

Modules as Objects in Newspeak

Gilad Bracha¹, Peter von der Ahé², Vassili Bykov³, Yaron Kashai⁴, William Maddox⁵, and Eliot Miranda⁶

¹ Ministry of Truth gilad@bracha.org

² peter@ahe.dk

³ smalltalkbigot@gmail.com

⁴ Cadence Design Systems yaron@cadence.com

⁵ Adobe Systems wmaddox@adobe.com

⁶ Teleplace eliot@teleplace.com

Abstract. We describe support for modularity in Newspeak, a programming language descended from Smalltalk [33] and Self [68]. 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 [44]. 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. Modularity in Newspeak is based exclusively on class nesting. There are no separate modularity constructs such as packages. 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.

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 [41] have engendered a large body of research aimed at relaxing their limitations ([58, 23, 37, 56, 50, 24]).

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.⁷

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].

As in Self, there is 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 [43]. 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 has proven to be elusive. A major contribution of this paper is to show how modularity can be achieved based *exclusively* on class nesting, without recourse to additional constructs such as packages, components, fragments etc.

Because there is no global namespace, a top level class declaration 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. Furthermore, module definitions can be compiled and loaded in any order.

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 [47].

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 [55, 28].

⁷ 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.

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.

To the best of our knowledge, *no other programming language supports all of these features.*

Specific technical contributions are a modularity mechanism that relies exclusively on class nesting, without recourse to additional constructs; a semantics that reduces the risk of inadvertent name capture via inheritance; 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 core 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`, shown in figure 1.

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. Comments appear between double quotes, and strings between single quotes.

The first pair of parentheses in the class declaration delimits the *instance initializer* where slots are declared and initialized.

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 ...`

```

class Point x: i y: j = ( " This section is the instance initializer "
| "declare slots"
public x ::= i. " ::= denotes slot initialization"
public y ::= j.
| )
(
public printString = (
^ 'x = ', x printString, ' y = ', y printString
)
)
)

```

Fig. 1. Cartesian Points

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 ... 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** (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). 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 chose to modify **Point** so that it uses polar coordinates, no modification to the **printString** method would be needed, as long as we preserved the interface of **Point** by providing methods x and y that compute the cartesian coordinates.

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 formal parameters of the primary factory are in scope in the instance initializer. In figure 1, the slot declarations include an *initialization clause* of the form ::= e where e is an arbitrary expression. For example x is initialized to the value of the formal parameter i using the syntax $x ::= i$.

The declaration of the primary factory automatically generates a corresponding method on the class object. When invoked, this method will allocate a fresh instance o, ensure that the instance initializer of the class is executed with **self** = o, and return the initialized instance o. To create a fully initialized instance of **Point** write, e.g.: **Point x: 42 y: 91**.

The factory method is somewhat similar to a traditional constructor. However, it has a significant advantage: its usage is indistinguishable from an ordinary method invocation. This allows us to substitute factory objects for classes (or one class for another) without modifying instance creation code. Instance creation is always performed via a late bound procedural interface. This eliminates the primary motivation for dependency injection frameworks [3].

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

2.1 Nested Classes

Newspeak class declarations 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.

An example of class nesting is `ShapeLibrary`, a class library for manipulating geometric shapes, shown in figure 2. `ShapeLibrary` has a number of nested classes within it. Several of these are outlined in the figure. We elide the details of some class declarations, replacing them with ellipses.

```
class ShapeLibrary usingPlatform: platform = (  
  | "We use = to define immutable slots".  
  private List = platform Collections List.  
  private Error = platform Exceptions Error.  
  private Point = platform Graphics Point.  
  |  
)  
(  
  public class Shape = (...)(...)  
  public class Circle = Shape (...)(...)  
  public class Rectangle = Shape (...)(...)  
)
```

Fig. 2. Outline of a shape library

Shapes are organized in a class hierarchy rooted in class `Shape`. The details of this code are unimportant to us here, except for one point: the classes refer to each other. For example both `Circle` and `Rectangle` inherit from `Shape` (the name of the superclass follows the equal sign in the class declaration). Recall, however, that a name such as `Shape` 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 `ShapeLibrary` by the declaration of the nested class `Shape`. The method is available to all code nested within `ShapeLibrary`. This is how an enclosing class provides a namespace for its nested classes.

