Extending a Dynamic Programming Language and Runtime Environment with Access Control

Philipp Tessenow, Tim Felgentreff, Gilad Bracha, Robert Hirschfeld

Technische Berichte Nr. 107

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 107

Philipp Tessenow |Tim Felgentreff | Gilad Bracha | Robert Hirschfeld

Extending a Dynamic Programming Language and Runtime Environment with Access Control

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2016
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-373-2

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URN urn:nbn:de:kobv:517-opus4-92560
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-92560

Complexity in software systems is a major factor driving development and maintenance costs. To master this complexity, software is divided into modules that can be developed and tested separately. In order to support this separation of modules, each module should provide a clean and concise public interface. Therefore, the ability to selectively hide functionality using access control is an important feature in a programming language intended for complex software systems.

Software systems are increasingly distributed, adding not only to their inherent complexity, but also presenting security challenges. The object-capability approach addresses these challenges by defining language properties providing only minimal capabilities to objects. One programming language that is based on the object-capability approach is Newspeak, a dynamic programming language designed for modularity and security. The Newspeak specification describes access control as one of Newspeak's properties, because it is a requirement for the object-capability approach. However, access control, as defined in the Newspeak specification, is currently not enforced in its implementation.

This work introduces an access control implementation for Newspeak, enabling the security of object-capabilities and enhancing modularity. We describe our implementation of access control for Newspeak. We adapted the runtime environment, the reflective system, the compiler toolchain, and the virtual machine. Finally, we describe a migration strategy for the existing Newspeak code base, so that our access control implementation can be integrated with minimal effort.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Complex problems require complex software solutions. Building such complex high-quality software is expensive in time, money, and its demand in experienced software engineers.

To handle the increasing complexity of software, programming approaches, such as object-orientation, aspects [24], or context-oriented programming [18] have been invented. Other programming patterns strive to enhance the overall architecture of programs. Such patterns reduce complexity by splitting the program code into modules that can be understood independently. Ideally, each module should serve one purpose, so that programs can be constructed by combining individually developed and tested modules.

1.1 Access Modifiers as an Instrument to Structure Programs

It is easier to implement, test, understand, and maintain modules that have clean and concise interfaces. The principle of information hiding [37] states that modules should be able to hide access to internal information, encouraging construction of clean and concise interfaces.

Modularization as a technique is used in many levels of the software and hardware stack: When looking at system architecture, a single computer or even groups of computers can act as modules. On a single computer, programs can be understood as modules. Within a program, another set of modules, namely functions, methods, aspects, layers, and many more, depending on the programming language, can be found.

To hide internal information of a module, different approaches are used in different levels of the software and hardware stack. For example, access to other computers can be controlled by physical wiring or by threat mitigation techniques on the various communication protocol layers [17]. On the layer of communication between programs, other ways of interaction, for example file and memory access, can be observed. Operating systems usually take care of the inter-program information restrictions, for example through address space separation [47], access right management on the file system [26], or Software Isolated Processes [10].

Within a program, *access control*— the ability to define access rights on classes, methods, or attributes— is one technique for separating the public interface of a module and its internals. Access control ensures that a module's private internals remain private by restricting access to it on a language level. It is one of the core properties of a programming language, aiding the programmer to better modularize programs. It has been shown that access control supports architectural goals like decoupling, ensuring invariants, and offering concise interfaces [32, 37]. Conventions have been developed for programming languages without access control—for example prefixing private methods with underscores [44]. However, such conventions can be circumvented. Access control, which is enforced at a language level, provides better modularization as it cannot be circumvented.

1.2 The Object-Capability Model

Access control not only facilitates modular software, but it is also an integral part of security. Security related goals, like confidentiality, are important in many domains. A program cannot necessarily trust its modules: A module might be a plug-in provided by an untrusted third party. Alternatively, it can be compromised by an attacker who used a security hole in that module. Either way, it is desirable that no module can extract secret information—accidentally or intentionally—from other modules.

Mark Miller proposed the object-capability model [32], offering a set of design rules that allow secure computation. In this model, an object is a combination of behavior and state, where object state consists of references to other objects. Thus, objects form a reference graph. This graph can only be altered by three operations:

- *A new object is instantiated*: This creates a new object reference. The newly created reference is only available to the object which initiated the object creation.
- *An object sends a message to another object*: To be able to do this, the message sender must hold a reference to the receiver. The receiver may get new object references from the arguments of the message send.
- *An object reference is destroyed*: This happens, for example, when the variable holding an object reference is overwritten by another reference.

In a language following the object-capability model, it is not possible to obtain object references in any other way. This implies that it must not be allowed to forge object references. For example, casting an arbitrary number to a pointer—typical for programming languages like C—violates the object-capability model. Additionally, the programming language must not allow mutable global state because object references can be shared that way.

If those preconditions are fulfilled, we can define each object to be a capability in the system. Every entity holding a capability is trusted to use the capability without further security checks, like access control lists [41]. This requires the program developer to carefully decide which object references are passed around.

One important pattern to develop such systems effectively is the proxy pattern [42]: For example, a programmer might want to give another module the capability to append information to a certain log file. By default, the file object handling the log file provides read and write access to any position of that file. In this case, a proxy object of the original file object, which only implements methods to append characters, would be created. The proxy holds a reference to the original file object to perform log writes, which must not be leaked. If access to that proxy object is restricted using access control, it is ensured that only the accessible interface is used. This way, the programmer can safely pass the proxy object to an untrusted module, giving only the capability to append information to that particular log file and nothing more.

To summarize, the object-capability model requires a number of features that need to be supported by the underlying programming language:

- Program execution happens solely via message sends between objects, where each object is a capability. Any entity holding a capability may make use of it without further security checks.
- Objects (capabilities) can be obtained only through creation or message passing. This implies that there is no mutable global state or reference forgery.
- Objects are able to restrict access to their interfaces. The accessible part of an object's interface may be used by anyone holding a reference to that object without further checks.

Consequently, access control is one of the core features a programming language must provide to support the object-capability model.

1.3 Access Control: A Key Feature of the Newspeak Programming Language

Newspeak[1] is a dynamic programming language in the tradition of Self [46] and Smalltalk [15]. Its security model is based on the object-capability model [4]. Consequently, Newspeak—by design—meets all object-capability requirements.

Newspeak is truly object-oriented. Everything, down to numbers, methods, or classes, is an ordinary object. Objects communicate solely via message sends—even when accessing their own structure [2].

Arbitrarily deeply nested classes are supported by Newspeak. Each class defines a separate namespace, which has access to methods defined in outer classes. Top-level classes, which we also refer to as *modules*, do not, by themselves, have access to any object references. Because of this design, there is no global namespace and, therefore, no global state. If a class needs object references it cannot obtain (e.g. access to other system libraries and modules), those references have to be given as parameters during the class initialization.

Newspeak's specification [4] defines access modifiers for classes, their attributes (slots), and methods. However, current implementations of the Newspeak programming language do not support them. This leads to some disadvantages when programming in Newspeak: The positive effects of access modifiers in a program's architecture are attenuated, because access modifiers are currently not enforced by

[1] http://www.newspeaklanguage.org/, last accessed June 18, 2015.

any Newspeak implementation. At the moment, programmers are encouraged to design class interfaces hiding their internal methods by defining them to be private. However, external classes can still access them. This allows programmers to violate well known principles, like the law of Demeter [27] and the principle of information hiding. Newspeak's security concept, the object-capability model, cannot be enforced without properly working access modifiers. The goal of this work, is to provide a working access control implementation for Newspeak.

In this work we show how to extend Newspeak's implementation to support access modifiers, enabling Newspeak to fully support the object-capability model. We study access modifier implementations of different languages and compare them with our implementation based on the findings. Additionally, we provide a way to convert the existing Newspeak source base and runtime to work with those access modifiers. This is important, because Newspeak is a self-supporting development environment: Newspeak's implementation, like its Compiler, Parser, or Code Editor, are written in Newspeak partially using wrong access modifiers.

1.4 Contributions

In this work, we propose an extension to the Newspeak implementation to enable access control for classes, methods, and slots. We present a way to migrate the existing Newspeak source base, which already contains some non-functioning access control statements, to facilitate the now enforced access modifiers.

There is a huge variety of prior work on access control implementations in several programming languages. Due to their partial similarity with Newspeak, we discuss the role of access modifiers in the language design of Java and Ruby. We compare access control implementations of Java and Ruby to our implementation.

The main contributions of our work are:

- a summary of access control semantics and implementations in selected languages,
- an application of access modifiers to the Newspeak virtual machine's compiled methods including a modified method lookup depending on the access modifier of compiled methods,
- an extension to the Newspeak implementation to fully support and reflect on access modifiers as originally stated in the Newspeak specification, and
- a design to migrate a self-supporting development environment like Newspeak to use access modifiers.

1.5 Structure

The remainder of this work is structured as follows: In chapter 2, we describe the Newspeak language design with special focus on Newspeak's access modifiers. We investigate the architecture of the Squeak-based implementation of Newspeak to show the changes necessary for supporting access modifiers.

We present our access control implementation in chapter 3. The implementation provides a way to store and enforce access modifiers in Newspeak without causing a severe performance impact. We enable the Newspeak runtime to reflect on access modifiers and discuss a way to migrate the Newspeak code base to support enforced access modifiers.

The actual implementation of access modifiers for Newspeak is presented in chapter 4. In that chapter, we show details of the changes to the Squeak virtual machine, details of how Newspeak stores and reflects on access modifiers, and how we migrated Newspeak environments to facilitate the new feature.

In chapter 5, we evaluate our implementation with a qualitative analysis. We follow the error propagation of access violations and verify the resulting system behavior. Furthermore, we analyze the performance impact of our implementation by running multiple benchmarks.

We have a look at related work in chapter 6 by comparing the access modifier-related language design of Java and Ruby to Newspeak's design. Furthermore, we discuss the differences of our implementation to two Java and Ruby implementations.

Finally, we conclude our work in chapter 7 and present potential future work.

2 The Design of the Newspeak Language with Emphasis on Access Modifiers

To study possible ways to implement access control for the Newspeak programming language, we provide some background on the current design and implementation of the Newspeak language and programming environment. We give a brief overview of general language concepts and offer deeper insights to access-control-related features of Newspeak.

Distinguishing properties of Newspeak are its object-orientation, module system, reflection application programming interface (API), and, most important for this work, security [2].

Security Newspeak strives to be secure by implementing the foundations of the object-capability model [32], which we introduced in section 1.2. Therefore, Newspeak fulfills the necessary language properties:

- It does not have any mutable global state. In particular there is no global namespace which could provide mutable state or ambient authority.
- All references, even references to classes, are solely obtained via message sends or object creation.
- Access to an object's interface can be restricted by defining accessor methods as `private` or `protected`.

Because of the first two language properties, it is not possible that objects can interact outside of the reference graph. The last property allows to partially restrict access to the reference graph. Especially the methods defined in `Object`, need to be handled carefully, because they can be a source of ambient authority.

Jan Sinschek's *Crimestop* project [43] is a capability system for Newspeak. A functioning access control implementation, as defined in this work, is an integral part of the success of such a system.

Object-Orientation Newspeak is truly object-oriented and message-based in the sense that all operations, even an object's access to its own structure, are performed by sending messages to objects [4]. Thus, direct access of an object's state, as in Java or Smalltalk, is not possible. Direct references to objects, also known as "instance variables" or "fields", are called "slots" in Newspeak. Because direct access to slots is not possible, not even from the object itself, accessor-methods are automatically created for slots.

Modularity Newspeak class definitions span an independent, immutable, and parameterizable namespace. A class definition is *independent* because it can be compiled, loaded, and deployed without the need of external dependencies. This makes it possible to compile Newspeak classes in any order. Class definitions themselves are stateless and, thus, inherently *immutable*. However, classes can have state, which is provided by arguments of their default initializer. This makes classes *parametric* and self-sustained. Furthermore, classes can be arbitrarily deeply nested and can contain three types of definitions (slots, methods, nested classes).

Top-level classes are often referred to as *modules*. A module can be instantiated multiple times without affecting other instances of the module in any way. This means that it is possible to instantiate a library with a given set of parameters and, later, instantiate and use the same library with a different set of parameters.

Reflection The reflection capabilities of Newspeak are provided by a system based on mirrors [6]. Newspeak's mirrors offer reflective capabilities for an object's *introspection*, the ability to reflect on its own structure, and *self-modification*, the ability to change its own program code or structure. Because reflection can be done using mirrors, regular objects only provide base functionality. Through the mirror system it is possible to separate meta-level functionality from base level functionality and thus provide reflection as a capability in the object-capability sense.

2.1 A Brief Introduction to Newspeak's Syntax

We briefly introduce those parts of the Newspeak syntax that are essential for understanding the access control implementation presented in this work using the example class in Listing 2.1. An extensive introduction to Newspeak's whole syntax can be found in the Newspeak specification [2]. The listing shows the definition of a top-level class `HelloWorld`, which provides the functionality to print a string to the standard output of the Newspeak process. A class definition consists of three parts: a class header which spans from line 1 to 8, an instance side declaration spanning from line 8 to line 15, and a class side declaration given on line 15.

The method header optionally starts with the access modifier of the class. Because the `HelloWorld` class is a top-level class, the access modifier is inherently set to `public` and must be omitted. Apart from the `class` keyword, which follows the omitted access modifier, and the class name, the most prominent part of a class header is the default initializer. Each class definition can specify a default initializer, which will be added as a class side method and is called to initialize the instances of that class. The initializer is split into two parts in the class definition: The signature of the initializer is given after the class name. Its body is given after the super class definition—the first pair of parenthesis in the listing.

Following the equals sign, a super class and the initializer to use on the super class can be specified optionally. In Listing 2.1 no super class has been specified, which means that the implicitly defined super class will be `Object` and the message `new` will be sent to initiate it.

Listing 2.1: A "hello world" program in Newspeak. The `HelloWorld` is a top-level class which takes a `platform` object as a parameter to its default initializer. One method, `hello` is defined, which prints the string "Hello world (x)" (where `x` is the number of times the method was called) to the standard output of the Newspeak process.

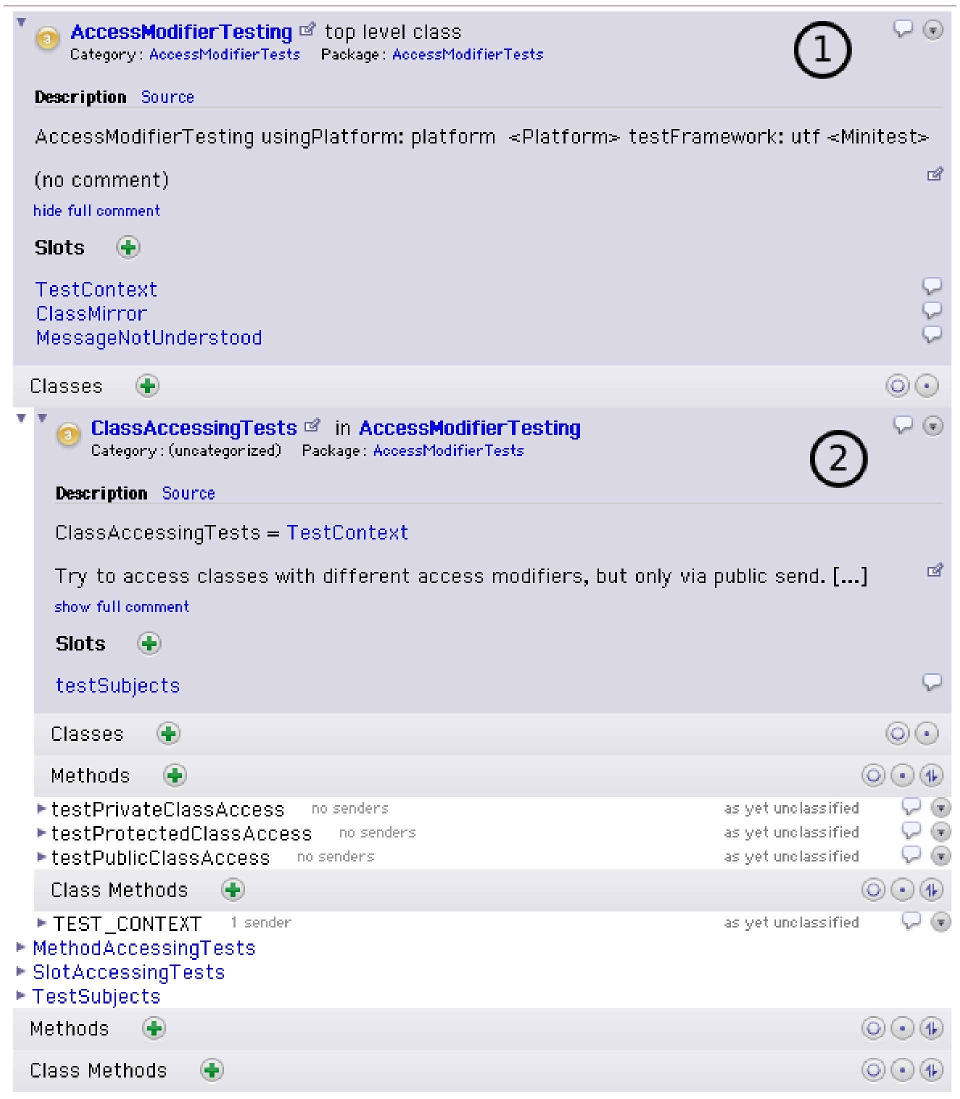
```
class HelloWorld usingPlatform: platform = (
  (* Outputs a friendly greeting to stdout. *)
  |
    OSProcess = platform squeak OSProcess.

    private timesCalled ::= 0.
  |
)('output'
  public hello = (
    timesCalled:: timesCalled + 1.
    OSProcess thisOSProcess stdOut
      nextPutAll: 'Hello world (', timesCalled, ')';
      nextPut: Character lf.
  )
) : ()
```

The default initializer is a special method. It is the entry point into a class and declares and initializes any state an instance of the class may have. The class can only access those objects which are given as arguments to the initializer or which are defined in an outer scope. For top-level classes, like our `HelloWorld` class, it is common to take a `platform` object as an argument to get references to objects of the Newspeak environment. Conceptually, the platform object is a map of module names to actual instantiated modules provided by the Newspeak environment.

Two slots are defined in the initializer in Listing 2.1: `OSProcess` and `timesCalled`. In the `OSProcess` slot, the `OSProcess` Smalltalk class is stored. That Smalltalk class is provided by the *Squeak* class. The `Squeak` class, which is obtained via the `platform` object like any other external dependency, provides an interface to classes of the underlying Squeak[1] system. Because Newspeak does not provide functionality to print a string to the standard output of the Newspeak process, the program relies on the `OSProcess` class of Squeak for that purpose. No access modifier is given for the `OSProcess` slot, which implicitly defines the slot as `protected`. The `timesCalled` slot is a `private` mutable slot used to count how often the "Hello world" message was printed. A mutable slot is declared with the `::=` assignment operator and an immutable slot is initialized once with the `=` assignment operator.

