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 23, 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 can be 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.

Nested classes can be parameterized, allowing for external configuration of classes, a form of dependency injection. Furthermore, parameterized classes go hand in hand with mixin modularity. Mixins are a form of inter-class code reuse and based on single inheritance. Traits, another form of inter-class code reuse with explicit conflict resolution can be implemented on top of mixins.

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

Die Zusammenfassung auf deutsch.

Acknowledgments

I owe everything to my cat.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API application programming interface

1. Introduction

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* [12], 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 wooden 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" [25]. This work describes a module system or 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*" [24].

Decomposability If a design method supports modular decomposability, it helps the programmer in breaking down big components into smaller one. These components 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 and 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.

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 [31]. 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 an object-oriented, class-based programming language and Squeak[1] is a Smalltalk-80 dialect. It was originally developed by Alan Kay, Dan Ingalls, and

[1] http://squeak.org

Adele Goldberg. Dan Ingalls described Smallktalk-80 as a project whose purpose is to "provide computer support for the creative spirit in everyone." In his article "Design Principles Behind Smalltalk" [21], 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 one (*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 store shared behavior and components in a designated place that allows other components to take advantage of it and eliminate 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 Work

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 deescribes 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 exisiting 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 in which 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 class 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 [16], 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 version control system, 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 in an application.

Squeak has packages [26], 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 (e.g., code reuse) for our system. See Section 6.1.2 for a detailed discussion of Squeak environments and why we did not use them in this system.

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 classes occur.

2.2. Dependency Managment

Dependency management describes the task of keeping track of dependencies and ensuring that required dependencies are available to the application in question. We destinguish 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 all 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. The application might then not work with a newer library, where the bug is fixed. Furthermore, the public API of a library might change with new versions, especially if it is a new major version.

Draft Draft Draft July 23, 2015 19:29 Draft Draft Draft

Therefore, we need a versioning mechanism in Matriona, that helps us to store and reference 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.

External Configuration In this case, the dependency management is delegated to the client of an application or library. What the application specifies is that it requires some dependency implementing a certain interface, but not what exact dependency it is exactly or in what version. This mechanism is also called *dependency injection* and used heavily in the Java world [35]. Dependency injection is also known as *inversion of control* [23], because it inverts the control of dependencies: it is shifted from the application or library in question to the user/client. 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 a source code management system and can be used to load new versions of the source code into an image. Metacello is a package management system (see Section 6.2.1), 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 [34].

External configuration can 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 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.

2.3. Hierarchical Decomposition

Smalltalk packages allow the programmer to group together what belongs together [13]. 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 [31]. 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 [3], which is in a basic form realized in Java packages, Ruby namespace module, 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.

It also allows for fine-grained dependency management: for example, it is considered good practice in many programming languages to keep import statements as small as possible. Import statements also act as documentation, giving the reader of the source code a rough idea of what the source code might do. Furthermore, if a functionality is nested in a submodule, it is likely that it is written in a more general way, such that it might be reused elsewhere in the application without bigger changes.

If the source code is functionally decomposed in a hierarchical way [39], 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 [40] (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 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 our work: classes as class members. Similar concepts are part of programming languages like Java, Ruby, Python, and Newspeak. Our concept follows closely the Newspeak notion of nested classes, but without making invasive changes to the Smalltalk programming language.

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.

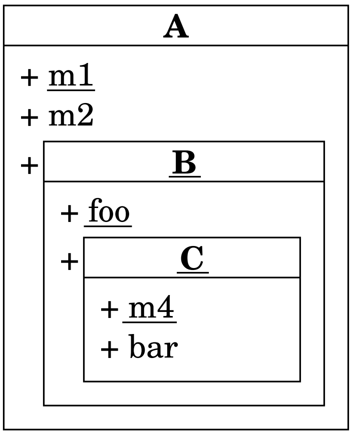


Figure 3.1.: Example: nested classes. A class can have class-side member classes.

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 resulting class and C to resulting meta class, and finally returns the resulting class. For performance reasons, our system also caches the result, meaning that a class is not generated twice.

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

[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.

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[2]. Consequently, its name is a concatenation of all class names on the path from the top-level class to class in question.

Notation and Example Figure 3.1 shows an example of nested classes in Squeak. `A` is a top-level class, i.e., it is part of the `globals` dictionary and known everywhere in the system; it can be referenced by just writing the identifier `A`. `A` has one instance method `m2` and two class methods `m1` and `B`. In accordance with UML notation, class-side method selectors are underlined.

`A class»B` is a class generator method that is associated with a set of instance methods {} and a set of class methods {`foo`, `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

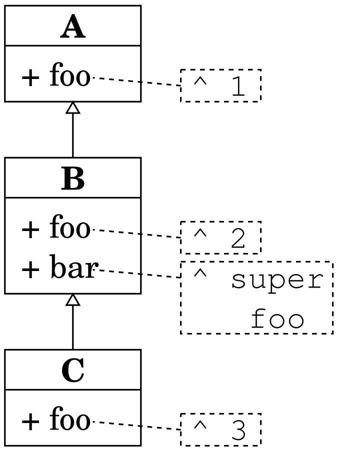


Figure 3.2.: Example: Binding of `super`. The method lookup starts at the superclass of the calling method's class.

It is sometimes necessary to access a method's lexical scope (i.e., the enclosing classes), in order to send messages to enclosing classes. For this reason, our system introduces new keywords, in addition to `self` and `super`, which are already present in every Smalltalk dialect. Figure 3.3 gives an overview of all method lookup-related keywords in the system.

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

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

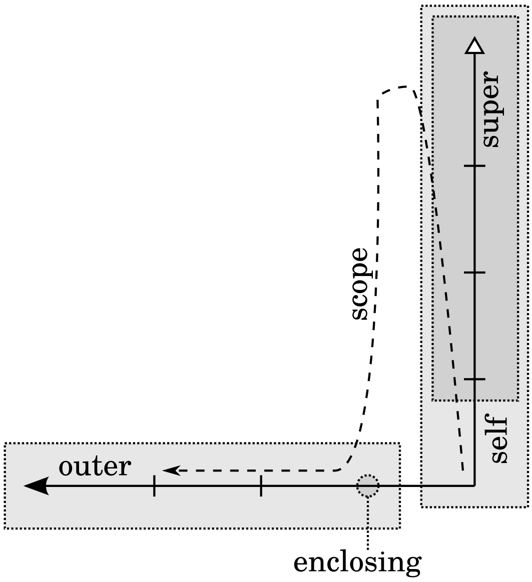


Figure 3.3.: Keywords for superclass and lexical scope access. The lookup starts at `self`, and continues with the lexical scope.

`super` Keyword This keyword is also used to make a message send within an object. Again, the receiver is the same object as the sender, but the lookup starts at superclass of the sender's method class. Note, that `super` is bound to the superclass of the method class, not the superclass of the receiver's class. For example, in Figure 3.2, `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`.

`enclosing` Keyword This keyword 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.

