

Objects as Modules in Newspeak

Gilad Bracha
Ministry of Truth
gilad@bracha.org

Peter von der Ahé
Google
ahe@google.com

Vassili Bykov
Google
vassili@google.com

Yaron Kashai
Cadence Design Systems
yaron@cadence.com

William Maddox
maddox@language-engines.com

Eliot Miranda
Qwaq
eliot@qwaq.com

Abstract

We describe support for modularity in Newspeak, a new programming language descended from Smalltalk [34] and Self [70]. Like Self, all computation — even an object’s own access to its internal structure — is performed by invoking methods on objects. However, like Smalltalk, Newspeak is class-based. Classes can be nested arbitrarily, as in Beta [47]. Since all names denote method invocations, all classes are virtual; in particular, superclasses are virtual, so all classes act as mixins. Unlike its predecessors, there is no static state in Newspeak, nor is there a global namespace. Top level classes act as module definitions, which are independent, immutable, self-contained parametric namespaces. They can be instantiated into modules which may be stateful and mutually recursive. Naturally, like its predecessors, Newspeak is reflective: a mirror library allows structured access to the program meta-level

Categories and Subject Descriptors D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features Classes and objects; Modules, packages; Inheritance.
; D.3.2 [PROGRAMMING LANGUAGES]: Language Classifications Object-oriented languages

General Terms Languages, Design

Keywords Virtual Classes, Mixins, Components, Dynamic Typing, Reflection, Capabilities, Sandboxes

1. Introduction

Modularity is a major issue in software design. Many programming languages provide features intended to support modularity. The state of the art leaves much to be desired however. Few languages support the use of modules as components. Typically, modules are not first class values in the language. They cannot be abstracted over. One cannot have multiple instances of a module definition in a single program, for example, nor are mutually recursive modules supported.

These deficits are not theoretical. In recent years they have spawned the evolution of a variety of extralinguistic tools (e.g., [1, 2, 4, 3]). Even relatively powerful module constructs such as those of ML [44] have engendered a large body of research aimed at relaxing their limitations ([60, 22, 40, 58, 52, 23]).

The Newspeak programming language [16] supports component-style modularity via a novel object-oriented language design, based on two key ideas:

1. All names are late bound.
2. There is no global namespace.

Point (1) holds because the only run time operation in Newspeak is virtual method invocation, known as *message send* in Smalltalk and Self parlance. We refer to this style of programming as *message-based programming*, and consider Newspeak to be a *message-based* language. We use the terms *method invocation* and *message send* interchangeably throughout this paper.¹

The exclusive use of messages in Newspeak strictly enforces the adage “program to an interface not an implementation”. The concept of interface is central to modularity [11].

¹ The term *message* is widely associated with asynchrony. However, messages may be synchronous as well. We consider virtual method invocations to be synchronous message sends. We plan to add support for asynchrony to Newspeak in the future.

As in Self, there is also no way to directly reference a slot (we follow Self in referring to storage locations as *slots*) since the only operation allowed is method invocation. Slot declarations implicitly introduce accessor methods.

Unlike Self, Newspeak is a class based language. As with slots, there is no way to refer directly to class declarations. Class declarations implicitly introduce accessor methods for the classes. This implies that classes are first class values, and that class names are dynamically bound, and subject to override just like methods. In other words, classes are always virtual [46].

In particular, superclass names are also dynamically bound, implying that a class declaration is never statically tied to a specific superclass. Every class declaration is in fact a mixin declaration [17].

Newspeak supports nested class declarations. Top level classes serve as *module definitions*, and their instances are *modules*. The idea that classes should be used to define modules is natural and obvious. However, its successful application relies crucially on the absence of a global namespace.

Because there is no global namespace, a top level class cannot refer to an enclosing scope; all names used within it must be defined by it or inherited. This enables a discipline whereby all external dependencies of a module declaration are explicitly listed at the top of the class.

Module definitions are stateless; even though Newspeak is an imperative language, it has no concept of static state.

It follows that any connection to the world outside the module must come via parameters supplied at instance creation. Modules therefore act as sandboxes, providing a natural fit with object-capability based security [49].

Of course, since modules are instances, they are first class. Side-by-side deployment of multiple instances of a module definition is trivial. Modules being objects, they conform to a procedural interface, and distinct implementations of the same module interface are possible as well.

The combination of virtual classes and class nesting allows entire class hierarchies to be defined, mixed in and overridden within modules, supporting class hierarchy inheritance [57, 27].

1.1 Contributions

The main contribution of this paper is the design and implementation of a programming language that combines support for:

- First class, mutually recursive modules.
- Side-by-side deployment of multiple module instances.
- Multiple simultaneous implementations of a module interface.
- Class hierarchy inheritance and mixins, including module mixins.
- Object capability based security and module sandboxing

- Full reflectivity.

These features are of considerable practical value, and many of them are the subject of ongoing research. To the best of our knowledge, no other programming language supports all of these features.

Specific technical contributions are the use of mirrors to resolve the inherent tension between reflectivity and security; a semantics that reduces the risk of inadvertent name capture via inheritance; a method lookup algorithm supporting these semantics that also incorporates dynamic enforcement of per object access control; and an instance initialization scheme that implicitly supports the factory pattern while decoupling the instance interface from the factory interface.

1.2 Roadmap

The next section introduces Newspeak and its key ideas by means of informal examples. Section 3 describes the semantics with more precision. Next, section 4 includes design rationale and usage guidelines. Section 5 describes our experience using Newspeak and the project's current status, followed by a discussion of related work (section 6), future work (7) and conclusions.

2. Newspeak by Example

Syntactically, Newspeak is closely related to Smalltalk and Self. For readers ignorant of Smalltalk syntax, we will explain its essentials below. We begin with a simple example which introduces the class Point.

```
class Point = ( "This is a comment"
  " The instance initializer "
  | x y | " Declare slots "
)
(
  public printString = (
    ' x = ', x printString, ' y = ', y printString
  )
)
```

The class defines two slots *x* and *y*. Slots are declared between vertical bars. Slots are similar to instance variables, except that they are never accessed directly. Slots are accessed only through automatically generated getters and setters. If *p* is a Point, *p x* and *p y* denote the values stored in *p*'s *x* and *y* slots respectively. Note that there is no dot between *p* and *x*; since method invocation is the only operation in Newspeak, it can be recognized implicitly by the compiler.

The first pair of parentheses in the class declaration delimits the *instance initializer* where slots are declared and initialized (if no explicit initializer is present, the slots are set to **nil**).

Setter methods are denoted by the slot name followed by a colon, so *p x: 91* sets the *x* coordinate of *p* to 91. This is an example of Smalltalk's keyword syntax. In general, the header of an *N*-ary method, *N* > 0, is declared *id1: p1 id2: p2 ... idN: pN*, where *id1: id2: ... idN:* is the name of the

method and $p_i, 1 \leq i \leq N$ are the formal parameters; each id_i is a *keyword*. An invocation of such a method is written $id1: e1 id2: e2 \dots idN: eN$, where the e_i are expressions denoting the actual arguments. The order of the keywords is significant and cannot be changed.

In addition to keyword methods, one may define methods whose name is composed of special symbols such as $+$, $\&$, $|$ etc. These methods always take one argument and are written using infix notation, e.g., $a * b$. These are known as *binary methods*. Our example shows the use of such a binary method: the $;$ method of `String` which implements string concatenation.

Within the body of `Point`, the names $x, y, x:$ and $y:$ are in scope and can be used directly, as shown in the method `printString`. However, x and y denote calls to the getter methods, not references to variables.

Newspeak programs enjoy the property of *representation independence* — one can change the layout of objects without any need to make further source changes anywhere in the program. For example, if we choose to modify `Point` so that it uses polar coordinates, no modification to the `printString` method is needed, as long as we preserve the interface of `Point` by providing methods x and y that compute the cartesian coordinates (note that the caret (^) is used to indicate that an expression should be returned from the method, just like the **return** keyword in conventional languages):

```
class Point = (
  | rho theta | " Declare slots "
) (
  public x = ( ^rho * theta cos)
  public y = ( ^rho * theta sin)
  public printString = (
    ^'x = ', x printString, ' y = ', y printString
  )
)
```

Newspeak classes can be nested within one another to arbitrary depth. Thus, a Newspeak class can have three kinds of members: slots, methods and classes. All references to names are always treated as method invocations, so any member declaration within a class can be overridden in a subclass. It is possible not only to override methods with methods, but to override slots, classes and methods with each other. For example, one can decide that in a particular subclass, a slot value should in fact be computed by a function, and simply override the slot with a method.