The side declaration contains definitions of nested classes and methods. Nested classes are not demonstrated in this example, because the syntax gets confusing easily when they are involved. Hopscotch, the development framework of Newspeak, solves this problem in an elegant way [7]. As seen in Figure 2.1, it displays nested classes as a separate class view nested in the view of the outer class.

[1]Squeak is a Smalltalk environment, on which a Newspeak implementation is based.



AccessModifierTesting top level class
Category: AccessModifierTests Package: AccessModifierTests
①

Description Source

AccessModifierTesting usingPlatform: platform <Platform> testFramework: utf <Minitest>

(no comment)
hide full comment

Slots

TestContext
ClassMirror
MessageNotUnderstood

Classes

ClassAccessingTests in AccessModifierTesting
Category: (uncategorized) Package: AccessModifierTests
②

Description Source

ClassAccessingTests = TestContext

Try to access classes with different access modifiers, but only via public send. [...]
show full comment

Slots

testSubjects

Classes

Methods

```
testPrivateClassAccess    no senders      as yet unclassified
testProtectedClassAccess  no senders      as yet unclassified
testPublicClassAccess     no senders      as yet unclassified
```

Class Methods

```
TEST_CONTEXT   1 sender                   as yet unclassified
```

MethodAccessingTests
SlotAccessingTests
TestSubjects

Methods

Class Methods

Figure 2.1: A screenshot of the Hopscotch development environment. The view of the `AccessModifierTesting` top-level class ① and its nested class `ClassAccessingTests` ② can be seen. Both within the same window and complete with documentation, slots, and methods.

Method definitions consist of a method signature and the method body. The signature is very similar to Smalltalk's method signatures in the way that the method selector can consist of multiple words with arguments in-lined between them. For example `usingPlatform: platform testFramework: utf` is the signature of the default initializer of the `AccessModifierTesting` class in Figure 2.1. It consists of two words and expects two arguments: `platform` and `utf`. The difference to Smalltalk method signatures is that Newspeak method signatures can be prepended by an access modifier. The `hello` method in Listing 2.1, for example, is `public`.

Within the method body, a variable is assigned at line 10. Actually, the assignment is not a direct access of the `timesCalled` slot. A normal message is send to a setter method, with the same effect as `timesCalled: timesCalled + 1` would have had. The difference to the two-colon notation is that it returns the assigned value and not the receiver, as a normal setter send would do, to allow more convenient chaining and avoid excessive use of parenthesis. Lines 11 to 13 of the method body are usual method sends, with the `nextPutAll:` and `nextPut:` method being send to the same receiver by using the semicolon.

There is one interesting thing, though. It was stated earlier that no state can be accessed unless it is defined in an outer scope or passed as a parameter. However, the `Character` class is accessed in line 13 even though it is not taken from the `platform` object. Similar to the call of the `OSProcess` getter method in line 11 with `self` being the receiver (actually, it is an implicit self send as described in subsection 2.3.4), a method named `Character` is called in line 13. The difference between those two is that `OSProcess` is an immutable slot of the class, for which a getter method is automatically defined by the system. This is not the case for the `Character` method. However, the `HelloWorld` class implicitly inherits from `Object`, which implements the `Character` method to return the `Character` class.

2.2 The Newspeak Metamodel

To discuss the consequences of access modifier applications, first we present the metamodel of Newspeak. We discuss how class inheritance is designed and the implications for the meta classes.

Class inheritance in Newspeak is designed as a composition of mixins. A mixin is an abstract subclass that implements extended behavior of a class [5]. Because the mixin is an *abstract* subclass, the super class is not necessarily known when creating the mixin. This makes mixin application—and therefore inheritance in Newspeak—late bound.

Each class declaration defines a mixin in Newspeak. This sets class declarations apart from classes—a class has a super class while a mixin does not. A mixin application creates a new class which has all slots and methods of the super class combined with those from the mixin. If there are name conflicts the mixin hides super class definitions.

A mixin application is illustrated in Figure 2.2 using the `HelloWorld` example as given in Listing 2.1. The `HelloWorld` class is a template for a family of `HelloWorld` in-

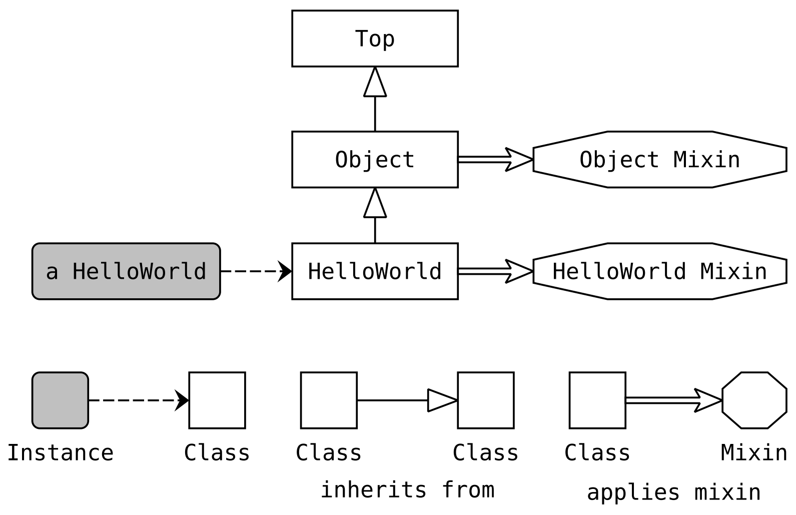


Figure 2.2: The `HelloWorld` class is built by the application of the `HelloWorld` mixin to the Object class. Thereby, `HelloWorld` inherits all behavior from `Object` which is not explicitly overwritten by the `HelloWorld` mixin. Similarly, the `Object` class is a product of the mixin application of the `Object` mixin to the empty class `Top`.

stances. They all implement the same functionality, which is defined in the `HelloWorld` class declaration and all super classes.

`HelloWorld` inherits from `Object`. The inheritance is implemented by defining a `HelloWorld` mixin implicitly through the class definition. The mixin is applied to the class returned by sending the message `Object` to self. Specifically at this point, inheritance is late bound. Because object state cannot be accessed directly, a getter method is created which returns the defined class. First, that getter method fetches the super class by executing an implicit send with the super class name as the selector. Then it applies the mixin from the class definition to that super class.

When following the super class chain of a class, every class, directly or indirectly, inherits from the `Object` class. The specific `Object` class depends on the underlying base system of the Newspeak implementation. In this case, where Squeak is used as the base system, the `Object` class is defined in the `KernelForSqueak` module.

The same mixin-inheritance principle applies to `Object`, as to any other class. Object inherits from an empty class `Top` by applying the `Object` mixin. This way, all behavior in Newspeak is described in mixins.

The empty class `Top`, from which the `Object` class should inherit according to the Newspeak specification, does not exist in the current Squeak based Newspeak implementation. In Figure 2.3, the actual meta model of the Squeak based implementation of Newspeak is shown. The super class of Newspeak's `Object` class is the Squeak class `ImplementationBase`. It lives in the Squeak part of the system and has its own meta class also living in Squeak. The `ImplementationBase` class is also not empty.

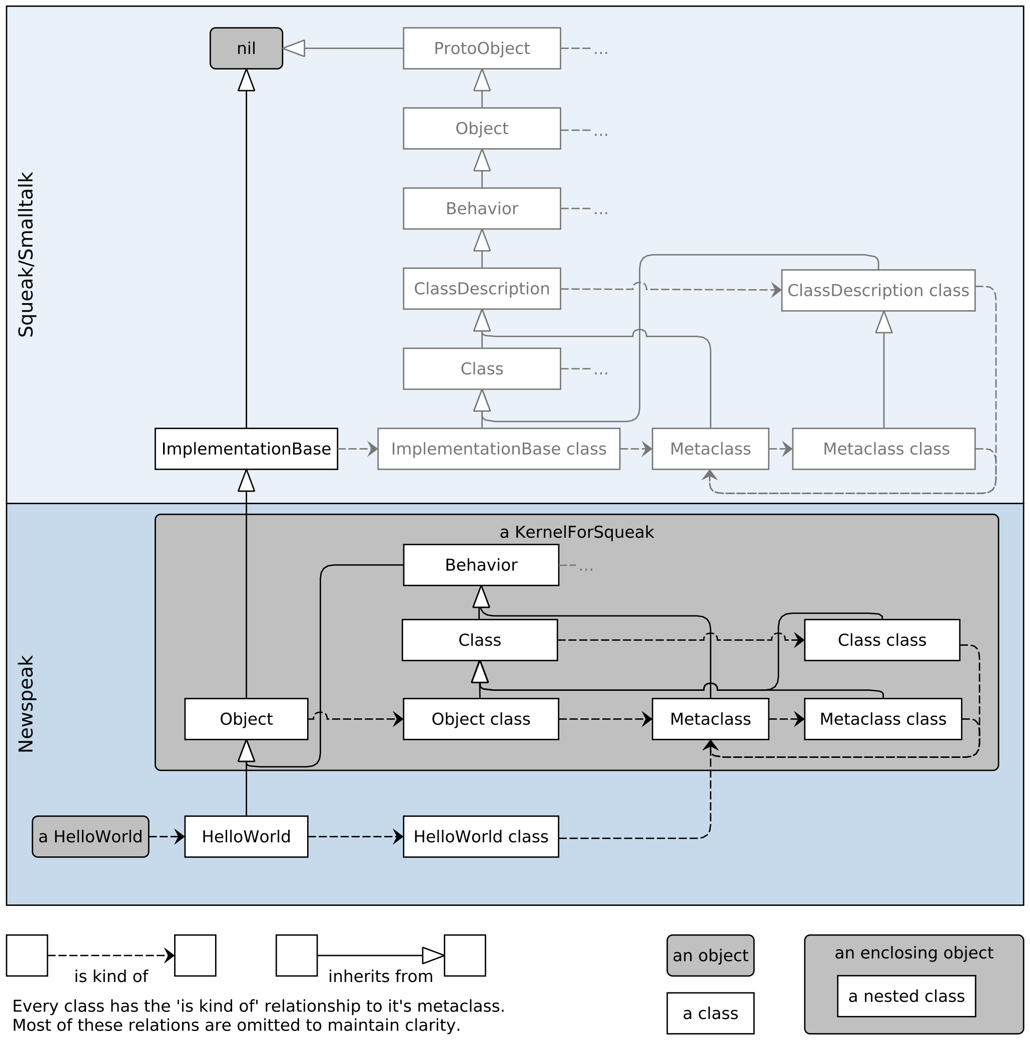


Figure 2.3: The Newspeak metamodel. Classes can either be top-level classes or be enclosed by enclosing objects. Classes are first-class citizens in Newspeak and, thus, have their own class (so called *meta class*). Classes like `Class`, `ClassDescription`, `Behavior`, or `Object` all have meta classes in Newspeak as well as in Smalltalk. Not all of them are explicitly shown in the figure.

Classes are regular objects like every other object in Newspeak. Thus, they have their classes, which are so called *meta classes*. In Figure 2.3 it can be seen that the class of the `HelloWorld` class is `HelloWorld class`, and that the class of `HelloWorld class` is `Metaclass`. The metaclass chain—the list of metaclasses a class has—has a loop, the so called *strange loop*. It can be found a the `Metaclass` class: The class of `Metaclass` is `Metaclass class` and vice versa.

Interestingly, the meta classes in Newspeak are similar to those in Squeak. The meta classes and their respective super class chain for Newspeak and Squeak are listed in Figure 2.3. Newspeak and Squeak have a very similar naming scheme and layout of their meta classes, which emphasizes that Smalltalk is one of the origins of Newspeak.

2.2.1 Newspeak's Exchangeable Base Systems

The Newspeak environment can be executed on top of several other programming languages and environments, which we call *base systems*. Currently supported base system languages are JavaScript [3], Dart [30], and Squeak/Smalltalk.

Unless developers explicitly care about the base system—for example if they want to use interface elements of the base system—they do not need to care about the fact that the Newspeak environment is running on top of another language. The Newspeak compiler compiles Newspeak entities, like classes or methods, directly into executable entities of the base system. Therefore, the Newspeak environment is split into multiple parts: independent Newspeak-only parts and parts which handle matters of the underlying base system.

The Newspeak kernel depends on the base system. It holds vital classes like `Object` or `Class` which are crucial to construct Newspeak objects. Therefore Newspeak has one kernel module for each base system, namely `KernelForSqueak`, `KernelForDart`, and `KernelForV8`. This means that if a class inherits from `Object`, the actually used `Object` is chosen from the kernel module that matches the current base system.

In the scope of this work and for the presented implementation, the Squeak base system is used. All findings of this work and in our implementation can be applied analogously to the other base systems with the exception of Squeak-specific details.

2.2.2 Reflections in the Mirror

Newspeak's reflection capabilities are based on a mirror system. Mirrors are objects that provide meta facilities for other objects. They are used to separate base-level functionality from meta-level functionality to support three important design principles: encapsulation, stratification, and correspondence [6].

The principle of *encapsulation* states that the implementation of a module should not depend on the actual implementation of other modules. Instead it should rely on a purposely exposed API of other modules. When respecting that principle, changes of used modules affect the implementation much less and enable an easier dependency management, for example by using semantic versioning. Good encapsulation often cannot be achieved with classical reflection APIs as shown in a case study [6].

Stratification is a principle of software engineering that states that meta-level functionality must be separated from base-level functionality. It yields the benefit that a feature does not impose costs if it is not used, because it is possible to not deploy it if it is separated.

Classical reflection APIS, which are woven into any object's base functionality, invoke different kinds of costs. A programmer looking at the interface of an object has to decide which method to use. Some of them providing meta level functionality, whereas the developer is looking to implement base level functionality. This may enlarge the probability of errors due to choosing the wrong methods and might enlarge development time and costs. Another cost factor is maintenance, especially for security. When meta-level facilities are easily reachable, it is often possible to access objects in an unforeseen way. This may enable an attacker to attack the whole system, after successfully exploiting a security hole in a single module. Separating the meta level functionality into mirrors has the benefit that it is possible to ship software, which should not use reflection, without the mirror system. This can also reduce the size of deployed systems.

Ontological correspondence states that the ontology of meta-level functionality should correspond to the ontology of the reflected system. Because computing environments are temporally separated between code at compile time and computations at run time, the mirror system should also separate between code and compile time. On the other hand a language implementation is structurally separated into different language constructs (like objects, classes, modules, types, or comments). Ideally the reflection system should provide one entity for each language construct.

The usage of mirrors in Newspeak solves some of the issues listed above. It is possible to ship applications without meta facility simply by not providing the mirror classes in the `platform` object. Newspeak's mirrors implement most meta-level facilities, but they do not implement *all* meta-level functions. For example the `class` method, which is implemented on `Object` and returns the class of the receiver, operates on the meta-level and is still present in every object's interface.

Newspeak provides a rich landscape of different mirrors. As outlined in Figure 2.4, which shows all mirrors which are relevant for Newspeak and the Squeak base system, there are many different types of mirrors for different purposes. For example, Newspeak's mirrors distinguish between different levels of the system architecture. For language constructs in the Newspeak language, appropriate mirrors in the `MirrorsForSqueak`[2] module can be found. Similar mirrors, but on a lower level, can be found in the `LowLevelMirrorsForSqueak` module, which translates between Newspeak and the Squeak base system. Furthermore, there are specific mirrors to handle Newspeak source code. This serves the structural as well as the temporal ontological correspondence. There are mirrors for all the different Newspeak language constructs on the source code level as well as on the runtime level.

[2]Fortunately, programmers do not have to know the base system their Newspeak application is running on to get the mirrors from the correct module. Instead, the correct mirror module is made available via the `platform` object when the `mirrors` message.

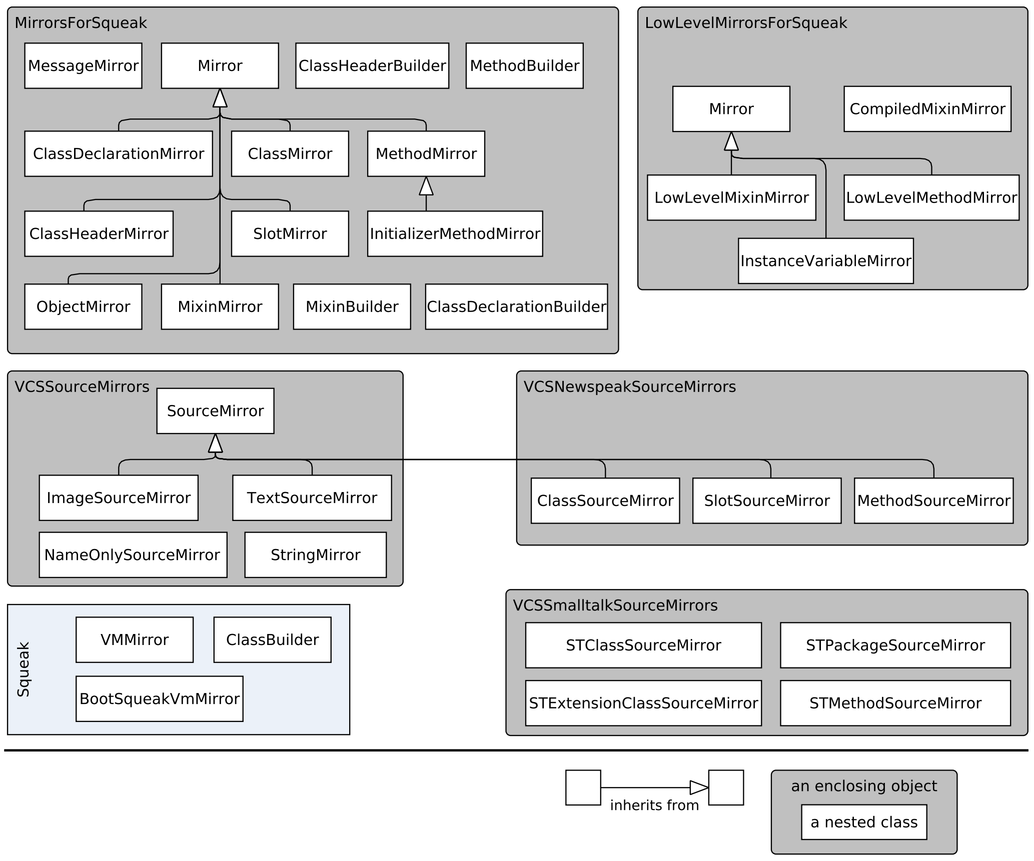


Figure 2.4: The Newspeak mirror landscape with focus on the Squeak base system. For a more detailed snapshot of currently implemented mirrors, see Figure A.1 in Appendix A.

