Modules in LOOM: Classes are not enough [*†]

Kim B. Bruce, Leaf Petersen[‡] and Joseph Vanderwaart
Williams College
Williamstown, MA, USA

April 13, 1998

Abstract

Pure object-oriented languages have tended to assume that classes can take the place of modules in programming in the large. We argue that modules are still needed in these languages and present the design of a module construct for our language $\mathcal{LOOM}$ which works well with the object-oriented paradigm. We discuss and illustrate the advantages of these modules over a class-only approach. In particular we illustrate the advantages of modules in supporting better control over information hiding, including the support of access like C++'s friends.
Research paper: language design

1 Introduction

As programmers and language designers have gotten more experience with object-oriented languages, it has become clearer that class boundaries are not always the correct abstraction layer for large programs. Interestingly it is the hybrid languages which have grown from ADT-style languages which have often provided the best support for modularity. Thus Ada 95 [Int95] and Modula-3 [CDG+88] have introduced module structures which are distinct from classes. Pure object-oriented languages like Smalltalk and Eiffel have generally chosen to identify classes with modules, though newer languages like Theta [DGLM94] (which can be seen as a descendant of the ADT-style language, Clu) and Java [AG96] have also chosen to provide modules (called packages in Java).

Modules provide three important functions in programming languages:

1. They provide a way of organizing code into manageable pieces and help with name-space management by only exporting names that are needed elsewhere.

2. They provide important abstraction barriers, grouping together definitions and declarations that fit logically together, and lessening the dependence of a unit on the implementation details of another unit.

3. They provide support for separate compilation, aiding the programmer in debugging large programs as well as making it possible to provide reusable code in libraries in such a way that the original source code need not be revealed.

[*]This research was partially supported by NSF grant CCR-9424123. The results in this paper are based in part on the Williams College senior honors thesis of the second author [Pet96].

[†]Corresponding author: Kim Bruce, Department of Computer Science, Williams College, Williamstown, MA 01267, USA. Phone: (413) 597-2273, FAX: (413) 597-4116, email: kim@cs.williams.edu.

[‡]Current address: School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA.

With strong language support for modules it should be possible to change the implementation of a type without changing its public interface and without requiring recompilation of other modules which import it. See [Jon96] for a more detailed discussion of modules.

Object-oriented languages like Eiffel, Smalltalk, and Java use classes as compilation boundaries as well as for name-space management, making it appear that classes can play the same role as modules. With [Szy92], however, we believe that this identification is a mistake and that classes are too fine a unit to serve usefully as module boundaries. Instead, we argue here that modules should be used to group together interrelated systems of classes (and perhaps other program units such as functions), and that modules, rather than classes should be the fundamental abstraction barrier in object-oriented languages.

The open-closed principle, as presented by Bertrand Meyer [Mey88], states that well-designed language constructs should be open to extension,but closed to modification. That is, once one has designed a construct (type, class, function, etc.), one should never have to go back and modify it. However, it should be possible to extend that construct by building a new construct which extends or overrides functionality from the original.

The open-closed principle was articulated to describe the use of classes in object-oriented languages, as it is possible to make incremental changes by defining new subclasses rather than modifying existing classes. In this paper we argue that this principle should also be applied to modules in that it should be possible to "open" a module in order to use the non-exported definitions in building constructs in a new module.

In this paper we propose a concrete model for modules in object-oriented languages, illustrating this by adding this construct to our earlier language $\mathcal{LOOM}$. We provide examples to illustrate the benefits of using this module construct rather than having classes take on the roles of modules.

In section 2 we briefly describe the features of the core $\mathcal{LOOM}$ language. The following section provides reasons why classes do not provide the same functionality required of modules. Section 4 introduces modules in $\mathcal{LOOM}$ and discusses the information hiding which occurs by providing access through module interfaces. In section 5 we argue that it should also be possible to "open" module implementations in other implementations in order to allow the programmer to extend types and classes defined earlier. Finally, we compare our work with the design of modules in other programming languages and conclude.