In the simple example above, it is possible to initialize a point's slots using its public interface. This is bad style however. It is better to initialize an object when it is created. Here is a more refined version of `Point` illustrating how this is done.

```
class Point x: i y: j = (
  " This section is the instance initializer "
  |
  public x ::= i. " ::= denotes slot initialization"
```

```
class CombinatorialParsing usingLib: platform = (
  | "We use = to set immutable slots"
  BlockContext = platform Kernel BlockContext.
  OrderedCollection =
    platform Collections OrderedCollection.
  Error = platform Exceptions Error.
  |
)
(
  class CombinatorialParser = (...)(...)
  class AlternatingParser = CombinatorialParser(...)(...)
  class SequentialParser = CombinatorialParser(...)(...)
  class WrappingParser = CombinatorialParser(...)(...)
)
```

Figure 1. Outline of a parser combinator library

```
public y ::= j.
| )
( " instance side "
  public printString = (
    ' x = ', x printString, ' y = ', y printString
  )
)
```

The class declaration evaluates to a *class object*. Instances may only be created by invoking a factory method on `Point`.

Every class has a single *primary factory*, in this case $x:y:$. If no factory name is given, it defaults to `new`. The primary factory method's header is declared immediately after the class name. The declaration of the primary factory automatically generates an implementation. This implementation will ensure that the instance initializer of the class is executed. The formal parameters of the primary factory are in scope in the instance initializer. To create a fully initialized instance of `Point` write, e.g.:

```
Point x: 42 y: 91
```

Having introduced basic syntax and terminology, let us examine a substantial example.

2.1 A Basic Module

Our next example is based on the parser combinator [37, 39, 72] library described in [15]. An outline of the library is shown in figure 1.

The actual class `CombinatorialParsing` includes 21 nested classes that constitute a class library implementing parser combinators. Several of these are shown in the outline above. The root of this hierarchy is `CombinatorialParser`. It defines methods that implement the various operations (combinators) on parsers. One such method is the alternation combinator, |

```
| p = (
  " The alternation combinator - denoted by | in BNF -
  but this is really the / operator of PEGs [32]"
  ^AlternatingParser new either: [self] or: p
)
```

The details of this code are unimportant to us here, except for one point: the code refers to another class in the library: `AlternatingParser`. Recall, however, that a name such as `AlternatingParser` cannot refer to a class directly. Rather, it is a method invocation that will return the desired class. This method is defined implicitly in class `CombinatorialParsing` by the declaration of the nested class `AlternatingParser`. The method is available to all code nested within `CombinatorialParsing`. This is how an enclosing class provides a namespace for its nested classes.

What of classes defined outside the topmost enclosing class? How are they accessed? Remember, top level classes do not have access to any surrounding scope.²

Consequently, all names that may be referenced inside a module definition must be defined by it or inherited from a superclass, which must also be a module definition. These names will include the names of the formal parameters of the module definition's factory method.

Consider the following method, taken from the nested class `SequentialParser`

```

, p = (
  | o |
  assert:[p isBlock].
  o: (OrderedCollection new
      addAll: parserFuns;
      add: p; yourself).
  ^SequentialParser new on: o
)

```

The name `OrderedCollection` is intended to refer to a class of the standard collections library. In the absence of a global namespace, there is no way to refer to this class directly. Instead, we have defined a slot named `OrderedCollection` in the surrounding top level class (figure 1).

When the module is instantiated, it should be passed an object representing the underlying platform as an argument to its factory method using `Lib`. This object will be the value of the factory method's formal parameter `platform`. During the initialization of the module, the slot will be initialized via the expression `platform Collections OrderedCollection`. This sends the message `Collections` to `platform`, presumably returning an object representing an instance of the platform's collection library. This object then responds to the message `OrderedCollection`, returning the desired class. The class is stored in the slot, and is available to code within the module definition via the slot's getter method.

The slot definition of `OrderedCollection` fills the role of an import statement, as do those of `Error` and `BlockContext`. Note that the parameters to the factory method are only in scope within the instance initializer. The programmer must take explicit action to make (parts of) them available to the

² We currently allow the superclass of a top level class to be drawn from a surrounding scope, as shown in our next example. We plan to change this and define classes like `BlocklessCombinatorialParsing` as mixins, to be applied to their superclasses separately.

```

class BlocklessCombinatorialParsing
  usingLib: platform =
  CombinatorialParsing usingLib: platform (
    |
    Dictionary = platform Collections Dictionary.
    List = platform Collections OrderedCollection.
    |
  )(
  class CombinatorialParser =
    super CombinatorialParser ( | name | )(…)
  class AlternatingParser =
    super AlternatingParser (…)(…)
  class SequentialParser =
    super SequentialParser (…)(…)
  class ExecutableGrammar =
    CombinatorialParser (…)(…)
  class ForwardReferenceParser =
    CombinatorialParser (…)(…)
  class ForwardWrappingParser =
    WrappingParser (…)(…)
)

```

Figure 2. Subclassing a Hierarchy

rest of the module. The preferred idiom is to extract individual classes and store them in slots, as shown here. It is then possible to determine the module's external dependencies at a glance, by looking at the instance initializer. Encouraging this idiom is the prime motivation for restricting the scope of the factory arguments to the initializer.

Since the entire library is embedded within a single class, it is possible to create multiple instances of the library and use them simultaneously (side-by-side deployment). For example, we could instantiate the library with different platform objects, which could provide different implementations of `OrderedCollection` with different characteristics (e.g., varying speed and memory requirements, logging capabilities etc.). Different module instances do not interfere with each other, because there is no static state. Module definitions are therefore re-entrant.

It is also possible to provide different implementations of the library. Since the library is accessed strictly via a procedural interface, functionally equivalent implementations may be used transparently to clients. For example, an implementation based on packrat parsing [31] could be deployed in parallel to the existing one, improving speed at the expense of memory — but no client of the library would need to be modified in any way.

2.2 Class Hierarchy Inheritance

We now extend our example to show how inheritance can be applied to an entire class hierarchy.

In `CombinatorialParsing`, combinators require that their arguments be closures. This is typical when implementing parser combinators in an eager language. Using closures in

this manner is undesirable: it is more verbose, and somewhat error prone. We will now use inheritance to produce a modified library where combinators work directly on parsers.

Figure 2 shows an outline of class `BlocklessCombinatorialParsing`. The name of the superclass, `CombinatorialParsing`, is followed by the message `usingLib: platform`. We do this in order to determine what parameters will be made available to the superclass' initializer. Before a subclass' instance initializer is executed, control passes to the instance initializer of its superclass in much the same way as constructors are chained in mainstream languages. Here we indicate that `platform` will be passed on to the superclass' instance initializer.

`BlocklessCombinatorialParsing` provides 3 new classes, and overrides 3 existing ones. Each overriding class subclasses the superclass' class of the same name. The overridden classes all have a common property: they include combinator methods that assert that the incoming parameter must be a closure. The purpose of the overriding classes is to eliminate these assertions. In addition, the class `CombinatorialParser` has added a slot, `name`.

Since `CombinatorialParser` is the superclass of all other classes in `CombinatorialParsing`, they all inherit the new slot. Let us see how this occurs.

When the class declaration for `CombinatorialParser` is overridden in `BlocklessCombinatorialParsing`, the automatically generated accessor method is overridden as well. Hence every attempt to access `CombinatorialParser` on an instance of `BlocklessCombinatorialParsing` will produce the overridden class rather than the original.

Classes are generated lazily. The first time a class is referenced, its accessor manufactures the class and caches the result. Subsequent accesses always return the same class object.

As an example, consider the class `WrappingParser`. This class is declared in `CombinatorialParsing` and inherited unchanged in `BlocklessCombinatorialParser`. It declares its superclass to be `CombinatorialParser`. The first attempt to use `WrappingParser` on an instance of `BlocklessCombinatorialParser` will cause the class to be generated; this will require accessing the superclass. The superclass clause denotes a method invocation, not a direct reference to a class, just like every other name in a Newspeak program. Therefore, the new class will be a subclass of the overridden version of `CombinatorialParser`, as intended. The same holds for all other subclasses of `CombinatorialParser` in the library, as one would expect.

