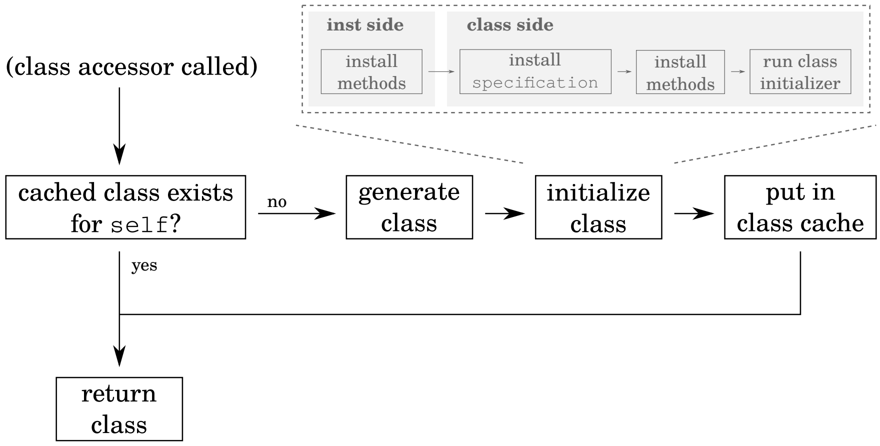


Figure 4.5.: Nested class extension initialization. An already existing and initialized class is initialized.

4.3. Anonymous Classes and Subclass Generation

In Smalltalk, new classes are created by subclassing an already existing class. Squeak has a 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.

Matriona reuses the class builder and adds functionality for creating anonymous subclasses. Anonymous subclasses [20] 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 4.6a 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 [23, 22]. 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'.
```

(a) Subclass notation in Squeak

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

(b) Subclass notation with nested classes

Figure 4.6.: Full notation for creating subclasses. Matriona provides abbreviations (convenience methods) in case no additional instance variables or class variables should be defined (e.g., `subclass`).

Figure 4.6b shows how subclasses are created in Matriona. `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.

4.4. 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` are bound to the enclosing class, 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 references to all enclosing classes in the lexical scope, and contains a `doesNotUnderstand:` handler, that forwards messages to the enclosing classes. If the enclosing class does not understand the message, the message is forwarded to the next enclosing class[6]. If at some point, a top-level class without an enclosing class is reached, the handler looks for an entry in the `globals` dictionary with the message's selector.

As an example, let us assume that we have classes nested as shown in Figure 3.1 and that all following message sends to `outer` happen in `A class»B class»C class»m4`. See Figure 4.7a for a visualization of the lookup.

- `outer foo`: lookup in `enclosing at: 1` (class `A B`) succeeds.
- `outer B`: lookup in `enclosing at: 1` fails, but lookup in `enclosing at: 2` (class `A`) succeeds.
- `outer A`: lookup in `enclosing at: 1` and `enclosing at: 2` 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`[7]. Looking up methods in the class of the method under compilation is not sufficient, because that method might be overridden in a subclass[8]. Therefore, we have to construct a `LexicalScope` object at runtime (instead of compile time) and pass it two objects: the array of all enclosing classes (contained in `outer`) and `self`.

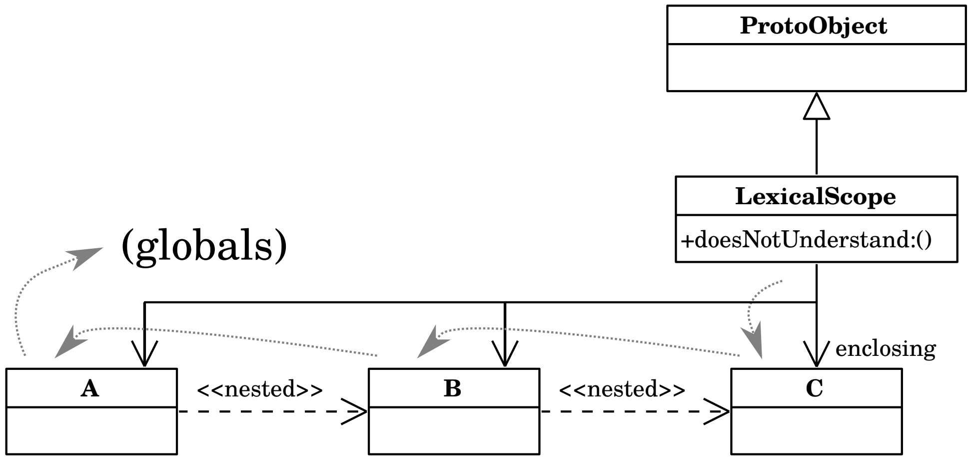
Figure 4.7b shows how the `scope` lookup works in a slightly modified example. Just as in the previous example, we assume that all message sends happen in `A class»B class»C class»m4`. However, `m4` is invoked on class `D`, which is a subclass of class `A B C`. Therefore, `self` is bound to `D`.

- `scope bar`: lookup in `self` succeeds: method `D class»bar`.
- `scope foo`: lookup in `self` fails, but lookup in `enclosing at: 1` (class `A B`) succeeds.

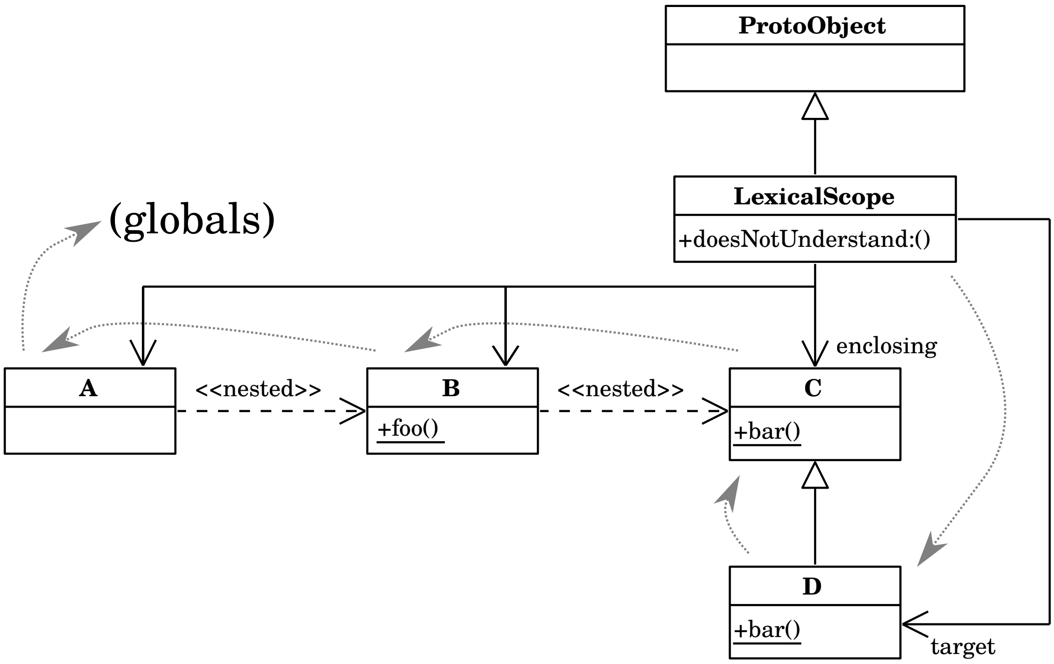
[6] The lexical scope of a method can only be determined by analyzing the structure of the meta model. For more details, see Section A.1.

[7] If `scope` is used in an instance method, the lookup starts at `self class`.

[8] `self` sends are late bound.



(a) `LexicalScope` for `outer` keyword



(b) `LexicalScope` for `scope` keyword

Figure 4.7.: Example: Method lookup using `LexicalScope`. Message sends to `scope` have an additional `target` object involved in the lookup, which is the receiver class in the context where `scope` appears in the source code. All association arrows actually reference the class object.

- The lookup for all other examples listed for `outer` (previous paragraph) yields the same result in this example.

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: outer`. 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.

4.5. 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 of the class is returned. For this reason, every class specification with a unary selector (unparameterized class) has an instance variable `classCache`, which contains cached class objects.

Caching Unparameterized Classes `classCache` is a dictionary mapping enclosing class objects to nested class objects. Every unparameterized class object can only have one instantiation. However, consider, for example, the situation in Figure 3.7b. In this case, `A2` caches two instances of `B`: one for `A1 class»B` (created and cached during the `super B` call) and one for `A2 class»B`. The former one contains only the methods defined in `A1 class»B`. The latter one is a subclass of the former one and contains all methods defined in `A2 class»B`. In this example, the corresponding class specification for `A2 class»B` has a class cache mapping `A1` to the former one and mapping `A2` to the latter one.

Parameterized Classes 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 4.8).

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 (or other globally reachable objects), no class would ever be garbage collected, because `SmallIntegers` are represented as tagged objects in Squeak [6, 43]. If parameterized classes are used as mixins, this is arguably less of a problem, because 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 4.9). We argue that mixins will most of the time be used in such a

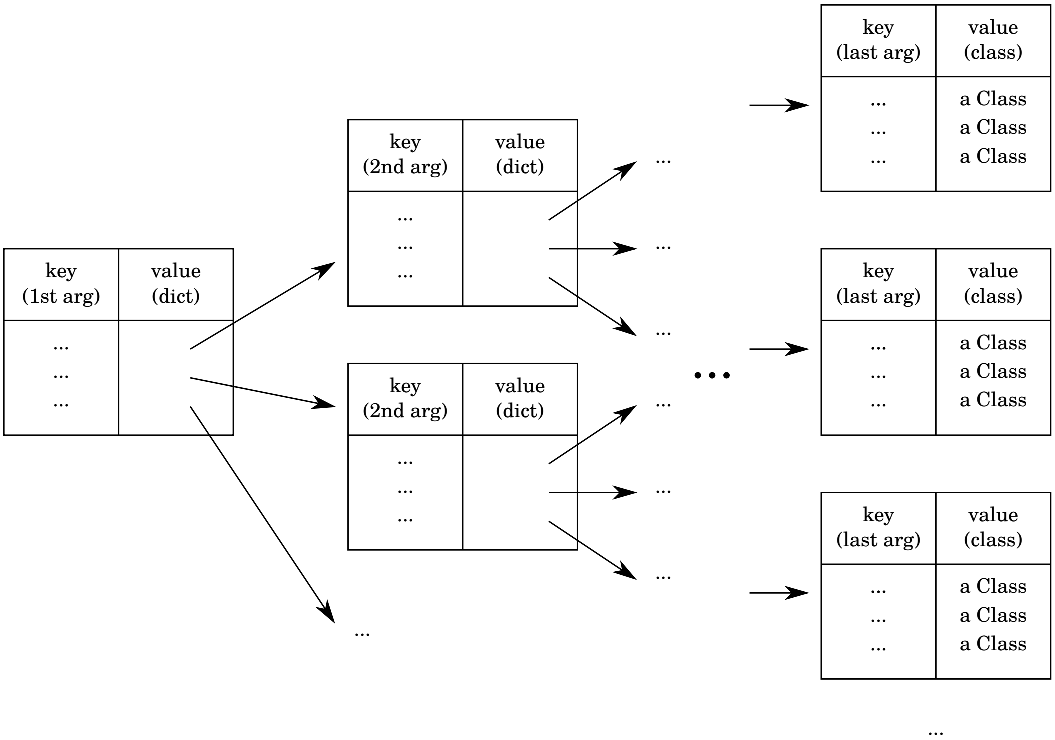


Figure 4.8.: Class cache for parameterized classes. The cache is a nested dictionary data structure, with an additional level of nesting per parameter.