2 $\mathcal{LOOM}$

$\mathcal{LOOM}$ [BFP97] is a provably type-safe, statically-typed, object-oriented programming language. Interesting features of LOOM include the use of the keyword `MyType` to stand for the type of `self`, and the use of matching, a relation more general that subtyping (at least in the presence of `MyType`) that corresponds to simply adding more methods to an object type.

The $\mathcal{LOOM}$ code in Figure 1 defines object types for singly and doubly-linked nodes holding integer values. The `include` in the definition of `DbleNode` (which is based on a similar construct in Rapide [KLM94a]), indicates that the new methods should be added to the methods in the other object type.

When a message is sent to an object, all occurrences of `MyType` in the signature of the message are replaced by the static type of the receiver. Thus if the message `setNext` is sent to a `DbleNode`, it will need to be provided with a parameter of type `DbleNode`, whereas the same message sent to a `Node` requires a parameter of type `Node`.

While this flexibility is extremely helpful (we would not want the result of sending `getNext()` to a `DbleNode` to return a `Node`), the use of `MyType` as the type of a parameter in methods causes

```
Node = ObjectType
      setValue: proc(integer);
      getValue: func():integer;
      setNext: proc(MyType);
      getNext: func():MyType;
   end;

DbleNode = ObjectType include Node
      setPrevious: proc(MyType newPrevious);
      getPrevious: func():MyType;
   end;
```

Figure 1: Object types for singly- and doubly-linked nodes.

the failure of subtyping. For example, `DbleNode` is not a subtype of `Node`, because `setNext` has a parameter of type `MyType`. As shown above, sending `setNext` to a `Node` with a parameter of type `Node` is fine, but sending `setNext` to a `DbleNode` with a parameter of type `Node` is not type safe because a `DbleNode` is required. Thus a `DbleNode` cannot masquerade as a `Node`, and hence they are not subtypes.

While the use of the `include` operator does not guarantee subtyping, it does guarantee the weaker *matching* relation. Object type `A` matches object type `B` (written `A` $<\#$ `B`) as long as every method in `B` also occurs in `A` with the same signature. Thus `DbleNode` *matches* `Node`, even though they are not subtypes.

$\mathcal{LOOM}$ does not support the notion of subtype. Thus, if an expression, `e`, has type `T`, then the value of `e` at run-time must be exactly the type `T`. $\mathcal{LOOM}$ does provide the possibility of supporting more flexible types by the provision of "hash types" of the form #`T`. If `e` has type #`T`, then at run-time, its value may be either of type `T` or of any type `T'` which matches `T`.

The $\mathcal{LOOM}$ classes in Figure 2 generate objects of type `Node` and `DbleNode`. In particular, the reader should notice that the `setNext` code for `DbleNode` would crash if executed with a parameter which is a `Node` rather than a `DbleNode`.

Most object-oriented languages have difficulty in statically type-checking sends of "binary methods", methods like `setNext` which take a parameter of type `MyType`. One problem is that the receiver of a message might be a subtype of the declared type (and hence its argument might also need to have type corresponding to the subtype). Because $\mathcal{LOOM}$ does not support subtypes, this is not a problem as long as the static type of the receiver is not a hash type, since the system knows the receiver's exact type. To avoid type errors $\mathcal{LOOM}$ will not allow a binary method to be sent to an object if the static type is only known to be a hash type.

$\mathcal{LOOM}$ also supports a form of parametric polymorphism where type variables may be constrained by forcing them to match a given type. Figure 3 contains a type `Comparable`, which contains methods for comparisons, and a parameterized class `OrderedList` which generates ordered lists. `OrderedList` takes as a parameter a type `T` which matches `Comparable`. The constraint ensures that elements of `T` can be compared. As with `Node`, `Comparable` has no subtypes because of the occurrence of `MyType` as a parameter in at least one of its methods. Thus the matching relation plays an important role in the language. Static type-checking rules ensure that programs using `MyType` and match-bounded polymorphism are type-safe.