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

`enclosing` is a keyword that evaluates to the method owner's enclosing class upon method compilation. Note, that `enclosing` is bound to the method's lexical scope, not the receiver 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 both cases). Note, that `B2 C bar2` calls `B2 foo`, because `bar2`'s lexically enclosing class is `B2`.

Note, that `enclosing` can be used for meta programming purposes; however, it should be avoided in general, 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

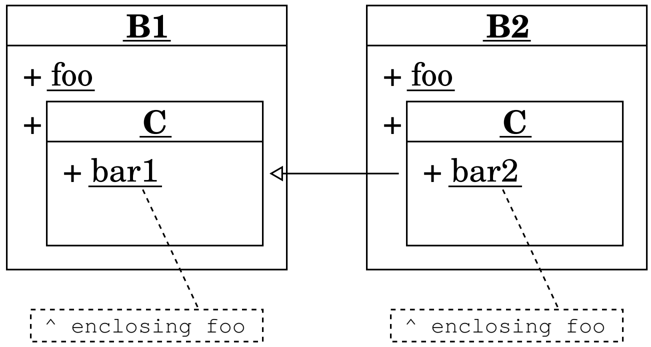


Figure 3.4.: Example: Binding of `enclosing`. The keyword is bound to enclosing class of the class where the method containing the keyword is contained.

have a `super` keyword that does the lookup only in the superclass[3] (single-level super). Matriona provides a `scope` keyword that should be used instead.

`enclosing` Method In addition to `enclosing`, every class in the system has a method `enclosing` that returns the enclosing class of the receiver[4], 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. The statement `enclosing enclosing` would be somewhat similar to a `super super` statement, which does not exist in Smalltalk.

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

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

[3]However, there is a method `Class»superclass`.
[4]The enclosing class of an object that is not a class is its class' enclosing class.

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

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

Matriona essentially first looks up the methods in `self`, then in the superclass hierarchy, and then in the lexical scope. This is also how the method lookup in Java works, also known as *comb semantics* [6]. 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 [7].

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

Implicit `scope` Receiver In our system, references to globals are in fact message sends with `scope` as implicit receiver. This makes it easier for Smalltalk programmers to write code in 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[5], 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 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.

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

3.3. Parameterized Classes

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

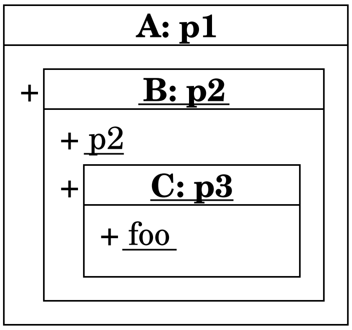


Figure 3.5.: Example: Parameterized classes. A class can have parameters accessible with message sends to `enclosing`.

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 3.5 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`.

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 and Instantiation

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 also be applied to already existing instantiations of the model, allowing giving programmers the feeling of working with a live system.

Smalltalk-80 Class/Meta Model Squeak already comes with a meta model: objects are instances of a classes, consequently, classes are also instances of a class. In Smalltalk, every class is an instance of its own meta class, which is in turn instance of `Metaclass` (Figure 4.1).

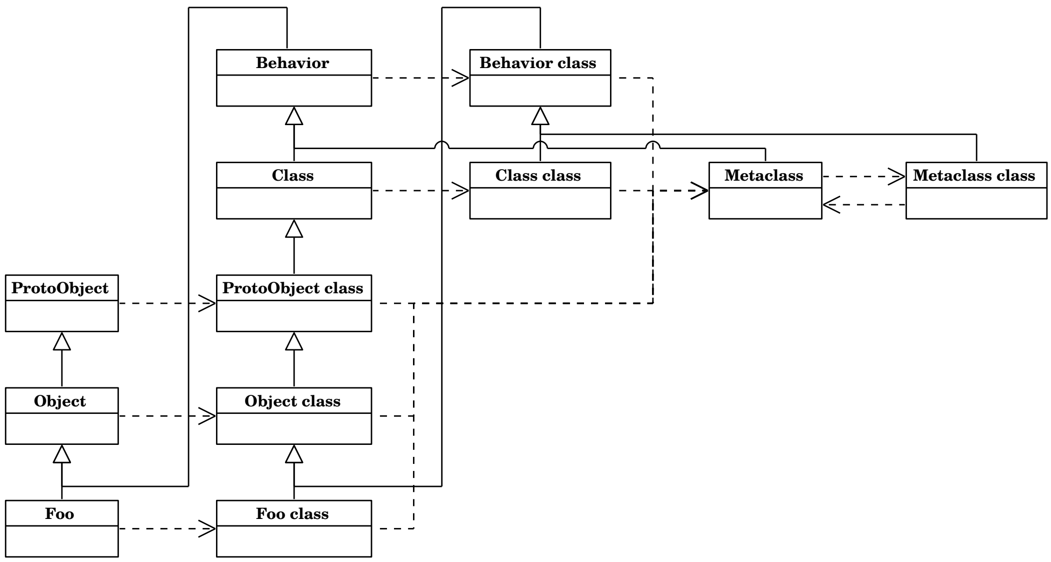


Figure 4.1.: Squeak class model. Every class is an instance of its meta class. Meta classes are instance of `Metaclass`. Meta classes and non-meta classes form a helix [10], connecting the meta class hierarchy with the non-meta class hierarchy.

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

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

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

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

Nested Classes Meta Model Figure 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 with its meta class specification and vice-versa. When a class specification is instantiated, the system generates both the class and the meta class. Meta class specifications cannot be instantiated.

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

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

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

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

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

Class Initialization Figure 4.3 illustrates how the system generates and initializes a nested class (class specification instantiation).

Add methods for class parameters.

Whenever a class accessor method is invoked, the method first checks if the class is already cached. If that is the case, it is returned. Otherwise, the class generator method called, returning an empty uninitialized class, i.e., all instance methods are still missing and only the superclass and the instance and class variables are

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

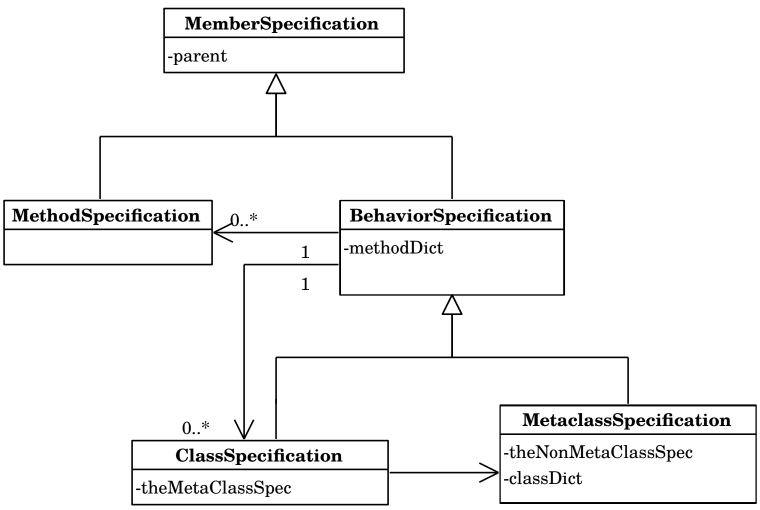


Figure 4.2.: Meta model for nested classes in Matriona. Class specifications can are containers for method specifications. Meta class specification can have additional nested class specifications.