```
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 4.9.: Example: Cached mixin application. The mixin application is uncached, because the mixin is a parameterized class. However, the aliased mixin application `MyCollection` is cached, because it is an unparameterized class.

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.

4.6. 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., existing instances must be migrated to the new class [18]. In that sense, Squeak and many other Smalltalk implementations [47] are different from other programming languages with an "edit/compile/run cycle" [40]: the programmer has the feeling that there is no difference between compile time and runtime.

For this reason, Matriona 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 (`instantiations`). This collection is in fact a `WeakIdentityKeyDictionary` and also used to cache arguments for parameterized classes (see Section 4.6.4). When a class specification is changed, all of its instantiations can be looked up easily and adapted one by one. Note, that this dictionary is different from the class cache, as it holds on to instantiations weakly.

4.6.1. Changing Instance/Class 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 [26, 32].

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.

4.6.2. Changing Instance/Class Variables

This kind of class change is more difficult to handle than method changes. Whenever an instance/class instance variable is added or removed, some methods might have to be recompiled, because instance variables are referenced with indices in the bytecode (see also Section 4.1).

Instance Variables Indexing In Figure 4.10, 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

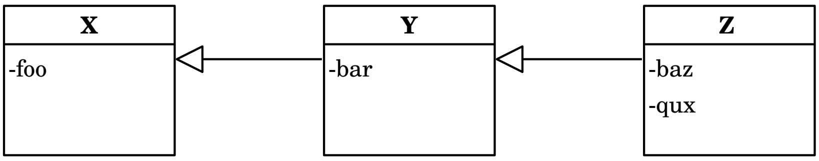


Figure 4.10.: Example: Instance variables indexing. Every instance variable has a zero-based index. The index of inherited instance variables is preserved.

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 referencing these instance variables) `X`, `Y`, and `Z` have to be recompiled. If instance variables in `Z` are changed, only `Z` has to be recompiled.

Definition of Instance Variables What is more interesting is how instance variables are defined in Matriona: they are part of the class generator method (Figure 4.6b). 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. Squeak has the same behavior: whenever an instance variable is changed, methods in the current class and all subclasses are recompiled. Section 4.6.4 gives an overview of the steps necessary for class migration.

4.6.3. Changing Target Class

The target class is the class that is returned by the class generator method. It is usually a new subclass. The superclass of a nested class can be changed by changing the receiver of the `subclass` message in the class generator method. Whenever the target class is changed, the class generator method must be executed a second time and the old class must be migrated to the new one. From a class migration point of view, it does not matter whether an instance variable or the target class was changed. The same class migration process follows.

4.6.4. Class Migration

Whenever an instance variable or the target class in a class generator method is changed, the changed class generator method must be executed a second time and the old class object must be migrated to the new class object. In this subsection, we describe some of the pitfalls in this process.

Class Argument Cache 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, every class specification maintains an argument cache (`instantiations`), mapping instantiations

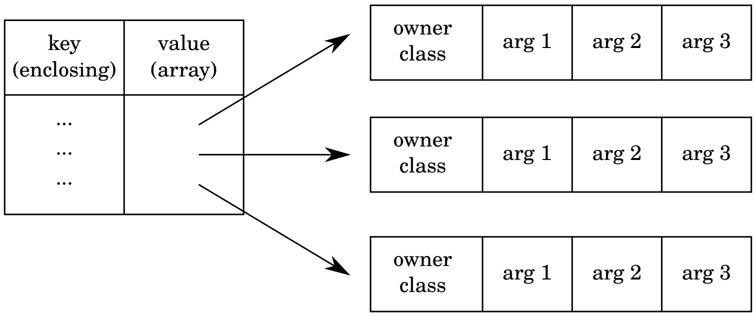


Figure 4.11.: Example: Argument cache. The cache is a dictionary mapping parameterized instantiations to an array of arguments that was used to generate the respective class.

(classes) to an array of arguments and the class that owns the nested class[9]. This argument cache is a `WeakIdentityKeyDictionary` and different from the dictionary data structure shown in Figure 4.8. That class cache would map arguments to instantiations. 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.

Class Migration Invoking the class generator method a second time usually generates a new class[10]. Therefore, all references to the old class have to be 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* [26] for more information.

At this point, we have to distinguish between class definitions and class extensions. We always migrate classes for a class definition. However, not all class extensions are migrated. Consider, for example, the case that the programmer created an alias for `String` and changed that alias to point to `SmallInteger`. In this case, we should not migrate all instances of `String` to `SmallInteger`. The rationale behind this example is that aliases and class extensions can be applied at multiple points throughout the program. Classes should never be migrated when such a class is changed, because other (unchanged) points in the program will be affected.

There is one exception to this rule. Whenever an inherited nested class is extended in a subclass, the class should be migrated, because every subclass has its own nested class that is different from the superclass' nested class, even though the nested class is defined in the superclass and extended in the subclass. Therefore, class extensions are migrated, only if the original class definition took place in the same object as the class extension in question.

[9]The owner class is not necessarily the enclosing class. The enclosing class is early bound, whereas the owner is the class to which the class accessor selector was sent.

[10]There are exceptions: for example, executing the method for an alias a second time does not generate a new class.

```
MyClass
    < class >
    ^ Object
```

(a) Class extension

```
MyClass
    < class >
    ^ Object subclass
```

(b) Class definition

```
MyClass
    < class >
    ^ super MyClass
```

(c) Class extension
(extending inherited nested class)

```
MyClass
    < class >
    ^ super MyClass subclass
```

(d) Class definition
(subclassing inherited nested class)

Figure 4.12.: Example: Class migration. In (b) and (d), a class is defined. Therefore, these classes will be migrated. In (a), a class is aliased. This class will not be migrated. In (c), a class defined in the same object (`self`) where it is extended, so this class will be migrated. We assume that the superclass in (c) actually defines the class and does not extend a class.

Figure 4.12 shows multiple examples. Figure 4.12c is the interesting case. An inherited nested class is extended. We assume, that the enclosing superclass defined `MyClass`. In that case, `MyClass` is defined in the same class as it is extended: it is defined in `super MyClass` and extended in `self MyClass` (same receiver in both cases).

Clearing Class Extension Caches Whenever a class is migrated, all class extensions have to be reapplied. During class migration, `becomeForward:` is used to replace all references to the old class with references to the new class, including class caches. After class migration, all class caches for class extensions for the new class are cleared. When the migrated class is later accessed through an alias or a class extension, all extension methods are reapplied.

Changing Class Extensions Whenever the class generator method for a class extension is changed, Matriona first undoes all changes to affected classes, i.e., it removes all methods that were added or replaced. It does at the moment not restore replaced methods. Colliding extension methods are a known problem in Smalltalk. Other techniques have been proposed, but are out of scope for this thesis (see Section 7.6). After changes to to affected classes have been undone, Matriona clears the class cache. Therefore, all class extensions are reapplied when the class is accessed again through the accessor method for the class extension. In case the class extension is an extension of an inherited nested class, the migration process takes place as previously described.

Overview Figure 4.13 gives a high-level overview of the entire class migration process.

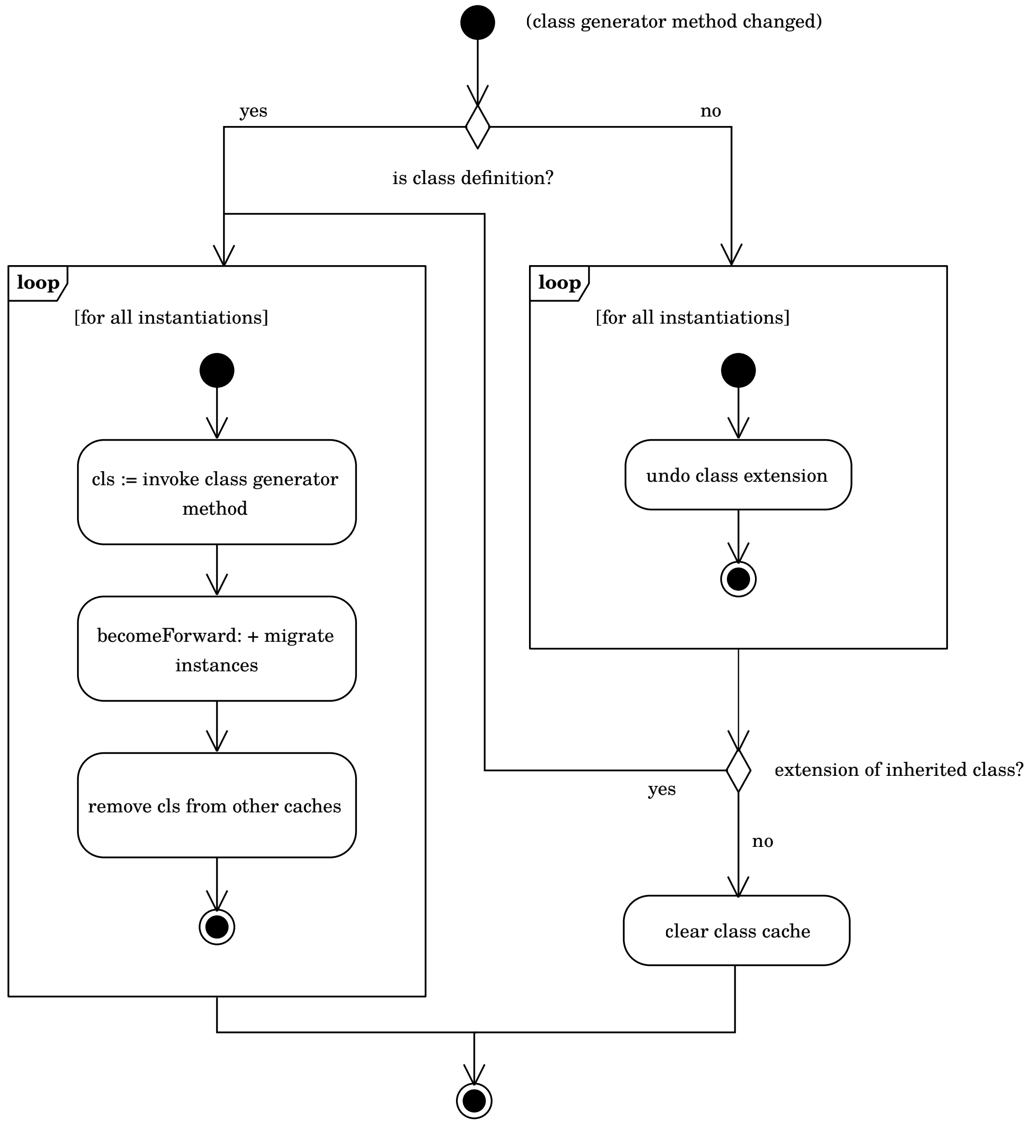


Figure 4.13.: High-level overview of the class migration process. Classes and instances are only migrated if the nested class is a class definition or a class extension of an inherited nested class.