Imports Code within a module must often make use of code defined by other modules. For example, `ShapeLibrary` requires utility classes such as `List`, defined by the standard collections library. In the absence of a global namespace, there is no way to refer to a class such as `List` directly. Instead, we have defined a slot named `List` inside `ShapeLibrary`.

The slot declarations used in figure 2 differ slightly from our earlier examples. Here, slots initialization uses `=` rather than `::=`. The use of `=` signifies that these are *immutable slots*, that will not be changed after they are initialized. No setter methods are generated for immutable slots, thus enforcing immutability.

When `ShapeLibrary` is instantiated, it expects an object representing the underlying platform as an argument to its factory method `usingPlatform:`. This object will be the value of the factory method's formal parameter `platform`. During the initialization of the module, the slot `List` will be initialized via the expression `platform Collections List`. 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 `List`, 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 `List` fills the role of an import statement, as do those of `Error` and `Point`. 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 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.

Modularity 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 `List` 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.

In all of the above scenarios, one can switch between libraries dynamically, store libraries in data structures, pass them as parameters etc., because libraries are represented as first class objects.

2.2 Class Hierarchy Inheritance

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

```
class ExtendShapes withShapes: shapes = (  
  | ShapeLibrary = shapes. |  
)(  
  public class ColorShapeLibrary usingPlatform: platform =  
    ShapeLibrary usingPlatform: platform (  
  )(  
    public class Shape = super Shape ( | color | )(...)  
  )  
)
```

Fig. 3. Subclassing a Hierarchy

Figure 3 shows the class `ExtendShapes`. The factory method for this class, `withShapes:`, takes a single argument, `shapes` which should be a shape library class such as `ShapeLibrary`. The instance initializer imports this library under the name `ShapeLibrary`.

Nested within `ExtendShapes` is class `ColorShapeLibrary`, which inherits from `ShapeLibrary`. This shows that it is possible to subclass an imported class. The name of the superclass, `ShapeLibrary`, is followed by the message `usingPlatform: 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.

`ColorShapeLibrary` overrides `Shape`. It defines its own class `Shape` that subclasses the superclass' class of the same name. The overriding class `Shape` has added a slot, `color`. Since `Shape` is the superclass of all other classes in `ShapeLibrary`, they all inherit the new slot. Let us see how this occurs.

When the class declaration for `Shape` is overridden in `ColorShapeLibrary`, the automatically generated accessor method is overridden as well. Hence every attempt to access `Shape` on an instance of `ColorShapeLibrary` 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 `Circle`. This class is declared in `ShapeLibrary` and inherited unchanged in `ColorShapeLibrary`. It declares its superclass to be `Shape`. The first attempt to use `Circle` on an instance of `ColorShapeLibrary` 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 `Shape`, as intended. The same holds for all other subclasses of `Shape` in the library, as one would expect.

2.3 Application Assembly and Deployment

In the absence of a global namespace, it may not be obvious how one combines separately developed top level classes into a single application. Here we discuss some options; other variations are possible.

An application is typically constructed by instantiating a top level class T representing the application as a whole. T will likely depend on a number of separately compiled module definitions; its factory method should take these, and only these, as arguments. The Newspeak IDE provides us with a namespace containing all classes used in development. It in this namespace that we will instantiate T . Use of the IDE namespace is analogous to how tools like `make` reference the components of an application utilizing the file system as a namespace.

Class T should have a method `main:args:` as its entry point. The method takes an object representing the underlying platform, and an array of command line arguments. The code in `main:args:` will instantiate the various module definitions imported by T , linking them together as required and then start up the application.

One deployment option is to use serialized objects as our binary format. We can serialize an instance of T , and later run it via a tool that deserializes it and invokes its `main:args:` method. This is similar to the classic C convention of invoking an application via a distinguished function `main()`. T is analogous to the C program, its `main:args:` method is analogous to the `main()` function, and the serialized instance is analogous to a binary file. Deserialization is the equivalent of linking and loading.

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 on the core of the semantics: method lookup and its interaction with class nesting. As illustrated in [42] such interaction can be subtle, especially in the presence of virtual classes. This section is targeted at readers with a keen interest in these subtleties; the rest of the paper can be understood independently.