The mirror landscape in Newspeak is heterogeneous and split into different modules. Some mirrors inherit from a `Mirror` class, other don't. There are even multiple different `Mirror` classes living in different modules. The advantage of this design is the independence of the different modules. It is possible to initialize one mirror module without initializing the other modules. Some mirrors, like the `VMMirror`, are not implemented in Newspeak, but as a Smalltalk class. This is necessary because the `VMMirror` needs to call Squeak VM primitives, which is not possible in Newspeak. One common convention is that mirrors can be initialized with the same default initializer: `reflecting: objectToBeReflected`.

Reflective languages in general can be divided into three categories [6]:

- *Introspection*: The program is able to examine its own structure.
- *Self-modification*: The program is able to change its own structure. This includes the ability to execute dynamically generated code.
- *Intercession*: The semantics of the underlying programming language can be changed.

Mirrors in Newspeak usually are capable of introspection and self-modification. The scope of the reflective capability depends on the actual mirrors. Some mirrors reflect source code, other on live objects, and others may only allow introspection of the public API of an object.

Another entity in Newspeak's reflective system are *builders*. They allow building new runtime entities, like classes or methods, which can be installed at run time in the live environment. By using builders it is possible, for example, to construct a new class—including methods and nested classes—in one step and install the finished class into the runtime environment in a second step.

2.3 Newspeak Message Send Types and Access Modifier Semantics

Newspeak is an object-oriented language where all interaction within and between objects happens solely via message sends. A message send *invokes* a method on an object. It consists of four parts which are crucial for the execution of the send:

- The *receiver* is the object in whose context the method of that message send will be executed. The receiver can be given explicitly or implicitly.
- The *sender* is the activation of the current method [2].
- A method can be accessible under a name in the context of the receiver. That name is the *selector* of the method.
- A method may require one or more *parameters* which will be passed along with the message send.

Newspeak provides different syntax for different types of message sends. One distinguishes between unary, binary, and keyword sends which are syntactically similar to the corresponding Smalltalk sends. An additional send syntax, the setter send, is syntactic sugar to set values on objects which enable chaining and eliminate excess parenthesis. The setter send `someSlot:: someValue` is equivalent to the more verbose expression `[:v | someSlot: v. v] value:(someValue)`.

According to the Newspeak specification, methods are defined by the mixin in which they are declared. A method is defined for a class if it is defined by the class' mixin or its super class. When invoked, the method is executed in the context of the receiver and passes control back to the sender if it is finished. In case no explicit return statement is given in the method, the receiver, which corresponds to `self` in the methods context, is returned.

Different types of message sends exists in the Newspeak language to provide functionality, security, and convenience: the ordinary send, self send, super send, outer send, and implicit receiver send. The message send types are distinguished by the way the receiver is defined:

- *Ordinary send*: The receiver for an ordinary send is defined by a (possibly parenthesized) expression. The expression, when evaluated, yields the receiver object. The execution of ordinary sends is described in subsection 2.3.1.
- *Self send*: Using the reserved word `self` as the receiver results in a self send. Self sends are further described in subsection 2.3.2.
- *Super send*: The reserved word `super` is used to do a super send which is described in subsection 2.3.3.
- *Outer send*: If the receiver has the form `outer N`, in which `N` is an identifier, the resulting send is an outer send as described in subsection 2.3.5.
- *Implicit receiver send*: It is possible to omit the receiver of a send (except for binary sends). In this case the receiver is defined implicitly as described in subsection 2.3.4.

2.3.1 Ordinary Send

Ordinary sends invoke `public` messages on the receiver object. They consist of an explicit receiver expression followed by a message clause. To execute an ordinary send, first the receiver expression is evaluated yielding the receiver object. Then the method matching the selector is looked up on the receiver's class.

The method lookup for ordinary sends searches the super class chain of the receiver for a method with a selector that matches the requested selector of the message send. We define the term *super class chain* of a class recursively as: A list of classes that contains a class and all super classes in the class hierarchy.

Method lookup always starts at the receiver's class. If a `public` method matching the selector of the message send is defined by the mixin of the receiver's class, it will be executed. Otherwise the lookup continues to the next class in the super class chain. The lookup stops if either a method with a matching selector was found or if all classes in the super class chain are searched unsuccessfully.

If no matching method could be found, the `doesNotUnderstand:` method is invoked with a message mirror of the failed message as an argument. This method is defined as a `protected` method on `Object` so that it is part of every object's internal behavior. The default implementation of `doesNotUnderstand:` raises a runtime error. Subclasses may overwrite it to implement custom behavior, for example to proxy message sends to other objects.

Considering the scope of implementing access modifiers, asynchronous sends are a special case of ordinary sends. An asynchronous send consist of a receiver expression followed by the asynchronous send token `<-:` and a message clause. Like ordinary sends, the receiver expression yields the receiver to which the message will be send. However, the message send is not executed synchronously but it is sent to the actor associated with the receiver. The message is placed in the message queue of the receiving actor following the actor semantics.

2.3.2 Self Send

Self sends are evaluated similarly to ordinary sends, except that they use the reserved word `self` instead of the receiver expression. The receiver of a self send is `self`—self is the object in whose context the current method is executed.

Like ordinary sends, self sends follow the super class chain to lookup the method with a matching receiver. Contrary to ordinary sends, they have more access rights. If the method class (the class in which the current method is implemented) has a `private` method with a matching selector that method is called. Otherwise, `protected` or `public` methods are looked up beginning at the class of `self`.

Listing 2.2: A message send example: The message `hellowWorld` is send to self in two ways: As an ordinary send (line 3) and as a self send (line 4).

```
1 | this |
2 this:: self.
3 this helloWorld. "only a public helloWorld method can be found this way"
4 self helloWorld. "helloWorld could have any access modifier"
```

In contrast to ordinary sends, self sends are not evaluated by evaluating the receiver first and then sending the message to the receiver. Instead of evaluating `self`, it is built-in that the message is send to the current `self`. This is necessary to ensure the access modifier semantics of self sends. If a self send would evaluate `self` to send a message to the result, it could not be distinguished from an ordinary send. The issue is illustrated in Listing 2.2 by showing two ways to send the message `helloWorld` to `self`. The first send is a verbose "self send" which evaluates `self` first and then sends a message to the result. In the actual send in line 3, one cannot distinguish the send from an ordinary send. Thus, the `helloWorld` method can only be accessed if it is `public`. In line 4, a self send is executed, which can access the method `helloWorld` even if it is `private`.

2.3.3 Super Send

Super sends consist of the reserved word `super` followed by a message clause. Very similar to self sends, they search the super class chain for a method whose selector matches the selector of the message send. In contrast to the self send, the lookup starts at the first super class in the chain (and not in the class of self). A super send can only find `protected` or `public` methods.

Even though the lookup starts at the first super class of the receiver, the method is executed in the context of the receiver itself.

2.3.4 Implicit Receiver Send

An implicit receiver send defines its receiver implicitly. Therefore, it only consists of a message clause.

The lookup of an implicit receiver send starts at the method class. Any method, including `private` and `protected` methods, whose selector matches the selector requested in the message send, is searched. If such method cannot be found in the method class, the lookup continues to the enclosing object's class with respect to the method class until the method was found or the top-level class was searched. In case the method could not be found in any enclosing object's class, the lookup continues as a self send.

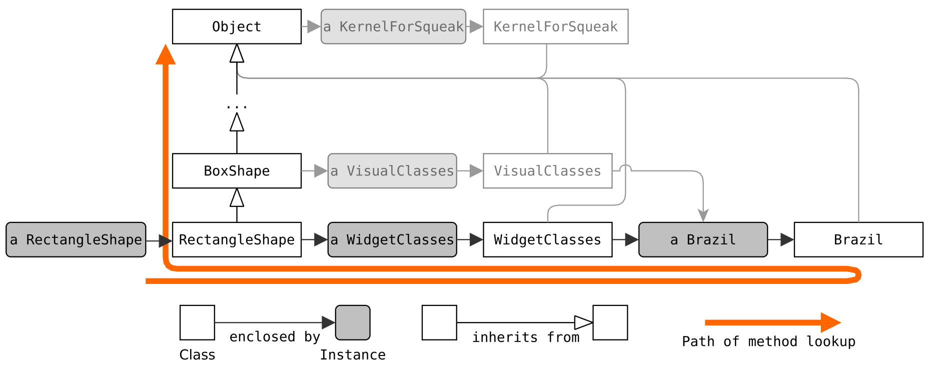


Figure 2.5: The Newspeak method lookup for an implicit receiver send. The lookup of a method beginning from the context of a `RectangleShape` object is shown. The inheritance chain of the `RectangleShape` class is displayed at the y-axis. The nesting of the class can be seen at the x-axis.

The lookup of implicit receiver sends is illustrated in Figure 2.5 starting from a RectangleShape object. The RectangleShape is part of Newspeak's graphical widget library *Brazil*. Therefore, it is nested in the WidgetClasses class and the top-level Brazil module. RectangleShape inherits its functionality from BoxShape as it represents a shape.

When a method in a RectangleShape object is implicitly called, the lookup first searches the enclosing objects. In this example the search for the method would start at RectangleShape and continue at WidgetClass and then Brazil. If the method selector in question was theDesktop, it would return a Desktop`ContainerClasses`Brazil class because the theDesktop method is implemented on the Brazil class. If the requested selector was size, no such method could be found in any enclosing class. Thus, the size method of RectangleShapes super class BoxShape would be found.

Super classes of enclosing classes, or enclosing classes of super classes will not be searched. This is illustrated in Figure 2.5 by the orange line — the path of method lookup.

Other examples of implicit receiver sends can be found in Listing 2.1: In line 10 timesCalled is called, which can be found immediately in self. In line 13 the method Character is called, which is implemented in the super class chain, specifically in Object.

2.3.5 Outer Send

An outer send has the form outer N, where outer is a reserved word and N is an identifier, followed by a message clause. The identifier N must be a lexically enclosing class.

Outer sends are commonly used to access methods in enclosing scopes. This is necessary because methods can be shadowed in the scope lookup. If, for example, WidgetClass and Brazil in Figure 2.5 both define a method with the selector x, an implicit receiver send would find the method defined in WidgetClass. To access the shadowed method x defined in Brazil, an outer send is used: outer Brazil x.

The lookup of an outer send starts at the class specified by the identifier N. If the method cannot be found in N, the lookup continues to search the super class chain of N. An outer send can find protected and public methods. If a private method is declared in N it can be found as well.

2.4 The Newspeak Compiler Architecture

Newspeak program code goes through different stages until it can be executed: It starts as plain text, which needs to be parsed and then compiled. The compiled entity can be made available to the system, ready to be used by other parts of the system.

2.4.1 Parser

The Parser takes Newspeak program code and produces an object graph. That object graph is the abstract syntax tree with which further actions, like code highlighting or code compilation, can be performed.

The Newspeak parsing uses the parser combinator library described by Bracha [1] and Geller [12]. Using parser combinators, the Newspeak parser rules can be read like Extended Backus-Naur Form (EBNF) while still being valid Newspeak code. This is achieved by dividing the parser into many sub-parsers and viewing EBNF operators as methods on those parsers.

Listing 2.3: A part of the Newspeak3 grammar as defined in the `NS3GRammar` class initializer. It shows the definition of a method declaration, a parser stored in the variable `methodDecl`, and how it is used to define further parsers.

```
NS3Grammar = ExecutableGrammar (
(* Grammar for Newspeak3 (without types). *)|
  ...
  methodDecl = accessModifier opt, messagePattern, equalSign, lparen,
      codeBody, rparen.
  category = string, methodDecl star.
  sideDecl =  lparen, nestedClassDecl star, category star, rparen.
  classDeclaration = (tokenFromSymbol: #class), classHeader, sideDecl,
      classSideDecl opt.
  ...
```

An excerpt of the Newspeak grammar is presented in Listing 2.3. It is defined in the `NS3Grammar` class initializer. The grammar is constructed of multiple parsers stored in instance variables. The grammar for a method declaration is stored in the `methodDecl` variable. Just as described in section 2.1, a method declaration consists of an optional access modifier, the method signature (here `methodPattern`), the equal sign, and the method body enclosed by parenthesis.

2.4.2 Compiler

One entity that uses the abstract syntax tree (AST) produced by the parser is the Newspeak compiler. It traverses the AST in multiple passes using a visitor pattern [12] to compile it into an executable form. The "executable form", as in Squeak, is a set of runtime objects like classes and methods.

Thus, the compiler is able to compile Newspeak code and construct low-level mirrors out of them. Those mirrors can later be installed into the Newspeak system by the AtomicInstaller. In the case of adding a slot to a class that means, for example, to compile the new class, swap all references to the old class with references to the new one and migrate all instances of that class to follow the new class layout.

2.4.3 The Squeak Virtual Machine as a Newspeak Interpreter

Newspeak runs on a derivative of the Squeak VM [34].[3] Consequently, the Newspeak compiler produces SqueakVM bytecodes with some modifications.

The most interesting differences between Squeak and Newspeak bytecodes in the scope of this work are the send bytecodes. In Smalltalk there are two kinds of possible send bytecodes: A normal send and a super send. Newspeak also knows the super send bytecode and normal send bytecode, which was introduced as *ordinary send* in subsection 2.3.1. However, Newspeak has more types of sends, namely the self send, implicit receiver send, and outer send. All sends are mapped to an ordinary send by the compiler. Therefore the compiler calculates the receiver beforehand and pushes it onto the stack as a normal ordinary send receiver.

Because outer sends and self sends are mapped using the `normalSendBytecode`, the current VM cannot know what type of send it executes. Without knowing the type of the message send, the VM has no way to know if a class, method, or slot with a specific access modifier can be reached. Therefore, access modifiers are currently not considered during method lookup.

[3]The Newspeak VMs can be downloaded here: http://www.mirandabanda.org/files/Cog/VM/, last accessed June 18, 2015.

3 An Access Modifier Design for Newspeak

In this work we propose a two step approach for adding the missing access modifier semantics to the Newspeak runtime.

Firstly, we propose changes to the Newspeak environment and its VM which result in a Newspeak runtime with access control enabled. We present a way to encode the access modifier information at the various levels of the Newspeak runtime. A way to enforce access modifiers during the method lookup in the VM is shown, as well as changes to the Newspeak runtime to enable reflection on access modifiers at runtime.

Secondly, we present an approach to migrate the existing code base. Existing Newspeak programs look up their methods as if they were `public`, even though they might be defined as `protected` or `private` in the source code. Our approach migrates Newspeak programs as well as the Newspeak runtime itself, to still function with enforced access modifiers.

3.1 Encoding the Access Modifier Information

The Newspeak grammar, and therefore its parser, already support access modifiers. Access modifiers can be used in Newspeak source code as described in the Newspeak grammar. It states that there are *four* possible ways to specify an access modifier: An access modifier can be defined either explicitly to be `public`, `protected`, or `private`, or implicitly by omitting the access modifier. If the access modifier is not defined, it is implicitly interpreted as if the entity is defined to be `protected`. Class, method, or slot definitions all define their access modifier as the first part of their grammar.[1] The parser parses the access modifiers and ignores them.

We changed the parser to not ignore access modifiers so that they are forwarded to the compiler. The compiler needs to encode the access modifier information in a way that enables the Newspeak runtime to reflect on access modifiers and react to an attempt of unauthorized class, method, or slot access.

[1]With the exception of top-level classes. They must not have an access modifier, because they are defined to be `public`. We discuss a simplification of the Newspeak grammar regarding this exception in Appendix C.

3.1.1 A Place to Store Access Modifier Information

One important fact is that Newspeak never directly accesses classes or slots. As outlined in chapter 2, an access to a class or slot is done by calling an automatically created getter or setter method. We conclude that it is sufficient to check access violations solely on methods, which greatly reduces the problem space.

It is a common pattern to check access modifiers and attempts to access protected resources at compile time (as it is done in Java [16]). Because Newspeak is a dynamic language, the compiler does not have enough information to decide whether a method call fulfills the access restrictions. It is possible, for example through meta programming, to alter the method lookup so that another method than the originally intended method is found. Therefore, access restrictions are checked at runtime on every method call. Because the different types of sends, have different method call semantics, the access modifier check also depends on the type of the method call.

Thus, it is convenient to store the access modifier information in the method object itself. The method object can be read by the VM to decide if it can be returned by the method lookup. Newspeak has a built-in reflective system for method objects which can be extended to store and read access modifiers from method objects.

3.1.2 Encoding Access Modifier Information in Method Objects

The method object, in which the access modifier information is to be stored, is an instance of the Squeak class `CompiledMethod` in the Squeak-based implementation of Newspeak. A compiled method consists of a 4 byte method header, method literals (4 byte for each literal), a variable number of bytecodes, and at least one byte for the method trailer.

The method trailer stores metadata for a method, for example a pointer to the method's source code. It would be possible to introduce a new kind of method trailer which encodes the method's access modifier. But since only two bits of information need to be stored for the three possible access modifiers, a method trailer introduces too much overhead. The encoded information of a method trailer is intended to be encoded and decode by a subclass of `CompiledMethodTrailer`. If the VM needs to read the access modifier of a method it either needs to instantiate such a method trailer instance, which would cause overhead, or needs to duplicate the logic to decode the access modifier of the trailer, which would introduce an unnecessary dependency to the Squeak code of the method trailer.

Another place to store metadata in a `CompiledMethod` is the method header. The method header has different layouts for different bytecode sets. Since the "alternate instruction set"[2] is used for current Squeak-based Newspeak implementations, we can focus on the method header layout of the alternate bytecode set.

[2]The alternate instruction set is explained in this forum thread http://forum.world.st/Multiple-Bytecode-Sets-td4651555.html written at Oct 16, 2012.

Table 3.1: The purpose of each bit of the 4 byte header of `CompiledMethods` for the alternate bytecode set. We propose to re-assign bit 28 and 29 to encode the access modifier of a method.

Bit Position	Old Header Layout	Proposed Header Layout
30	flag-bit for the alternate bytecode set	no change
29	unused	access modifier flags
28	user-level flag-bit[1]	
27–24	number of arguments (4 bit)	no change
23–18	number of temporary variables (6 bit)	no change
17	flag-bit indicating a large frame size	no change
16	flag-bit: method has a primitive	no change
15–0	number of literals (16 bit)	no change

[1] The flag-bit is ignored by the virtual machine and is only used ad-hoc by user-level programs.

In Table 3.1 the proposed re-assignment of bit 28 and 29 of the method header to encode the access modifier of a `CompiledMethod` is shown. Bit 29 is currently not assigned to any official purpose - this bit can be used without further complications. Bit 28 is used as a user-level flag by any Squeak program that may need a bit in the method header. Currently, it is not used in the Squeak image. To the best of our knowledge there is no Newspeak application using that bit, so we re-assign bit 28 in our context.