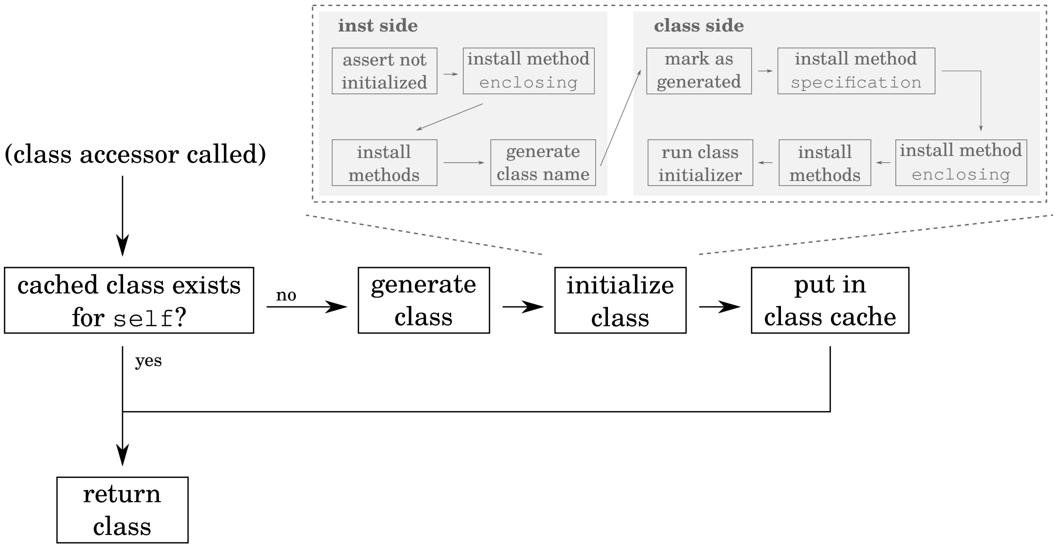


Figure 4.3.: Nested class initialization. Classes are generated lazily and initialized using the Matriona meta model.

set up correctly. The following list gives an overview of the steps necessary for initializing a class.

1. Install `enclosing` instance method. This method returns the enclosing class.
2. Install/compile all instance methods listed in the class specification.
3. Generate the class name. The class name is a concatenation of the enclosing class' name and the selector of this class' accessor method. It is stored as an instance variable at `Class`. Note, that every class object is an instance of its meta class, which is a subclass of `Class` (Figure 4.1).
4. Add a marker method to the meta class to mark it as generated. This makes is easy to check if a class is an ordinary (legacy) Smalltalk class or was generated within our system.
5. Install `specification` class method. This method returns the class specification, which is useful for meta programming purposes.
6. Install `enclosing` class method. This method is identical to the instance method.
7. Install/compile all class methods listed in the meta class specification.
8. Send `initialize` to the class object.

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

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

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

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

4.2. Anonymous Classes and Subclass Generation

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

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

Subclass Notation Figure 4.4a 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 [15, 14]. 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`.