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*, a description of a class that is abstract with respect to any superclass. There can be many classes that *correspond* 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, but we will not explore this feature further here.

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. When there is no risk of confusion, we may continue to use the term *class* rather than the more verbose *class declaration*, e.g., when we speak of a method being declared in a class.

3.2 Nested Classes and Enclosing Objects

Nested Classes are Per Instance 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 class `ColorShapeLibrary` in figure 3. The superclass of `ColorShapeLibrary` is imported by the surrounding module definition. Each instance of the enclosing class may have distinct 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.

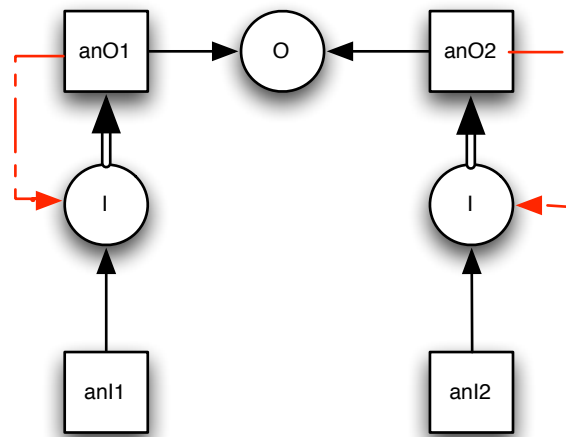
Enclosing Objects and Lexical Scope The relationship between an instance *o* of an enclosing class and its nested classes is bidirectional: each such nested class has *o* as its *enclosing object*.

The relationship between enclosing objects and nested classes is illustrated in Figure 4, which depicts a top level class `O` whose declaration includes a nested class declaration `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 class' enclosing object is the dynamic representation of the lexical scope immediately enclosing the class' declaration. Since lexical scope plays an important role in Newspeak method lookup, there is a close connection between enclosing objects and message sends.

3.3 Method Lookup

In most object-oriented programming languages, if the receiver of a message is **self** 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 [42, 61]. 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



Legend

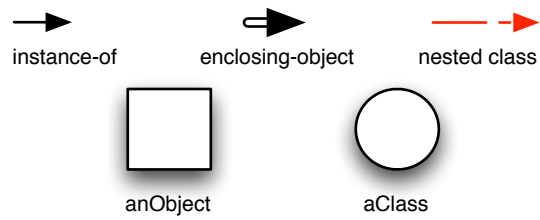


Fig. 4. Enclosing classes, nested classes and their instances

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 [35] and formalized in [61]. It is called “comb semantics” in NewtonScript [62].⁸

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 5 for an illustration.

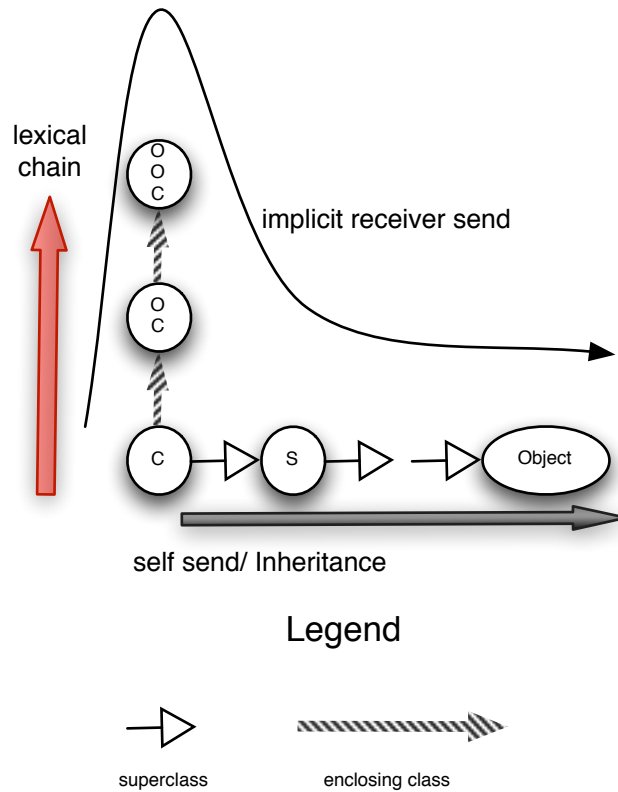


Fig. 5. Method lookup

⁸ NewtonScript has no lexical nesting, but its lookup semantics are nevertheless essentially the same.