We propose to assign the different access modifiers bit 28 and 29 as shown in Table 3.2. If both bits are `0`, the access modifier is not defined, so that the implementation is backwards compatible. A compiler that doesn't know about access modifiers would typically not set those bits, implicitly assigning the method the `undefined` access modifier state. The VM should handle the `undefined` modifier as if it was `public`, so that our extension remains as compatible as possible with the old header.

Table 3.2: The proposed access modifier encoding.

Bit 29	Bit 28	Assigned Access Modifier
0	0	undefined[1]
0	1	private
1	0	protected
1	1	public

[1] Should be handled by the virtual machine as if it was `public`.

3.2 Enforcing Access Modifiers

The VM handles the execution of each method send in the Squeak-based implementation of Newspeak. Therefore, we need to adapt the VM to validate that message sends conform to the attached access modifier semantics.

3.2.1 Modifications of the Method Lookup

When the VM needs to execute a message send, it does the following abstract steps:

1. Consume the send's bytecode, the selector, receiver, and arguments from the stack.
2. Find the object to start the method lookup, which depends on the type of the send.
3. When handling an implicit receiver send, look up the method in the lexical scope. If a method was found, prepare and then execute the new method. In that case do not execute further steps.
4. Look up the superclass chain for the method. If a method could be found, prepare and then execute the new method. In that case do not execute further steps.
5. No matching method can be found. Start a new message send with the selector `doesNotUnderstand:` to handle this case in Squeak/Newspeak.

We propose to alter the method lookup phase of the VM to also check for a method's access modifier. At this phase, the VM already has access to the method object, including the header in which the access modifier is stored. In each phase of the lookup the valid access modifiers for each message send type are known.

The implicit receiver send is the only send type that can lookup the lexical scope as described in item 3. This step does not need to be altered in any way because every method that is found is allowed to be executed independently of its access modifier as explained in subsection 2.3.4.

The method lookup in the superclass chain needs to be adapted. If a method is found, the VM needs to decide if the method is allowed to be executed depending on its access modifier. A `public` method can be executed for every send type. Only self sends, super sends, outer sends, and implicit receiver sends are allowed to execute a `protected` method. For self sends and outer sends, `private` methods can be executed only if the method is found in exactly the same class where the method lookup started. Thus, the VM needs to have access to three entities to decide whether to execute a method: the type of the send, the class where the method lookup started, and the class in which the method with a matching selector was found.

3.2.2 Possible Reactions to an Illegal Method Call

In this section we discuss how the Newspeak language should react on a method call with insufficient access rights. We discuss several reactions and finally conclude on why the behavior outlined in the Newspeak specification is the most appropriate reaction.

One reaction could be to silently ignore method calls with insufficient access rights. Another option is to let the program halt. Both ways are obviously impractical and inflexible. A better approach is to raise a runtime error, which can be handled by the Newspeak program.

Raising a Special Runtime Error A new error class could be created for that purpose, for example `AccessModifierMismatchError`. The error would include information about the message (which includes the sender and selector) and the allowed and actual access modifier to indicate the mismatch. When this error is to be raised, the VM could follow the pattern of the `MessageNotUnderstood` error. Instead of sending `doesNotUnderstand: aMessage` it would call the

```
accessModifierMismatch: aMessage expectedAccessModifier: modifier
```

method as a self send on the receiver. It assumes that the method is implemented on the `Object` class and raises the `MessageNotUnderstood` error when called. Subclasses of `Object` may override the method to react on an access modifier mismatch in a custom way.

However, a separate Error for an access modifier mismatch has one drawback: It reveals the existence of a `private` or `protected` method even though the method should be hidden from an ordinary message send. This can lead to undesired behavior, such as exposing the existence of a proxy which intended to fully mimic an object, and might affect a programs security.

Listing 3.1: The implementation of a proxy class. The proxy forwards all messages it receives to the `target` object. It also implements at least two non-public messages: `target` and `doesNotUnderstand`. The existence of those messages must not be revealed so that the proxy can effectively act like the `target` object.

```
class Proxy on: anObject = (
  (* A proxy that forwards ALL messages to the target object. *)
  | private target = anObject. |
)
('as yet unclassified'
  doesNotUnderstand: aMessage = (
    ↑ aMessage sendTo: target
  )
) : ()
```

A proxy class, as listed in Listing 3.1, which has some `private` methods to handle its internals, accidentally reveals the existence of its internal methods if an `AccessModifierMismatchError` would be thrown. An object that owns a proxy, could distinguish between the target and the proxy on that target by its behavior, which might be a security concern. It requires more work to implement such a proxy with the existence of a separate access violation error. The call-forwarding logic, which is implemented in `doesNotUnderstand:` in the proxy, needs to be duplicated in the error handler.

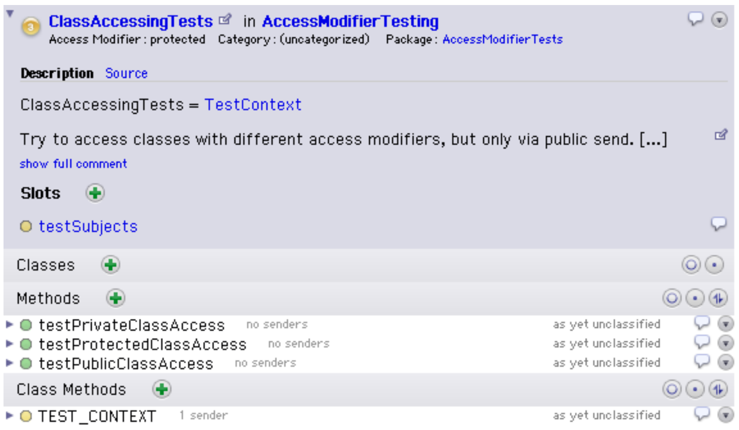
Denying the Existence of Not-Accessible Methods We conclude that an object should act as if a called method does not exist if it cannot be reached due to its access modifier. During the method lookup the existence of a method is only acknowledged if it exists and if its access modifier allows execution. Like the previous solution, this solution allows flexible meta programming—by overwriting the `doesNotUnderstand:` method of a class—while keeping the existence of an internal method a secret.

When overwriting a method of a superclass, it is possible to give the new method a different access modifier than the shadowed method of the superclass. This is not a problem if the new method is more accessible—for example when the superclass defines a `private` method which is defined as `protected` in a subclass. Although, the Newspeak specification states that "subclasses should not, as a matter of good practice, reduce the accessibility of inherited methods" [2], the case of reducing a method's accessibility has to be handled. Because the existence of a method is denied on a method call with insufficient accessibility, the VM should continue the method lookup in the hope to find another matching method in a superclass. Therefore, we agree with the Newspeak specification: If a method with a matching selector but an insufficient access modifier is found, the method lookup should skip the method and continue the lookup.

3.3 IDE support of Access Modifiers in Newspeak

To write Newspeak programs that use access modifiers, program code not only needs to be compiled with access modifiers. The system should also provide tool support to developers. Thus, it should be possible to reflect on the compiled entity and get information about its access modifier. Therefore, the reflective system of Newspeak needs to be extended. Newspeak's developer tools—like the class browser and code editor—need to inform the developer about the access modifiers of classes, slots, or methods.

Ideally, the programming environment does not only communicate access modifiers through source code, but also in a more concise way. The Hopscotch IDE has two views for classes and methods: an expanded view showing all the details and program code, and a summary view which should fit in one line of text. As visualized in Figure 3.1, we indicate the access modifier of a class, slot, or method in the summary view with a colored symbol. The color indicates the access modifier:



ClassAccessingTests in AccessModifierTesting
Access Modifier: protected Category: (uncategorized) Package: AccessModifierTests

Description Source

ClassAccessingTests = TestContext

Try to access classes with different access modifiers, but only via public send. [...]
show full comment

Slots
testSubjects

Classes

Methods

testPrivateClassAccess	no senders	as yet unclassified
testProtectedClassAccess	no senders	as yet unclassified
testPublicClassAccess	no senders	as yet unclassified

Class Methods

TEST_CONTEXT	1 sender	as yet unclassified

Figure 3.1: The modified code editor with the proposed access modifier extensions: The `ClassAccessingTests` class displayed is expanded in the Hopscotch IDE. The slots and methods of the class with color-coded access modifiers are shown.

green for `public`, yellow for `protected`, and red for `private`. In the extended view, slot and method definitions expose their access modifier directly in the source code. For extended class views, as seen in Figure 3.1, the class declaration is not shown. Therefore we textually indicate the access modifier just below the class name.

3.4 Migration to an Environment with Enforced Access Modifiers

Before we began our work on Newspeak, the parser of Newspeak was capable of parsing access modifiers—they had no effect, but the source code was valid. For the scope of this work we assume that this behavior of the parser made programmers write their programs using access modifiers, but without the ability to verify their validity. For example, the words "public", "protected", and "private" can be found 1740 times in the 86129 lines of the Newspeak code[3] distributed with the Newspeak image and VM.

If we enabled access modifiers in Newspeak, existing Newspeak programs, including the programming IDE and Newspeak core, would break. We do not want to break the existing Newspeak code base, which is why we need a way to migrate the source code. Therefore, we propose to modify the VM so that it optionally prints a warning to its standard output instead of raising an error when a method send cannot be executed due to insufficient access modifiers. Message sends will succeed as if every class, slot, or method was `public`. The access modifier warnings are collected from the VMs standard output into a file. After some time of using Newspeak, for

[3] As of commit `2bde0a10` from 2013-09-19 in the Newspeak source code repository.

example by running the test suite, compiling programs, or browsing through source code, a potentially long list of access modifier violations are collected in that file.

We contribute a Newspeak program which can parse such a file and upgrades access modifiers in the Newspeak program code. This program needs to know the receiver of the message send, the selector of the message, and the implementor of the method. Provided with that information for each failed method send, the program can find the current access modifier. If it finds an access modifier violation for a `private` method, slot, or class, the access modifier will be upgraded to `protected`. Similarly, a `protected` access modifier will be upgrade to `public`. When upgrading an access modifier, a comment is inserted which states that the access modifier was automatically upgraded and lists the old access modifier. This makes the review of the programs changes easier. Finally, changes made by the upgrade program can be merged back into the Newspeak source code repository.

There are multiple possible strategies to upgrade access modifiers. One strategy is to first downgrade all access modifiers—even those stated implicitly by not specifying a modifier in the source code—to `private`. In a second step, the upgrade procedure can be run as described previously to upgrade the access modifiers until no more errors appear. This strategy has the advantage that the access modifier of every entity in the Newspeak environment is specified with as little access rights as possible while still having a working Newspeak environment with access modifiers enforced. However, it is possible that the interface of a specific Newspeak class was intentionally made `public` to be used by other (potentially not loaded) Newspeak programs. The strategy would alter the interface of Newspeak classes without considering the *intention* of the interface.

We decided to use another strategy that does not change intentionally placed access modifiers. It only downgrades access modifiers which were defined implicitly. This would leave every intentionally set access modifier intact, while downgrading the access modifiers which were not specified previously. In the second phase, where the access modifiers are upgraded again, all access modifiers are considered again.

Because we change the interface of Newspeak classes with the upgrade program, changes made by that program need to be reviewed by the Newspeak community. The number of access modifier changes is high, which is why they should be split into chunks of related changes. We assume that a smaller set of related access modifier changes leads to a better code review. Unfortunately, we do not have a sufficient grouping algorithm at hand which works automatically. Therefore, we group the changes differently: The upgrade program is run multiple times. For each run another part of Newspeak's functionality is executed, the access modifier violations are collected and fixed. The resulting source code changes are grouped into one change request that can be reviewed by the community. We assume that grouping makes it easier for the code reviewers to review the access modifier changes made by our algorithm in a given context.

4 Implementation

This chapter provides some details on how we implemented the solutions outlined in chapter 3. Therefore, we present in section 4.1 how the access modifier information was encoded, in section 4.2 how we enforced access modifiers in the VM, and in section 4.3 how the Newspeak runtime environment can reflect on access modifiers. Finally, we present details about our implementation of the access modifier upgrade program in section 4.4.

4.1 Encoding the Access Modifier Information

As presented in section 3.1, we encode the access modifier information in `CompiledMethod` objects. To that end, we change the interface of the `CompiledMethod` class to include an `accessModifier` getter and setter method. Both methods access the method header of the `CompiledMethod` object.

The `#accessModifier:` setter is presented in Listing 4.1. If the `CompiledMethod` uses the alternative bytecode set, the access modifier, which is either the symbol `#private`, `#protected`, or `#public`, is converted into the encoded value as per Table 3.2. Then the access modifier bits in the method header are set to `00` with a bit mask. The new access modifier bit can then be set on the method header with a bit-wise `or` operation. The getter method for the access modifier retrieves the access modifier in a similar way. In addition to the getter and setter methods for access modifiers, we added the convenience predicates `isPublic`, `isProtected`, and `isPrivate`.

Just like the accessor methods for the Newspeak runtime, which are implemented in the `CompiledMethod` class, there is an accessor method in the VM to reflect on access modifiers (shown in Listing 4.2). Two pragmas[1] are defined in the source of the `accessModifierFlags` method. The first pragma (line 7) states that the method should be inlined into methods which call the getter, whereas the second pragma (line 8) defines a variable `cogMethod` in the generated C source code. The accessor reads the encoded method header of the currently handled method, which is stored in the global variable `newMethod`, and returns it. The `newMethod` variable is set on every message send during the method lookup. The `accessModifierFlags` method receives the methodHeader of the method. Therefore it needs to differentiate between normal `CompiledMethod` objects and methods that have been compiled by the

[1]A pragma is an annotation attached to a `CompiledMethod` in Smalltalk. The concept of pragmas is used in Slang to modify the C source code generation of Slang methods.

Listing 4.1: The implementation of the setter method for the access modifier of a `CompiledMethod`. It is written in Smalltalk, because it is a Smalltalk class.

```
accessModifier: accessModifier
 "set header bits 29-28 in Alternate Bytecode Set"
 | accessModifierBits header |
 self usesAlternateBytecodeSet ifTrue: ["accessModifier bits:
  00 undefined
  01 private
  10 protected
  11 public"
 "we rely on the fact that indexOf: returns 0 for objects not in the array"
 accessModifierBits := (#(#private #protected #public) indexOf:
     accessModifier).
 header := self header bitAnd: (3 bitShift: 28) bitInvert. "header with reset
      accessModifier bits"
 self objectAt: 1 put: (header bitOr: (accessModifierBits bitShift: 28))].
```

just-in-time compiler (JIT) because they have a different layout (lines 11 to 14). Finally, it takes the header and extracts and returns the access modifier bits (lines 17, 18).

4.2 Enforcing Access Modifiers

To enforce the access modifier semantics of Newspeak, two changes need to be implemented in the VM: First, new message send bytecodes need to be added so that we can distinguish a self or outer send from an ordinary send. Second, the method lookup of the VM needs to be altered so that it considers access modifiers.

As described in section 2.3, the different message send types in Newspeak have different access modifier semantics. Therefore, the VM needs to know which message send type is currently executed to decide whether the access modifier of a method is sufficient or not. Prior to this work, all the send types were functional in Newspeak, but the self send and outer send were emulated using the ordinary send by the Newspeak compiler. We changed the self send and outer send so that they use their own bytecode, removing the need to explicitly push the receiver.

In case of implicit receiver sends and outer sends, the receiver cannot be known at compile time. Therefore the Newspeak derivative of the Squeak VM introduced `ImplicitReceiverBytecode` and `PushPseudoVariableOrOuterBytecode` as new bytecodes. Both can search the lexical scope of an object to find and push the receiver of the message send onto the stack.

To demonstrate this behavior of the Newspeak compiler for the different send types, we use a method containing an ordinary send, a self send, and an outer send[2] in Listing 4.3. The method was compiled two times: using the old Newspeak compiler

[2]We have omitted implicit receiver sends and super sends in the listing, because they produce the same bytecode for the old and new compiler.

Listing 4.2: The implementation of the getter method for the access modifier of a `CompiledMethod` in the VM. It is implemented in the `StackInterpreter` class and written in the Slang language, a subset of Smalltalk which is compiled to C to build the Newspeak virtual machine.

```
accessModifierFlags
 ”fetches the access modifier flags, returns a number:
  0 - undefined (treated equal to public)
  1 - private
  2 - protected
  3 - public”
 <inline: true>
 <var: #cogMethod type: #’CogMethod *’>
 | methodHeader cogMethod |
 methodHeader := self rawHeaderOf: newMethod.
 methodHeader := (self isCogMethodReference: methodHeader)
  ifTrue: [ cogMethod := self cCoerceSimple: methodHeader to: #’CogMethod *’.
   cogMethod methodHeader ]
  ifFalse: [ self headerOf: newMethod ].
 (self headerIndicatesAlternateBytecodeSet: methodHeader)
  ifTrue: [↑ (methodHeader >> 29) bitAnd: 3 ]
  ifFalse:[↑ 0 ]
```

and using the Newspeak compiler with our modifications. The bytecodes produced by the old compiler are shown in Listing 4.4, whereas the bytecodes produced by the new compiler are shown in Listing 4.5.

Ordinary Sends In the original Newspeak VM an ordinary send consists of two steps: First, the receiver is pushed onto the stack with a push bytecode, then a message send bytecode is used to execute a message send to the receiver. Because the `printString` method does not take any parameters, no parameters need to be pushed onto the stack prior to the message send bytecode. The self send is very similar to an ordinary send. It first pushes the receiver (self) onto the stack and then issues an ordinary send. Thus, the VM cannot distinguish if the send was originally an ordinary send or self send. The same problem occurs with outer sends. Again, the receiver is pushed onto the stack with a special push bytecode, which can push an enclosing object. After the receiver was pushed, an ordinary send is issued to simulate the outer send.

To let the VM know which message send type is used, we introduce two additional send bytecodes: the self send bytecode and the outer send bytecode. The self send bytecode is a two-byte bytecode where the first byte consists of the bit pattern `<11110101>` identifying the bytecode as a self send. The second byte consists of the pattern `<iiiiijjj>`, which uses 5 bits to encode the position of the selector in the literal table, and 3 bits to encode the number of parameters for the message send.

Listing 4.3: A method containing an ordinary send, a self send and an outer send. The class implementing this method is nested in the `MyApplication` class, so that the outer send works. The `printString` method is implemented on `Object` and, thus, can be send to every object.

```
testTheCompiler = (
  5 printString.
  self printString.
  outer MyApplication printString.
)
```

Listing 4.4: The method listed in Listing 4.3 compiled with the unmodified Newspeak compiler. The self send and the outer send are implemented using the ordinary send bytecode.

```
<E5 05> pushConstant: 5
<71> send: printString
<DC> pop
<4C> self
<71> send: printString
<DC> pop
<E1 FF 4D> pushExplicitOuter: 1
<71> send: printString
<DC> pop
<D8> returnSelf
```

Listing 4.5: The method listed in Listing 4.3 compiled with the modified Newspeak compiler. The self send and the outer send are implemented using their own bytecodes.

```
<E5 05> pushConstant: 5
<71> send: printString
<DC> pop
<F5 08> selfSend: printString
<DC> pop
<F8 08 01> outerSend:
    printString (depth: 1)
<DC> pop
<4C> self
<D9> returnTop
```

Using extension bytecodes[3] higher literal numbers and parameter counts can be encoded.