Figure 4.4b shows how subclasses are created in our system. `NewClass` is a class generator method and also the name of the new class. Therefore, it is no longer necessary to pass a symbol with the name of the new class to the `subclass:` method. Note, that the `<class>` pragma is necessary to distinguish between class generator methods and regular methods, which might accidentially return a class. Only in the former case, a class specification object is created.

```
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 4.4.: 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`).

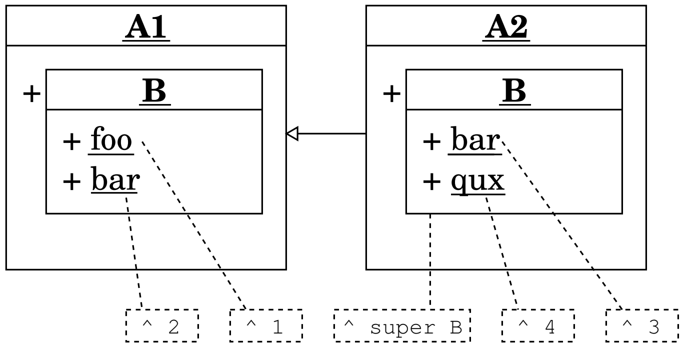
4.3. Inheriting Nested Classes

Nested classes are accessed using accessor methods which return the generated class. They are similar to class instance variable 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 object. Nested classes can be overridden in a subclass of the enclosing class, just as any regular method can be overridden. The following paragraphs give an overview of how a subclass of an enclosing class can customize the nested class.

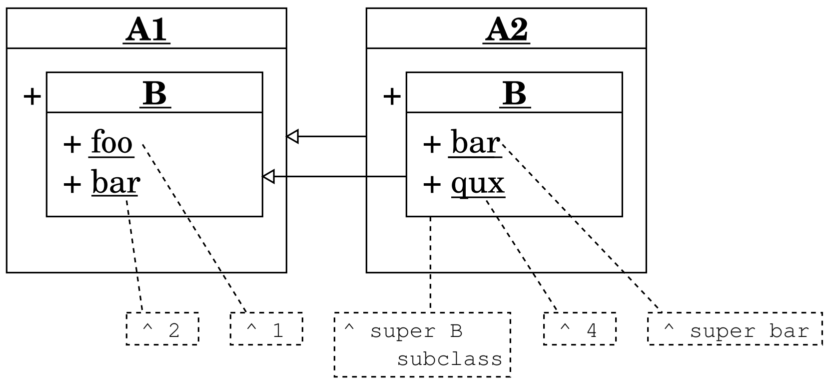
Override with Nested Class A subclass of an enclosing class can define a new nested class. The programmer simply adds a new class generator method with the same selector to the subclass. The superclass will keep using the old nested class, whereas the subclass will use the new one, because the method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

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 (without the `class` pragma) to the subclass.

Extend Inherited Nested Class without Subclassing A subclass can extend the inherited nested class, i.e., 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 in the class specification of the nested class in the subclass. Duplicate methods will be replaced, similarly to extension methods in Squeak.



(a) Extending inherited nested classing



(b) Subclassing inherited nested classing

Figure 4.5.: Example: Extending and subclassing nested classes. Subclassing inherited nested classes leads to parallel class hierarchies.

Figure 4.5a 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 replaces in `A2 B`.
- `A2 B qux`: returns 4.

Note, that `A1 B` and `A2 B` have the same superclass, but are different class object. `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`[3]. That class is then initialized with the class specification for `A2 B` (without making a subclass), i.e., all methods defined in `A2 B`'s class specification are added. A second call to `A1 B` will not return the previous generated and extended class for `A2`, because the class cache is implemented on a per-receiver basis.

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 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 4.5b shows an example for subclassing a nested class, which is similar to Figure 4.5a. 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 1, because the `super` call invokes `A1 class»B class»foo`.

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

Implementation of `outer` During compilation, all references to `outer` are bound to an instance of `LexicalScope`. This class is a subclass of `ProtoObject`, holds a reference to the enclosing class, and contains a `doesNotUnderstand:` handler, that forwards messages to the enclosing class. If the enclosing class does not

[3]Caches are receiver-specific.

understand the message, the message is forwarded to that class' enclosing class. If at some point, a top-level class without an enclosing class is reached, the handler looks for an entry in the `globals` dictionary with the message's selector.

As an example, let us assume that we have classes nested as shown in Figure 3.1 and that all following message sends to `outer` happen in `A class»B class»C class»m4`. See Figure 4.6a for a visualization of the lookup.

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

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

Figure 4.6b shows how the `scope` lookup works in a slightly modified example. Just as in the previous example, we assume that all message sends happen in `A class»B class»C class»m4`. However, `m4` is invoked on class `D`, which is a subclass of class `A B C`. Therefore, `self` is bound to `D`.

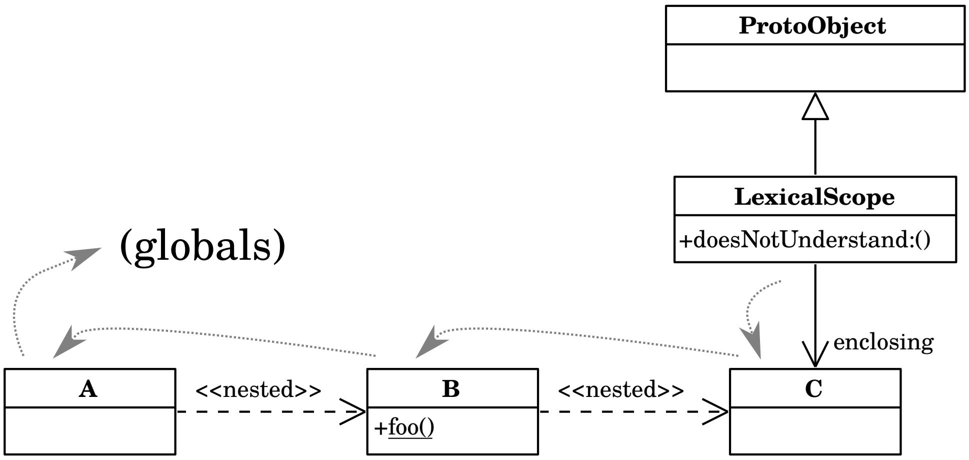
- `scope bar`: lookup in `self` succeeds: method `D class»bar`.
- `scope foo`: lookup in `self` fails, but lookup in `enclosing` (class `A B`) succeeds.
- The lookup for the all examples listed for `outer` (previous paragraph) yields the same result here.

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

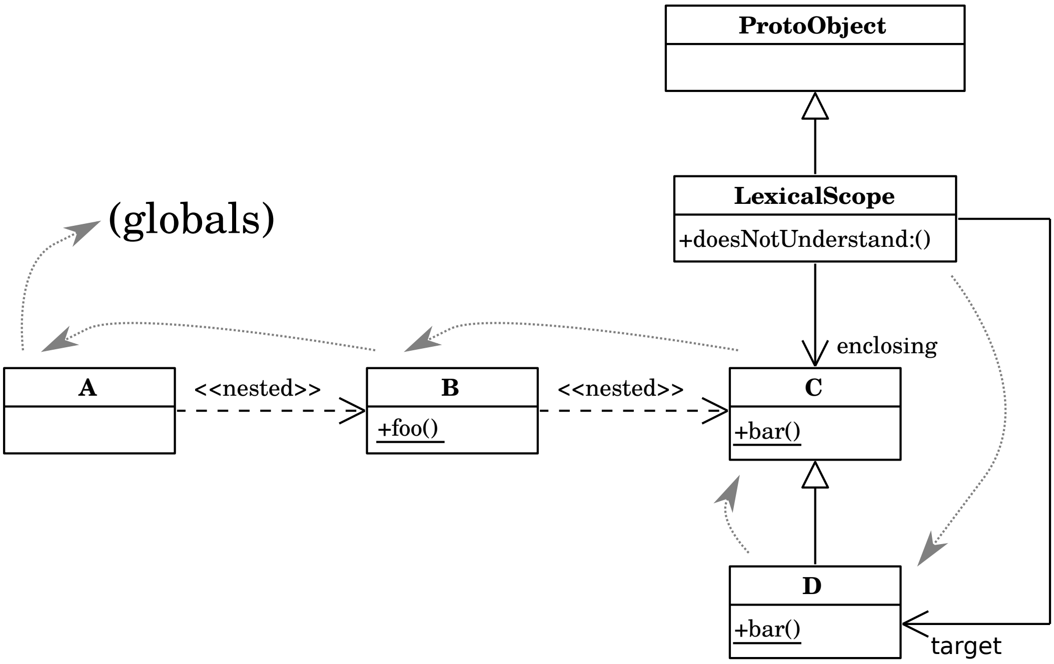
4.5. Class Caching

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

Cache must map self to instantiation



(a) `LexicalScope` for `outer` keyword



(b) `LexicalScope` for `scope` keyword

Figure 4.6.: Example: Method lookup using `LexicalScope`. Message sends to `scope` have an additional `target` object involved in the lookup, which is the receiver in the context where `scope` appears in the source code.

ject. 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 4.7).

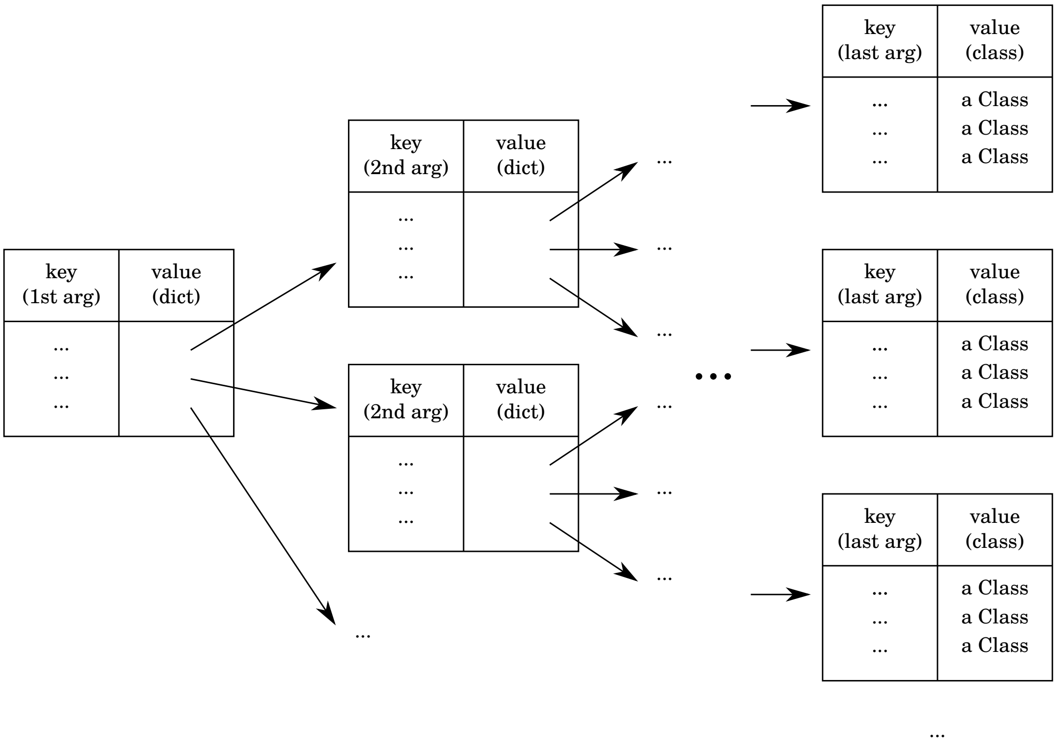


Figure 4.7.: Class Cache stored in `ClassSpecification`

In this case, class objects are garbage collected once there is no reference to at least one of the arguments in the system anymore. However, it depends on how exactly parameterized classes are used. If parameterized classes are used heavily, for example with `SmallIntegers` as parameters, no class would ever be garbage collected, because `SmallIntegers` are represented as tagged objects in Squeak [4, 30]. If parameterized classes are used as mixins, this is arguably less of a problem, because the the number of base classes to which a mixin is applied is usually not excessively large. However, note, that mixin applications can easily be cached by aliasing them as an unparameterized class (Figure 4.8). We argue that mixins will most of the time be used in such a way, because writing the mixin application explicitly is more verbose and hinders readability; in addition, the programmer might want to add additional methods to the mixin application, in which case the mixin application must be subclassed or aliased as described, anyway.