2.3 Nesting Models Architecture

Our next example is a compiler for Newspeak, outlined in figure 3. The factory method accepts three parameters: a platform object, a parsing module that provides the front end of the compiler, and a mirror module that defines the reflection system accessed by the compiler.

```
Newspeak2Compilation usingPlatform: platform
                             newspeakParser: ns2Parser
                             mirrorLib: mirrors = (
|
astModule = ns2Parser ASTModule.
Collection = platform Collections Collection.
Parser = ns2Parser Parser.
ASTTool = astModule ASTTool.
AST = astModule AST.
SendAST = astModule SendAST.
... "other imported classes such as VarDeclAST etc."
"Module variables"
parser = Parser new.
getterPool = Array new: 256.
setterPool = Array new: 256.
protected StringAST = astModule StringAST.
|
)
class Compiler = (...)(
  class AST2ByteCodeCompiler = ASTTool (...)(
    class CodeGenerator = (...)(...)
  )
  class Rewriter = ASTTool (...)(
    class BlockLocalReturnAST = AST (...)(...)
    class ImplicitRecvSendAST = SendAST (...)(...)
    class ParameterAST = VarDeclAST (...)(...)
    class ReceiverAST = VariableAST (...)(...)
    class SuperSendAST = SendAST (...)(...)
    class TemporaryAST = VarDeclAST (...)(...)
  )
  class ScopeBuilder = ASTTool (...)(...)
)
class Scope = (...)(...)
class SymbolTableEntry = (...)(...)
)
```

Figure 3. A Compiler

The module definition has several nested classes, providing symbol tables, scopes etc. Here, we focus on the `Compiler` class that performs the actual compilation. Its main nested classes correspond to phases of compilation: `ScopeBuilder` constructs scopes, `Rewriter` transforms the AST into a form more suitable for compilation, and the class `AST2ByteCodeCompiler` actually produces code.

The rewriting phase makes use of a large number of classes representing specialized AST nodes of the lower level representation that is fed into the byte code compiler. There is no need to expose these classes to other phases of the compilation, and so it is convenient to nest them inside `Rewriter`.

The byte code compiler has several nested classes, the most important being `CodeGenerator`. Again, the code generator is of no interest to other parts of the compiler, and is

best nested within `AST2ByteCodeCompiler`, where it logically belongs.

In both cases, we see classes nested three levels deep inside the compilation module; this nesting structures reflects the architectural structure of the application.

2.4 Assembling an Application

Since modules have no access to a shared global namespace, it may not be immediately obvious how a complete program is actually assembled and run. There is a need for some place to glue different modules together, by instantiating module definitions and feeding in any arguments their factories require. We will now show how this may be accomplished.

Throughout this section, assume that factory arguments such as `platform`, `parsingModule` etc. are defined as slots in the surrounding scope. We will also use the syntax `m::e` in place of `m:(e)` to reduce the need for parentheses.

Consider the task of deploying a stand alone Newspeak compiler. To construct a working compiler, one must instantiate `Newspeak2Compilation`:

```
Newspeak2Compilation usingPlatform: platform
                      newspeakParser: parsingModule
                      mirrorLib: mirrors.
```

We'll need a parsing module and a mirror module as arguments. The mirror module is easy to create:

```
mirrors:: NewspeakMirrors usingPlatform: platform.
```

The parsing module is more involved:

```
parsingModule:: Newspeak2Parsing usingLib: platform
                 ast: astModule
                 grammar: ns2Grammar.
```

Apart from the `platform`, we need an AST module as one parameter, and the grammar itself as the other. We want the grammar separate from the parser, so we can use it for other things (e.g., pretty printing).

Generating an AST module is simple:

```
astModule:: Newspeak2AST usingPlatform: platform.
```

The grammar does not require access to the `platform`, but it does need a parser combinator library:

```
ns2Grammar::
  Newspeak2Grammar parserLib: parserLib.
```

The parser combinator library we use is `BlocklessCombinatorialParsing`.

```
parserLib::
  BlocklessCombinatorialParsing usingLib: platform.
```

We can combine all the above steps in a single method `main:args:` as shown in figure 4. The method has two parameters: a `platform` object and an array of arguments.

In the figure, `CompilerApp` imports the various module definitions that constitute our application, and `main:args:` is used to link them together.

```
class CompilerApp compiler: c mirrors: m parsing: ns2p
asts: a grammar: g parserLib: p runner: r = (
  |
  BlocklessCombinatorialParsing = p.
  Newspeak2Grammar = g.
  Newspeak2AST = a.
  Newspeak2Parsing = ns2p.
  NewspeakMirrors = m.
  Newspeak2Compilation = c.
  CompilerRunner = r.
  |
  )(
  public main: platform args: argv = (
    |
    parserLib
    ns2Grammar
    astModule
    parsingModule
    mirrors
    compilerModule
    ns2compiler
    |
    parserLib:: BlocklessCombinatorialParsing
                 usingLib: platform.
    ns2Grammar:: Newspeak2Grammar
                 parserLib: parserLib.
    astModule:: Newspeak2AST usingPlatform: platform.
    parsingModule:: Newspeak2Parsing
                    usingLib: platform
                    ast: astModule
                    grammar: ns2Grammar.
    mirrors:: NewspeakMirrors usingPlatform: platform.
    compilerModule:: Newspeak2Compilation
                    usingPlatform: platform
                    newspeakParser: parsingModule
                    mirrorLib: mirrors.
    ns2compiler:: compilerModule Compiler new.
    (CompilerRunner
     for: ns2compiler usingPlatform: platform)
     compileFile: (argv at: 1).
  ))
```

Figure 4. Deploying the compiler as an application

```
CompilerApp
  compiler: Newspeak2Compilation
  mirrors: NewspeakMirrors
  parsing: Newspeak2Parsing
  asts: Newspeak2AST
  grammar: Newspeak2Grammar
  parserLib: BlocklessCombinatorialParsing
  runner: CompilerRunner
```

Figure 5. Packaging the application

Our IDE provides a namespace containing our application’s top level classes. Using such a namespace, we can evaluate the code shown in figure 5.³

The result is an object *o* which points at all of our application’s module definitions and has a `main:args:` method that, given a valid platform object, will build an instance of our application — a Newspeak compiler — and run it. The serialized form of *o* will serve as our deployment format.

When serializing *o* we cut off the object graph at Object and a few core system classes (e.g., Class, String etc.), replacing them with symbolic links. The corresponding objects will be provided by the host platform upon deserialization. Since *o* refers only to module definitions (its own class and the arguments we passed to its factory, all of which were module definitions as well), and module definitions are stateless, the resulting serialized form contains the necessary code and little else.

To run the application, deserialize the serialized form into an object (hooking up the missing references to Object, String etc.) and call its `main:args:` method, passing it a platform object and any arguments.

This is similar to the classic C convention of invoking an application via a distinguished function `main()`. CompilerApp is analogous to the C program, its `main:args:` method is analogous to the `main()` function, and the serialized instance of CompilerApp is analogous to a binary file. Deserialization is the equivalent of linking and loading.

However, our approach differs in subtle yet crucial ways.

- Linkage is trivial. Only a small fixed set of system objects needs to be linked in by the deserializer.
- The module’s connection to the outside world is entirely mediated by capabilities passed in at run time under the control of the module’s user.
- The module is a first class object in the language and may be manipulated both directly and reflectively by the environment in which it is deployed.

³ The language defines no global namespace. However, application building tools such as compilers, linkers etc. always need a mechanism to obtain their inputs in order to support separate compilation. The namespace provided by the IDE is analogous to a traditional utility like `make` using the file system to name source and binary files.

Obviously, variations are possible, e.g., the `main:args:` method could load required modules via the network, either eagerly or lazily.

We have now established an intuition as to how modularity works in Newspeak and what it can achieve. It is time to discuss the semantics of Newspeak in detail.

3. Semantics

The semantics of the Newspeak language are defined in [16]. Here we focus only on those features of Newspeak that are directly relevant to modularity.

3.1 Classes, Declarations and Mixins

In Newspeak, it is important to distinguish between a *class* and a *class declaration*. In a traditional object oriented language there is a 1:1 correspondence between a class declaration (a syntactic entity) and a class (a run time entity). In Newspeak, however, superclasses are dynamically bound, so a class declaration does not uniquely determine a class. Instead, a class declaration induces a *mixin*. There can be many classes corresponding to a given declaration. Each such class is an application of the mixin induced by its declaration to some superclass.

It is possible to explicitly extract and apply the mixin associated with a class declaration.

We use intuitive terms such as *top level class*, *nested class* and *enclosing class* to refer to classes that correspond, respectively, to top level, nested or enclosing class declarations.

Each instance of an enclosing class has its own distinct set of nested classes. As an example of why this is necessary, consider the case of a nested class whose superclass is imported by the surrounding module definition (e.g., `SuperSendAST` in figure 3). Each module may have its own imports, leading to a completely different nested class.

More generally, the notion that a class is an attribute of an object, just like a method or slot, is motivated by modeling considerations, as in Beta.