The newly introduced bytecodes are produced by our modified Newspeak compiler as shown in Listing 4.5. The self send consists only of one two-byte bytecode. Similarly, the outer send consists of one three-byte bytecode. It is notable that for both bytecodes the receiver is not set by the compiler anymore; it is implicitly given through the specific message send bytecode. This way the higher access rights of outer sends and self sends can only be used on receivers following the self send or outer send semantics. If the sends were implemented to first push the receiver and then execute the send (as it was done previously), a security risk would be present: It would be possible to push *any* receiver and execute a self send on it.

[3]There can be up to two extension bytecodes following a normal bytecode. Both extension bytecodes have two bytes, the first identifying the bytecode. The second byte encodes an arbitrary number used to extend the previous bytecode.

Self Sends and Outer Sends The VM side implementation of the self send and outer send bytecode are very similar to the implementation of the ordinary send bytecode. The send bytecode and extension bytecodes are consumed from the stack, the selector is taken from the literal table, and the specified number of parameters are taken from the stack. The difference to the ordinary send is that the receiver is computed instead of taking it from the stack. For the self send, the receiver of the send is always the receiver of the currently executing method. For the outer send, the receiver is looked up using the same algorithm as the `pushExplicitOuter` bytecode used by the unmodified compiler. To handle the self send and outer send bytecodes, which have not pushed a receiver onto the stack, the calculated receiver is put on the stack retroactively. This is done because the VM assumes that the receiver is placed on the stack at various places, for example when removing the remains of a method execution from the stack. We decided to push the absent receiver onto the stack because it adds less complexity to the VM implementation than adding this corner case to the various places which rely on the position of the receiver.

Method Lookup Since we refactored the VM to use different bytecodes for the different types of message sends in Newspeak, it can enforce the access modifier semantics of Newspeak during the method lookup. We found that, even though there are five types of message sends, there are only two different ways required to handle the method lookup. When looking for a method in the lexical scope, all methods can be found no matter which access modifier they have. When looking for a method in the superclass chain, there are two cases: First, only `public` methods can be found (an ordinary send is executed). Second, all methods can be found if the method is found directly in the receiver object. If the method is defined at any other place, only `protected` or `public` method can be found (when executing a self send, outer send, implicit receiver send, or super send—when executing a super send the lookup starts at the superclass of the receiver).

The existing method lookup functionality in the VM is altered to find only `public` methods and raise an error if any `private` or `protected` method with a matching selector is found. Additionally, a second lookup was added to the VM which allows all methods to be found if the method exists in the receiver. An error is raised if a `private` method is found at any other place. This behavior is not conform to the Newspeak specification (no error should be raised, but the method lookup should continue ignoring the `private` method). The difference between our implementation and the Newspeak specification is a known limitation as described in section 5.3.

We present a more detailed explanation of the method lookup implementation of the interpreter VM in Appendix B.

4.3 Reflecting on Access Modifiers at Runtime

To enable Newspeak programs—for example, programming tools like the class browser—to reflect on access modifiers, Newspeak's mirror system has to be adapted.

All access modifiers are encoded in `CompiledMethod` objects. This information has to be available for method-related, slot-related, and class-related mirrors.

A `MethodMirror` directly reflects on a method. For the Squeak-based implementation of Newspeak the method mirror holds a reference to a `CompiledMethod`. The mirror can obtain the access modifier directly from the `CompiledMethod` it holds.

`SlotMirrors` are a work in progress in the current version of Newspeak. Currently, they are read only. They do not hold a reference to any other mirror and, therefore, cannot dynamically determine their access modifier. Instead, they are initialized by other mirrors with their name, mutability, and access modifier. A `SlotMirror` can forward the access modifier it got during initialization when asked for its access modifier. To enable this behavior, it needs to be extended by a slot capable of holding the access modifier and by adapting the initializer of the `SlotMirror` class to access the access modifier as a parameter. The `MixinMirror` and the `ClassHeaderMirror` are the only mirrors in the Squeak based implementation of Newspeak that currently instantiate `SlotMirror` objects. They have been adapted to use the new initializer of the `SlotMirror`.

Class-related mirrors, which need to be able to give information about their reflectee's access modifier, are the `ClassDeclarationMirror` and `ClassMirror`.

The `ClassDeclarationMirror` needs to be able to reflect on its access modifier because it belongs to the declaration of a class. However, it does not have a reference to an actively used class. Therefore, it does not have an accessor method from which it can deduce its accessibility. Naturally, a `ClassDeclarationMirror` would infer its access modifier from the class declaration.

However, in the current Newspeak implementation only the class header, which does not include the access modifier information, is saved. This needs to be changed in a future version of Newspeak. We choose to implement a work around. The `ClassDeclarationMirror` infers its access modifier by fetching its enclosing class—via the instance mixin it was initialized with—and then searching the method dictionary of the enclosing class for the getter method pointing to itself. The method dictionary of a class contains all methods, even those generated to access slots or classes. Because the access modifier was saved in the method header, the access modifier of a class can be inferred by its accessor method. If there is no enclosing class, we assume that we have a top-level class which are defined to be `public`.

The downside of this approach is that a mixin can be applied multiple times. Therefore, a mixin might return the enclosing class of where it was defined, and not necessarily where the class declaration was built. Because multiple applications of a mixin are allowed in the specification, this workaround has to be removed as part of future work. This is a known limitation as outlined in section 5.3.

The version control system of Newspeak, called Memory Hole [25], can save versions of Newspeak code in different repositories—currently Mercurial [36], git [29], and file-system-based repositories are supported. To serialize Newspeak classes and their slots, methods, and inner classes, a separate set of mirrors is used. The Newspeak source code mirrors are located in the `VCSNewspeakSourceMirrors` module. They can reflect on classes, slots, and methods and have been extended by us similar to the previously described mirrors. Using the extended mirrors, the version control system of Newspeak is able to display added or removed access modifiers and is able to merge them.

4.4 Migration to an Environment with Enforced Access Modifiers

This section describes how the Newspeak code base can be migrated to use the now functional access modifiers. A migration is necessary because of the use of access modifiers in the Newspeak core.

An unchanged Newspeak image cannot correctly start with enforced access modifiers. It fails due to an `MessageNotUnderstood` error before any Newspeak window can be displayed. Similarly, it cannot handle the thrown error, because the user interface for the debugger cannot be displayed without raising another `MessageNotUnderstood` error. A similar issue arises when bootstrapping a new Newspeak image.[4]

The implementation effort for the access modifier migration is split into two parts: We have modified the VM to only check the access modifiers when a specific option is enabled and, secondly, we have developed a Newspeak program which can find access violations and attempts to fix them.

Optional Access Modifier Checks in the Virtual Machine To let the Newspeak image start and bootstrap, we added an option to enable the access modifier checks. If this option is not given, the VM will not check for access modifiers. This way developers can migrate their source code to work with enabled access modifiers. The option may be enabled per default or removed completely in a later version (so that it is permanently enabled).

We have added the command line argument `−enforceam`, which can be given when starting the VM. If this argument is given, the VM sets the variable `enforceAccessModifiers` to `1`, which means that access modifiers will be enforced.

The VM has been adapted to check for access modifier during the method lookup, but only call the `doesNotUnderstand` method if the `−enforceam` flag is set. If it is not set, a warning containing the method selector, the receiver, and the class where the method was found is printed to the standard output of the VM.

The Access Modifier Migration Program The second part of the code migration effort is a Newspeak program that can upgrade access modifiers. It reads the access violation outputted by the virtual machine and searches the method, class, or slot that was called with insufficient access rights. The access modifier of that method, class,

[4] Newspeak images are created starting from a Squeak image, which is modified to create a full Newspeak image. Therefore, a pre-compiled environment (including a compiler) is loaded and then all Newspeak sources are loaded, compiled, and installed. This process is called bootstrapping.

Listing 4.6: The part of the migration program that calculates the new source code of a method when migrating it

```
modifiedMethodSourceOf: methodMirror = (
  | source oldModifier newModifier |
  (* Changes the method’s access modifier.
     Also adds a comment to the source naming the previous modifier.*)

  source:: methodMirror source asString.
  methodMirror reflectee usesAlternateBytecodeSet ifFalse: [ ↑ source ].

  oldModifier:: methodMirror accessModifier.
  newModifier:: newAccessModiferForMethod: methodMirror.
  oldModifier = newModifier ifTrue: [ ↑ source ].

  (source startsWith: (oldModifier, ’ ’)) ifTrue: [
    source replaceFrom: 1 to: (oldModifier size)
           with: ’         ’ startingAt: 1.
  ].

  (Regex string: source prefixMatchesRegex:
    ’\s*\(\* was (private|public|protected) \*\) ’) ifTrue: [
    ↑ (newModifier, ’ ’), (source withBlanksTrimmed)
  ] ifFalse: [
    ↑ (newModifier, ’ (* was ’,oldModifier,’ *) ’),
      (source withBlanksTrimmed)
  ].
)
```

or slot will be upgraded (a `private` access modifier will be changed to `protected`, a `protected` access modifier will be changed to `public`).

For a given class, slot, or method, which has been identified to have a too restrictive access modifier, the source code will be modified and the entity will be re-compiled. For example, when a method shall be upgraded, the `modifiedMethodSourceOf:` method of the upgrade program is used to migrate the method source. The `modifiedMethodSourceOf`, as listed in Listing 4.6, only modifies the method source if the method uses the alternate bytecode set, because only those methods use the modified method header. Then it calculates the new access modifier using the `newAccessModifierForMethod:` method that implements a strategy to upgrade the access modifier. The strategy is to simply to upgrade the access modifier from `private` to `protected` and from `protected` to `public`. After finding the new access modifier, the source code is changed. When changing the source code, an additional comment is inserted which states the old access modifier (except there already is such a comment). The comment documents the automated access modifier change and is intended to help to review the change. This approach takes the Newspeak implementation (while not considering access modifiers) as a reference and adapts the access modifiers according to their usage. Finally, the method returns the new source code which can then be compiled and installed in the Newspeak system. A similar approach is implemented for slots and class definitions, which changes the class header source instead of the method source.

Our approach adapts the access modifiers used in the observed Newspeak system to how the classes, slots, or methods are used. It is possible, though, that the access modifiers are set intentionally and the current implementation shouldn't use a specific `private` or `protected` entity. This case is not covered by our approach. However, the changes done by the upgrade program are commented and meant to be reviewed. We assume that an undesirable access modifier upgrade will be identified during review.

The described process only upgrades access modifiers that are too strict. However, there might be access modifiers that are too loose—for example a method which does not have an access modifier (because they have not made a difference in the current implementation anyway), which should be `private`. We assume that classes, slots, or methods, which have no access modifier set explicitly, can fall into that category. Therefore, our upgrade program is run once with a downgrade-strategy to downgrade every not explicitly specified access modifier. The previously described upgrade process can then be applied to the now `private` entities.

The downgrade approach works well for methods and slots because they store their access modifier in the source code. If the access modifier is stored in the source code, the program can distinguish a `protected` entity from an entity that does not have an access modifier and is, therefore, implicitly defined as `protected`. Unfortunately, class definitions do not store their access modifiers in the class header source. Because our program has no way to distinguish a class which is explicitly set to `protected` from a class which has no access modifier, it currently downgrades all `protected` classes.

5 Evaluation

In this chapter our approach to implement access modifiers for Newspeak is evaluated. The evaluation is two-fold: We evaluate the performance impact of our implementation compared to the current Newspeak implementation without access modifier support. Secondly, our access modifier implementation is evaluated qualitatively by inspecting the errors that occur during image startup. We inspect the behavior of the error propagation, fix the access modifier, and test if the error is resolved. Finally, we point out known limitations of our approach as well as of the current implementation.

5.1 Performance Impact

We expect performance to be negatively affected by our modifications to the VM and to the Newspeak image. It was one of the implicit goals to minimize this effect. This section presents the results of multiple performance benchmarks. The performance benchmarks were executed using a combination of different VMs and Newspeak images:

1. An *unmodified CogVM* VM is run using an *unmodified image*.
2. In the second benchmark run, the JIT is disabled. Apart from that the VM and the image is still unmodified.
3. The *modified CogVM*, a VM that was compiled with our modifications, is used in the third run. The image is still unmodified.
4. In the last run we use the modified VM in combination with a *modified image*. The modified image was built with our modifications.

The *unmodified CogVM* VM was built from the version `VMMaker.oscog—eem.335` in the VMMaker repository.[1] The *modified CogVM* is based on the same version, but includes our changes described in chapter 4. Newspeak images are based on the Newspeak repository[2] mercurial version `e651d4001702` and the Newspeak bootstrapping repository[3] mercurial version `f86ca238aeb4`. The *unmodified image* directly bootstraped from these repositories. The *modified image* is the same Newspeak image with our modifications as described in chapter 4.

[1]http://source.squeak.org/VMMaker, last accessed June 18, 2015.
[2]https://bitbucket.org/newspeaklanguage/newspeak_bleeding_edge, last accessed June 18, 2015.
[3]https://bitbucket.org/newspeaklanguage/nsboot_bleeding_edge, last accessed June 18, 2015.

The JIT was disabled in all but the first benchmark runs, by passing the parameter `−cogmaxlits 0` to the VM. The parameter sets the number of literals a method should have to be just-in-time compiled. Setting it to `0` disabled the JIT. Currently, our modifications do not work with the JIT. Because of that, the enabled and disabled JITs are compared first to see which part of the performance drop is due to the disabled JIT and not due to our modifications. All measurements, except the first, are taken with the JIT disabled. This way it is possible to observe the performance drop that comes from disabling the JIT when comparing the first pair of results.

The results of two benchmarks are presented in this section, a microbenchmark *SlotWrite* that measures slot writes and another benchmark *ParserCombinators* that measures Newspeak's parser combinator module.

All measurements are performed in a ThinkPad T430s (CPU: Intel® Core™ i7-3520M CPU @ 2.90GHz × 4; 16GB RAM) running on a 64bit Ubuntu 14.04 with the Linux kernel version `3.13.0−35−generic`. All measured values on those two benchmarks and all other benchmarks in the Newspeak benchmark repository are listed in Appendix D.

SlotWrite The SlotWrite benchmark is a microbenchmark that performs 1.000.000 slot writes to a `protected` slot per benchmark run.

The benchmark results are visualized in Figure 5.1. Again, each result is the average of 10 benchmark runs with the same configuration. The standard deviation of all runs is shown as a black error handle on every bar.

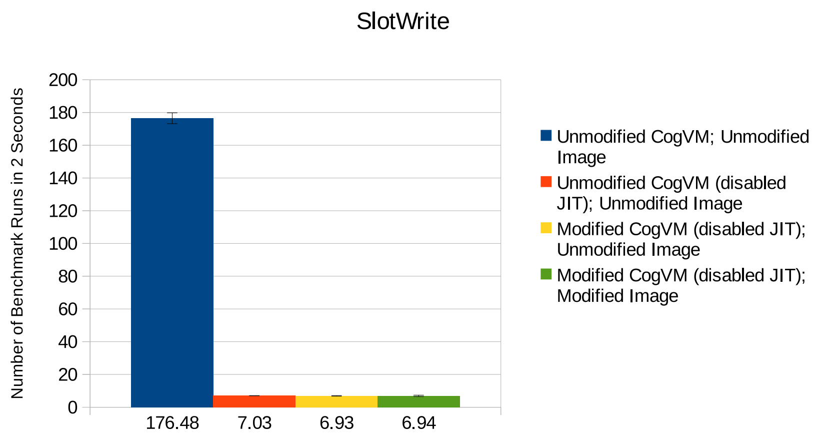


Figure 5.1: The results of the SlotWrite benchmark for different VM and image combinations. The average results of the benchmark runs are plotted along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

It can be observed that disabling the JIT hugely affects runtime performance. When comparing the difference of the unmodified VM with the modified VM (both with JIT disabled), a slight performance drop of around 1.4 percent can be observed. We conclude that our modifications do not have much impact on slot writes.

ParserCombinators The ParserCombinator benchmark is best described by citing its class comment: "A macrobenchmark based on the Newspeak's Combinatorial-Parsing. This benchmark parses and evaluates a fixed string with a simple arithmetic expression grammar. These parser combinators use explicitly initialized forward reference parsers rather than mirrors to handle the cycles in the productions. They also do not use of any platform streams to avoid API differences."

The benchmark results are visualized in Figure 5.2. Again, each result is the average of 10 benchmark runs with the same configuration. The standard deviation of all runs is shown as a black error handle on every bar.

Disabling the JIT reduces the performance of the ParserCombinators benchmark. Thus, we expect a huge performance increase when adapting our changes to the JIT. When comparing the unmodified and modified VM (both with an unmodified image and disabled JIT) no difference in the runtime performance could be observed. This is because even when using the modified VM, the image only sends the old bytecodes. When using the modified image, a performance drop of 10 percent can be observed because the new bytecodes are used.

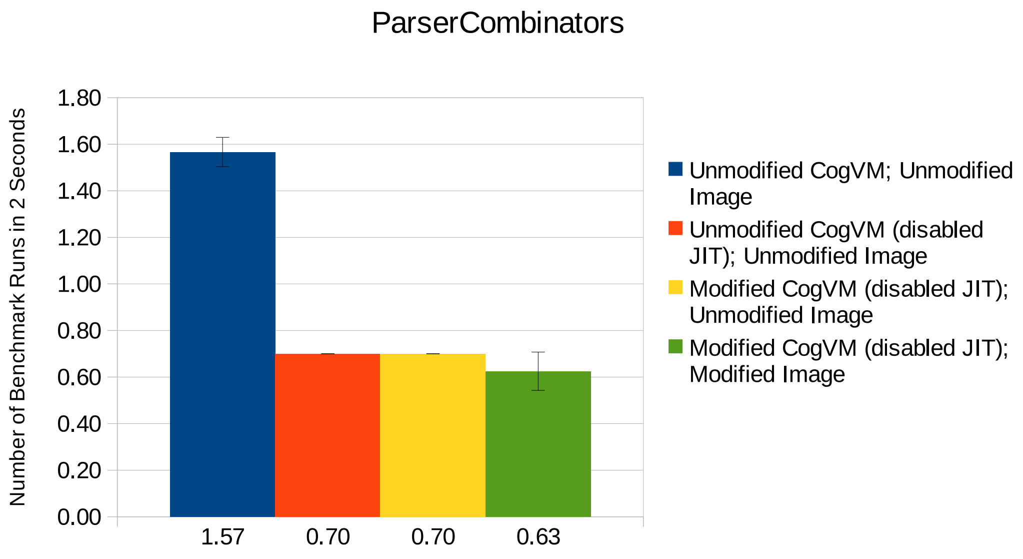


Figure 5.2: The results of the ParserCombinators benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

5.2 Analysis of Access Violations

This section presents the impact of our modifications to the Newspeak image and VM. The startup process of a Newspeak image is observed with functioning access modifiers, but an unmigrated code base. The access violation errors that are logged are analyzed and retraced to the source code. After fixing the cause of the access violation, we show that the error disappears and that our modifications are working.