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();}
  }
}
```

The expectation is that `new Outer.Inner().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;} }
```

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.

We now examine the semantics of method lookup in more detail. 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: methods matching the send 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.

Implicit Receiver Sends An implicit receiver send is equivalent to an ordinary send to the implicit receiver. The question is how to determine the receiver when it is implicit. Figure 6 gives pseudo-code for computing the implicit receiver.

```
0 function implicitReceiver(s, r, d, m) {
1   if declares(d, m) return r;
2   if (enclosingDeclaration(d) = nil) return s;
3   var cls := class(r); // 3
4   while (declaration(cls) ≠ d) cls := superClass(cls);
5   return implicitReceiver(s, enclosingObject(cls), enclosingDeclaration(d), m);
6 }
```

Fig. 6. Determining implicit receivers

The code above makes use of six auxiliary functions: *declares*(*d*, *m*) is a predicate that evaluates to true if the declaration *d* includes a declaration of a member named *m*; *enclosingDeclaration*(*d*) returns the declaration lexically enclosing the declaration *d*, or **nil** if *d* is a top level class declaration; *class*(*o*) is the class of the object *o*; *declaration*(*c*) is the class declaration corresponding to class *c*; *superClass*(*c*) is the superclass of *c*; and finally, *enclosingObject*(*c*) is the enclosing object of class *c*.

The function *implicitReceiver* takes four arguments: the receiver of the current method, *s* (i.e., **self**); a candidate receiver object *r*; a candidate class declaration *d*; and a message name *m*. The search for an implicit receiver begins by invoking *implicitReceiver* with **self** as both the current and the candidate receiver, the class declaration immediately enclosing the call site as the candidate declaration and the name of the message being sent as the final argument.

The function requires the invariant that the class of *r*, or some superclass, corresponds to *d*. The invariant ensures that if *d* declares a member named *m*, *r* can respond to the message; *r*'s class chain includes a class *C* corresponding to *d* and *d* includes a matching member. This invariant holds for the initial arguments, and is preserved when the function recurses.

Line 1 of figure 6 tests if the candidate declaration declares a member named *m*. If so, the candidate receiver is the desired result and we are done; if not, we need a new candidate declaration, representing the next lexical level. The new candidate declaration is the enclosing declaration of *d*. If the enclosing declaration is **nil**, *d* must represent a top level declaration, meaning that no matching member has been found in the lexical scope. The desired member can only be inherited, and the implicit receiver must be **self** (line 2).

Otherwise we must determine a new candidate receiver. On lines 3 and 4, we scan up the superclass chain of *r*, looking for a class *cls* corresponding to the declaration *d*. The next candidate receiver will be the enclosing object of *cls*.

The scan is necessary because the class of *r* need not correspond to the declaration *d*; the send may be in inherited code, in which case the enclosing declaration is that of a superclass of *r*. On line 5, we pass the new candidates to a recursive call of *implicitReceiver*, with *s* and *m* unchanged.

4 Discussion

4.1 Types

Newspeak is a dynamically typed language. We favor the development of optional/pluggable type systems [15] for Newspeak, but have not pursued the topic.

The fact that we are not bound by a type discipline has given us the freedom to experiment with idioms that are challenging to typecheck. An example is cross-family inheritance. We routinely define virtual classes that inherit from classes originating in other modules, an essential practice given the mechanisms we use to "import" classes between modules. Cross-family inheritance is severely restricted in systems such as [32, 20].

4.2 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 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. 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.

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.

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.

4.3 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 declaration, how do we decide when a class declaration 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, its declaration should likely 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. We are still learning about these issues, and further experimentation is needed.

4.4 Mutual Recursion

Newspeak makes it possible to create mutually recursive modules. However, this is not yet 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 [54].

We plan to introduce better support for mutual recursion among modules using *simultaneous slot declarations*, a generalization of `letrec`. Interested readers are referred to the Newspeak language specification [16] for details.