3.2 Method Lookup

Given that the only runtime operation in Newspeak is method invocation, defining the meaning of method invocation is central to the semantics of Newspeak. Ordinary message sends involve a receiver and a message consisting of a method name and a (possibly empty) set of arguments. The meaning of such sends is standard, as in Smalltalk, except for the handling of access control.

For simplicity, we defer the discussion of access control to section 3.3. For now, assume all members are public. Under this assumption, ordinary sends are looked up a standard single inheritance class chain, starting with the class of the receiver.

More interesting are *implicit receiver sends* — sends written without an explicit receiver. Before we can explain

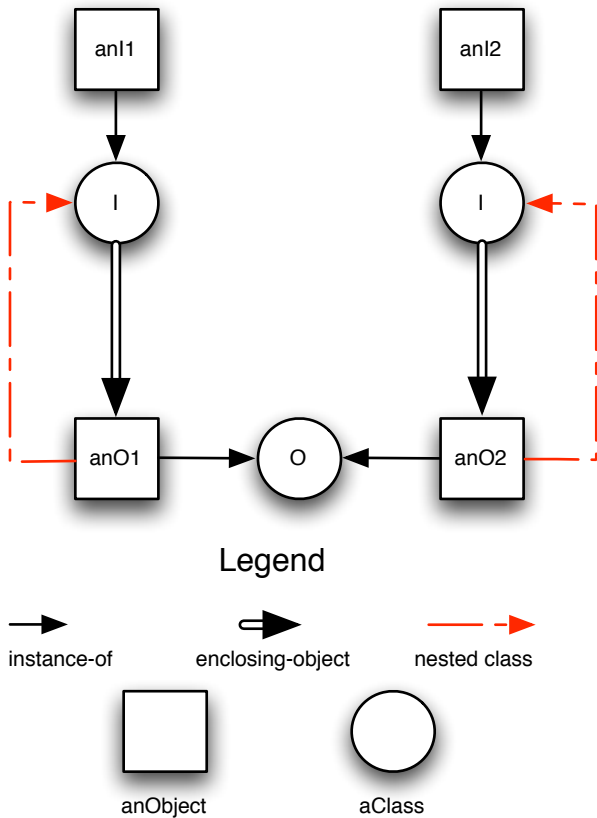


Figure 6. Enclosing classes, nested classes and their instances

how implicit receiver sends work, we must introduce the concept of enclosing objects.

3.2.1 Enclosing Objects

The relationship between an instance *o* of an enclosing class and its nested classes is bidirectional: each such nested class has *o* as its *enclosing object*.

The relationship between enclosing objects and nested classes is illustrated in Figure 6, which depicts a top level class *O* with a nested class *I*. *O* is shown with two instances, *anO1* and *anO2*. Each such instance has its own class *I*. Each *I* class can have its own instances - in this case *anI1* and *anI2* respectively.

A method invoked upon an object may originate in one of a number of classes — the class of the object or any superclass thereof. Each such class could, potentially, be declared inside a distinct lexical scope, and be the result of sending a message to a distinct enclosing object. Hence each may have its own enclosing object which may differ from the enclosing objects of any other class in the superclass chain.

A class' enclosing object is the dynamic representation of the lexical scope of its immediately enclosing class.

```

evalImplicit(message, classDeclaration, receiver) {
  implicitReceiver(r, d) {
    if (d = nil) return receiver; // 1
    if d.declares(message.name) return r; // 2
    var cls := r.class; // 3
    while (cls.declaration ≠ d) cls := cls.superClass; // 4
    return implicitReceiver(cls.enclosingObject, // 5
                          d.enclosingDeclaration);
  }
  return // 6
  implicitReceiver(receiver, classDeclaration).message
}

```

Figure 7. Method lookup for implicit receiver sends

As an example, consider an instance of CombinatorialParser (figure 1) executing the method

```

| p = (
  " The alternation combinator - denoted by | in BNF -
  but this is really the / operator of PEGs"
  ^AlternatingParser new either: [self] or: p
)

```

AlternatingParser refers to the enclosing lexical context — the class declaration of CombinatorialParsing. Specifically, the intended reference is the slot AlternatingParser. The slot is specific to a particular instance of CombinatorialParsing — the enclosing object of CombinatorialParser.

Evidently, there is a close connection between enclosing objects and implicit receiver sends.

3.2.2 Implicit Receiver Sends

The value of an implicit receiver send is determined via an ordinary send to the implicit receiver (see line 6 of figure 7). Our first step then, is to determine the implicit receiver.

The search for an implicit receiver begins by assuming that a matching method declaration will be found in the class declaration immediately enclosing the currently executing method, in which case the implicit receiver is **self**. If this is not the case, we recurse up the lexical chain, examining the enclosing class declaration, implying that the implicit receiver would be the enclosing object of the class corresponding to the current class declaration. This process continues until we reach the top level. If a matching method is not found at that point, the method must be inherited and the implicit receiver is **self** after all. The motivation for this design is treated further in section 4.2.

We now examine the semantics in more detail. Figure 7 gives pseudo-code that defines the semantics of implicit receiver sends. The heart of the process is the nested function *implicitReceiver*. It takes a candidate receiver object *r* and a candidate class declaration *d* as arguments. The function requires the invariant that the class of *r*, or some superclass, corresponds to *d*, unless *d* is **nil**.

A **nil** declaration indicates that we have passed the outermost lexical level, in which case the implicit receiver is **self**

(line 1). Otherwise, if the candidate declaration includes the method, then the receiver is the candidate receiver and we are done (line 2). Because of our invariant, we know that r can respond to the message; its class chain includes a class C corresponding to d — and d includes a matching method.

If d does not include a matching method, then we need to continue our search. The next candidate declaration will be the declaration enclosing the current one (line 5). The new candidate receiver is the enclosing object of C . We know that this new pair of candidate receiver and declaration meets our invariant. Furthermore, because of our invariant, we are guaranteed to find C on the superclass chain of the candidate receiver (line 4). We can then pass its enclosing object to the recursive call (line 5).

3.3 Access Control

Access control changes the description given above slightly. The semantics of access control are necessarily somewhat speculative at this point. The implementation of these features is incomplete, and we need experience with them before they are finalized.

Here are the semantic modifications dictated by our model of access control:

Ordinary Sends. Ordinary sends may only invoke **public** methods. If a **protected** or **private** method is found, lookup fails.⁴

Self sends. Sends whose receiver is the literal **self** may invoke all methods of the immediately enclosing class, but only **public** and **protected** methods of other classes. When searching these classes, if a **private** method is found lookup fails.

Super Sends. Super sends may invoke both **public** and **protected** methods. Lookup begins with the superclass of the class in which the current method was found. If a **private** method is found, lookup fails.

Implicit Receiver Sends. If a **private** method is found in a surrounding lexical scope, that method is invoked directly on the candidate receiver. Otherwise, message lookup is performed starting with the class of the implicit receiver. If a **private** method is found lookup fails.

4. Discussion

4.1 Types

Newspeak is a dynamically typed language. We favor the development of optional/pluggable type systems [14] for Newspeak, but have not pursued the topic. Our experience so far gives no indication that the lack of static typing is a significant drawback to the use of the language.

The fact that we are not bound by a type discipline has given us the freedom to experiment with idioms that are

⁴Strictly speaking, the `doesNotUnderstand:` method is invoked. By default this raises an error, but it can be profitably overridden.