We prepared a Newspeak VM and bootstrapped a Newspeak image with our modifications. The Newspeak source in the image contains our modifications, but the access modifiers in the image are not yet migrated. When starting the modified VM with the modified image and the `−enforceam` parameter (which activates the access modifier checks), a blank window appears. No Newspeak window can be shown and the Newspeak process occupies a CPU core by 100 %. The error log on the standard output of the VM is listed in Listing 5.1.

Listing 5.1: The error log of the modified Newspeak image and VM before upgrading the access modifiers. The last two lines are repeated in an endless loop.

```
# RuntimeForSqueak'Platform'3400>blackMarket
# RuntimeForSqueak'Platform'3400 class(KernelForSqueak'Class'248)>name
# RuntimeForSqueak'Platform'3400>resetForNewImageSession
# RuntimeForSqueak'Platform'3400 class(KernelForSqueak'Object'248)>
    printString
# KernelForSqueak'Metaclass'248(KernelForSqueak'Object'248)>printString
# KernelForSqueak'Metaclass'248 class(KernelForSqueak'Object248)>printString
```

The first error in the error log appears on a `Platform` object on which the `blackMarket` method was called.[4] In the log the method appears to be `protected`, which is indicated by the `#`-symbol before the class name (`+` would be `public`; − would be `private`). When opening the image, it can be observed that `blackMarket` is a slot on the `Platform` class and is indeed `protected`. Because `blackMarket` is intended to be a `public` API slot, its access modifier should be changed to `public`.

But because `blackMarket` is not `public`, our modifications invoke the `doesNotUnderstand` method. The default implementation of `doesNotUnderstand:` in the `Object` class initializes a new `MessageNotUnderstood` error which is instantly signaled. Because no method handles the error, Newspeak's attempts to log the error into a log file. This is done in the `NsFFISessionManager` class in the `prepareForNewVmSession` method, which handles the image startup. While writing the error log, the `name` method is called in `Class`. The `name` method is

[4]The `blackMarket` method was called while initializing a `NativeSession` object during the image startup.

`protected`. Again, our modifications do not allow method execution and invoke `doesNotUnderstand`, which results in the second line of the error log in Listing 5.1.

The remaining lines in the error log come from handling the resulting error, which fails again and again, until the execution is caught in an endless loop. The last two lines in the error log are repeated endlessly.

Listing 5.2: The error log of the modified Newspeak image and VM after upgrading the access modifier of the `blackMarket` method of the `Platform` class. The last two lines are repeated in an endless loop.

```
# Brazil'ContainerClasses'24>Desktop
# Brazil'ContainerClasses'24 class(KernelForSqueak'Class'48)>name
# Platform>resetForNewImageSession
# Platform class(KernelForSqueak'Object'48)>printString
# KernelForSqueak'Metaclass'248(KernelForSqueak'Object'48)>printString
# KernelForSqueak'Metaclass'248 class(KernelForSqueak'Object48)>printString
```

When migrating the access modifier of the `blackMarket` method of the `Platform` class (which cause the original error) to `public`, the error log is different. The original error does not appear. Instead, the `Desktop` slot at the `Brazil`ContainerClasses` class cannot be accessed. As shown in the new error log in Listing 5.2, again the image cannot recover due to other access violations in a similar way as in the first try.

From those two examples, we observed that our modifications worked for two ordinary sends, which attempted to call a `protected` method.

5.3 Known Limitations

Although our modification to the Newspeak VM and image work in principle, some limitations are still present. This section presents limitations of our approach as well as of the current implementation.

Access Modifier Semantics As discussed in section 3.2.2, the VM should react to a call to an not-accessible method by ignoring the not-accessible method and continuing the lookup. Our implementation currently violates the specification, because it aborts the lookup when such a method is called and directly calls the `doesNotUnderstand:` method. We took this approach because it makes the migration easier (calls of not-accessible methods are caught early). However, our implementation needs to be changed to obey the specification in future work.

Just-in-Time Compiler It is clear from the performance benchmarks that our implementation is much slower than the usual Newspeak implementation because the

JIT needs to be disabled. This is because our modifications of the JIT produce crashes of the VM. However, we are certain that our modification can be applied to the JIT once the cause of the crashes is eliminated.

MixinMirrors We changed the `ClassDeclarationMirror` so that it can reflect on the access modifier of its class. As discussed in section 4.3, the downside of this approach is that a mixin can be used—by definition—multiple times to extend a class through subclassing. Therefore, a mixin might return the enclosing class of where it was defined, and not necessarily where the class declaration was built. Because multiple applications of a mixin are allowed in the specification, this workaround has to be removed as part of future work.

Migration As pointed out in section 4.4, the migration program produces a lot of changes. Those changes need to be grouped and submitted manually. Furthermore, the migration program only fixes access violations that occurred during the testing phase. It cannot detect access violations which did not throw an error during the migration.

6 Related Work

Practical languages employ a wide variety of mechanisms to protect program entities from unqualified access. We give representative examples of protection mechanisms from JavaScript, Smalltalk, and C++.

Some languages do not directly provide access control on a language level. Instead, programmers use other languages features to control information access. Examples of such languages are JavaScript and Smalltalk.

JavaScript JavaScript [8] is a prototype-based language that treats all attributes of a prototype as `public`. All attributes (and therefore methods) of a prototype can be accessed by any other prototype. JavaScript does not directly offer means for access control. A commonly used way to make methods `private` is to define methods as local variables inside of closures [38]. This way those methods are only accessible inside the closure's scope which effectively hides them from outer access. In comparison with Newspeak, JavaScript has no direct support for access modifiers. There is, for example, no built-in way to define `protected` entities.

Smalltalk Smalltalk [15] objects consists of data, so called slots, and methods. In Smalltalk an object's methods can be accessed from any other object through message sends. Slots, however, are only accessible by the object those slots belong to. Therefore, access control in Smalltalk is implicit by a basic policy. Fine grained access control for slots is possible by making slots available to other entities by using getter and setter methods. Methods, however, cannot be made `private`. Methods which are intended for internal use are often put in a method category with the name "private". This is only a convention that communicates that other objects should not call those methods — the language does not enforce access control. Through reflection even slots can be accessed by other objects. Newspeak is secure in this regard as it offers reflection only through mirrors, which can be managed in an object-capability way.

C++ Many programming languages exist that explicitly feature access control. One of them is C++ [45], which has access modifier keywords built into the language. It is object-orientation and offers explicit access control through the following access modifier keywords which can be given to members of classes:

- `Public` members can be used by any functions.
- `Protected` members can be used by functions of the same class or by subclasses. Additionally, they can be used by `friend` classes.
- `Private` members can only be used by the same class or `friend` classes.

The protection provided by the C++ access control can be circumvented either by directly accessing the memory of protected data, or by performing a type conversion (type cast) [31]. Type conversion and direct memory access is not possible in Newspeak.

⁂

Among the other popular languages with explicit access control are Java and Ruby. The following sections provides a closer look into the access control implementation of those languages.

6.1 The Access Modifier Implementation of Ruby

Ruby [11] is an object-oriented scripting language designed for high-level general-purpose programming. It aims to have a rich syntax and emphasizes meta programming. Ruby provides explicit access control for methods.[1] In Ruby, the class body is executed—just as if it was defined in a method—to define, for example, methods or attributes. Thus, the words `public`, `protected` and `private` are not keywords. Instead they are implemented as class-side functions.

The semantics of the access modifiers in Ruby are as follows:

- `Public` methods are accessible from everywhere.
- `Protected` methods can only be invoked from within methods of the same class or subclasses.
- `Private` methods can only be invoked through an implicit receiver send from within methods of the same class or subclasses.

The de facto reference implementation of the Ruby language is implemented in C and is commonly called MRI (Matz' Ruby Interpreter—Yukihiro "Matz" Matsumoto is the creator of the language) [40]. Many other Ruby implementations have been developed [22, 23, 39]. We want to examine a Ruby implementation called MagLev [13]. MagLev is built on top of VMware's GemStone/S 3.1 VM [14]. GemStone is a Smalltalk implementation similar to Newspeak. The MagLev Ruby implementation is based on an extended Smalltalk VM, just as Newspeak is based on a Smalltalk VM. Therefore we can compare how the access modifier semantics of Ruby are implemented in MagLev and compare the findings to our implementation.

Defining Access Controlled Methods Ruby classes and modules can have `private`, `protected`, and `public` methods. As shown in Listing 6.1, the access modifier

[1]Ruby also provides access control for constants. Because the constant lookup is different than the method lookup and because space in this work is limited, we focus on access modifiers for methods.

Listing 6.1: An example Person class written in Ruby. A person may have a birthday, which is `private` and has a `private` getter/setter method. A `public` method exists which returns the age of the person in years.

```ruby
require 'date'

class Person
public
  def age_in_years
    now = Time.now.utc.to_date
    age = now.year - birthday.year
    # for days in the year before the birthday
    age - ((now.month > birthday.month ||
           (now.month == birthday.month && now.day >= birthday.day)) ? 0 : 1)
  end

private
  def birthday(date = nil)
    if date
      @birthday = date
    else
      @birthday ||= Date.parse('1903-06-25')
    end
  end
end
```

directives do not need to be directly attached to the methods. Instead, an access modifier directive controls all following methods until another directive occurs or the class definition ends. Classes and modules define a `methodProtection` slot, which encodes the last access modifier directive seen during method compilation. In our example in Listing 6.1 the `methodProtection` slot is initialized with `0` (which stands for `public`). The `public` method, which is executed as the first statement in the class body, re-sets the value to `0`. All following methods are defined as `public`, which makes the `age_in_years` function `public`. A `private` call follows, which sets the `methodProtection` slot to `2` (which stands for `private`) and makes the following methods `private`.

Method Lookup and Access Violations Three execution levels exist in MagLev: The GemStone Smalltalk VM, the Smalltalk environment, and the Ruby environment. Ruby-methods are written in Ruby, but compiled to Smalltalk-level method objects. The GemStone VM can execute Smalltalk methods as well as Ruby methods and is responsible for the method lookup of Ruby methods.

The access modifier of a Ruby method is saved in the Smalltalk-side method object. The `rubyInfo` slot of the `GsComMethNode` class, which is the intermediate representation for the compiler of a `CompiledMethod`, holds a bit map which encodes the access modifier of a method. It makes the access modifier information available through the `methodProtection` getter method, so that the VM can decide whether the method can be found by a method call or not.

The method lookup code of the GemStone VM checks for the accessibility of a method, if it is found in a Ruby-method dictionary. If a method is not accessible, the

VM executes the Smalltalk method `_doesNotUnderstand`. The `_doesNotUnderstand` method does a dispatch depending on the language environment. If the current environment is the Ruby environment, it calls the Ruby-level method `method_missing`. The `method_missing` method is the default Ruby handler for methods calls that could not be found — very similar to `doesNotUnderstand` in Newspeak.

Methods that could not be found due to an access restriction also result in a `method_missing` call. To provide the reason of the failing method call in the error message, the VM provides an primitive to get the "lastDNUProtection" variable. It encodes whether the last `doesNotUnderstand` (or `method_missing`) was called because of a `protected` or `private` method or if the method was not present.

The Ruby access control for methods in MagLev is implemented very similar to our implementation. The access modifier bits are stored in the Smalltalk compiled method object. MagLev stores them in a slot whereas our implementations encodes the access modifier in the method header. The error handling is similar, too, except that MagLev provides the information that a method could not be executed because the access rights were insufficient. This reveals the existence of a not accessible method. Our semantics hide the fact that the method exists.

6.2 The Access Modifier Implementation of Java

Java [16] is an object-oriented language that also provides explicit access control. Similar to Newspeak, program code in Java is compiled into bytecode and then executed by the Java VM [28]. The reference implementation of the Java language is maintained by Oracle [35].

Java's access modifiers semantics are as follows:

- `Private` members are only accessible if the member is defined within the same class as the calling code.
- `Package Private` is the default modifier and is chosen if no modifier was given. Members defined with this modifier are only accessible from code that is within the immediately enclosing package they are defined in.
- `Protected` members are accessible like package private members. Additionally, the are accessible from subclasses.
- `Public` members are accessible from everywhere within the same compilation unit.

In contrast to Ruby or Newspeak, accessibility in Java is a static property that is determined at compile time [16]. Thus, access modifiers are easily circumvented by using reflection.

Besides the reference implementation, there are multiple other Java implementations, for example STX:LIBJAVA [21], which is a Java environment implemented within the Smalltalk/X VM [9]. STX:LIBJAVA allows runtime access control [19, 20]. It lazily loads references to classes, methods, or fields that are not in the current class.

Listing 6.2: Part of the STX:LIBJAVA code which resolves references to methods or fields of a class. The resolver takes the accessibility of methods and fields into account when referencing them. Note: The `privilegedAccessQuery` method is a way to bypass access control, for example for certain tests. The `hasMagicAccessRights` method returns true if the class inherits from "sun.reflect.MagicAccessorImpl".

```
!JavaResolver methodsFor:'common helpers'!

checkPermissionsForMethodOrField: aJavaMethodOrField from: accessingJavaClass
     to: resolvedJavaClass
  ...

  accessingJavaClass hasMagicAccessRights ifTrue: [ ↑true ].

  (self checkPermissionsFrom: accessingJavaClass to: resolvedJavaClass)
      ifFalse: [
    JavaVM privilegedAccessQuery query ifTrue: [ ↑ true ].
    ↑ false
  ].
  aJavaMethodOrField isPublic ifTrue: [ ↑ true ].
  ((aJavaMethodOrField isProtected
    and: [
      resolvedJavaClass javaComponentClass
        equalsOrIsSubclassOf: aJavaMethodOrField javaClass
    ])
      and: [
        accessingJavaClass javaComponentClass
          equalsOrIsSubclassOf: aJavaMethodOrField javaClass
      ])
      ifTrue: [ ↑ true ].
  ((
  aJavaMethodOrField isPrivate not
    and: [ resolvedJavaClass javaPackage = accessingJavaClass javaPackage ])
      and: [ resolvedJavaClass classLoader = accessingJavaClass classLoader
          ])
      ifTrue: [ ↑ true ].
  (aJavaMethodOrField isPrivate
    and: [ aJavaMethodOrField javaClass name = accessingJavaClass name ])
      ifTrue: [ ↑ true ].

  "/a little bit too verbose here just so it's clear what's in query"

  JavaVM privilegedAccessQuery query ifTrue: [ ↑ true ] ifFalse: [ ↑ false ].
```

The loading process of references also includes an accessibility check as shown in Listing 6.2. The permission check from the listing implements Java's access modifier semantics for fields and methods. If the method returns `false`, an `IllegalAccessError` is thrown.

When a reference is resolved, which usually happens lazily on the first time a reference is used, the resulting entity is stored in the constant pool of a classfile. The reference is cached there for future use. If a class was changed at runtime (STX:LIBJAVA provides a live coding environment for Java) the reference in the classfile is changed to the new class. Existing instances keep their classes (so they do not need to be migrated), but new objects get the changed class. This way access modifiers can be changed at runtime.

The access control implementation in Java is not suitable for a dynamic runtime environment like Newspeak. However, STX:LIBJAVA solves this problem and makes access modifiers possible in a dynamic Java environment. Still, access control can be bypassed through reflection, which is not possible with our access modifier implementation for Newspeak.

7 Summary and Conclusions

In this work, we have shown the importance of an access control implementation for Newspeak. Although access control was alreay specified in the Newspeak specification, it was not implemented. We presented a way to implement the Newspeak specification in this regard and discussed design decisions relevant to the implementation.

We have shown that our approach is feasible. At the time of writing, our work is partially merged into the Newspeak code base. Some parts of our implementation, especially the changes to the VM, need minor improvements to be mergable.

Since access modifiers were available in the syntax but were not enforced, we concluded that the existing implicit and explicit access modifiers in the Newspeak code base need to be migrated. We have described an approach to migrate access modifiers so that the Newspeak image works with enforced access modifiers.

We evaluated the performance of our implementation using multiple benchmarks. The benchmark results of a micro and a macro benchmark are presented in this work, showing no severe performance penalty. We have compared the access modifier design of Newspeak to other programming languages and, in more detail, have compared our implementation to the access control implementation of a Java and a Ruby derivative.

Access modifiers in a language environment offer much potential for improvements from the language level up to the tooling:

Virtual Machine We proposed changes to the VM, implementing access control for Newspeak. Since we started the implementation, the Newspeak VM source base progressed a lot due to the Spur project of Eliot Miranda [33]. Spur is a project to rewrite parts of the Squeak/Newspeak VM, affecting large parts of it. To make our changes useful to a greater public, they should be (re-)implemented on top of Spur.

Performing Sends The Squeak-based implementation of Newspeak allows to perform message sends through meta programming. This functionality is implemented in the `ImplementationBase` class by reusing the `perform:` method of Squeak. It allows to do an ordinary send to the object on which the `perform:` method is called on. A possible future work is to implement similar methods to perform other sends so that `protected` or `private` methods can be executed through meta programming. Those methods should not be implemented as a global authority because they provide extra capabilities.

Bytecodes We introduced two new bytecodes in this work: The self send bytecode and the outer send bytecode. We assume that the self send bytecode is a frequently used bytecode. Thus, a possible future work would be to obtain data on the uses of self sends. Based on that data, common uses of self sends could be implemented with a one-byte bytecode (our current implementation of the self send bytecode needs two bytes). This would reduce the average size of `CompiledMethods`.

⁂

We conclude that our approach to access control in Newspeak successfully implements the language specification. Our modifications to the Newspeak source code have been merged into the main development branch. Some modifications, like the modifications to the VM, need further refinements. The performance impact of our modifications are moderate — especially when the VM modifications are ported to the JIT.

Our implementation is a keystone for Newpeak. A functioning access control does not only help to implement the Newspeak specification, it also enables security concepts like the object-capability approach.

References

[1] G. Bracha. “Executable grammars in Newspeak”. In: *Electronic Notes in Theoretical Computer Science* 193 (2007).