4.7. Integration in Squeak

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

4.7.1. Module Repository

At the moment, there is a separate *module repository* for Matriona. 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 written in Matriona should be listed in the Smalltalk `globals` dictionary, replacing system classes with their counterparts written in Matriona, which would also make the module repository obsolete.

4.7.2. IDE Support

Matriona 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 [55], a framework for dataflow-driven tool development, and shown in Figure 4.14. It supports creating and deleting methods and nested classes, but basic refactoring functionality and functionality such as browsing senders and receivers is still missing.

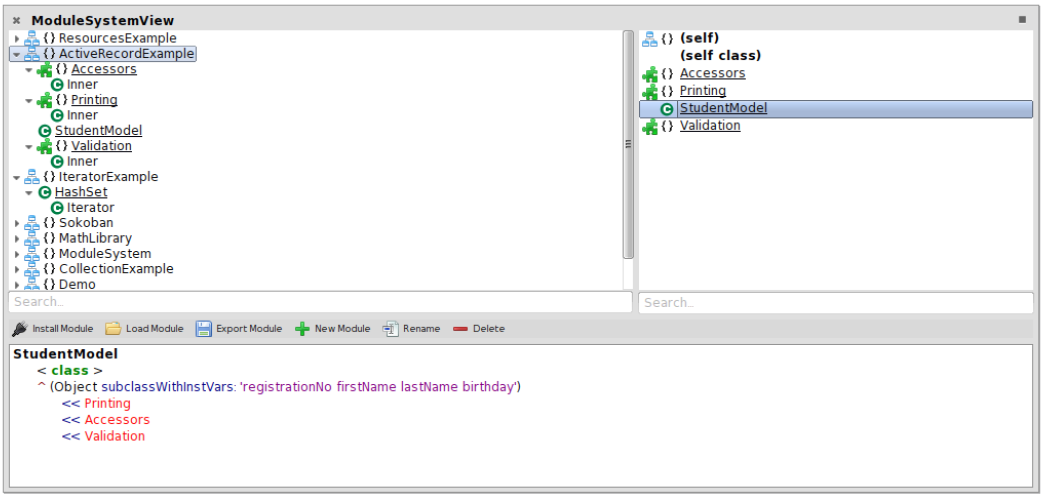


Figure 4.14.: Class Browser for Nested Classes

Matriona is also integrated with the Squeak workspace and the test runner (Figure 4.15). 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.

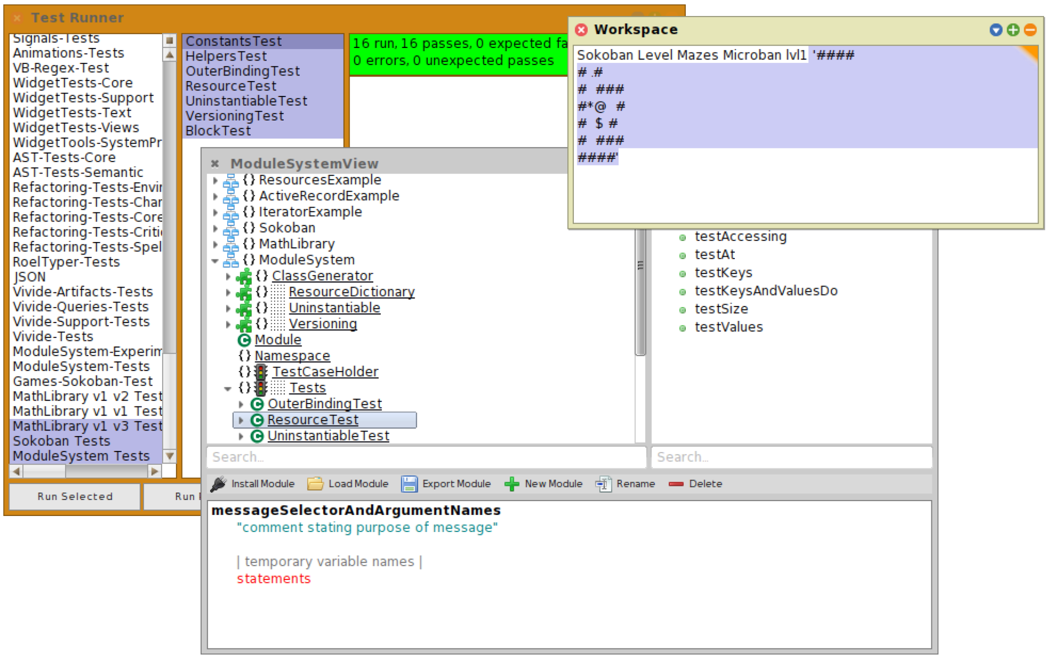


Figure 4.15.: Integration in Squeak

4.7.3. 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 with keywords that were introduced with Matriona, 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 4.4). 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 Matriona.

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

4.8. Source Code Management

Squeak uses Monticello as a source code packaging tool. Monticello can import and export code on a per-package basis. It can supports file system directories, HTTP URLs, and FTP URLs as repositories (i.e., the place where the code is loaded from and stored at). Whenever the programmer wants to get a new version of the source code, two options are available: packages can be loaded which will overwrite all local changes, and packages can be merged which will preserve local changes and only update new methods and classes. In case of merge conflicts, the programmer has to decide which version to load (old method or new method).

SqueakSource[11] used to be a remote repository for Monticello projects (groups of packages). It is now deprecated and SmalltalkHub[12] is one possible replacement.

Matriona is not integrated with Monticello, because Monticello does currently not support class nesting. Many changes would be necessary in the import/export code, the user interface (e.g., merge window), and the backend repository; for example, SqueakSource allows browsing the code on its website, which does no longer work with class nesting.

Source Code Format Matriona comes with its own import/export functionality and supports as only repository the file system as of now. The exported format is similar to the FileTree[13] format. There is a directory for every class. Inside that directory, there are `instance` and `class` directories storing instance methods and class methods, respectively. For every nested class, the corresponding `class` directory contains a source code file for the class generator method and a directory for the nested class. For every method, there is a file containing its source code with the selector of the method as file name (colons are replaced with dots).

```
SpaceCleanup.st
└─ SpaceCleanup
   ├─ instance
   └─ class
      ├─ Game.st
      └─ Game
         ├─ instance
         │  ├─ initialize.st
         │  └─ startGameWithPlayers..st
         ├─ class
         └─ open.st
```

Figure 4.16.: Example: Source code export

Figure 4.16 illustrates what the exported format looks like. The top-level module is `SpaceCleanup`. It does not have any methods on the instance side, but a regular method `open` on class side, as well as a nested class `Game`. That nested class has instance methods `initialize` and `startGameWithPlayers:`.

[11] http://www.squeaksource.com/
[12] http://www.smalltalkhub.com/
[13] https://github.com/dalehenrich/filetree

Source Code Repository At the moment, we use git and GitHub to store modules written in our system, but any other external source code management system, such as Subversion or Mercurial, can be used. Matriona does only support loading and saving, but not merging. It does also not store metadata associated with methods or classes, e.g., the author of a method or when it was changed. Instead, we rely on the underlying source code management system.

The following list gives an overview of how to load new changes into the system.

1. Export local changes (if any).
2. Get the latest source code from the remote repository (e.g., `git pull`).
3. Resolve merge conflicts on the file system, if any.
4. Import the entire module[14].

The following list gives an overview of how local changes can be stored in a repository.

1. Export the entire module.
2. Get the latest source code from the remote repository (e.g., `git pull`).
3. Resolve merge conflicts on the file system, if any.
4. Send local working copy to the remote repository (e.g., `git commit` and `git push`).

[14]This step only recompiles changed methods.

5. Use Cases

In this chapter, we show how Matriona can be used in applications and describe how it can solve the problems presented in Section 2.

5.1. Avoiding Duplicate Class Names

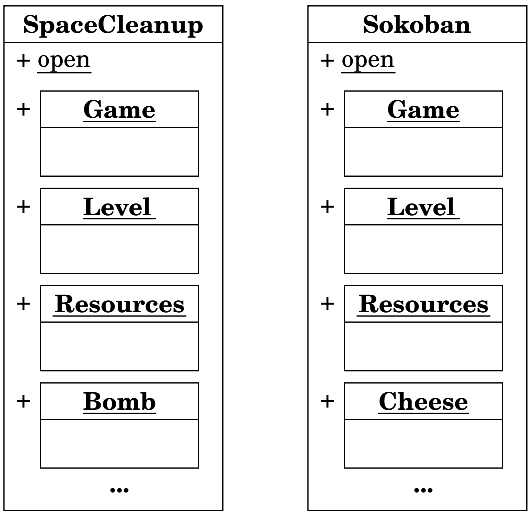


Figure 5.1.: Example: Avoiding duplicate class names. Every nested class has a unique fully qualified name.

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 (SpaceCleanup), providing classes `Game`, `Level`, `Resources` among others. The second game is a Sokoban game, and has three classes with the same name. Without Matriona, this would be a problem: as soon as another class with the same name is installed, the old one is overwritten with the new one.

With Matriona, two classes with the same name can coexist in the same image, as long as they are nested within different classes (Figure 5.1).

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

5.2. 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.