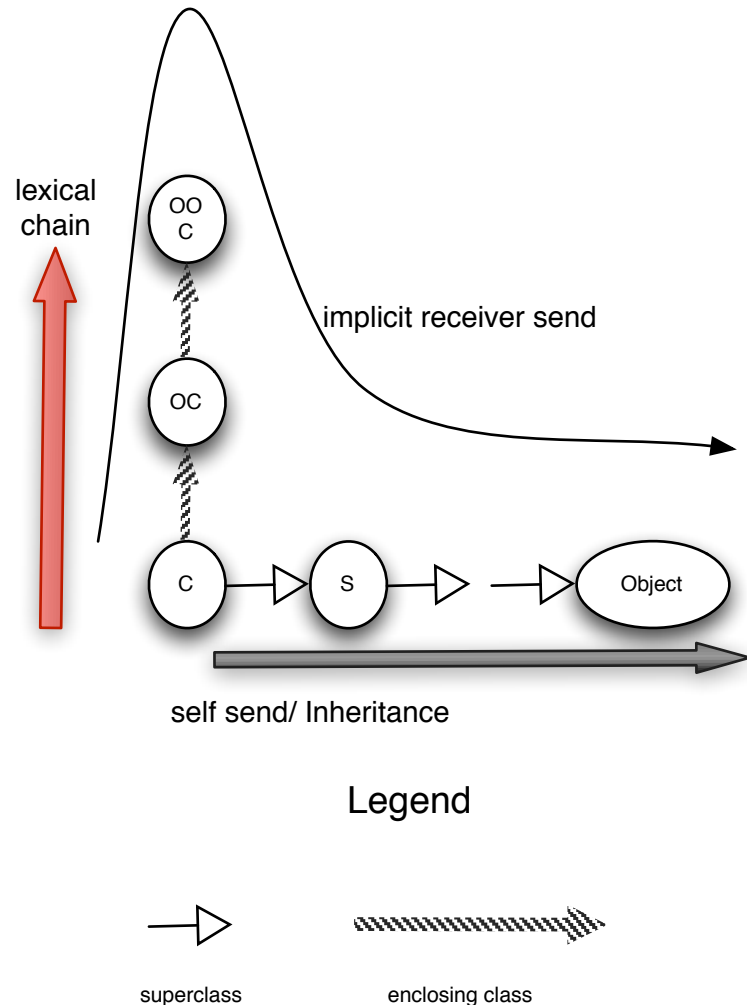


Figure 8. Method lookup

challenging to typecheck. We routinely define virtual classes that inherit from classes originating in other modules, a practice that is severely restricted in systems such as [33, 20]. Indeed, such cross-family inheritance is inevitable given the mechanisms we use to "import" classes between modules.

4.2 Method Lookup

In most object-oriented programming languages, if the receiver of a message is **self** (aka **this**) it can be omitted. In the presence of class nesting, if the receiver is implicit (i.e., omitted), it may be either **self** or some enclosing object. In statically typed languages, the lexical level of the receiver is determined at compile time [45, 63]. In dynamically typed languages, it is usually done at run time as part of the method lookup process. Typically, one starts the lookup with the class of **self** (or **self** itself in a prototype based language) and proceeds up its inheritance chain; if no method is found, one jumps to the enclosing lexical level and recurses.

This notion is described in detail in the Java Language Specification [36] and formalized in [63]. It is called "comb semantics" in NewtonScript [64].⁵

Newspeak differs in that lookup proceeds up the lexical scope chain (starting with the lexically deepest activation record) and only if no lexically visible matching method is found do we proceed up the inheritance chain of **self**. In no case do we search the inheritance chains of enclosing objects. See figure 8 for an illustration.

The motivation for our design is the desire to avoid inadvertent capture of method names by superclasses. In both Beta and Java, situations such as the following (illustrated in Java) can arise:

```
class Sup { }
class Outer {
  int m(){ return 91}
  class Inner extends Sup {
    int foo(){return m();}
    // case 1: new Outer.Inner().foo() = 91
  }
}
```

The expectation is that a call to `foo` will yield the result 91, because it returns the result of the call to `m`, which is defined in the enclosing scope.

Consider what happens if one now modifies the definition of `Sup`:

```
class Sup {
  int m(){ return 42}
  // case 2: new Outer.Inner().foo() = 42
}
```

The result of calling `foo` is now 42.

The behavior of the subclass has changed in a way that its designer could not anticipate, which is clearly undesirable. Of course, inheritance in general suffers from modularity problems, but there is no reason to aggravate them further.

In Newspeak, code is immune to capture of lexically scoped names due to changes in inherited libraries. We believe this approach is more robust in the face of program evolution, especially on larger scales.

4.3 Inheritance and Modularity

There is an inherent conflict between inheritance and modularity. In the absence of inheritance, we can always add functionality to a module definition without fear of breaking its clients. With inheritance, any new method added may conflict with a subclass that has independently (and unbeknownst to the author of the superclass) added a method of the same name but unrelated semantics.

Despite this problem we have opted to use classes with inheritance as our sole modularity mechanism. Our initial plan was to make modules and classes distinct; classes would

support inheritance but modules would not. We found that module definitions were very much like classes:

- They are first class objects that are instantiated multiple times via factory method invocations.
- They contain slot declarations that describe the state of their instances.
- They contain class declarations nested within them. Each module instance needs to have its own set of nested classes.

It seemed very undesirable to introduce two very similar yet subtly distinct constructs. It also seemed a pity to forego the ability to mix in module definitions.

Ultimately, we realized that in a networked world, the risk of conflict with subclasses is manageable. We expect to be able to search for and locate every subclass of a given class, worldwide, much as one does in one's own IDE. The conflict can be therefore be avoided pre-emptively.

This scenario is still somewhat in the future, but we believe it is not too distant a future. Newspeak is intended to be a network aware "cloud computing" language, operating in precisely such an environment. Hence, we chose not to compromise our design by introducing a split between classes and module definitions.

4.4 Further Binding

Existing work on virtual classes inherits the idea of *further binding* from Beta. This means that when a nested class overrides another, it does not replace it; rather, the original nested class is replaced by a composition of itself with and the overriding nested class. This composition is typically defined as a form of subclassing or mixin composition, and may involve a linearization of the class hierarchy.

In Newspeak, class overriding is identical to classical method overriding.

Further binding is partly motivated as a means of enforcing a measure of semantic consistency when classes are overridden. This is not the dominant consideration in our design. Rather, our chief concern is flexibility.

We allow programmers to completely replace one implementation of a class by another. It is also possible to replace a class by a method that generates a class (say, based on network data), or by a slot that stores a class. Further binding makes it difficult to support such practices.

While further binding is a common idiom, we believe it is more general to provide a simple uniform primitive in the language. When further binding of a nested class *C* is desired, it can be done by explicitly declaring the overriding class' super class as super *C*. This allows for mixin composition of class hierarchies, but does not preclude other use cases.

⁵NewtonScript has no lexical nesting, but its lookup semantics are nevertheless essentially the same.

4.5 Considerations in Module Design

The Newspeak modularity system raises questions about how to best design modular programs. How does one decide what constitutes a module? Within a module, how do we decide when a class should be nested in another?

An extensive study of these questions is far beyond what can be covered here. Nevertheless, we have some comments.

Beta is the only language that supports virtual classes for which there is a large body of programming experience over time. The Beta community's approach is driven by modeling considerations, and we believe this holds for Newspeak as well.

Conceptually, a nested class is an attribute of a specific object. The nesting structure should model the objects in question. More specifically, if a class requires access to the internals of an object, it likely should be nested in the object's class.

Sometimes a class is only useful in the context of another class. Encapsulation considerations will encourage us to nest the former within the latter. This is not always desirable however. Pragmatically, this may lead to the creation of many identical classes — one per instance of the enclosing class. If the enclosing class has many instances, this can be problematic. This situation likely reflects a modeling error; perhaps there is a missing intermediate class representing the appropriate enclosing class.

The same considerations may be helpful when examining a module as a whole. If a class is not a property of the module, it likely deserves an independent existence. An example would be our AST module `Newspeak2AST` which is independent of the compiler, because ASTs have utility in other contexts.

In summary, we are still learning about these issues, and further experimentation is needed.

4.6 Mutual Recursion

Newspeak makes it possible to create mutually recursive modules. However, this is not as convenient as it should be. Given two mutually recursive modules, one of their definitions must be instantiated before the other, yet each expects the other as an argument to its factory method.

We can work around this problem by introducing a proxy object for one of the modules. The proxy is passed to the factory of the first module in lieu of the second module. We can then instantiate the second module, and set the proxy to delegate to it. As long as the initialization of the two modules is well founded, this solution is workable, albeit awkward. An alternative is the use of mixin composition [56].

We hope to introduce better support for mutual recursion among modules in future work.

4.7 Versioning and Lifecycle Issues

In this paper we do not consider questions such as module versioning, module update, module repository management

etc. [1, 2]. There is no doubt a need for standardized ways of dealing with these problems, and suitable APIs should evolve to handle them.

Because Newspeak is dynamically typed and reflective, and modules are first class objects, we expect that such management issues can be handled in libraries via metaprogramming.

4.8 Relation to the Object-Capability Security Model

Newspeak's security model is founded on the object-capability model [49]. In this model, the authority to perform an operation (which may have potential security implications) is provided exclusively through objects that act as *capabilities*. This places several requirements on the programming language. These include:

1. Objects must provide true data abstraction; they must be able to hide their internals from other objects — even other objects of the same class.
2. There must be no static state (e.g., static or global variables). Such state can be accessed by code that was not explicitly authorized to do so, providing *ambient authority*.

With respect to point (1), Newspeak supports object-level encapsulation. Object members that are private or protected can only be referenced within the scope of the object. This is not the case in mainstream object-oriented languages such as C++, Java or C# (in Javascript, closures are used for this purpose; we find this awkward).