[2] G. Bracha. *Newspeak Programming Language Draft Specification Version 0.091*. URL: http://bracha.org/newspeak-spec.pdf (last accessed 2014-09-30).

[3] G. Bracha. *NS2JS*. URL: https://groups.google.com/d/msg/newspeaklanguage/m6z3Lx8NHcY/TivKuz-LcJYJ (last accessed 2014-09-30).

[4] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Miranda. *The Newspeak Programming Platform*. Technical report. Cadence Design Systems, 2008.

[5] G. Bracha and W. Cook. “Mixin-based Inheritance”. In: *SIGPLAN Notices* 25.10 (1990).

[6] G. Bracha and D. Ungar. “Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages”. In: *ACM SIGPLAN Notices* 39.10 (2004).

[7] V. Bykov. “Hopscotch: Towards user interface composition”. In: *Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*. 2008.

[8] ECMA. *ECMA-262: ECMAScript Language Specification*. Geneva, Switzerland, June 2015.

[9] eXept Software AG. *Smalltalk/X - Object-oriented programming language*. URL: http://www.exept.de/en/products/smalltalkx (last accessed 2014-09-30).

[10] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. “Language Support for Fast and Reliable Message-Based Communication in Singularity OS”. In: *ACM SIGOPS Operating Systems Review*. 2006.

[11] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O’Reilly Media, Inc., 2008. ISBN: 978-0-596-55465-1.

[12] F. Geller, R. Hirschfeld, and G. Bracha. *Pattern Matching for an object-oriented and dynamically typed programming language*. Technical report. Universitätsverlag Potsdam, 2010.

[13] GemStone Systems. *MagLev*. URL: https://maglev.github.io/ (last accessed 2014-09-30).

[14] GemTalk Systems. *GemStone/S Product Line*. URL: http://gemtalksystems.com/index.php/products/gemstones/ (last accessed 2014-09-30).

[15] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983. ISBN: 978-0-201-11371-6.

[16] J. Gosling, B. Joy, G. L. J. Steele, G. Bracha, and A. Buckley. *The Java Language Specification Java SE 8 Edition*. Addison-Wesley Professional, 2014. ISBN: 978-0-133-90079-8.

[17] E. G. Haffner, T. Engel, and C. Meinel. "Techniques for Securing Networks Against Criminal Attacks". In: *International Conference on Internet Computing*. 2000.

[18] R. Hirschfeld, P. Costanza, and O. Nierstrasz. "Context-Oriented Programming". In: *Journal of Object Technology* 7.3 (2008).

[19] M. Hlopko. "Java implementation for Smalltalk/X VM". Master's thesis. Czech Technical University in Prague, 2011.

[20] M. Hlopko, J. Kurš, and J. Vraný. "Towards a Runtime Code Update in Java". In: *Proceedings of the 13th Annual International Workshop on Databases, Texts, Specifications, and Objects*. 2013.

[21] M. Hlopko, J. Kurš, J. Vraný, and C. Gittinger. "On the Integration of Smalltalk and Java". In: *Science of Computer Programming* (2014).

[22] *JRuby a Java powered Ruby implementation*. URL: http://jruby.org/ (last accessed 2014-09-30).

[23] A. Junod, R. Bazinet, and D. Bernier. *Professional IronRuby*. John Wiley & Sons, Incorporated, 2010. ISBN: 978-0-470-37708-6.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. 1997.

[25] M. Kleine, R. Hirschfeld, and G. Bracha. *An Abstraction for Version Control Systems*. Technical report. Universitätsverlag Potsdam, 2012.

[26] B. W. Lampson. "Protection". In: *ACM SIGOPS Operating Systems Review* 8.1 (1974).

[27] K. J. Lieberherr and A. J. Riel. "Demeter: A CASE Study of Software Growth through Parameterized Classes". In: *Proceedings of the 10th International Conference on Software Engineering*. 1988.

[28] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification Java SE 8 Edition*. Addison-Wesley Professional, 2014. ISBN: 978-0-133-92272-1.

[29] J. Loeliger and M. McCullough. *Version Control with Git - Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc., 2012. ISBN: 978-1-449-31638-9.

[30] R. Macnak. *Newspeak-to-Dart Compilation*. URL: https://docs.google.com/document/d/1pU_nautpK49pJzwZkJM1NhACcad0Hjq0rjIomLSWQ1Y/edit (last accessed 2014-09-30).

[31] Microsoft MSDN. *Controlling Access to Class Members*. URL: http://msdn.microsoft.com/en-us/library/zsc61976(v=vs.100).aspx (last accessed 2014-09-30).

[32] M. S. Miller and J. S. Shapiro. "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control". PhD thesis. Johns Hopkins University, 2006.

[33] E. Miranda. *Spur*. URL: http://www.mirandabanda.org/cogblog/category/spur/page/3/ (last accessed 2014-09-30).

[34] E. Miranda. "The Cog Smalltalk Virtual Machine". In: *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages*. 2011.

[35] Oracle. *Java Software*. URL: https://www.oracle.com/java/index.html (last accessed 2014-09-30).

[36] B. O'Sullivan. *Distributed Revision Control with Mercurial*. Mercurial project, 2007.

[37] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Communications of the ACM* 15.12 (1972).

[38] M. Pennisi. *Information Hiding in JavaScript*. URL: http://bocoup.com/weblog/info-hiding-in-js/ (last accessed 2014-09-30).

[39] E. Phoenix. *Rubinius: The Ruby Virtual Machine*. URL: http://rubini.us/ (last accessed 2014-09-30).

[40] *Ruby Programming Language*. URL: https://www.ruby-lang.org (last accessed 2014-09-30).

[41] R. S. Sandhu and P. Samarati. "Access Control: Principle and Practice". In: *Communications Magazine, IEEE* 9 (1994).

[42] M. Shapiro. "Structure and Encapsulation in Distributed Systems: The Proxy Principle". In: *Proceedings of the 6th Int. Conf. on Distributed Computing Systems (ICDCS)*. 1986.

[43] J. Sinschek, E. Bodden, and M. Mezini. "Injecting Security into Untrusted Components". Unpublished.

[44] S. Stefanov. *JavaScript Patterns*. 1st. O'Reilly Media, Inc., Sept. 2010. ISBN: 978-1-449-39694-7.

[45] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2013. ISBN: 978-0-133-52285-3.

[46] D. Ungar and R. B. Smith. "Self: The Power of Simplicity". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. 1987.

[47] B. J. Walker, R. A. Kemmerer, and G. J. Popek. "Specification and Verification of the UCLA Unixt Security Kernel". In: *Communications of the ACM* 23.2 (1980).

A Newspeak's Mirror Landscape

Newspeak provides a rich mirror landscape. In Figure A.1 we list all implemented Mirrors in the Newspeak codebase at the time of writing this work. We also listed builder classes, which are similar to mirrors, except that they do not only reflect on objects, but also help to construct new objects.

There is a multitude of modules which contain mirrors, builders, and other reflection related classes. Each module serves different purposes. See subsection 2.2.2 for a description of Newspeak's mirror system.

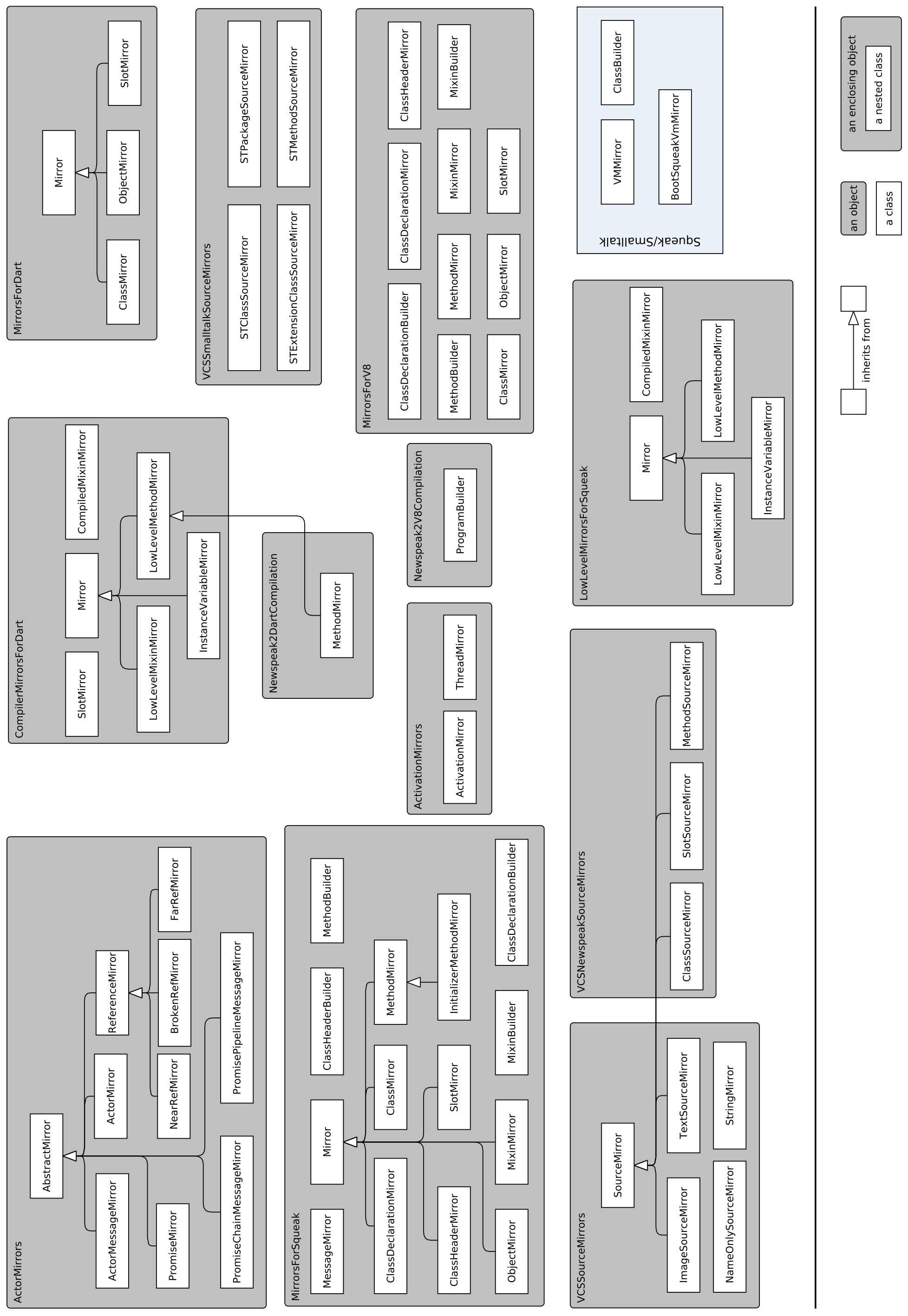


Figure A.1: The Newspeak mirror landscape.

B The Implementation of the Method Lookup in the Virtual Machine

Newspeaks method lookup funcitonality is implemented in the VM. In fact, there are different VMs in Newspeak. Most notably, an interpreter VM and a JIT VM. The method shown in Listing B.1 implements a part of the method lookup of the Newspeak interpreter VM. Its first parameter `class` indicates the class where the method is currently searched in. The second parameter `originClass` indicates the class where the method lookup originally started.

Listing B.1: The `lookupMethodInClass:StartLookupFrom:` method implemented at the `StackInterpreter` class. It implements the method lookup for a given selector and `class`, knowing that the lookup originally started at the `startingClass`.

```
lookupMethodInClass: class startLookupFrom: originClass
  | currentClass dictionary found |
  <inline: false>
  self assert: class ~= objectMemory nilObject.
  currentClass := class.
  [currentClass ~= objectMemory nilObject]
    whileTrue:
    [dictionary := objectMemory fetchPointer: MethodDictionaryIndex ofObject:
         currentClass.
    found := self lookupMethodInDictionary: dictionary.
    found ifTrue: [ ↑ self currentClassUnlessAccessRestricted: currentClass
                                class: class
                                originClass: originClass].
    currentClass := self superclassOf: currentClass].

  "Could not find #doesNotUnderstand: -- unrecoverable error."
  messageSelector = (objectMemory splObj: SelectorDoesNotUnderstand) ifTrue:
    [self error: 'Recursive not understood error encountered'].

  "Cound not find a normal message -- raise exception #doesNotUnderstand:"
  self createActualMessageTo: class.
  messageSelector := objectMemory splObj: SelectorDoesNotUnderstand.
  self sendBreak: messageSelector + BaseHeaderSize
    point: (objectMemory lengthOf: messageSelector)
    receiver: nil.
  ↑self privateLookupMethodInClass: class startLookupFrom: class
```

It searches the method dictionary of a class for a method with a matching selector. If such a method is found, it will be checked against the access modifier semantics with the help of the `currentClassUnlessAccessRestricted` method listed in Listing B.2. If the check succeeded, the class holding the right method will be returned.

Listing B.2: The implementation of `currentClassUnlessAccessRestricted`, which either prints a warning to the standard output of the VM, or calls the `doesNotUnderstandMethod` depending on the `enforceAccessModifiers` option.

```
currentClassUnlessAccessRestricted: currentClass class: class originClass:
    originClass
  (currentClass = originClass or: [self newMethodIsPrivate not]) ifFalse: [
    enforceAccessModifiers ifTrue: [
      "Cannot access method; pretend not to understand -- raise exception #
          doesNotUnderstand:"
      self createActualMessageTo: class.
      messageSelector := objectMemory splObj: SelectorDoesNotUnderstand.
      self sendBreak: messageSelector + BaseHeaderSize
        point: (objectMemory lengthOf: messageSelector)
        receiver: nil.
      "Lookup the DNU; allowing all methods to be found"
      ↑self privateLookupMethodInClass: currentClass
            startLookupFrom: originClass ]
    ifFalse: [
      self print: 'Not accessible method         ';
             printMessageAccessModifier;
             print: ' ';
         printActivationNameForSelector: messageSelector
         startClass: originClass; cr.]].
  ↑currentClass
```

Otherwise the `doesNotUnderstand` method is called on the reciever, either because the access modifier was not sufficient, or because no matching method could be found. To call the `doesNotUnderstand` method, a new method lookup is started. If the `doesNotUnderstand` method cannot be found (which should not happen because it is part of the `Object` class) a hard error is raised.

C Possible Simplifications of the Newspeak Grammar

There are different expectations on an access control implementation for the Newspeak runtime. Some are defined explicitly in the Newspeak specification, others implicitly in the way of the intended development experience and the nature of the dynamic Newspeak runtime.

The explicitly defined requirements in the specification are the method lookup, which we have already discussed in section 2.3, and the Newspeak grammar rules. Relevant grammar rules are those for class, method, and slot definitions, because they are the only entities having an access modifier.

Newspeak's grammar states that there are *four* possible ways to specify an access modifier: It can be defined either explicitly to be `public`, `protected`, or `private`, or implicitly by omitting the access modifier. If the access modifier is not defined, it is implicitly interpreted as if the entity is defined to be `protected`. Class, method, or slot definitions all define their access modifier as the first part of their grammar.

The only exception to that rule is the top-level class declaration. Top-level classes are by definition `public`, which is why the grammar does not allow to specify any access modifier. This exception might confuse a Newspeak programmer. It also requires to consider corner cases in the grammar definition and compilation.

The Newspeak grammar currently has two very similar class definitions: `toplevelClass` and `nestedClassDecl`, both listed in Listing C.1. Both declarations basically consist of a `classDeclaration`, which offers unification potential. An example for that unification is shown in Listing C.2.
The simplification of the grammar is an ongoing discussion and a potential future work.

Listing C.1: An excerpt of the Newspeak grammar featuring class definitions. Only the grammar productions which lead from `compilationUnit` to `nestedClassDecl` are shown.

```
compilationUnit =  languageId, toplevelClass, eoi.
toplevelClass = classCategory, classDeclaration.
classDeclaration = (tokenFromSymbol: #class), classHeader, sideDecl,
                   classSideDecl opt.
sideDecl =  lparen, nestedClassDecl star, category star, rparen.
nestedClassDecl = accessModifier opt, classDeclaration.
```

Listing C.2: A possible refactoring of the Newspeak grammar shown in Listing C.1. The `nestedClassDecl` is not necessary anymore, because it was merged with the `classDeclaration`. It would be a compilation error if the access modifiers was `private` or `protected`.

```
compilationUnit =  languageId, toplevelClass, eoi.
toplevelClass = classCategory, classDeclaration.
classDeclaration = accessModifier opt, (tokenFromSymbol: #class), classHeader
    , sideDecl, classSideDecl opt.
sideDecl =  lparen, classDeclaration star, category star, rparen.
```

D Performance Benchmarks

This chapter presents more details and the underlying data of the performance benchmarks presented in section 5.1. The description of every benchmark in the Newspeak benchmark suite is taken from the class comment of the benchmark class. For every benchmark the results are shown in a figure comparing run times across different VM and image combinations. Finally we provide the measured data.

D.1 Benchmark Results

The performance benchmark were executed using a combination of different VMs and Newspeak images as explained in section 5.1.

Every paragraph in the remainder of this sections briefly describes one of the benchmarks from Newspeak's benchmark suite[1] as of git version `4737002`. A short description of every benchmark is given along with the performance graphs. The description is usually taken from the class comment of the corresponding benchmark class.

ClosureDefFibonacci A fibonacci microbenchmark stressing closure allocation, closure evaluation and integer arithmetic. The benchmark results are visualized in Figure D.1.

ClosureFibonacci A fibonacci microbenchmark stressing closure evalutation and integer arithmetic. The benchmark results are visualized in Figure D.2.

DeltaBlue One-way constraint solver, originally written in Smalltalk by John Maloney and Mario Wolczko. The benchmark results are visualized in Figure D.3.

MethodFibonacci A microbenchmark stressing method invocations and integer arithmetic. The benchmark results are visualized in Figure D.4.

NLRImmediate A microbenchmark performing non-local returns. The benchmark results are visualized in Figure D.5.

NLRLoop A microbenchmark performing non-local returns in a loop. The benchmark results are visualized in Figure D.6.

[1] http://bitbucket.org/newspeaklanguage/benchmarks, last accessed June 18, 2015.