5.2.1. Representing Module Versions

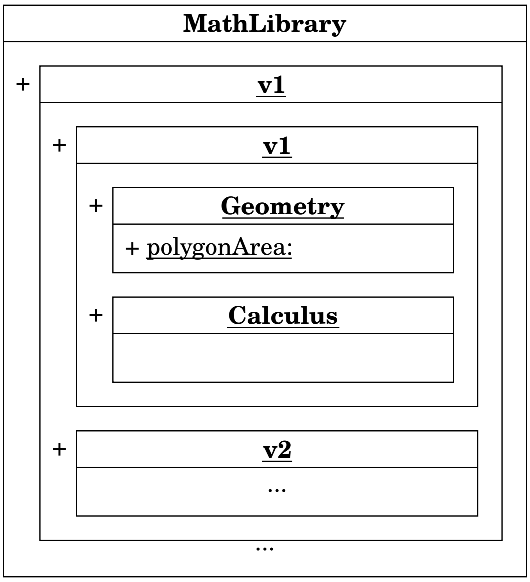


Figure 5.2.: Example: Module versioning. A version is represented by a nested class.

Figure 5.2 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 on 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`.

New Versions In case the programmer wants to add a new version, Matriona will in future releases provide a mechanism to copy a base version and give it a new name. The copied base version can then be modified. Already released versions should not be changed in the future. Instead, a new version should be released.

For development purposes, it is useful to have a special version called `dev`. Programmers can collaboratively work on this version. Once the version should be released, the programmer can make a copy of the entire class and give it a new name: the new version number.

Matriona does at the moment not support delta updates. A new version is always an entire copy of an application, even if just a few methods changed. We might consider delta updates in future releases of Matriona, such that a new version is essentially the previous version and a set of changed methods/classes. Of course, this requires having the entire application history installed in the image.

5.2.2. 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 5.3.: Defining class aliases

Figure 5.3 shows how class aliases 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.

Classes aliases, as described in this paragraph, are similar to import statements in other programming languages, and are a form of internal dependency management.

Helper Methods In Figure 5.2, 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.

In order to get the latest installed version with major version `1`, the programmer could write `MathLibrary v1 latest`. Future versions of Matriona might automatically download and install missing versions, instead of throwing an error message.

5.2.3. Squeak Versioning

With Matriona, it is theoretically possible to run multiple versions of Squeak in one image. The basic idea is that every Squeak version is nested in a different version class. The screenshot in Figure 5.4 shows two versions of the system browser running in the same image. The old system browser lacks many features, such as syntax highlighting or buttons for senders/implementors.

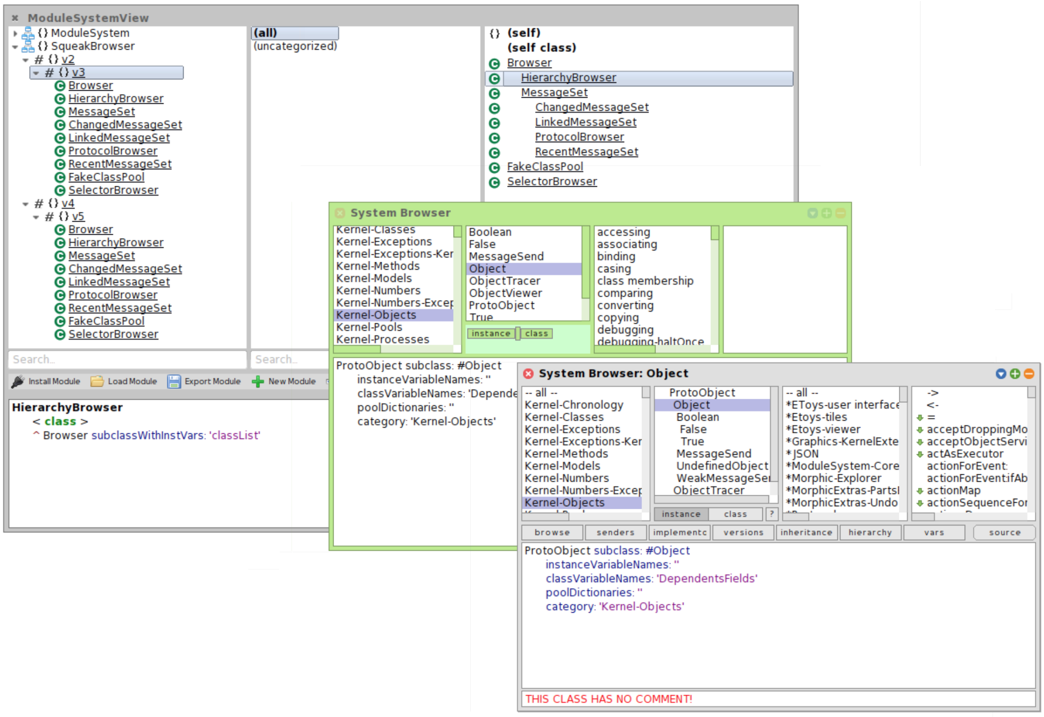


Figure 5.4.: Example: Squeak system browser in different versions. The browser in the front is a modern Squeak 4.5 system browser. The browser in the back is a system browser from Squeak 2.3.

When we tried running bigger system libraries, such as Morphic, in different versions in one image, we encountered the following difficulties.

- Many system libraries are not written in a modular way. For example, they use global state. Whenever global state is stored on other classes or in the `globals` dictionary (e.g. `Smalltalk globals at: #World`), the library circumvents Matriona.

- Some classes should not exist multiple times in one image. For example, `Array` and `String` are classes that the virtual machine knows about[1]. Whenever an argument is passed to a primitive, the virtual machine expects that it is an instance of a class it knows about. Similarly, whenever a primitive returns a value, it is an instance of the version the image knows about.

Future work might investigate how multiple Squeak versions can be run in a single image.

5.2.4. 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 [13]. 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.

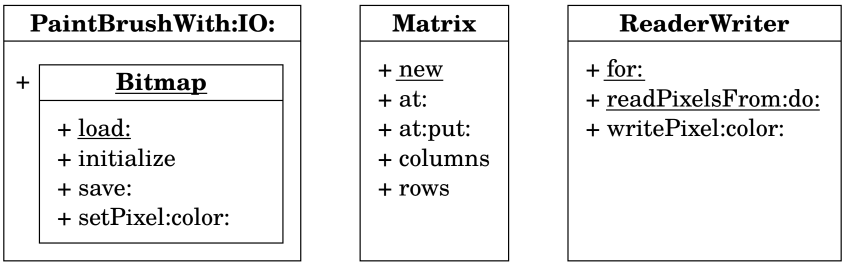
In our system, methods inside parameterized classes can reference arguments provided to the class accessor method. The idea is that, instead of referencing classes in the global namespace, the programmer references these arguments. The user of the module can then decide which exact implementation he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

Example Figure 5.5 shows part of the implementation of a 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 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.

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.

External configuration as shown in this example is similar to a constructor that accepts class objects as parameters and constructs an instance of the class with the

[1] `SmalltalkImage»specialObjectsArray` calls primitive 129 and returns an array of 56 unique special objects that the VM knows about.



(a) Overview of the `PaintBrushWith:IO:` module and dependent interfaces

```
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.
```

(b) Source code for configurable application

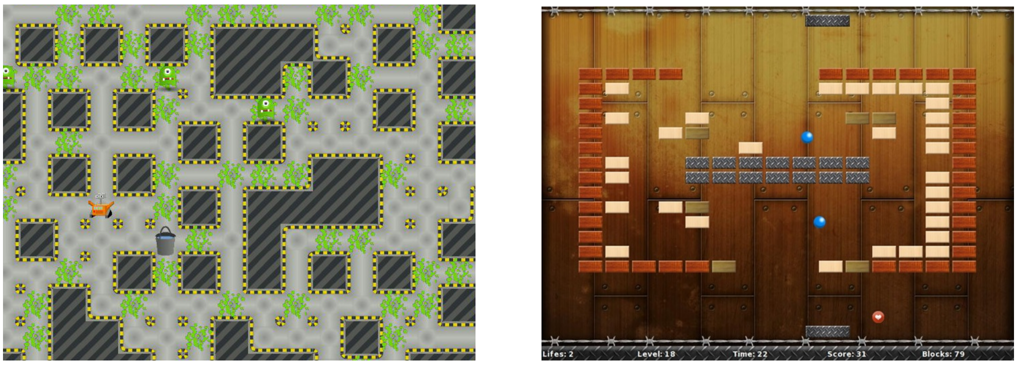
Figure 5.5.: Parameterized classes for external module configuration

class objects stored in instance variables. The difference to this approach is that, in our system, also class methods are bound to the passed arguments, because a new class object is constructed instead of an instance of a class. Furthermore, our system allows creating new nested classes with the argument as a superclass (mixins).

5.3. Hierarchical Decomposition

One of the benefits of hierarchical decomposition is better readability and better understandability. Proper class nesting makes it easier for readers of the source code to understand which classes belong together and to find the class containing a certain functionality.

As an example, we took two simple computer games written in Squeak: SpaceCleanup[2], a bomberman clone, and Breakout[3] (Figure 5.6).



(a) SpaceCleanup **(b)** Breakout

Figure 5.6.: Example games as subjects for hierarchical decomposition

Figure 5.7 shows the original source code of SpaceCleanup and the source code after we introduced class nesting. The original source code already made use of packages, which can be compared to a single level of class nesting. The refactored source code is mostly unchanged, except for class name changes. It is interesting to see is that the class structure is already much more readable by just getting rid of all namespace prefixes. We can not only get rid of class prefixes, but also suffixes. For example, `Builder`, `Item`, and `State` suffixes are omitted. It is now also possible to group classes together that belong together logically. For example, both level builders are nested within `SpaceCleanup Level`. Similarly, all items are nested in `SpaceCleanup Level Item` (which is also the superclass of all items), which makes sense because a level consists of tiles and every tile can have items. An item cannot be used without a tile and a tile is never used outside a level. Note, that there exist two classes with the name `Random`, but they are nested in different classes.