4.5 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.

4.6 Access Control

The Newspeak specification [16] defines object-level access modifiers. Object members that are **private** or **protected** can only be referenced within the scope of the object. Enforcing such access control complicates the semantics only slightly. However, this feature has not yet been implemented, so we refrain from discussing it in detail in this paper.

4.7 Relation to the Object-Capability Security Model

Newspeak's security model is founded on the object-capability model [47]. 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*.

Point (1) is sometimes addressed using closures (e.g., in Javascript). We find this awkward, and intend to support access control on object members as discussed in section 4.6 above.

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 [34]. 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 [38]. 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,⁹ 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* [48, 16]. Unless code has access to an alien, it cannot call foreign code.

⁹ Caveat: the current implementation uses Squeak classes, and so does not meet this requirement.

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

An implementation of Newspeak is available 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, where Newspeak was originally developed.

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, especially with respect to modularity.

6 Related Work

Newspeak is a direct descendant of Smalltalk [33]. 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. Most fundamentally, Smalltalk distinguishes between method invocation and variable access — it is not a purely 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, though there have been research efforts such as [12].

The idea of message-based programming comes to us from Self [68]. 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 [67]. 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 [44]. 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 ([36], 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 [59]). 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 [26] is similar to Beta, but extends patterns to mixins. This allows gbeta to support class hierarchy inheritance [28]. 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 [27]. In contrast, in Newspeak, class overrides behave like classical method overrides.

The closest relatives of Newspeak modules are Units ([30], [31]) 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 class declarations are mixins, since their superclasses are always virtual. In contrast to Newspeak, Units and classes are distinct. Units do not support inheritance.

A related area of research are languages that support components [64, 69, 6, 39, 40, 57, 45]. Newspeak modules are not strictly components because of their use of inheritance. Component research often emphasizes static checking of component compatibility and component life cycle issues. These topics are beyond the scope of this paper (see sections 4.1 and 4.5).

AmbientTalk [22] is a language designed to support the programming of mobile systems. It is message based (emphasizing asynchrony) and uses mirrors extensively [49].

No discussion of related work on modularity would be complete without reference to the ML module system [41]. 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 [58, 23, 37, 56, 50]. In particular MixML [24] 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, 25]) are closely related to the ideas of Jigsaw [14]. Mixin modules compose via inheritance-like operators and naturally support mutual recursion.

CaesarJ [32, 9], and J& [52, 51] 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 class declarations 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 [29], 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 4.2 for more detail.

Scala [53] supports virtual types but does not support virtual classes (though there is a plan to add such support). Virtual types [66] 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 usually 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 ([35], 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 [65] 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 [60] 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 [47]. The language E [63] is based on that model. Newspeak supports reflection, whereas E does not. The Joe-E language [46] is a capability enabled language inspired by E but based on Java, that supports a subset of Java's introspection facilities. To our knowledge, Newspeak is the first system to combine the object-capability model with full reflectivity.

7 Future Work

There are several areas, especially access control and mutual recursion, where the language does not fully conform to its specification. We also have plans for supporting actor based concurrency, value types, and perhaps a purely functional language subset.

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 4.7).

Extending Newspeak module definitions to support a range of module combinators like those described in [14] 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 [13]. In that context, we would hope to address issues such as module distribution and update.

8 Conclusions

The Newspeak programming language supports a number of desirable features:

- Module definitions are first class objects. They may be stored, passed as arguments, returned from computations etc.
- Class hierarchy inheritance: entire module definitions, including complete class library hierarchies may be extended and modified via inheritance.
- All external dependencies of a module definition are explicit.
- Module definitions are parametric with respect to their dependencies, and can be compiled, loaded and 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.
- Mixins. Module definitions and the classes within them can be mixed 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 based on an exceedingly simple foundation. We have created a practical modular language with no global namespace, using pervasive late binding, with classes as the only modularity construct. These ideas can clearly be applied to other languages and contexts. We look forward to future work in areas such as typechecking and security to fully realize the potential of this research.