```
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 4.8.: Cached Mixin Application Example

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., exisiting instances must be migrated to the new class [11]. In that sense, Squeak and many other Smalltalk implementations [33] are different from other programming languages with an "edit/compile/run cycle" [27]: the programmer has the feeling that there is no difference between compile time and runtime.

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

Changing Instance Methods Whenever an instance method is added, removed, or changed, the system retrieves the collection of all instantiations, and performs the corresponding change on the class object. This does not require creating a new class object, but merely changing the method dictionary using Squeak's meta object protocol [17, 22].

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

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

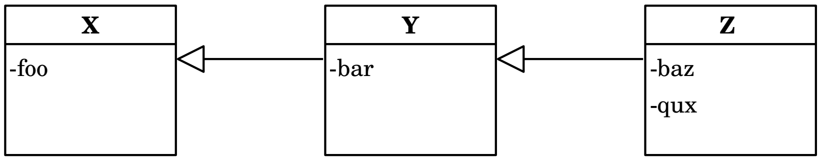


Figure 4.9.: Instance Variable Indexing

In Figure 4.9, class `X` has instance variable `foo`, class `Y` has instance variables `foo` and `bar`, and class `Z` has instance variables `foo`, `bar`, `baz`, and `qux`. Instance variables are indexed according to the superclass hierarchy. Therefore, `foo` has index 0, `bar` has index 1, `baz` has index 2, and `qux` has index 3. These indices are used in the bytecode instead of string literals or symbols. Therefore, when instance variables are changed in `X`, all classes (methods in these classes) `X`, `Y`, and `Z` have to be recompiled. If instance variables in `Z` are changed, only `Z` has to be recompiled.

What is more interesting is how instance variables are defined in our system: they are part of the class generator method (Figure 4.4b). Therefore, the system has to execute that method a second time whenever it is changed. The method returns a new class object which must be initialized again, i.e., all methods are recompiled. Note, Squeak has the same behavior: whenever an instance variable is changed, methods in the current class and all subclasses are recompiled.

Class generator methods for unparameterized classes can just be invoked without any parameters. However, in order to update parameterized classes, the system has to cache the arguments provided to the class generator method when the class was generated. Therefore, in addition to the class cache, every class specification maintains an argument cache, mapping instantiations (classes) to an array of arguments. This argument cache is a `WeakIdentityKeyDictionary` and different from the dictionary data structure shown in Figure 4.7. That class cache would map arguments to instantiations, but this cache maps instantiations to arguments. Whenever there is no reference to an instantiation in the image anymore, the array of arguments can be garbage collected, because nobody can access the class anymore; therefore, this class does not have to be updated.

Note, that invoking the class generator method a second time might generate a new class. Therefore, all references to the old class have to replaced with references to the new class using the `becomeForward:` method. Also, all instances of the old class have to be migrated to the new class. This is no different from what Squeak does when an instance variable is added or removed, and not described in any more detail in this work. We encourage the reader to consult the *Smalltalk Blue Book* [17] for more information.

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

4.7. Integration in Squeak

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

4.7.1. Module Repository

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

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

4.7.2. IDE Support

Our system comes with a proof-of-concept implementation of a class browser. The existing system browser cannot be used, because it cannot handle class nesting. Our class browser is written in Vivide [37], a framework for dataflow-driven tool development, and shown in Figure 4.10. It supports creating and deleting methods and nested classes, but basic refactoring functionality and functionality such as browsing senders and receivers is still missing.

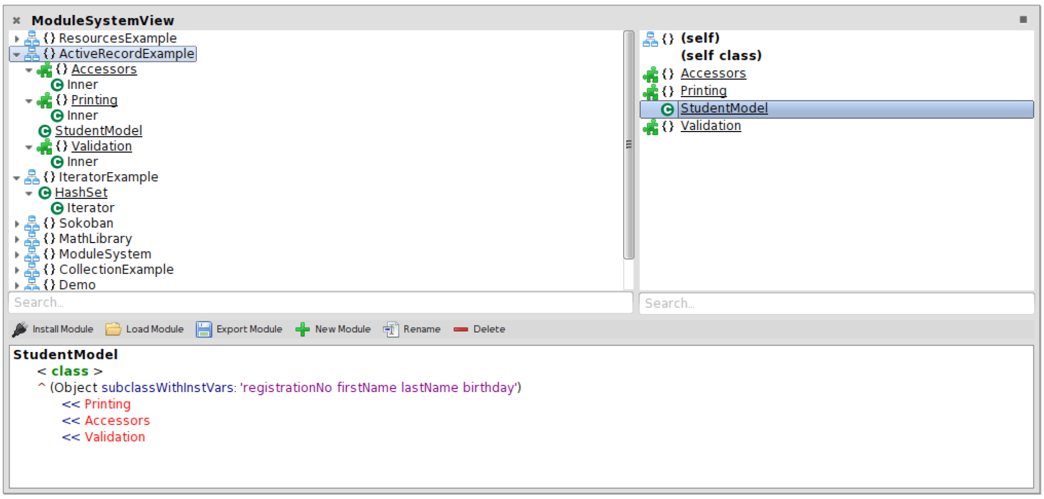


Figure 4.10.: Class Browser for Nested Classes

Our system is also integrated with the Squeak workspace and the test runner (Figure 4.11). 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.