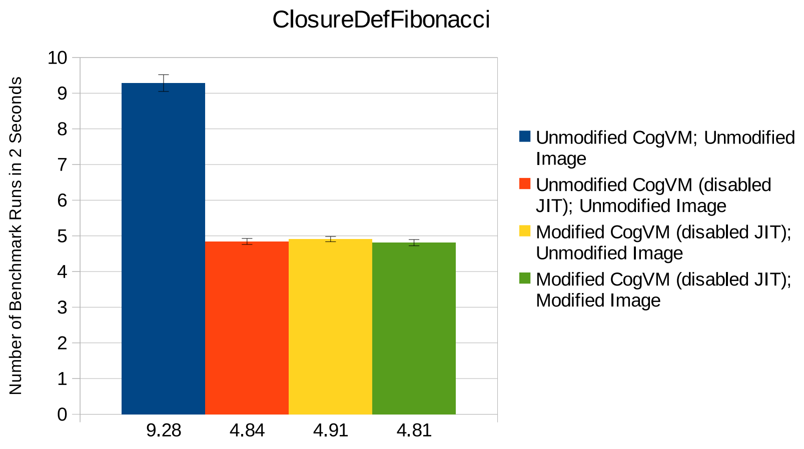


Figure D.1: The results of the ClosureDefFibonacci benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

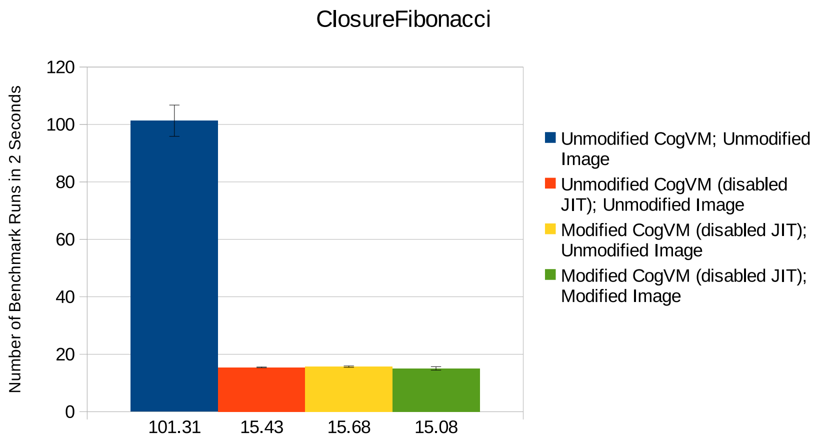


Figure D.2: The results of the ClosureFibonacci benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

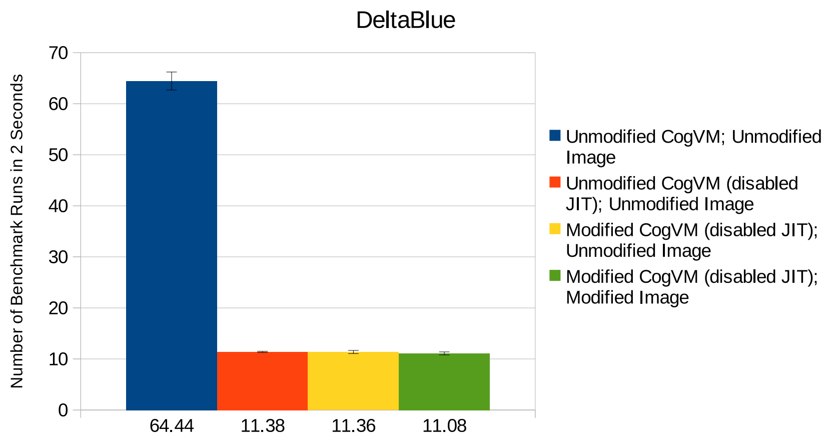


Figure D.3: The results of the DeltaBlue benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

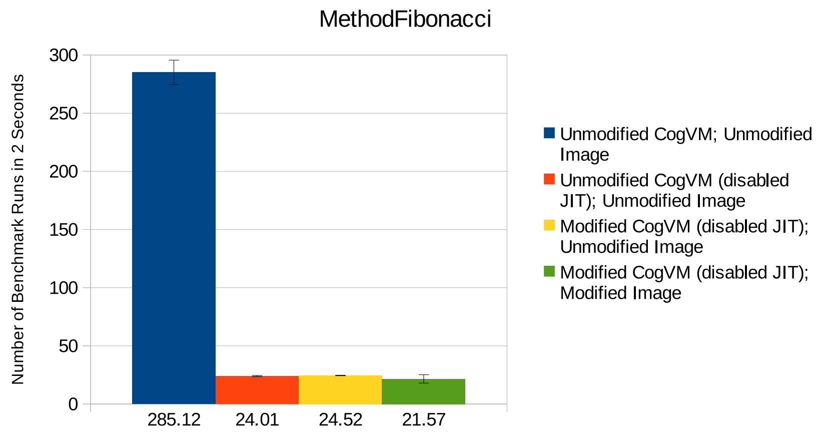


Figure D.4: The results of the MethodFibonacci benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

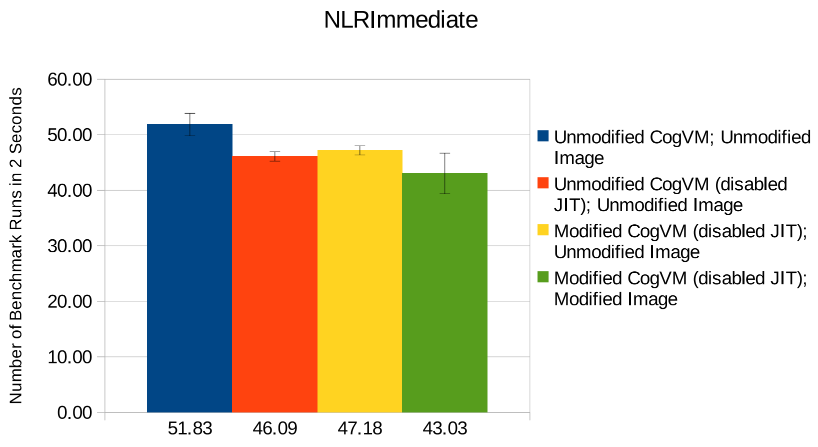


Figure D.5: The results of the NLRImmediate benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

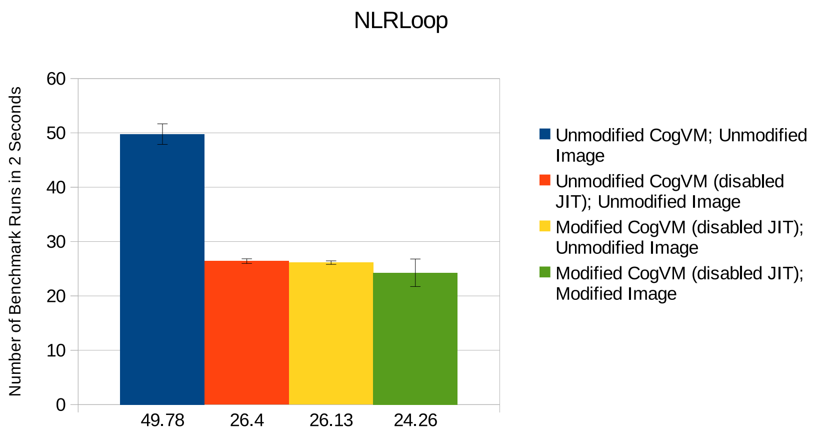


Figure D.6: The results of the NLRLoop benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

Richards An OS kernel simulation benchmark, originally written by Martin Richards in BCPL. The benchmark results are visualized in Figure D.7.

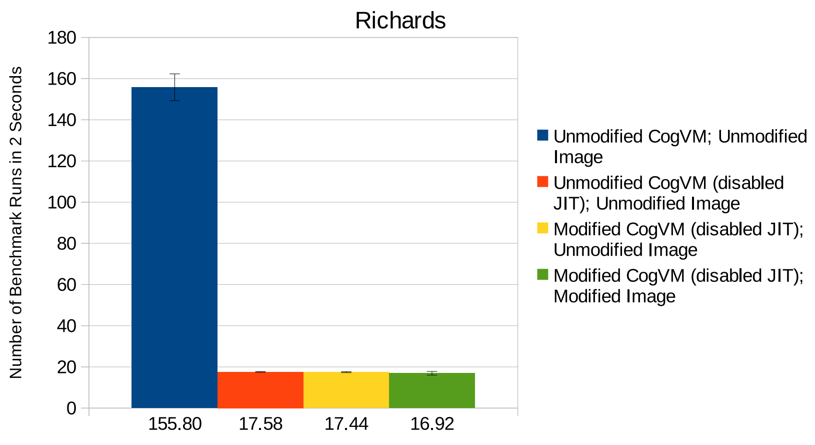


Figure D.7: The results of the Richards benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

SlotRead A microbenchmark that performs many slot reads. The benchmark results are visualized in Figure D.8.

Splay This benchmark is based on a JavaScript log processing module used by the V8 profiler to generate execution time profiles for runs of JavaScript applications, and it effectively measures how fast the JavaScript engine is at allocating nodes and reclaiming the memory used for old nodes. Because of the way splay trees work, the engine also has to deal with a lot of changes to the large tree object graph. The benchmark results are visualized in Figure D.9.

D.2 Benchmark Result Data

The exact measurements taken from the benchmarks are shown in the following tables: Table D.1, Table D.2, Table D.3, and Table D.4.

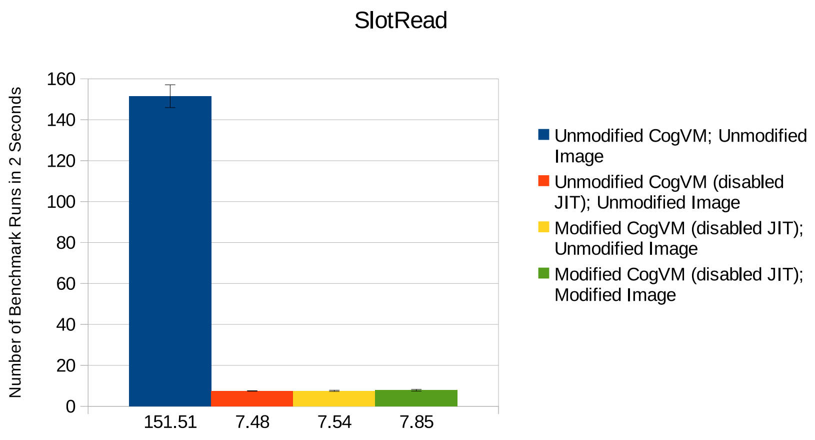


Figure D.8: The results of the SlotRead benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

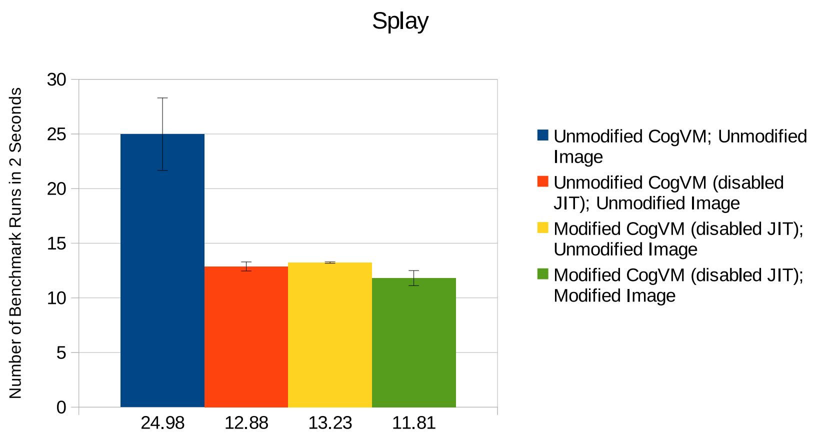


Figure D.9: The results of the Splay benchmark for different VM and image combinations. The average results of the benchmark runs are plottet along the x-axis. The y-axis shows the number of performed benchmark executions in 2 seconds runtime.

Table D.1: Benchmark results for the unmodified virtual machine and the unmodified Newspeak image. Every available benchmark was run 10 times. The result shows the number or benchmark runs within 2 seconds. Minimum, maximum, and average values are shown as well as the standard deviation.

Name	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Min	Max	Average	Std. Deviation
ClosureDefFibonacci	65.1	66.7	66.6	65.3	65	65.8	65.1	65	66.4	65.1	65	66.7	65.65	0.70
ClosureFibonacci	500	503	505.5	497.5	490.5	471.5	502	489	492	491	471.5	505.5	493.25	9.90
DeltaBlue	140.8	144.5	145.3	145	138	139	136.8	141	141.3	144.5	136.8	145.3	141.53	3.09
MethodFibonacci	773	793	766	759	750	755	674.5	735	782	760	674.5	793	751.25	32.60
NLRImmediate	948.5	954.5	983	973.5	938	946.5	938	967.5	980.5	964	938	983	959.58	16.71
NLRLoop	582.5	575	571.5	577	577.5	560	561.5	580.5	577	569	560	582.5	572.83	7.62
ParserCombinators	16	16.3	16.1	15.6	14.9	15.5	14.5	16	15.6	16.1	14.5	16.3	15.62	0.58
Richards	295.2	290.2	280.5	284.2	278.5	279.7	284	285.5	278.5	291.5	278.5	295.2	285.13	5.85
SlotRead	166.5	168.1	167.1	165	165.8	166.8	165.3	153.8	153.3	165.1	153.3	168.1	163.18	5.43
SlotWrite	171.6	171.1	196.5	172.6	189.5	193	173.5	186.3	178.8	172.1	171.1	196.5	181.05	9.88
Splay	77.9	76.7	77.3	76.4	77.9	77.3	76.4	76.4	77.1	75.5	75.5	77.9	76.86	0.75

Table D.2: Benchmark results for the unmodified virtual machine with disabled just-in-time compiler and the unmodified Newspeak image. Every available benchmark was run 10 times. The result shows the number or benchmark runs within 2 seconds. Minimum, maximum, and average values are shown as well as the standard deviation.

Name	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Min	Max	Average	Std. Deviation
ClosureDefFibonacci	8.7	9.5	9.5	9.4	9.4	9.4	9.3	9.2	9.4	9.4	8.7	9.5	9.28	0.23
ClosureFibonacci	87	102.2	104.2	103.5	103.5	103.9	105.5	104.5	104.6	104.3	87	105.5	101.31	5.45
DeltaBlue	61.5	67	66	66.8	64.4	63.8	64.3	64.9	63.5	62.6	61.5	67	64.44	1.77
MethodFibonacci	260.5	297.5	300	289	285	288.2	286	284.5	286.2	284	260.5	300	285.12	10.54
NLRImmediate	46.5	52.9	53.1	53.2	53.1	53.1	52.9	52.6	52.6	52.3	46.5	53.2	51.83	2.03
NLRLoop	44.9	51.2	51.2	50.8	50.1	50.7	51.1	50	51	50.2	44.9	51.2	49.78	1.89
ParserCombinators	1.4	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.4	1.6	1.57	0.06
Richards	165.5	148.5	154	167.8	147.8	154.1	154.5	156	152	153.8	147.8	167.8	155.80	6.50
SlotRead	156	160.8	160	144.7	148.8	148.5	149	148.8	148	148	144.7	160.8	151.51	5.57
SlotWrite	170.6	176.3	172.4	173.8	178.3	178.5	179.1	180.3	179.3	178.3	170.6	180.3	176.48	3.30
Splay	31.9	26.1	26.1	26	21.5	26.5	22.5	21.5	22.3	22	21.5	31.9	24.98	3.32

Table D.3: Benchmark results for the modified virtual machine with disabled just-in-time compiler and the unmodified Newspeak image. Every available benchmark was run 10 times. The result shows the number or benchmark runs within 2 seconds. Minimum, maximum, and average values are shown as well as the standard deviation.

Name	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Min	Max	Average	Std. Deviation
ClosureDefFibonacci	5	4.9	4.8	4.8	4.9	5	4.9	5	4.9	4.9	4.8	5	4.91	0.07
ClosureFibonacci	15.5	15.8	15.5	15.8	15.6	16.1	15.9	15.7	15.8	15.2	15.2	16.1	15.68	0.25
DeltaBlue	11.8	11.6	10.9	11.2	11.4	11.5	11.5	11.5	11.5	10.8	10.8	11.8	11.36	0.31
MethodFibonacci	24.5	24.8	24.5	24.3	24.1	24.5	24.5	24.4	24.5	25	24.1	25	24.52	0.25
NLRImmediate	48.2	47.1	47.7	46.4	47.1	48.2	47.2	47.7	47.4	45.5	45.5	48.2	47.18	0.82
NLRLoop	26.6	26.6	25.9	25.5	26.1	26.2	26	26.3	26.2	26	25.5	26.6	26.13	0.33
ParserCombinators	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.70	0.00
Richards	17.3	17.5	17.2	17.2	17.9	17.4	17.1	17.7	17.7	17.3	17.1	17.9	17.44	0.26
SlotRead	7.7	7.8	6.9	7.8	7.8	7.5	7.4	7.8	7.4	7.7	6.9	7.8	7.54	0.29
SlotWrite	7	7.1	7	7	7	6.8	7.2	6.8	6.9	6.6	6.6	7.2	6.93	0.17
Splay	13.3	13.3	13.2	13.2	13.3	13.2	13.2	13.1	13.3	13.2	13.1	13.3	13.23	0.07

Table D.4: Benchmark results for the modified virtual machine with disabled just-in-time compiler and the modified Newspeak image. Every available benchmark was run 10 times. The result shows the number or benchmark runs within 2 seconds. Minimum, maximum, and average values are shown as well as the standard deviation.

Name	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**Min**	**Max**	**Average**	**Std. Deviation**
ClosureDefFibonacci	4.9	4.9	4.8	4.7	4.7	4.8	4.7	4.8	4.9	4.9	4.7	4.9	4.81	0.09
ClosureFibonacci	15.5	15.3	15.8	14.7	15.3	15.4	14.9	15.1	15.7	13.7	13.7	15.8	15.08	0.61
DeltaBlue	11.3	11.2	11.5	10.8	10.9	11.4	10.7	11	11.3	10.7	10.7	11.5	11.08	0.30
MethodFibonacci	24.5	24.3	23.5	14.2	23.3	20.7	23	17.3	24.6	24.6	14.2	24.6	21.57	3.57
NLRImmediate	48	43.3	41.8	40	36.2	40.8	44.3	43.7	47.4	46.6	36.2	48	43.03	3.66
NLRLoop	26.9	25.1	25.5	25	18.4	22	25	25.4	26.4	26.1	18.4	26.9	24.26	2.54
ParserCombinators	0.7	0.6	0.6	0.7	0.5	0.6	0.5	0.7	0.7	0.7	0.5	0.7	0.63	0.08
Richards	18	17.7	17.2	16.9	16.8	16.1	15.4	16	17.9	17.6	15.4	18	16.92	0.89
SlotRead	8.4	7.6	7.8	8.1	7.4	7.9	7.2	7.6	8.3	8.3	7.2	8.4	7.85	0.41
SlotWrite	7.3	6.9	7.2	6.8	6.1	7.2	6.5	7.1	7.4	7.3	6.1	7.4	6.94	0.41
Splay	12.8	10.8	12.6	12.2	10.9	11.9	11.6	11.2	11.8	12.3	10.8	12.8	11.81	0.69

ISBN 978-3-86956-373-2
ISSN 1613-5652