Now consider point (2). A typical example of ambient authority might be a class `File` with a constructor that takes a file name and returns an instance that can access a file in the local file system. This is a standard design, but in a situation where file system access must be restricted, requires authorization checks on every access.

Systems that combine security consciousness with pervasive ambient authority, such as Java, pay a high run time cost for such checks, since they may require a traversal of the entire call stack to ensure that no unauthorized caller, however indirect, might retrieve information about a file's contents or even its existence [35]. As a result, these checks are often disabled, completely undermining security.

An alternative approach is a sandbox model, where only operations deemed safe are provided. Java also supports sandboxing via class loaders [41]. However, class loaders are complex and brittle; they can introduce interoperability problems because they create incompatible types, and they do not compose well.

As we have already noted, there is no static state in Newspeak, addressing point (2). In Newspeak, each module runs in its own sandbox, created explicitly by providing the module with the desired capabilities (i.e., objects provided as parameters to the factory when the object was constructed). There is usually no need for explicit (and costly) security

checks on individual operations to ensure that the caller has the appropriate authority to invoke them. The fact that the caller holds a reference to an object that can perform the operation conveys the necessary authority.

Here are some example uses of the object-capability model.

Reflection. Newspeak provides a mirror library supporting both introspection and self modification. If classes support only base level operations such as instance creation and subclassing,⁶ code can perform reflective operations only via mirror objects. Mirrors serve as capabilities for reflection.

Foreign function calls. The ability to call foreign functions is a practical necessity in most systems. However, once code has access to a foreign function, it is very difficult to give any security guarantees. Newspeak does not have a language feature (such as native methods or **extern** functions) supporting foreign function calls. Instead, foreign calls are mediated via objects known as *aliens* [50, 16]. Unless code has access to an alien, it cannot call foreign code.

Notwithstanding all of the above, the current Newspeak prototype does not provide any security guarantees. The language provides a foundation for security, but a secure system requires a lot more: careful API design, security audits, a secure binary format etc. These problems are topics for future work.

5. Status and Experience

Newspeak was developed as part of an application development project at Cadence Design Systems. That project no longer exists, for reasons unrelated to Newspeak. The language is now being developed as an open source project.

An implementation of Newspeak is freely available on the web at <http://newspeaklanguage.org>. The current prototype is built upon Squeak Smalltalk. It includes a fairly complete implementation of the Newspeak language as described here. However, the implemented language deviates from its specification in some minor respects. The most important of these is that access control has not been fully implemented yet.

Newspeak has been used to write its own IDE (including compiler, debugger, class browsers, object inspectors, unit testing framework, , a mirror based reflection API etc.), a portable GUI tool kit [19], an object serializer/deserializer, a parser combinator library, a regular expression package, core libraries for collections, streams, strings and files, parts of our foreign function interface as well as CAD application code at Cadence.

Some of the modules cited above were designed and coded directly in Newspeak. In other cases, we have ported code from Smalltalk or from earlier interim Newspeak dialects. This porting process is still ongoing; we hope to be

⁶ Caveat: the current implementation uses Squeak classes, and so does not meet this requirement.

able to create a small standalone platform and IDE written entirely in Newspeak.

In our experience porting Smalltalk code to Newspeak is surprisingly straightforward. For example, we have ported much of the Strongtalk core libraries (which are an implementation of the original Smalltalk blue book libraries) and encountered relatively few issues. Some of these ports have been done by programmers with no prior exposure to Newspeak or even Smalltalk.

Our experience with Newspeak is necessarily limited, but decidedly non-trivial. The Newspeak platform as a whole represents approximately 8 person years of work. Seasoned Smalltalk and Beta developers find Newspeak a highly desirable language to work with, with significant advantages.

Newspeak's nested classes provide a natural organization for the problem domain, where every element of a program has a natural place. For the same reasons, porting Smalltalk code to Newspeak tends to produce better structured code.

Newspeak's lightweight syntax and reflectivity make it an excellent medium for defining internal DSLs, which help programmers work at a level of abstraction more suited to the problem domain, and relatively free of solution-space artifacts.

6. Related Work

Newspeak is a direct descendant of Smalltalk [34]. The Newspeak language differs from Smalltalk in a number of important ways, however. Unlike Smalltalk, Newspeak has an intentional, syntactic representation of classes; this is crucial in supporting nested classes, which are not present in Smalltalk. Smalltalk has a global namespace and abundant static state, including global variables, class variables, pool variables and class instance variables as part of globally accessible classes. Most fundamentally, Smalltalk distinguishes between method invocation and variable access — it is not a message based language. These differences lead to a different semantics of method lookup, and to the notion of modularity described in this paper. Smalltalk itself has no linguistic notion of modularity.

The idea of message-based programming comes to us from Self [70]. However, Self has a global namespace, known as *the lobby*, with stateful elements within it. Self is also a prototype based language, whereas Newspeak is class based. Self's notion of modularity is extralinguistic; a tool, the *transporter*, interprets metadata that helps it subdivide the object graph [69].

Newspeak owes several other features to Self, in particular the idea of mirrors [18], which is crucial to combining reflectivity with the object-capability security model.

The third major influence on Newspeak is Beta [47]. Beta introduced the idea of virtual classes. However, Beta is not a message based language, and classes must be declared to be virtual explicitly. In addition, superclasses are never treated as virtual, and so Beta does not support mixins. Beta

programs are composed via its fragment system ([38], Part III). There is always a top level which introduces static state and a global namespace into Beta programs. Beta has a mandatory static type system, and is not a reflective language (though see [61]). All these things make it very different from Newspeak.

Beta is also unusual in its concept of *pattern*, which unifies types, classes, methods, functions and procedures. In Newspeak, classes and methods are distinct.

The language gbeta [25] is similar to Beta, but extends patterns to mixins. This allows gbeta to support class hierarchy inheritance [27]. The gbeta semantics dictate a very specific, structured model of how classes nested in a subclass compose with the corresponding nested classes in the superclass [26]. In contrast, in Newspeak, class overrides behave like classical method overrides.

The closest relatives of Newspeak modules are Units ([29], [30]) which are parameterized in a way that is similar to top level Newspeak classes. Units enable mixins when a class' superclass is a parameter to the Unit, whereas in Newspeak, all classes are mixins, since their superclasses are always virtual. In contrast to Newspeak, Units and classes are distinct. Units do not support inheritance. See section 4.3 for discussion.

A related area of research are languages that support components [66, 73, 6, 42, 43, 59, 48]. Newspeak modules are not strictly components because of their use of inheritance (again, see section 4.3). Component research often emphasizes static checking of component compatibility and component life cycle issues. These topics are beyond the scope of this paper (but see sections 4.1 and 4.7).

AmbientTalk [71] is a language designed to support the programming of mobile systems. It is message based (emphasizing asynchrony) and uses mirrors extensively [51].

No discussion of related work on modularity would be complete without reference to the ML module system [44]. ML modules are very different from Newspeak classes. They are statically typed and can contain types as members. Conversely, they cannot be mutually recursive and are not first class values. ML structures can only be abstracted over via special constructs called functors. Functors cannot be abstracted over at all, though there is a considerable literature on how to overcome these limitations [60, 22, 40, 58, 52]. In particular MixML [23] supports higher order mutually recursive modules with type components. However, MixML modules are not first class values, cannot be reflected upon, and do not support overriding.

Mixin modules ([8, 7, 24]) are closely related to the ideas of [13]. Mixin modules compose via inheritance-like operators and naturally support mutual recursion.

CaesarJ [33, 9], and J& [54, 53] are both quite similar to Newspeak in that classes may nest and be overridden. Both languages are based on Java, and neither attempts to use nested classes as a comprehensive modularity mechanism

as we have done here. Instead, they rely upon Java's global namespace and must deal with the associated static state. Each of these two languages provides a static type system.

In CaesarJ, classes cannot inherit from classes defined in other modules, or from classes defined at different nesting levels. These restrictions preclude importing superclasses in the manner we have shown here.

Both gbeta and CaesarJ are based on the type system of [28], though gbeta allows inheritance across families.

In J&, as in Java, nested classes are properties of classes, not objects. J& allows inheritance across type families, but does not support dynamic binding of superclasses from other class families. As a result, it cannot express our import mechanism either.

Tribe [20] provides a typed semantic framework largely independent of a specific host language. Tribe supports only limited forms of cross-family inheritance via its *adoption* construct, which relies on the notion of a top level scope. This is clearly at odds with our approach, where no top level scope exists, and unrestricted use of inheritance is extensively relied upon.