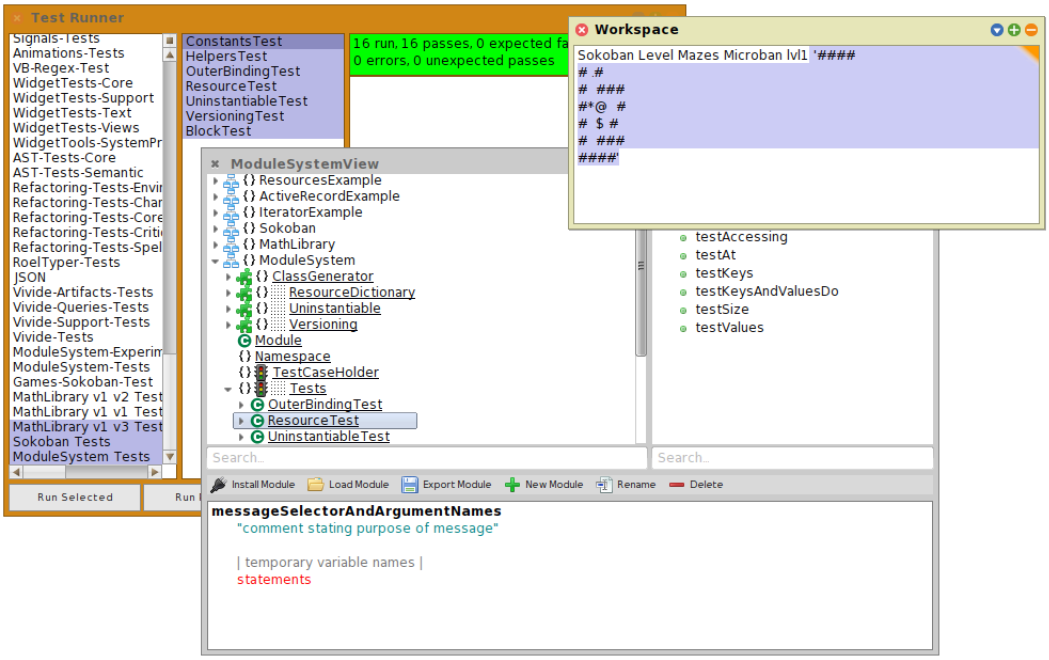


Figure 4.11.: Integration in Squeak

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 keywords that were introduced with our system, such as `outer` and `enclosing`, because they are bound in the Squeak environment of the class.

What is still an issue is that the debugger shows slightly different source code from what the programmer wrote. For example, class references are prepended with the `scope` keyword. In addition, whenever the scope keyword is used, code must be inserted that generates a new instance of `LexicalScope`, because `scope` cannot be bound at compile time (see Section 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 our system.

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[4] used to be a remote repository for Monticello projects (groups of packages). It is now deprecated and SmalltalkHub[5] is one possible replacement.

Our system 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 Our system 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[6] format. There is a directory for every class. Inside that directory, there are `instance` and `class` directories storing instance method 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).

ModuleSpec should also have a MethodSpec

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

Figure 4.12.: Source code export example

[4] http://www.squeaksource.com/
[5] http://www.smalltalkhub.com/
[6] https://github.com/dalehenrich/filetree

Figure 4.12 illustrates what the exported format looks like. The top-level module is `Bomberman`. 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 to instance methods `initialize` and `startGameWithPlayers:`.

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 systen, such as Subversion or Mercurial, can be used. Our system 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 lastest 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[7].

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`).

[7]This step only recompiles changed methods.

5. Use Cases

5.1. Avoiding Class Name Clashes

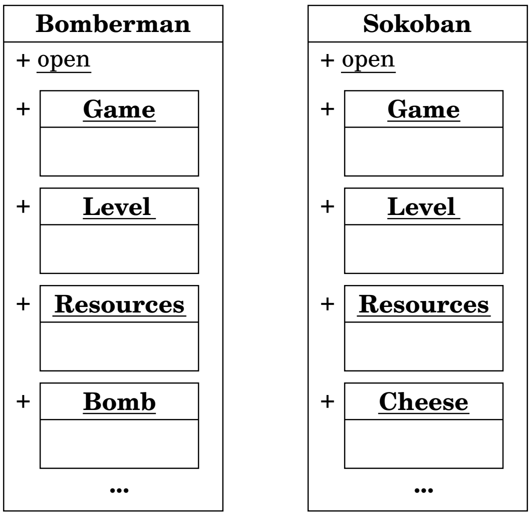


Figure 5.1.: Resolving class name clashes with class nesting

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

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

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

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

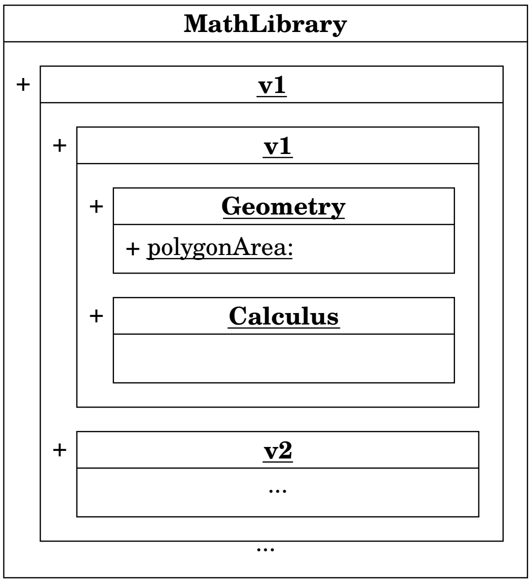


Figure 5.2.: Module versioning

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 in numbers.

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

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

Helper Methods In Figure 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.

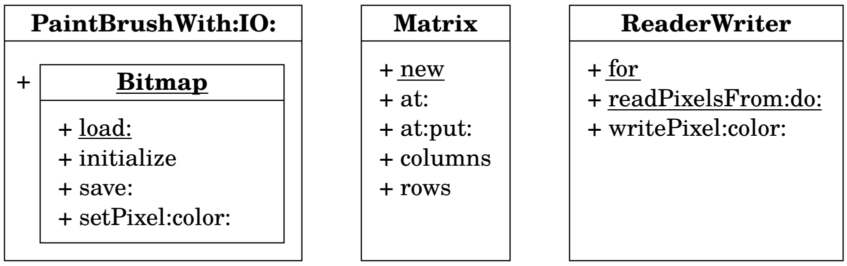
- `Versioning»myLatest`: returns the latest version contained as a nested class in the receiver. For example, `MathLibrary myLatest` returns `MathLibrary v1`.
- `Versioning»latest`: returns the lastest version in the receiver recursively. For example, `MathLibrary latest` returns `MathLibrary v1 v2`.
- `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.