[2] https://github.com/matthias-springer/space-cleanup
[3] https://github.com/fniephaus/BroBreakout

- **SpaceCleanup**
  - EventDispatcher
  - Level
    - *Builder*
      - GridPattern
      - Random
    - Tile
      - *Item*
        - Bucket
        - *Destructible*
        - Floor
        - Monster
          - *Strategy*
            - MoveToPlayer
            - Random
        - *Moving*
        - PickUp
        - Player
        - Portal
        - Slime
        - Wall
        - Water
  - *State*
    - Building
    - Configuring
    - GameOver
    - GameWon
    - Paused
    - Running
  - Resources
  - UI
    - CheatWindow
    - ConfigurationWindow
    - Controls
    - GameInformation

(a) Refactored project structure

- **SpaceCleanup-Core**
  - ScuEventDispatcher
  - ScuGame
  - ScuGameBuildState
  - ScuGameConfigState
  - ScuGameOverState
  - ScuGamePausedState
  - ScuGameRunningState
  - *ScuGameState*
  - ScuGameWonState
  - *ScuMonsterStrategy*
  - ScuMonsterRandomStrategy
  - ScuMonsterToPlayerStrategy
- **SpaceCleanup-Items**
  - ScuBucket
  - *ScuDestructibleItem*
  - ScuFloor
  - *ScuItem*
  - ScuMonster
  - *ScuMovingItem*
  - ScuPickUpItem
  - ScuPlayer
  - ScuPortal
  - ScuSlime
  - ScuWall
  - ScuWater
- **SpaceCleanup-Level**
  - ScuLevel
  - *ScuLevelBuilder*
  - ScuGridPatternLevelBuilder
  - ScuRandomLevelBuilder
  - ScuTile
- **SpaceCleanup-Resources**
  - ScuResourceManager
- **SpaceCleanup-UI**
  - ScuCheatWindow
  - ScuConfigurationWindow
  - ScuControls
  - ScuGameInformation

(b) Original project structure

Figure 5.7.: SpaceCleanup game implementation with/without hierarchical decomposition

Figure 5.8 shows the original source code of Breakout, as well as a refactored version. We did not change the source code, except for class names. All block-related classes are stored as nested classes in the class `Breakout Block`, which is also used to represent regular blocks in the game, that can be destroyed using `Racket`. `Breakout Block Boundary` represents a special block used for the undestroyable border of the game. All power ups are represented as nested classes in the abstract superclass `Breakout Powerup`. The structure of the refactored version is much clearer, because the original version did not take advantage of packages, probably due to the relatively small number of classes.

```
Breakout
├── Level
│   ├── Ball
│   ├── Block
│   │   ├── Boundary
│   │   ├── Explosion
│   │   └── Racket
│   ├── Builder
│   ├── Powerup
│   │   ├── Accelerate
│   │   ├── Ball
│   │   ├── Decelerate
│   │   ├── Enlarge
│   │   └── Shrink
│   └── World
├── View
│   ├── Level
│   ├── Menu
│   └── Welcome
├── LevelStatistics
│   └── Item
└── MenuLabel
```

(a) Refactored project structure

```
BroBreakout
├── BroBall
├── BroBlock
├── BroBoundary
├── BroBreakout
├── BroExplosion
├── BroLevelBuilder
├── BroLevelStatistics
├── BroLevelStatisticsItem
├── BroLevelView
├── BroLevelWorld
├── BroMenuLabel
├── BroMenuView
├── BroPowerup
├── BroPowerupAccelerate
├── BroPowerupBall
├── BroPowerupDecelerate
├── BroPowerupEnlarge
├── BroPowerupShrink
├── BroRacket
├── BroView
└── BroWelcomeView
```

(b) Original project structure

Figure 5.8.: Breakout game implementation with/without hierarchical decomposition

5.4. Mixin Modularity with Parameterized Classes

Parameterized classes can be used to build mixins. Mixins are not a special feature of this system: they are an application of Matriona 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 [11, 3], i.e., it is a class transformator (also called *abstract subclass* [14]).

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 method is a set of instance-side methods and a set of class-side methods. These are the methods that will be added when applying the mixin.

Recursive Mixin Application A mixin can make sure another mixin is applied upon its application. This is done by 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 5.9 shows an example of parameterized classes and how they can be used to build mixins.

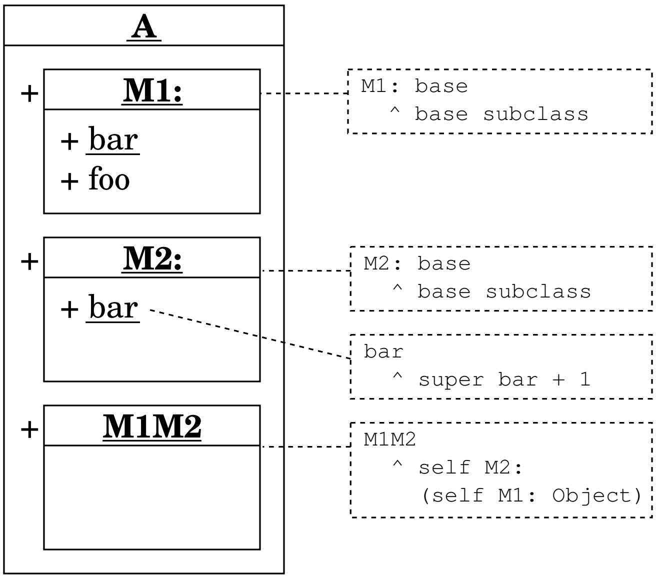


Figure 5.9.: Implementation of Mixins with Nested Classes

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 objects, 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 5.5.

5.5. Unparameterized Class Generator Pattern

The syntax used for mixin application has a few shortcomings. For example, the statements `self A: (self B: Object))` means that mixin `B:` is applied to `Object`, and then mixin `A:` is applied to that result. The problem is that the source code statement does not reflect the order of mixin applications: the statement has to be interpreted from right to left. Another problem is that `A:` and `B:` are parameterized classes and parameterized classe cannot be referenced using an implicit `scope` receiver. Therefore, the programmer always has to write `scope` explicitly.

Both problems can be solved by wrapping the mixin in an unparameterized class and adding a helper method to `Class`. We assume that the name of the parameterized nested class is always `Mixin:`. Then, the helper method « can be defined as shown in Figure 5.10.

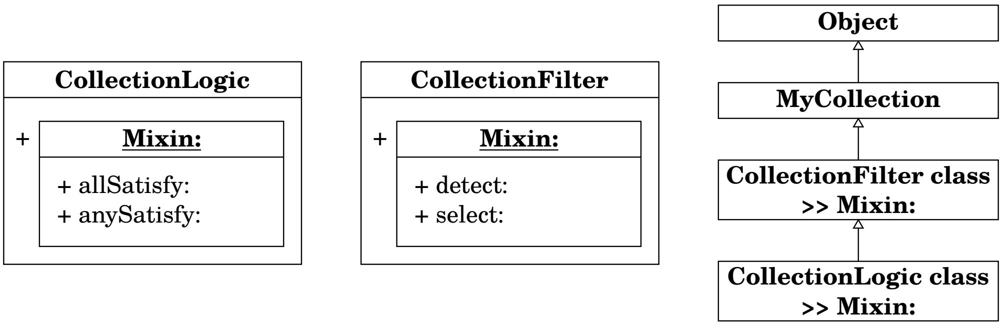
```
Class»« aMixin
   ^ aMixin Mixin: self
```

Figure 5.10.: Helper method on `Class` for unparameterized mixin wrapper classes

Example Figure 5.11 shows two mixins and a base class: `CollectionLogic` is a mixin that adds the methods `allSatisfy:` and `anySatisfy:`, and `CollectionFilter` is a mixin that adds the methods `detect:` and `select:`. All of these four methods can be implemented based on `do:`, which iterates through all elements of a collection. Both mixins can be applied to classes providing at least that method.

`CollectionLogic` and `CollectionFilter` are wrappers around mixins, making it possible to access them like any unparameterized class. When a mixin is applied using the « syntax, the receiver is used as an argument for the `Mixin:` method. Therefore, the name of the actual mixin must always be `Mixin:`, as long as, « is implemented as shown in Figure 5.10. Note, that « inverses the order of receiver and argument, which is why the statement in `FullCollection` can be read from left to right: first `CollectionFilter` and then `CollectionLogic` is applied to `MyCollection`.

Pre-Include Hooks and Post-Include Hooks The unparameterized class generator pattern allows the definition of pre-include hooks and post-include hooks. A pre-include hook is a method defined on the mixin wrapper, which is executed before the mixin was applied, with the base class as an argument. Similarly, a



(a) Class diagram showing mixin and result of mixin application

```
CollectionLogic class»Mixin: base
   < class >
   ^ base subclass

(CollectionLogic class»Mixin: base)»allSatisfy: aBlock
   self do: [ :each |
      (aBlock value: each) ifFalse: [ ^ false ] ].
   ^ true

(CollectionLogic class»Mixin: base)»anySatisfy: aBlock
   self do: [ :each |
      (aBlock value: each) ifTrue: [ ^ true ] ].
   ^ false

" (implementation of CollectionLogic omitted) "

MyCollection»do: aBlock
   " Some implementation "

FullCollection
   < class >
   ^ MyCollection << CollectionFilter << CollectionLogic
```

(b) Definition and application of mixins

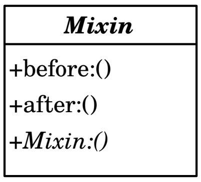
Figure 5.11.: Unparameterized class generator pattern

post-include hook is executed after the mixin was applied, with the resulting class as an argument.

Note, that the programmer can already write arbitrary code at the point where the mixin is applied. However, pre-include hooks and post-include hooks are provided by the mixin itself, and not by the user of a mixin.

```
Class»« aMixin
   | result |
   aMixin before: self.
   result := aMixin Mixin: self.
   aMixin after: result.
   ^ result
```

(a) Mixin wrapper application



(b) Mixin wrapper base class

Figure 5.12.: Implementation of pre-incude hooks and post-include hooks for mixins

Figure 5.12 shows how these hooks are implemented. Mixins with a pre-include hook or a post-include hook should be a subclass of the abstract class `Mixin`. This class provides empty `before:` and `after:` methods which should be overridden in subclasses and contain the pre-include hook and/or post-include hook.

In the previous paragraph, the unparameterized class generator pattern was presented as a tool to increase code readability. With regards to include hooks, this pattern is more: it is necessary to have some kind of wrapping. Include hooks should not be defined on the mixin function itself, because all methods defined on the mixin function are added during mixin application. This is usually not desirable.

5.6. Mixins as Composable Pieces of Behavior

Mixins, as described in the last two sections, are class transformators. Given an existing class, they output a new subclass with additional or changed behavior. In Figure 5.11, we started with `MyCollection`, a class containing only the `do:` method, and added additional behavior to it, resulting in the class `FullCollection`.

Here is another point of view on the same situation: combine behavior from `CollectionFilter` and `CollectionLogic`, add an implementation of `do:`, and call it `FullCollection` (Figure 5.13).