Acknowledgements

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 Felix Geller, John Hedditch, 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=dd2fhx4z_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., Tobin-Hochstadt, S.: The Fortress language specification (2005), available from <http://research.sun.com/projects/plrg/>

7. Ancona, D., Fagorzi, S., Moggi, E., Zucca, E.: Mixin modules and computational effects. In: Goos, G., Hartmanis, J., van Leeuwen, J. (eds.) ICALP 2003 - Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 2719, pp. 224–238. Springer Verlag (2003)
8. Ancona, D., Zucca, E.: A calculus of module systems. *Journ. of Functional Programming* 12(2), 91–132 (2002), <http://www.disi.unige.it/person/AnconaD/Software/Java/CMS.html>
9. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of CaesarJ. *Transactions on AOSD I, LNCS 3880*, 135 – 173 (2006), <http://www.daimi.au.dk/ko/papers/overview-of-caesarj-2005.pdf>
10. Awad, I.: Module identification and loading for Javascript, available at <http://docs.google.com/Doc?id=dfgxb7gk61d47876fr>
11. Baldwin, C.Y., Clark, K.B.: *Design Rules: The Power of Modularity*. MIT Press (2000)
12. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: controlling visibility of class extensions. *Computer Languages, Systems & Structures* 31 (October 2005)
13. 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>
14. Bracha, G.: *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, University of Utah (1992)
15. 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>
16. Bracha, G.: The Newspeak programming language specification, version 0.05 (2009), available at <http://bracha.org/newspeak-spec.pdf>
17. Bracha, G., 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., 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 (Jul 2008), ECOOP 2008 International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)
20. Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: a simple virtual class calculus. In: *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. pp. 121–134. ACM, New York, NY, USA (2007)
21. Corwin, J., Bacon, D.F., Grove, D., 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 programing, systems, languages, and applications*. pp. 241–254. ACM, New York, NY, USA (2003)
22. van Cutsem, T.: *Ambient References: Object Designation in Mobile Ad Hoc Networks*. Ph.D. thesis, Vrije Universiteit Brussel (2008)
23. Dreyer, D., Crary, K., 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*. pp. 236–249. ACM, New York, NY, USA (2003)
24. Dreyer, D., Rossberg, A.: Mixin' up the ML module system. In: *Proc. of the ACM SIGPLAN International Conference on Functional Programming* (2008)

25. Duggan, D., 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)
26. Ernst, E.: gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. Ph.D. thesis, Department of Computer Science, University of Aarhus, Århus, Denmark (1999)
27. Ernst, E.: Propagating class and method combination. In: Proceedings ECOOP'99. pp. 67–91. LNCS 1628, Springer-Verlag, Lisboa, Portugal (Jun 1999)
28. Ernst, E.: Higher-order hierarchies. In: Cardelli, L. (ed.) Proceedings ECOOP 2003. pp. 303–329. LNCS 2743, Springer-Verlag, Heidelberg, Germany (Jul 2003)
29. Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: Proc. of the ACM Symp. on Principles of Programming Languages (2006)
30. Findler, R.B., Flatt, M.: Modular object-oriented programming with units and mixins. In: Proc. of the ACM SIGPLAN International Conference on Functional Programming. pp. 94–104 (1998)
31. Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 236–248 (1998)
32. Gasiunas, V., Mezini, M., Ostermann, K.: Dependent classes. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07). ACM Press (2007), <http://www.daimi.au.dk/~ko/papers/depcls-oopsla07.pdf>
33. Goldberg, A., Robson, D.: Smalltalk-80: the Language and Its Implementation. Addison-Wesley (1983)
34. Gong, L., Ellison, G., Dageforde, M.: Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation (2nd Edition). Addison-Wesley, Reading, Massachusetts (2003)
35. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Third Edition. Addison-Wesley, Reading, Massachusetts (2005)
36. Knudsen, J.L., Lofgren, M., Madsen, O.L., Magnusson, B.: Object-oriented environments: The Mjolner approach. Prentice-Hall (1994)
37. Leroy, X.: A proposal for recursive modules in Objective Caml, http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf
38. Liang, S., 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)
39. Liu, Y.D., Smith, S.: Interaction-based programming with classages. In: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (2005)
40. Liu, Y.D., Smith, S.: A formal framework for component deployment. In: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (2006)
41. MacQueen, D.: Modules for Standard ML. In: Proc. of the ACM Conf. on Lisp and Functional Programming. pp. 198–207 (Aug 1984)
42. 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. pp. 114–131. ACM, New York, NY, USA (1999)
43. Madsen, O.L., Møller-Pedersen, B.: Virtual classes: A powerful mechanism in object-oriented programming. In: Proceedings OOPSLA '89, ACM SIGPLAN No-