Common to most of the work on virtual classes is the idea of *further binding*; that is, within a subclass, nested classes implicitly subclass classes with the same name in the superclass. In contrast, in Newspeak, class overriding follows exactly the same rules as method overriding. See section 4.4 for more detail.

Scala [55] supports virtual types but does not support virtual classes (though there is a plan to add such support). Virtual types [68] are only peripherally related to this paper, because we do not consider typechecking.

Scala encourages the use of objects as modules in a style similar to Newspeak, but Scala has a global scope that can include stateful objects, effectively providing static state. Scala uses traditional imports for namespace management, and access control is per type rather than per object.

Java supports a rich variety of nested classes. Among these, inner classes are closest to Newspeak's nested classes because inner classes can access their lexical scope via their enclosing instance ([36], section 8.1.3). Unlike Newspeak nested classes, inner classes in Java are never virtual, and cannot be used to express mixins etc. Virtual types have been proposed for Java [67] however, as have extensions supporting modularity [21].

Large frameworks for managing modules have been proposed in the context of Java ([1, 2]). They provide assorted life cycle management services for modules, and rely on class loaders to provide dynamic namespaces.

The *Securable Modules* proposal [5, 10] for Javascript supports first class modules as objects with no surrounding namespace, enabling object-capability style security. A disciplined use of the facility appears to allow modules in the style we advocate. However, the proposal support imports

via its requires function, which encourages code to hardwire external dependencies within itself.

Mixin layers [62] allow entire groups of mixins to be manipulated and composed as a unit. They also specify a recursive composition semantics for the nested mixins, similar to further binding. Mixin layers emphasize cross cutting concerns, as does aspect orientation.

Some features of aspect oriented programming (AOP), such as before/after advice, can be closely paralleled in Newspeak using mixins. Other AOP features are not supported in Newspeak.

The object capability model is described in [49]. The language E [65] is based on that model. Newspeak supports reflection, whereas E does not. To our knowledge, Newspeak is the first system to combine the object-capability model with full reflectivity.

7. Future Work

We plan to bootstrap a complete platform written entirely in Newspeak to gain further experience with the language. In the process we expect our implementation to mature so that it fully conforms to its specification. This should include support for actor based concurrency, value types, and access control.

A major area for future research is typechecking of Newspeak code. This is a challenging problem. We expect typecheckers for Newspeak to fit in a pluggable type framework based on general metadata support in the language, as suggested in the Newspeak specification.

It would be interesting to develop a formal semantics for Newspeak. The language is simple enough that a completely accurate formal semantics can be defined for it.

Realizing a truly secure system based on Newspeak is a significant challenge that we would like to undertake at a later stage (see section 4.8).

Extending Newspeak module definitions to support a range of module combinators like those described in [13] is also attractive.

The project's long term goal is to create a secure, efficient and above all high productivity platform for cloud computing, broadly interpreted [12]. In that context, we would hope to address issues such as module distribution and update. A less ambitious direction would investigate more conventional approaches involving module versioning.

8. Conclusions

The Newspeak programming language's modularity constructs support a number of desirable features:

- Module definitions are first class objects. They may be stored, passed as arguments, returned from computations etc.
- All external dependencies of a module definition are explicit.

- Module definitions are parametric with respect to their dependencies, and can be deployed independently of them.
- Side by side deployment: multiple instances of a module definition can co-exist simultaneously.
- Alternate module implementations conforming to the same interface can co-exist simultaneously.
- Support for virtual classes, with no restrictions on inheritance, providing flexibility and leading to
- Class hierarchy inheritance: entire module definitions, including complete class library hierarchies may be extended and modified via inheritance
- Mixins. Module definitions and the classes within them can be mixed in in multiple places.
- Support for the object capability model, providing a foundation for security.
- Reflection. Module definitions may be dynamically generated, introspected and modified via reflection.

This combination of features is unique to Newspeak, and has been achieved by means of a novel programming language design. This design is based on an exceedingly simple foundation using the notions of objects and message passing, combined with the absence of global scope. Dynamic typing gives our system a degree of simplicity and flexibility that has proven difficult to obtain otherwise.

Further work is necessary to fully realize the potential of the Newspeak language. Nevertheless, the system is already of both practical and academic value. The language has been implemented with the exception of minor details. The Newspeak platform is publicly available, and a considerable body of software has been written using it. We believe Newspeak makes a meaningful contribution to object oriented language design, and the ideas described here are likely to contribute to other languages and spawn further research.

Acknowledgments

Alex Buckley, Raffaello Giuliatti, Yardena Meymann, James Noble, Kenneth Russell, and Mads Torgersen provided valuable feedback on earlier drafts of this paper. We are also grateful to Matthias Kleine, Ryan Macnak, Yardena Meymann, Stephen Pair and David Pennell for their code contributions to the Newspeak open source project.

The following individuals deserve our thanks for their encouragement and support of our work: Neal Gafter, Robert Hirschfeld, Erik Meijer, Karyn Phan, and Allen Wirfs-Brock.

Above all, we thank our former colleagues at Cadence, Pramod Chandraiah, Sandeep Desai, Allen Goldberg, Douglas McPherson, Meirav Nitzan, Bob Westergaard and Guang

Yang, who took the plunge and wrote application code using a new and immature language and platform.

References

- [1] www.osgi.org.
- [2] Java module system. Java Community Process JSR 277, now defunct, described at <http://jcp.org/en/jsr/detail?id=277>.
- [3] Java on guice: Guice 1.0 user's guide. Available at http://docs.google.com/Doc?id=dd2fmx4z_5df5hw8.
- [4] OpenJDK project Jigsaw. <http://openjdk.java.net/projects/jigsaw>.
- [5] Securable modules proposal for Javascript. Available at <https://wiki.mozilla.org/ServerJS/Modules/SecurableModules>.
- [6] ALLEN, E., CHASE, D., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE, G., AND TOBIN-HOCHSTADT, S. The Fortress language specification, 2005. available from <http://research.sun.com/projects/plrg/>.
- [7] ANCONA, D., FAGORZI, S., MOGGI, E., AND ZUCCA, E. Mixin modules and computational effects. In *ICALP 2003 - Automata, Languages and Programming* (2003), G. Goos, J. Hartmanis, and J. van Leeuwen, Eds., vol. 2719 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 224–238.
- [8] ANCONA, D., AND ZUCCA, E. A calculus of module systems. *Journ. of Functional Programming* 12, 2 (2002), 91–132.
- [9] ARACIC, I., GASIUNAS, V., MEZINI, M., AND OSTERMANN, K. Overview of CaesarJ. *Transactions on AOSD I, LNCS 3880* (2006), 135 – 173.
- [10] AWAD, I. Module identification and loading for Javascript. Available at <http://docs.google.com/Doc?id=dfgxb7gk61d47876fr>.
- [11] BALDWIN, C. Y., AND CLARK, K. B. *Design Rules: The Power of Modularity*. MIT Press, 2000.
- [12] BRACHA, G. Objects as software services. Invited talk at OOPSLA 2005 Dynamic Languages Symposium; updated video available at <http://video.google.com/videoplay?docid=162051834912297779>. Unpublished manuscript available at <http://bracha.org/objectsAsSoftwareServices.pdf>.
- [13] BRACHA, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
- [14] BRACHA, G. Pluggable type systems, Oct. 2004. OOPSLA Workshop on Revival of Dynamic Languages. Available at <http://pico.vub.ac.be/%7Ewdmeuter/RDL04/papers/Bracha.pdf>.
- [15] BRACHA, G. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.* 193 (2007), 3–18.
- [16] BRACHA, G. The Newspeak programming language specification, 2008. Available at <http://bracha.org/newspeak-spec.pdf>.
- [17] BRACHA, G., AND COOK, W. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990).
- [18] BRACHA, G., AND UNGAR, D. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 2004).
- [19] BYKOV, V. Hopscotch: Towards user interface composition, July 2008. ECOOP 2008 International Workshop on Advanced Software Development Tools and Techniques (WASDeTT).
- [20] CLARKE, D., DROSSOPOULOU, S., NOBLE, J., AND WRIGSTAD, T. Tribe: a simple virtual class calculus. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development* (New York, NY, USA, 2007), ACM, pp. 121–134.
- [21] CORWIN, J., BACON, D. F., GROVE, D., AND MURTHY, C. MJ: a rational module system for java and its applications. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2003), ACM, pp. 241–254.
- [22] DREYER, D., CRARY, K., AND HARPER, R. A type system for higher-order modules. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), ACM, pp. 236–249.
- [23] DREYER, D., AND ROSSBERG, A. Mixin' up the ML module system. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming* (2008).
- [24] DUGGAN, D., AND TECHAUBOL, C.-C. Modular mixin-based inheritance for application frameworks. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (2001).
- [25] ERNST, E. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.
- [26] ERNST, E. Propagating class and method combination. In *Proceedings ECOOP'99* (Lisboa, Portugal, June 1999), LNCS 1628, Springer-Verlag, pp. 67–91.
- [27] ERNST, E. Higher-order hierarchies. In *Proceedings ECOOP 2003* (Heidelberg, Germany, July 2003), L. Cardelli, Ed., LNCS 2743, Springer-Verlag, pp. 303–329.
- [28] ERNST, E., OSTERMANN, K., AND COOK, W. R. A virtual class calculus. In *Proc. of the ACM Symp. on Principles of Programming Languages* (2006).
- [29] FINDLER, R. B., AND FLATT, M. Modular object-oriented programming with units and mixins. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming* (1998), pp. 94–104.
- [30] FLATT, M., AND FELLEISEN, M. Units: Cool modules for HOT languages. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1998), pp. 236–248.
- [31] FORD, B. Packrat parsing: Simple, powerful, lazy, linear time. In *Proc. of the ACM SIGPLAN International Confer-*