For readability reasons, our implementation provides a simplified notation that combines this kind of mixin application and subclassing (Figure 5.14). This new notation first applies mixins, and creates a subclass of the result afterwards. Note, that the notation reflects the order of subclassing: at first, `Mixin1` is applied, then `Mixin2`, then `Mixin3`, and finally a subclass is created with the additional methods defined on `NewClass`.

The idea of mixins used as composable units of behavior is similar to traits [52]. However, there are some minor differences.

```
FullCollection
   < class >
   ^ (Object
      << CollectionFilter
      << CollectionLogic) subclass

FullCollection»do: aBlock
   " Some implementation "
```

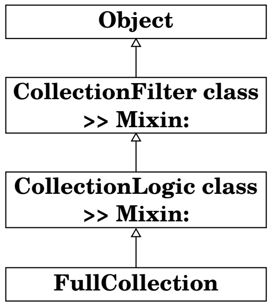


Figure 5.13.: Mixins as composable pieces of behavior

```
NewClass
   < class >
   ^ Object
      mixin: Mixin1
      mixin: Mixin2
      mixin: Mixin3
      subclassWithInstVars: 'foo bar'
      classVars: 'Bar'
      classInstVars: 'Foo'
```

(a) Mixin and subclass template

```
FullCollection
   < class >
   ^ Object
      mixin: CollectionFilter
      mixin: CollectionLogic
      subclassWithInstVars: ''
```

(b) Mixin and subclass example

Figure 5.14.: Simplified notation for using subclassing and mixins

- Mixins are not flat, but create an inheritance hierarchy. E.g., `FullCollection superclass` returns an application of `CollectionLogic class»Mixin:` and not `Object`.
- No explicit conflict resolution is required. Traits raise an error whenever a method is added multiple times and the conflict is not resolved manually in the resulting class. The last applied mixin, on the other hand, overwrites predefined methods with the same name, and allows calling the original implementation using `super`.

5.7. Traits

Traits are similar to mixins and allow behavior to be shared among multiple classes. They can be implemented with class nesting and parameterized classes. The basic idea is to have a parameterized class for every trait, adding trait methods to the target class, but without subclassing it first. Every trait has a pre-include hook and post-include hook. The pre-include hook creates a set of all selectors for the target class. The post-include hook checks if a method was overwritten when it was applied by comparing the set of selectors with the selectors provided by the inner parameterized class. If that is the case, the trait replaces that method with a method that throws an error message, telling the programmer that the conflict has to be resolved. The method provided by the trait is aliased with a selector containing the trait name. In a resolved method (which overwrites the method that throws the conflict error), the programmer can call aliased trait methods.

The idea of traits is not yet fully fledged out at the moment. This section is meant to give a rough idea of what else could be done with Matriona (see Section A.2 for details). The described approach still has a few shortcomings.

- Conflicts errors are thrown when the resolved method is called and not during trait application.
- Adding new methods to Traits that were already applied can break these applications. Every trait is essentially a class extension and adding a new method will add the method to all trait applications, whether or not the method already exists (no conflict resolution).

5.8. Extension Methods

There are cases, in which the functionality of an already existing class in a different module must be extended or changed. For example, this is the case when a bug in another library must be fixed. The programmer typically writes a method that replaces the existing one with the bug. Sometimes, extension methods are also used add additional behavior. For example, the `Morphic` package adds the convenience method `asStringMorph` to `String`. Sometimes it is sufficient to create a subclass of the class in question, and add the changed behavior only to the subclass. However, there are cases where the application code is not in control of instance creation.

An extension method can be added in Matriona by creating a nested class whose class generator method returns an already existing class instead of a new subclass (class extension).

Consider, for example, that we want to add a method `asString` to the top-level class `FullCollection` in Figure 5.14. Figure 5.15 shows how to define a method returning the string concatenation of all elements in the collection.

```
MyApplication class»FullCollection
   < class >
   ^ Repository FullCollection

MyApplication class»FullCollection»asString
   ^ String streamContents: [ :stream |
      self do: [ :each | stream nextPutAll: each asString ] ]
```

Figure 5.15.: Extension methods using nested classes

Note, that it is not possible to add extension methods to all parameterized classes or class specifications. Extension methods can only be added to concrete classes (i.e., class objects). For example, it is not possible, to add an extension method to all classes that are generated by `PaintBrushWith:IO:` in Figure 5.5; only a concrete class object (instantiation) can be extended.

Extension methods are dangerous because changes to existing methods could break other code relying on the old behavior. Numerous alternatives have been proposed, and we provide a brief overview of some of them in Section 7.6.

6. Related Work

In this chapter, we describe how the problems listed in Section 2 were solved in other programming languages, and compare their approaches with Matriona.

6.1. Duplicate Class Names

In this section, we describe how other programming languages address the problem of duplicate class names. For example, this problem can occur if multiple libraries provide classes with the same name. When referencing these classes, it is then no longer obvious which class was meant.

6.1.1. 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 [16]. 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 [29]. 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.

In Matriona, a namespace is an uninstantiable nested class. Instead of imports, Matriona supports aliases. Wildcard aliases/imports are not supported in Matriona. Nested classes can be accessed using message sends instead of extending the Smalltalk syntax with a namespace notation.

Java Packages and Nested Classes The Java programming language has a concept of packages. A package is set of classes, interfaces, and packages, 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 [4].

- *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).
- *Non-static member class:* a class that belongs to an instance of another class, i.e., it is a non-static 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. At first glance, it seems that every instance of a class has its own non-static member class; however, all of these classes must inherit from the same class (which can be resolved at compile time). Effectively, all non-static member classes are the same class, with the only exception that they are bound to different enclosing objects; every class has a field holding a reference to the enclosing instance [24].
- *Anonymous class:* a class without a name. In older Java versions, it was frequently used as a substitute for missing block closures [51]. 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 (without using the `java.lang.Class`) 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, non-static member classes are not allowed to have static members which are not final [27]. Furthermore, a subclass cannot override a member class definition [28]; it can just define its own member class. The difference is that overriding implies late binding, which is not the case in Java.

Jx In *Jx*, Nystrom et al. changed the Java language is such a way, that subclasses can enhance member classes [42]: the new member class overrides the original one and is always a subclass and a subtype of the member class in the superclass. This is equivalent to subclassing inherited nested classes in Matriona. In Jx, there is no way to completely override a nested class or to extend it without subclassing. The subclass relationship is established implicitly, without using the `extends` keyword.

References to classes are late bound in a way that, depending on the context, a reference to a nested class can be a reference to the original nested class (in the enclosing superclass) or a reference to the enhanced nested class (in the enclosing subclass).

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 [2]. 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 is 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 (non-static 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 (e.g., read from the file system) overwrites all previous definitions. This process is often used deliberately in Ruby, in order to change the behavior of a library or an application, e.g., to fix a known bug (*monkey patching*) [1].

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 an `__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 dots as separators.

Modules in other packages can be imported by writing their fully qualified name or using a path relative to the current module [56].

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 a subclass 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.

6.1.2. Squeak Environments

A Squeak environment is a mapping of symbols to global objects. Squeak environments were introduced with Squeak 4.5 [54] and make it possible to have multiple `globals` dictionaries, effectively establishing namespaces. In fact, `Smalltalk globals` is an environment. Every class has an `environment` instance variable determining the environment it belongs to.

Environments establish an association between global identifiers and objects at compile time. For example, if the programmer writes `Object new` in a method, Squeak looks up `#Object` in the environment of the class in which the method is compiled and adds a reference to the result of the lookup in the environment as a method literal. Environments are integrated into the Squeak code base; e.g., the debugger looks up symbols in the corresponding environment when evaluating a code snippet. However, environments lack IDE support at the moment. For example, new environments cannot be created in the system browser as of now.

Name Policies An environment can be imported into another environment. This process copies over all name bindings from the source environment to the target environment. Subsequent changes to the source environment are not reflected in the target environment. In order to solve name conflicts during namespace imports, class names can be changed during import using name policies.

- `AllNamePolicy`: Class names are not changed during import.
- `ExplicitNamePolicy`: The programmer can specify an alias for every class using a dictionary.
- `AddPrefixNamePolicy`: A static prefix is added to every class name during import.
- `RemovePrefixNamePolicy`: A static prefix is removed from every class during import.

Example Consider, for example, that we want to have two applications `SpaceCleanup` and `Breakout` installed in a Squeak image, and both applications provide duplicate class names (e.g. `Game`), as described in Figure 5.1. The programmer has to define separate environments for SpaceCleanup and Breakout, containing only classes from the respective applications, and importing the system environment, such that system classes like `String` or `Morph` are available. The methods in each application can reference `Game` directly, because the corresponding environment does not contain bindings for the other application.

When the programmer wants to use classes from either one of the two applications, the environment has to be imported into the environment of the classes that need to reference the application classes. If both applications are needed, a name policy must be specified to resolve conflicts. References to classes in the application must then be replaced with the resolved class name (e.g., with prefix).

Squeak Environments in Matriona Environments are used in Matriona to implement the method `specification` and the keywords `enclosing` and `outer`. Every class has its own environment and these three identifiers are bound to the corresponding objects. Matriona does, however, not use environments for class lookup for the following reasons.

- Classes are accessed using message sends. Having early-bound classes breaks this notion conceptually, because message sends are always late bound.
- Parameterized classes cannot be early bound (bound at compile time), because instantiations of a parameterized class do not exist until the corresponding class generator method was invoked with the corresponding arguments (which are only known at runtime).
- Lazy class initialization is not possible with environments, making source code imports slower, because all referenced class would be created immediately during the import procedure.
- Early-bound classes make it more difficult to handle source code changes. Consider, for example, that a method references a nested class contained in the second-level enclosing class, and a class with the same name is added to the first-level enclosing class. In this case, Matriona would have to recompile the

method (in order to change the binding) and must, therefore, have a cache of all methods that reference a class.

6.1.3. Newspeak Modules

Newspeak is a programming language that is inspired by Self and Smalltalk. In Newspeak, classes can be nested, establishing a hierarchical namespace. That namespace is, however, not global [13, 8]. All references to external libraries or applications are message sends to a special `platform` object [12], which is constructed by the application developer, and deserialized and imported when the application is installed. Access to external libraries and also the system libraries is only possible through `platform`. Basic language classes like `String`, as well as Squeak classes that have not been transformed to Newspeak classes, are an exception: they can be accessed using `platform blackMarket` [38].

Method Lookup In Newspeak, all names are late bound. Nested classes can be accessed by sending a message to the enclosing object. The receiver of a message is implicit, i.e., the programmer does not have to write `self message`, but just `message`. The lookup mechanism first looks for a corresponding method or nested class in `self`, then in the lexical scope of the method, and finally in the superclass hierarchy [12]. Instance variables can only be accessed through automatically-generated accessor methods.