- tices. pp. 397–406 (Oct 1989), published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10
44. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the Beta Programming Language. Addison-Wesley (1993)
 45. McDirmid, S., Flatt, M., 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)
 46. Mettler, A., Wagner, D.: The Joe-E language specification, version 1.0. Tech. Rep. UCB/EECS-2008-91, University of California at Berkeley (2008), available at <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-91.pdf>
 47. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006)
 48. Miranda, E.: Newspeak FFI internal documentation, available at <http://wiki.squeak.org/squeak/uploads/6100/Alien%20FFI.pdf>
 49. Mostinckx, S., Van Cutsem, T., Timbermont, S., Tanter, E.: Mirages: behavioral intercession in a mirror-based architecture. In: DLS '07: Proceedings of the 2007 symposium on Dynamic languages. pp. 89–100. ACM, New York, NY, USA (2007)
 50. Nakata, K., Garrigue, J.: Recursive modules for programming. In: ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming. pp. 74–86. ACM, New York, NY, USA (2006)
 51. Nystrom, N., Qi, X., Myers, A.C.: J&: Software composition with nested intersection. In: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (2006)
 52. Nystrom, N.J.: Programming Languages for Scalable Software Extension and Composition. Ph.D. thesis, Dept. of Computer Science, Cornell University (2007)
 53. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press, Mountain View, California (2008)
 54. Odersky, M., Zenger, M.: Scalable component abstractions. In: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications. pp. 41–57 (2005)
 55. Ossher, H., Harrison, W.: Combination of inheritance hierarchies. In: Proceedings OOPSLA '92, ACM SIGPLAN Notices. pp. 25–40 (Oct 1992), published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10
 56. Ramsey, N., Fisher, K., Govereau, P.: An expressive language of signatures. In: ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming. pp. 27–40. ACM, New York, NY, USA (2005)
 57. Rinat, R., Smith, S.: Modular internet programming with cells. In: European Conference on Object-Oriented Programming (2002)
 58. Russo, C.V.: Recursive structures for standard ML. In: ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming. pp. 50–61. ACM, New York, NY, USA (2001)
 59. Schmidt, R.W.: Metabeta: Model and implementation. Tech. rep., Department of Computer Science, Aarhus University (April 1996)
 60. Smaragdakis, Y., Batory, D.: Mixin layers: A object-oriented implementation technique for refinements and collaboration-based designs. ACM Transactions on Software Engineering and Methodologies (2002)
 61. Smith, M., Drossopoulou, S.: Inner Classes visit Aliasing. In: ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2003) (2003), <http://www.cs.kun.nl/erikpoll/ftfjp/2003.html>

62. Smith, W.R.: NewtonScript: Prototypes on the Palm, pp. 109 – 139. Springer-Verlag (1999), in *Prototype-Based Programming: Concepts, Languages and Applications*, Noble, Taivalsaari and Moore, editors
63. Stiegler, M.: E in a walnut (2000), available at <http://www.skyhunter.com/marcs/ewalnut.html>, or from <http://www.erights.org/>
64. Szyperski, C.A.: *Component Software: Beyond Object-Oriented Programming*, 2nd Edition. Addison-Wesley, Reading, Massachusetts (2002)
65. Thorup, K.K.: Genericity in Java with virtual types. In: *European Conference on Object-Oriented Programming*. pp. 444–471 (1997)
66. Thorup, K.K., Torgersen, M.: Unifying genericity: Combining the benefits of virtual types and parameterized classes. In: Guerraoui, R. (ed.) *Proceedings ECOOP'99*. pp. 186–204. LCNS 1628, Springer-Verlag, Lisbon, Portugal (Jun 1999)
67. 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)
68. Ungar, D., Smith, R.: SELF: The power of simplicity. In: *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct 1987)
69. Zenger, M.: Type-safe prototype-based component evolution. In: *European Conference on Object-Oriented Programming* (2002)