5.2.3. External Configuration

Parameterized classes can not only be used to build mixins, but also externally configuarable modules. The basic idea is taken from Newspeak, where all module dependencies are encapsulated in a `platform` object. This platform object is installed along with the application source code and contains all libraries that the application depends on in the correct version [8]. 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 5.4 shows part of the implementation of simple drawing application. `PaintbrushWith:IO:` is a parameterized top-level class which takes as arguments a matrix implementation and a file IO library. The matrix implementation is used for storing the pixels inside the application. In the simplest case, this is could be the class `Matrix` from the Squeak standard library. It could, however, also be a class which stores pixels in a compressed form (e.g., using run-length encoding), but has `at:`, `at:put:`, `rows`, and `columns` as public API methods. `ReaderWriter` must be a class or object that supports reading and writing files on a pixel-by-pixel basis. Depending on which IO class the user of the library provides to `PaintbrushWith:IO:`, the application might for example generate JPEG files or PNG files.



(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 5.4.: Parameterized classes for external module configuration

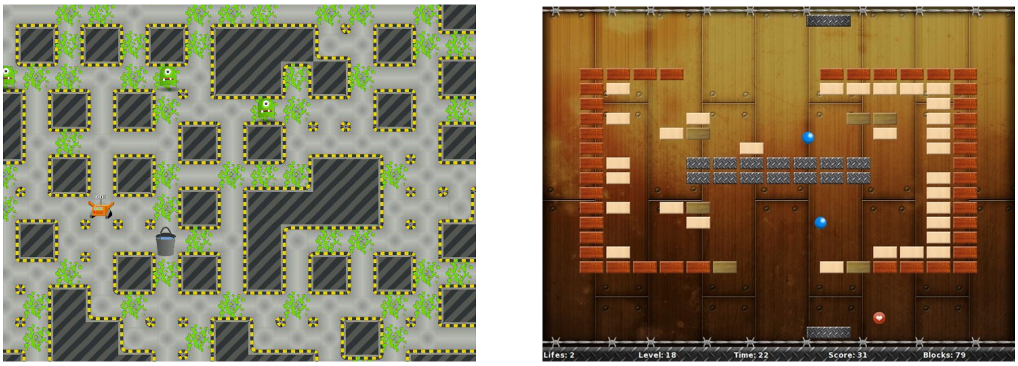
It is important to understand that the implementation of `PaintbrushWith:IO:` is entirely decoupled from the pixel data structure representation and the import/-export functionality. It is up to the user of `PaintbrushWith:IO:` to configure the class as needed.

External configuration as shown in this example is similar to a constructor that accepts class objects as parameters and constructs and instance of the class with the 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[1], a bomberman clone, and Breakout[2] (Figure 5.5).



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

Figure 5.5.: Example games as subjects for hierarchical decomposition

Figure 5.6 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. What 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, the `Builder`, `Item`, and `State` suffixes are omitted. It is now also possible to group classes together that belong together logically. For

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

example, all both level builders are nested within `SpaceCleanup Level`. Similarly, all items are nested in `SpaceCleanup Level Items` (which is also a 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.

Figure 5.7 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 design undestroyable borders of the game. All power ups are represented as classes and stored 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.

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

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 our system and come for free by just having class nesting as described in the previous sections; they are an immediate consequence of parameterized classes. A mixin is a function that takes as an input a class and outputs a subclass with additional behavior, i.e., it is a class transformator.

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

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

Example Figure 5.8 shows an example of parameterized classes and how they can be used to build mixins.

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*

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 5.6.: SpaceCleanup game implementation with/without hierarchical decomposition

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 5.7.: Breakout game implementation with/without hierarchical decomposition

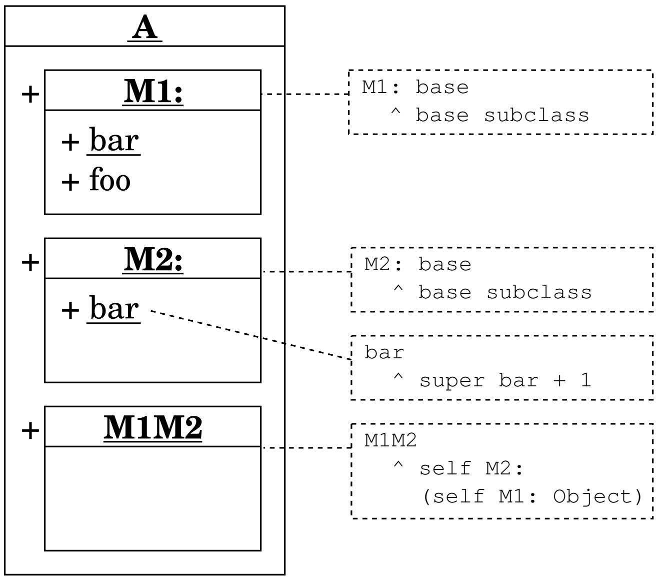


Figure 5.8.: Implementation of Mixins with Nested Classes

`A M1:`, whose superclass is `Object`. Note, that `A M1:` and `A M2:` are not specific classes: we use this notation as a name for *some* application of `A class»M1:` and `A class»M2:`, respectively. Therefore, even if two classes have the same name, they are not necessarily the same class if they names contain a colon.

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

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

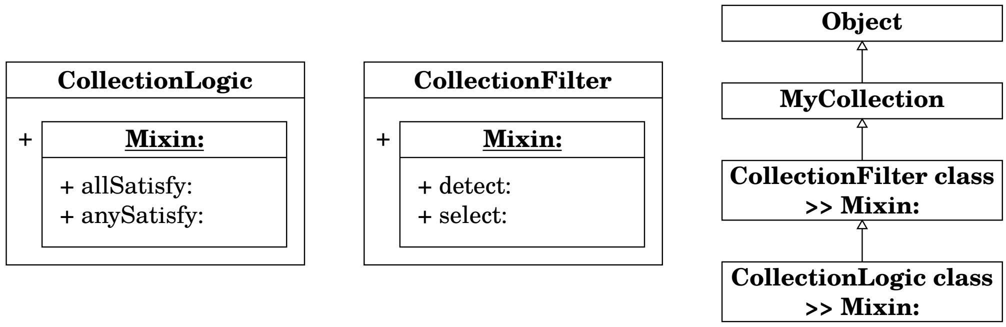
Both problems can be solved by wrapping the mixin in an unparameterized class and adding a helper method to `Class`. We assume that name of the parameterized nested class is always `Mixin:`. Then, the helper method « can be defined as shown in Figure 5.9.