- ence on *Functional Programming* (September 2002), pp. 36–47.
- [32] FORD, B. Parsing expression grammars: A recognition-based syntactic foundation. In *Proc. of the ACM Symp. on Principles of Programming Languages* (January 2004), pp. 111–122.
- [33] GASIUNAS, V., MEZINI, M., AND OSTERMANN, K. Dependent classes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOP-SLA'07)* (2007), ACM Press.
- [34] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.
- [35] GONG, L., ELLISON, G., AND DAGEFORDE, M. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation (2nd Edition)*. Addison-Wesley, Reading, Massachusetts, 2003.
- [36] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Massachusetts, 2005.
- [37] HUTTON, G., AND MEIJER, E. Monadic parser combinators. *Journal of Functional Programming* 8 (July 1998), 437–444.
- [38] KNUDSEN, J. L., LOFGREN, M., MADSEN, O. L., AND (EDITORS), B. M. *Object-oriented environments: The Mjolner approach*. Prentice-Hall, 1994.
- [39] LEIJEN, D., AND MEIJER, E. Parsec: Direct style monadic parser combinators for the real world. Tech. Rep. UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.
- [40] LEROY, X. A proposal for recursive modules in Objective Caml. http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf.
- [41] LIANG, S., AND BRACHA, G. Dynamic class loading in the Java virtual machine. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1998).
- [42] LIU, Y. D., AND SMITH, S. Interaction-based programming with classages. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (2005).
- [43] LIU, Y. D., AND SMITH, S. A formal framework for component deployment. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (2006).
- [44] MACQUEEN, D. Modules for Standard ML. In *Proc. of the ACM Conf. on Lisp and Functional Programming* (Aug. 1984), pp. 198–207.
- [45] MADSEN, O. L. Semantic analysis of virtual classes and nested classes. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1999), ACM, pp. 114–131.
- [46] MADSEN, O. L., AND MØLLER-PEDERSEN, B. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices* (Oct. 1989), pp. 397–406. Published as Proceedings OOP-SLA '89, ACM SIGPLAN Notices, volume 24, number 10.
- [47] MADSEN, O. L., MØLLER-PEDERSEN, B., AND NYGAARD, K. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.
- [48] MCDIRMIID, S., FLATT, M., AND HSIEH, W. Jiazzi: New age components for old fashioned java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (2001).
- [49] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
- [50] MIRANDA, E. Newspeak FFI internal documentation. Available at <http://wiki.squeak.org/squeak/uploads/6100/Alien%20FFI.pdf>.
- [51] MOSTINCKX, S., VAN CUTSEM, T., TIMBERMONT, S., AND TANTER, E. Mirages: behavioral intercession in a mirror-based architecture. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages* (New York, NY, USA, 2007), ACM, pp. 89–100.
- [52] NAKATA, K., AND GARRIGUE, J. Recursive modules for programming. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2006), ACM, pp. 74–86.
- [53] NYSTROM, N., QI, X., AND MYERS, A. C. J&: Software composition with nested intersection. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (2006).
- [54] NYSTROM, N. J. *Programming Languages for Scalable Software Extension and Composition*. PhD thesis, Dept. of Computer Science, Cornell University, 2007.
- [55] ODERSKY, M., SPOON, L., AND VENNERS, B. *Programming in Scala*. Artima Press, Mountain View, California, 2008.
- [56] ODERSKY, M., AND ZENGER, M. Scalable component abstractions. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (2005), pp. 41–57.
- [57] OSSHER, H., AND HARRISON, W. Combination of inheritance hierarchies. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices* (Oct. 1992), pp. 25–40. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.
- [58] RAMSEY, N., FISHER, K., AND GOVEREAU, P. An expressive language of signatures. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2005), ACM, pp. 27–40.
- [59] RINAT, R., AND SMITH, S. Modular internet programming with cells. In *European Conference on Object-Oriented Programming* (2002).
- [60] RUSSO, C. V. Recursive structures for standard ML. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2001), ACM, pp. 50–61.

- [61] SCHMIDT, R. W. Metabeta: Model and implementation. Tech. rep., Department of Computer Science, Aarhus University, April 1996.
- [62] SMARAGDAKIS, Y., AND BATORY, D. Mixin layers: A object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodologies* (2002).
- [63] SMITH, M., AND DROSSOPOULOU, S. Inner Classes visit Aliasing. In *ECOOP Workshop on Formal Techniques for Java Programs (FTJJP 2003)* (2003).
- [64] SMITH, W. R. *NewtonScript: Prototypes on the Palm*. pp. 109 – 139. In *Prototype-Based Programming: Concepts, Languages and Applications*, Noble, Taivalsaari and Moore, editors, Springer-Verlag 1999.
- [65] STIEGLER, M. E in a walnut, 2000. available at <http://www.skyhunter.com/marcs/ewalnut.html>, or from <http://www.erights.org/>.
- [66] SZYPERSKI, C. A. *Component Software: Beyond Object-Oriented Programming, 2nd Edition*. Addison-Wesley, Reading, Massachusetts, 2002.
- [67] THORUP, K. K. In *European Conference on Object-Oriented Programming* (1997), pp. 444–471.
- [68] THORUP, K. K., AND TORGENSEN, M. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proceedings ECOOP'99* (Lisbon, Portugal, June 1999), R. Guerraoui, Ed., LCNS 1628, Springer-Verlag, pp. 186–204.
- [69] UNGAR, D. Annotating objects for transport to other worlds. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 1995).
- [70] UNGAR, D., AND SMITH, R. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 1987).
- [71] VAN CUTSEM, T. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, 2008.
- [72] WADLER, P. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture* (New York, NY, USA, 1985), Springer-Verlag New York, Inc., pp. 113–128.
- [73] ZENGER, M. Type-safe prototype-based component evolution. In *European Conference on Object-Oriented Programming* (2002).

Appendix A: Newspeak Syntax Guide

Newspeak expressions include, in order of decreasing precedence:

Primary expressions. These are literals and identifiers and parenthesized expressions. An identifier `id` always denotes a method invocation, i.e., `id()` in mainstream notation. The parentheses are not needed, because there is no need to distinguish between variables and methods.

Unary expressions. These are of the form `x foo`, where `x` is either a primary or unary expression. A unary expression `x foo` is equivalent to the conventional `x.foo()`. The dot between the receiver and the method name is unnecessary, as method invocation is the sole operation.

Binary expressions. Expressions of the form `a op b`, where `op` is an operator symbol such as `+`, `|`, `<<` etc. and `a` and `b` are primary, unary or binary expressions. Example: `6 + 3 factorial`, which evaluates to 12. It is equivalent to the conventional `6 + 3.factorial()` or, more accurately `6.plus(3.factorial())`. An operator is simply a method with a single argument, whose name is non-alphanumeric. All operators have the same precedence: `6 + 6 * 6` is 72, not 42.

Keyword expressions. In general, the header of an N -ary method, $N > 0$, is declared `id1: p1 id2: p2 ... idN: pN`, where `id1:id2:...idN` is the name of the method and $p_i, 1 \leq i < N$ are the formal parameters; each id_i is a *keyword*. An invocation of such a method is written `id1: e1 id2: e2 ... idN: eN`, where the e_i are primary, unary or binary expressions denoting the actual arguments. The order of the keywords is significant and cannot be changed.

Examples:

Windows version: `7 + 0i`, which would normally be written as `Windows(7.plus(0.i()))`.

add: 1 to: Cobol, which in standard syntax would be written `add(1, Cobol())`.

Comments appear between double quotes.