Matriona supports implicit receivers only for unary messages and looks up methods using comb semantics: the lookup starts in `self`, continues in the superclass hierarchy, and finally traverses the lexical scope of the method. Non-unary selectors cannot have implicit receivers as this would change the Smalltalk syntax. For example, `message` is a valid Smalltalk statement, but `message: #foo` is not. It is, however, a valid Newspeak statement. In Matriona, we encourage programmers to make use of implicit receivers only when a class is referenced. In fact, all message sends with implicit receivers are replaced with message sends to `scope` by the compiler.

Nested Classes In Newspeak, nested classes can be defined on the class side and on the instance side. Since all names in Newspeak are late bound, all classes are in fact two mixins: one mixin for the instance side, and one mixin for the class side. Every class is essentially represented by a superclass statement and two mixins that will be applied to the evaluation of the superclass statement (and its meta class) [9].

6.2. Dependency Management

This section gives an overview of how programmers can use external dependencies in other programming languages. Dependency management describes not only the process of how dependencies are installed and organized, but also how they can be included and referenced in an application.

We first describe three methods for referencing external dependencies. Afterwards, we give an overview of how dependencies are installed, stored, and organized in other programming languages.

6.2.1. Explicit Dependencies

This is the simplest form of dependency management. A dependency is referenced by writing down its fully qualified name in the source code. Consider, for example, that a Java programmer wants to write a Paintbrush application, similar to the example shown in Figure 5.5. This application requires a library for reading and writing picture/image files. An external PNG reader/writer library can be referenced by writing down its fully qualified name, e.g., `ar.com.hjg.pngj.PngReader`[1]. In this section, is it not important how we can ensure that the class `PngReader` is loaded and available (see Section 6.2.4). What is important is that the programmer explicitly referenced the dependency. Therefore, the application is coupled to that dependency. Changing the dependency requires changing the source code of the application. Note, that referencing dependencies explicitly is not possible in programming languages without a global namespace (e.g., Newspeak).

6.2.2. Dependency Injection

An alternative to explicit dependencies is dependency injection. Instead of referencing dependencies explicitly using their fully qualified name (or using an alias), the programmer writes down a list of all dependencies at one central position: the *injector* knows about all dependencies and ensures that clients have access to dependencies when needed. Whenever a dependency is required in the source code, the programmer uses an implementation-independent interface instead of the concrete implementation (if the language is statically typed) and adds a source code annotation. The source code annotation ensures that the system *injects* the dependency [49]. This makes dependency management easier, because dependencies are listed at one central position, whereas they were scattered across the application in the previous example.

Google Guice Guice[2] is a framework for dependency managment in Java. The programmer has to create and define a so-called *module*[3], which binds interfaces to implementations [58]. Consider, for example, that there is an interface `ImageReader` that is implemented by `PngReader`. Figure 6.1 shows how the programmer defines the module binding `ImageReader` to `PngReader` and uses the reader in `Paintbrush`. Note, that `Paintbrush` does not reference `PngReader` directly, but just an abstract

[1] PNGJ is a Java library for reading and writing PNG images.
[2] https://github.com/google/guice
[3] Modules in Guice are not to be confused with modules in Matriona.

interface. The `Paintbrush` class is decoupled from the concrete reader class. The PNG reader class could easily be replaced with a reader class reading a different file format by just modifying the module, as long as the new reader class also implements the interface `ImageReader`.

```
import ar.com.hjg.pngj.PngReader;
import com.google.inject.AbstractModule;
import org.imageformats.ImageReader;

public class PaintbrushModule extends AbstractModule {
   @Override
   public void configure() {
      bind(ImageReader.class).to(PngReader.class);
   }
}
```

```
import org.imageformats.ImageReader;

public class Paintbrush {
   @Inject
   private ImageReader reader;

   public void loadImage(String fileName) {
      /* ... */
      Bitmap bitmap = reader.readFile(fileName);
      /* ... */
   }
}
```

Figure 6.1.: Example: Dependency injection with Google Guice

Note, that the example in this paragraph shows only the very basic functionality of Google Guice. More advanced features are available, for example, ensuring that an injected implementation is a singleton instance. Dependency injection is also used heavily in Java test cases, to ensure that a test uses a mock implementation [60]. Another popular dependency injection framework for Java is the Spring Framework[4].

Seuss Seuss is a framework for dependency injection in Pharo/Smalltalk [53]. Whenever a dependency is required in a class, an instance variable and a corresponding setter method should be added. The setter method must have an `inject` pragma, telling the framework that a dependency must be injected upon instance creation. For example, if a class requires an image reader class as a dependency, the programmer could add an instance variable setter method with the `<inject: #ImageReader>` pragma. The framework allows binding the symbol `#ImageReader` to a concrete object at a different position in the code.

The authors of Seuss argue, that Seuss can help getting rid of static methods, which are often used as accessor methods for *globally visible services*, where the

[4] http://projects.spring.io/spring-framework/

corresponding class acts as a namespace. Seuss can also make test code simpler, because implicit dependencies are resolved and delegated to the injector. Furthermore, the abstract factory pattern becomes obsolete.

6.2.3. External Configuration in Newspeak

In Newspeak, methods cannot access other top-level classes, because there is no global namespace. At the same time, there is no form of dependency injection that would provide dependencies where needed. Instead, Newspeak has the notion of a `platform`, a dictionary-like data structure containing references to all dependencies. During module/class instantiation, all dependencies should be acquired from `platform`, which is passed as an argument in the constructor, and stored in slots (instance variables), so that they can be used within modules [12]. This is necessary because, in contrast to Matriona, class bodies (methods etc.) in Newspeak do not have access to class parameters [9].

A platform object should be bundled together with an application. It is created by the developer of an application and then serialized to disk, together with the source code of the application. Whenever a user installs the application, all dependencies are installed along with the application code. Since there is no global namespace, the platform acts as a sandbox. Different applications and platforms cannot interfere with each other.

Parameterized classes in Newspeak can be used for external configuration of classes in a way that is similar to Matriona. Whenever a class is paramterized and stores its arguments in slots, methods in the current class and nested classes can access these slots, because Newspeak automatically creates accessor methods for all slots. When `scope` in Matriona does the method lookup, it first searches for methods in the class, and then checks if there is a parameter for the class with that name. Afterwards, it continues the lookup in the enclosing class. An implicit receiver lookup in Newspeak immediately finds the corresponding accessor method.

6.2.4. Dependency Installation

In this section, we briefly describe dependency/package management systems for different programming languages.

Metacello Metacello is a package management system for Smalltalk. Add content

Maven A Java class loader loads compiled Java classes into a running virtual machine. It can be used to load classes dynamically at runtime by specifying their names. Maven is a *software project management and comprehension tool*. It stores dependencies in a repository on the file system and loads them using a Java class loader. Maven projects have a `pom.xml` configuration file that contains a listing of all dependencies required by the project. When the project is run, Maven ensures that all dependencies are available in the repository and downloads them from a remote server, otherwise. It then compiles the project and runs it.

Maven is a widely-used tool, not only for open-source projects, but also for enterprise applications. In 2014, the Maven central repository hosted more than 17000 projects and more than 115000 versions in total, amounting to about 265 GB of data [36].

Every Maven dependency declared in `pom.xml` should have a version. A version can be an exact version number (e.g., `[1.1]`) or a version range. For example, if all versions smaller or equal to `1.0` are acceptable, the programmer can write `(,1.0]`. Another example is `[1.0,2.0)`, meaning that all versions between `1.0` (including) and `2.0` (excluding) are acceptable [62]. The way Maven specifies versions is similar to Matriona. However, Maven cannot load more than one version of a library at a time. There would be no way to reference a certain version of a library in the Java code, because all the programmer does is writing down the fully qualified name of a class contained in a dependency. The version number is usually not part of the fully qualified class name. In Matriona, it is.

Maven dependencies are transitive. If `A` requires `B`, and `B` requires `C`, then adding `A` as a dependency will automatically add `B` and `C` as dependencies. The programmer does not have to specify these dependencies explicitly.

RubyGems RubyGems is a package manager for Ruby. Libraries and applications are contained in *gems*. Gems can be installed using the command line tool `gem` and are hosted at a central repository[5]. The programmer has to *require* (import) the package `rubygems` in his application. Afterwards, installed RubyGems can be imported by adding corresponding `require` statements. Specific versions of a gem can be imported by adding a `gem` statement in front of the `require` statement. For example, `gem "extlib", ">= 1.0.8"`, followed by `require "extlib"` imports the library `extlib` in a version that is guranteed to be greater or equal to `1.0.8` [50].

Bundler[6] is a dependency manager for RubyGems. The programmer can add a `Gemfile` to the root directory of an application. All dependencies are automatically downloaded and installed when the programmer executes the command line statement `bundle install`.

6.3. Readability and Understandability

In object-oriented, class-based programming languages, source code is typically structured on multiple levels. Classes are used to group common behavior for a set of objects. Inside a class, methods are used to divide source code into smaller, more managable pieces. In this section, we give an overview of how classes can be structured in other programming languages, in order to increase readability and understandability of source code.

[5] http://rubygems.org
[6] http://bundler.io/

6.3.1. Smalltalk Packages

In Smalltalk, packages are used as deployment units. Usually, the programmer can already tell by the name of a package, what the responsibilities of a certain package are. For example, in Figure 5.7b, all item classes are contained in the package `SpaceCleanup-Items`. Similarly, all UI-related classes are contained in `SpaceCleanup-UI`. Packages make it easier to find a certain class whose name is unknown to the programmer. They also make it easier to understand in what context a certain class is used.

6.3.2. Hierarchical Decomposition

As described in Section 6.1.1, many programming languages such as Java, Python, Ruby, or Newspeak, have a concept of packages, namespaces, and/or nested classes. These concepts allow for a form of hierarchical decomposition. Smalltalk packages allow the programmer to put classes in a certain package, according to their responsibilites. The mentioned concepts make it possible to structure classes on a more accurate level. Packages, namespaces, and nested classes act as a form of information hiding, because implementation details are hidden from the programmer. Only when examining the next nested level, the programmer is confronted with another level of details. In Section 2.3, we give an overview of the benefits of hierarchical decomposition.

6.4. Code Reuse

6.4.1. Multiple Inheritance

In programming languages with multiple inheritance, a class can be a subclass of more than just one superclass. Examples of programming languages supporting multiple inheritance are C++, Eiffel, or Python. Multiple inheritance is controversial because of the *diamond problem*: imagine that a class inherits from two classes and both classes provide the same method. Which implementation should be used in the subclass? In C++, this problem is solved by specifying explicitly, which implementation to use. In Python, the order of superclasses matters (C3 linearization).