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

Figure 5.9.: Helper method on `Class` for unparameterized mixin wrapper classes

Example Figure 5.10 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. Consequently, these four methods are written in such a way, meaning that both mixins can be applied classes providing at lease this 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 sent to the `Mixin:` method. Therefore, the name of the actual mixin must always be `Mixin:`, as long as, « is implemented as shown in Figure 5.9. Note, that « inverses the order of receiver and argument,



(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 5.10.: Unparameterized class generator pattern

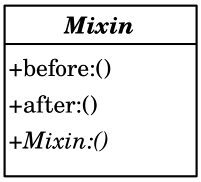
which is why the statement in `FullCollection` can be read from left to right: first `CollectionFilter` and then `CollectionLogic` is applied to `MyCollection`.

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 post-commit 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 5.11.: Implementation of pre-incude hooks and post-include hooks for mixins

Figure 5.11 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 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 shold 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 out a new subclass with additional or changed behavior. In Figure 5.10, 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 5.12).

For readability reasons, our implementation provides a simplified notation that combines this kind of mixin application and subclassing (Figure 5.13). This new

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

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

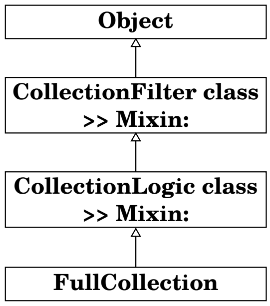


Figure 5.12.: Mixins as composable pieces of behavior

notation first applied 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`.

```
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 5.13.: Simplified notation for using subclassing and mixins

The idea of mixins used as composable units of behavior is similar to traits [36]. However, there are some minor differences.

- 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, but allows calling the original implementation using `super`.

5.7. Traits

5.8. Extension Methods

There are cases, in which the functionality of an already 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 our system by creating a nested classes whose class generator method returns an already existing class instead of a new subclass.

Consider, for example, that we want to add a method `asString` to the top-level class `CachedCollection` in Figure 5.10. Figure 5.14 shows how do 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 5.14.: 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 5.4; 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

6.1. Class Name Clashes

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 [9]. 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 [20]. Wildcard imports are possible, importing all members of a namespace. Imported members can be referenced within a namespace as if they were part of that namespace. A namespace member can also be defined as *private*; such a member cannot be imported, but always has to be referenced using its fully qualified name or using a relative name.

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

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

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

Java supports the concept of nested classes: a class can either be a top-level class or a class that is nested within another member. There are four different kinds of nested classes [2].

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

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

Java imposes certain restrictions on member classes. For example, nonstatic member classes are not allowed to have static member which are not final [18]. Furthermore, a subclass cannot override a member class definition [19]; it can just define its own member class. The difference is that overriding implies late binding, which is not the case in Java. With *Jx*, Nystrom et al. changed the Java language is such a way, that subclasses can enhance member classes [29]: the new member class overrides the original one and is always a subclass and a subtype of the member class in the superclass. Jx also allows changing the superclass of a member class in a subclass of the enclosing class, a form of mixin modularity.

Ruby Modules Ruby has the concept of classes and modules. Modules are classes which are not instantiable. They can be included in classes and be used as mixins. Modules and classes can be nested in each other, defining a namespace. Classes and modules can be accessed using their fully qualified name, which is the concatenation of their names with two colons as separator. For example, if class `B` is nested in class `A`, `B`'s fully qualified named is `A::B`. Classes and modules can also be accessed using relative names. For example, when accessing `A::B`, Ruby first looks for `A` in the current class/module. If there no such member, it looks in the enclosing class/module.

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

In Ruby, classes and modules can be extended after they have been defined. In case of an accidential class/module name clash, the two (or more) classes/modules are effectively merged. In case of colliding methods, the method that was last seen (read from the file) overwrites all previous definitions. This process is often used deliberately in Ruby, in order change the behavior of a library or application, e.g., to fix a known bug (*monkey patching*) [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 a `__init__.py` source code file is a package. Packages can contain other packages and modules. Packages can be imported just like modules. The fully qualified name of a module is the concatenation of all package names and the module name, with a dot as a separator.

Modules in other packages can be imported by writing their fully qualified name or using a path relative to the current module [38].

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

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

6.1.2. Squeak Environments

6.1.3. Newspeak Modules

6.2. Dependency Management

6.2.1. Metacello

6.2.2. Java Class Loader

6.2.3. Separate Compilation

6.2.4. Dependency Injection

6.2.5. External Configuration in Newspeak

6.3. Readability and Understandability

6.3.1. Smalltalk Packages

6.3.2. Hierarchical Decomposition

Java, Python, Ruby, Newspeak, ...

6.3.3. Information Hiding with Interfaces

6.4. Code Reuse

6.4.1. Multiple Inheritance

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 [5]. Theya are often used together with collections [32]. 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 6.1.: Generic array implementation using Java generics

Figure 6.1 shows how Java generics are used in practise. `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. Therefore, Java actually allocates a storage array of type `Object[]`. Therefore, 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 [28]: `(T[]) new Object[size]`.

6.4.5. C++ Templates

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

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: 0470189487, 9780470189481.

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

[3] Matthias Blume and Andrew W. Appel. “Hierarchical Modularity”. In: *ACM Trans. Program. Lang. Syst.* 21.4 (July 1999), pp. 813–847. ISSN: 0164-0925. DOI: 10.1145/325478.325518.

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

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

[6] 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.

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

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

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

[10] 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. DOI: 10.1145/74877.74921.

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

[12] M. Dixon-Kennedy. *Encyclopedia of Russian and Slavic Myth and Legend*. ABC-CLIO, 1998, p. 187. ISBN: 9781576070635.

[13] Bruce Eckel. *Thinking in Java*. 3rd. Prentice Hall Professional Technical Reference, 2002, p. 331. ISBN: 0131002872.

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

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

[16] 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.

[17] 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.

[18] 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: 0133260224, 9780133260229.

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

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

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

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

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

[24] Bertrand Meyer. *Object-Oriented Software Construction*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0136290493.

[25] G.J. Myers. *Composite/structured design*. Van Nostrand Reinhold, 1978. ISBN: 9780442805845.

[26] Oscar Nierstrasz, Stphane Ducasse, and Damien Pollet. *Squeak by Example*. Square Bracket Associates, 2009. ISBN: 3952334103, 9783952334102.

[27] 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.

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

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

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

[31] 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. DOI: 10.1145/361598.361623.

[32] 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. DOI: 10.1145/1985441.1985446.

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

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

[35] Dhanji R. Prasanna. *Dependency Injection*. 1st. Greenwich, CT, USA: Manning Publications Co., 2009. ISBN: 193398855X, 9781933988559.

[36] 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. DOI: 10.1007/978-3-540-45070-2_12.

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

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

[39] Frank F. Tsui and Orlando Karam. *Essentials of Software Engineering, Second Edition*. 2nd. USA: Jones and Bartlett Publishers, Inc., 2009, p. 139. ISBN: 0763785342, 9780763785345.

[40] 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. DOI: 10.1007/978-3-642-29645-1_14.

Appendix A.

First Unimportant stuff.

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

Eigenständigkeitserklärung

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

Potsdam, den 23. Juli 2015

Matthias Springer

Todo list