6.4.2. Mixins

Ruby Modules, Python Multiple Inheritance, Newspeak, Jigsaw

6.4.3. Traits

Squeak implementation

6.4.4. Java Generics

Java generics allow classes and interfaces to be parameterized by one or multiple classes and interfaces for type checking reasons [7, 15]. They are often used together with collections [45]. Generic parameters are defined as part of the class or interface definition. When a class or interface is used, the programmer can pass classes and interfaces as arguments.

```
class Array<T> {
   T[] storage;

   public List(int size) {
      storage = /* ??? */;
   }

   T get(int index) {
      return storage[index];
   }

   void set(int index, T value) {
      storage[index] = value;
   }
}

Array<String> arr = new Array<String>(100);
```

Figure 6.2.: Generic array implementation using Java generics

Figure 6.2 shows how Java generics are used in practice. `T` is the generic parameter of the class `Array`. The compiler ensures that only arguments with the correct type `T` can be passed to `set()` and knows that `get()` can only return objects of type `T`.

One shortcoming of Java generics is type erasure: generic type information is only known at compile time, but not at runtime. In contrast to C++ templates, there is only one `Array` class, regardless of how often the class is parameterized with different arguments [33]. Therefore, Java actually stores a reference to an array of type `Object[]`. It is difficult to initialize `storage` to an array of type `T`. In fact, the statement `new T[size]` does not compile. What the programmer could write instead is an unchecked type cast [41]: `(T[]) new Object[size]`.

In Matriona, a new class is created every time a parameterized accessor method is executed. Furthermore, arguments passed to the accessor method are available at runtime using message sends to `scope`.

6.4.5. C++ Templates

C++ templates allow classes to be parameterized with generic types. In contrast to Java generics, C++ generates a copy of the template, whenever it is used with a concrete type [59]. Consequently, every instantiation of a C++ templated class generates a new class, whereas all instantiations of a Java generic class are the same class (type erasure).

C++ templates are similar to parameterized classes in Matriona in a sense that a new class is generated whenever a template/parameterized class is instantiated. However, new classes in Matriona can be generated at runtime, whereas C++ templates are generated statically at compile time, as if they were a preprocessor transformation.

7. Future Work

7.1. Class as Instance-side Members

7.2. Bytecode Transformation instead of Recompilation

7.3. Adding Instance Variables

7.4. Undo Changes

Remove added methods from old class if definition is changed (extension methds).

7.5. Squeak Integration

7.6. Extension Methods

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

7.7. Dependency Management

8. Summary

comparison with Newspeak: many ideas taken from it, but too complex

Bibliography

[1] Edward Benson. *The Art of Rails (Programmer to Programmer)*. Birmingham, UK, UK: Wrox Press Ltd., 2008. ISBN: 978-0-4701-8948-1.

[2] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. "Analyzing Module Diversity". In: *Journal of Universal Computer Science* 11.10 (Oct. 28, 2005), pp. 1613–1644.

[3] Lorenzo Bettini, Viviana Bono, and Silvia Likavec. "A Core Calculus of Higher-order Mixins and Classes". In: *Proceedings of the 2004 ACM Symposium on Applied Computing*. SAC '04. Nicosia, Cyprus: ACM, 2004, pp. 1508–1509. ISBN: 1-58113-812-1.

[4] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 978-0-321-35668-0.

[5] Matthias Blume and Andrew W. Appel. "Hierarchical Modularity". In: *ACM Trans. Program. Lang. Syst.* 21.4 (July 1999), pp. 813–847. ISSN: 0164-0925.

[6] 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.

[7] Gilad Bracha. "Generics in the Java programming language". In: *Sun Microsystems, java.sun.com* (2004), pp. 1–23.

[8] Gilad Bracha. "Modules: Dreams and Reality". In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. AOSD '11. Porto de Galinhas, Brazil: ACM, 2011, pp. 283–284. ISBN: 978-1-4503-0605-8.

[9] Gilad Bracha. *Newspeak Programming Language Draft Specification Version 0.095* http://bracha.org/newspeak-spec.pdf. 2015.

[10] Gilad Bracha. "On the interaction of method lookup and scope with inheritance and nesting". In: *In 3rd ECOOP Workshop on Dynamic Languages and Applications (DYLA*. Citeseer. 2007.

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

[12] 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.

[13] Gilad Bracha, Peter Ahe, Vassili Bykov, Yaron Kashai, and Eliot Miranda. “The newspeak programming platform”. In: *Cadence Design Systems* (2008).

[14] Gilad Bracha and William Cook. “Mixin-based Inheritance”. In: *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA/ECOOP ’90. Ottawa, Canada: ACM, 1990, pp. 303–311. ISBN: 0-89791-411-2.

[15] Gilad Bracha, Sun Microsystems, Norman Cohen Ibm, Christian Kemper Inprise, Martin Odersky Epfl, David Stoutamire, and Sun Microsystems. *Adding generics to the java programming language: Public draft specification, version 2.0*. Tech. rep. 2003.

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

[17] J.-P. Briot and P. Cointe. “Programming with Explicit Metaclasses in Smalltalk-80”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA ’89. New Orleans, Louisiana, USA: ACM, 1989, pp. 419–431. ISBN: 0-89791-333-7.

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

[19] M. Dixon-Kennedy. *Encyclopedia of Russian and Slavic Myth and Legend*. ABC-CLIO, 1998, p. 187. ISBN: 978-1-5760-7063-5.

[20] Stéphane Ducasse. “Evaluating message passing control techniques in Smalltalk”. In: *Journal of Object-Oriented Programming (JOOP* 12 (1999), pp. 39–44.

[21] Bruce Eckel. *Thinking in Java*. 3rd. Prentice Hall Professional Technical Reference, 2002, p. 331. ISBN: 0-13100-287-2.

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

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

[24] David Flanagan. *Java In A Nutshell, 5th Edition*. O’Reilly Media, Inc., 2005. ISBN: 0-59600-773-6.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[26] 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.

[27] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. 1st. Addison-Wesley Professional, 2013. ISBN: 978-0-1332-6022-9.

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

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

[30] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '97. Atlanta, Georgia, USA: ACM, 1997, pp. 318–326. ISBN: 0-89791-908-4.

[31] Daniel H. Ingalls. "Design Principles Behind Smalltalk". In: *Byte* 6.8 (Aug. 1981), pp. 286–298.

[32] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-26211-158-6.

[33] Scott Lembcke, Sam BeVier, and Elena Machkasova. "Specialization of Java Generic Types". In: *Midwest Instruction and Computing Symposium*. 2006.

[34] *Martin Fowler: Inversion of Control Containers and the Dependency Injection Pattern*. `http://www.martinfowler.com/articles/injection.html`. Accessed: 2015-07-23.

[35] Bertrand Meyer. *Object-Oriented Software Construction*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0-13629-049-3.

[36] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. "The Bug Catalog of the Maven Ecosystem". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 372–375. ISBN: 978-1-4503-2863-0.

[37] G.J. Myers. *Composite/structured design*. Van Nostrand Reinhold, 1978. ISBN: 978-0-4428-0584-5.

[38] *Newspeak 101 – A guide for the Perplexed, January 2014 Update Release*. `https://medium.com/newspeak-documentation/newspeak-101-1fe7a924d726`. Accessed: 2015-07-27.

[39] Oscar Nierstrasz, Stphane Ducasse, and Damien Pollet. *Squeak by Example*. Square Bracket Associates, 2009. ISBN: 978-3-9523-3410-2.

[40] 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.

[41] Jaime Niño. "The cost of erasure in Java generics type system". In: *Journal of Computing Sciences in Colleges* 22.5 (2007), pp. 2–11.

[42] 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.

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

[44] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782.

[45] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. "Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 3–12. ISBN: 978-1-4503-0574-7.

[46] Ondřej Pavlata. "Ruby Object Model–The S1 structure". In: (2012).

[47] 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.

[48] *Pharo by Example, Draft Chapter Metacello*. http://pharobyexample.org/drafts/Metacello.pdf. Accessed: 2015-07-23.

[49] Dhanji R. Prasanna. *Dependency Injection*. 1st. Greenwich, CT, USA: Manning Publications Co., 2009. ISBN: 978-1-9339-8855-9.

[50] *RubyGems Guide: Pattern*. http://guides.rubygems.org/patterns/. Accessed: 2015-07-28.

[51] Dorin Sandu and Dwight Deugo. "The Lambda Pattern". In: *Proceedings of the 1999 Pattern Languages of Programming Conference*. 1999.

[52] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and AndrewP. Black. "Traits: Composable Units of Behaviour". English. In: *ECOOP 2003 – Object-Oriented Programming*. Ed. by Luca Cardelli. Vol. 2743. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 248–274. ISBN: 978-3-540-40531-3.

[53] Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz. "Seuss: Decoupling responsibilities from static methods for fine-grained configurability". In: *Journal of Object Technology* 11.1 (Apr. 2012), 3:1–23. ISSN: 1660-1769.

[54] *Squeak 4.5 Release Notes*. http://wiki.squeak.org/squeak/6189. Accessed: 2015-07-27.

[55] 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.

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

[57] Frank F. Tsui and Orlando Karam. *Essentials of Software Engineering, Second Edition*. 2nd. USA: Jones and Bartlett Publishers, Inc., 2009, p. 139. ISBN: 978-0-7637-8534-5.

[58] Robbie Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress, 2008. ISBN: 978-1-5905-9997-6.

[59] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Pearson Education, 2002. ISBN: 978-0-6723-3405-4.

[60] Hong Yul Yang, Ewan Tempero, and Hayden Melton. “An Empirical Study into Use of Dependency Injection in Java”. In: *Proceedings of the 19th Australian Conference on Software Engineering*. ASWEC ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 239–247. ISBN: 978-0-7695-3100-7.

[61] Stefan Zugal, Jakob Pinggera, Barbara Weber, Jan Mendling, and HajoA. Reijers. “Assessing the Impact of Hierarchy on Model Understandability – A Cognitive Perspective”. English. In: *Models in Software Engineering*. Ed. by Jörg Kienzle. Vol. 7167. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 123–133. ISBN: 978-3-642-29644-4.

[62] Jason Van Zyl. *Maven - The Definitive Guide*. O’Reilly, 2008, pp. I–XIV, 1–452. ISBN: 978-0-596-51733-5.

Appendix A.

Implementation Details

A.1. Determining the Lexical Scope

A.2. Traits

Eigenständigkeitserklärung

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

Potsdam, den 31. Juli 2015

Matthias Springer

Todo list