Another important feature of $\mathcal{LOOM}$ is the separation between classes and types. As seen

```
class NodeClass(v:integer)
    var
       val=v:integer;
       next=nil:mytype;
    methods
        function getNext():mytype
        begin
          return next;
        end;
        function getVal():integer
        begin
           return val;
        end;
        procedure setNext(nxt:mytype)
        begin
           next:=nxt;
        end;
        procedure setVal(vl:integer)
        begin
           val:=vl;
        end;
end;

class DbleNodeClass(v:integer) inherit NodeClass(v) modifying setNext;
    var
       prev=nil:mytype;
    methods
       function getPrev():mytype
        begin
           return prev;
        end;
       procedure setPrev(prv:mytype)
        begin
           prev:=prv;
        end;
       procedure setNext(nxt:mytype)
        begin
           super.setNext(nxt);
           nxt.setPrev(self);
        end;
    end;
```

Figure 2: Classes for singly- and doubly-linked nodes.

```
Comparable = ObjectType
   lessThan: func(MyType): boolean;
   greaterThan: func(MyType): boolean;
   equal: func(MyType): boolean;
end;

class OrderedList(T <# Comparable)
   ...
end;
```

Figure 3: Match-bounded polymorphism in $\mathcal{LOOM}$.

above, classes do not play the role of types in $\mathcal{LOOM}$. Variables and parameters are given object types (which play a role similar to interfaces in Java). Several distinct classes may generate objects of the same type. For instance, classes representing different implementations of points may still generate objects of the same type as long as their public interface is the same.

Readers should consult [BFP97] for more details on $\mathcal{LOOM}$.

3 Why classes should not be modules

Szyperski [Szy92] (see also [Szy96]) makes compelling arguments that, in spite of the similarities between classes and modules, classes should not be seen as a replacement for modules. In the introduction we listed three important roles for modules: managing name spaces, providing abstraction barriers, and providing support for separate compilation.

The primary function of classes, on the other hand, is as extensible generators of objects. That is, classes are used to create new objects, and, using inheritance, support the definition of new classes which are incremental modifications of the original.

While these lists of capabilities of modules and classes have little or no overlap, classes do have several capabilities that they share with modules. One is that methods and instance variables in classes can be provided with restrictions on their visibility, providing some control over the name space outside the class, and providing an abstraction barrier. For example methods and instance variables labeled "public" in C++ and Java are visible to all users of the class, whereas "private" components are only available inside other methods of the class, and "protected" methods are also available in subclasses, but are not visible to clients. Moreover, in many languages classes form compilation units. Thus classes provide some support for name management, abstraction, and separate compilation.

However there are important ways in which classes and modules play completely different roles [Szy92].

1. Modules are compile-time abstractions which essentially disappear at run-time, while classes (or at least their constructors) are run-time values. Thus in order to use a feature of a class, one must first create an object from the class. Thus if a collection of mathematical functions or constants is packaged in a class, one must create an object of that class and then obtain the functions or constants by sending messages to the object, even though the methods are not in any way specialized to the object. Realizing the awkwardness of this, most object-oriented languages have provided alternative mechanisms to support these needs. For example, Java

has static methods and Smalltalk has class methods. In ADT style languages, such functions and constants would instead be placed in a module and imported. This is a much more natural way to provide access to these innately static features rather than extending class constructs in order to artificially support them.

2. Class-based languages quite naturally tend to look at design at the class level. In fact as we move up to programming in the large we discover that we are often dealing with systems of classes. In order to support these properly we must be able to define mutually recursive classes whose invariants depend on objects from several classes. In a class-centric language it is not at all clear where to put such invariants as there is no natural structure which can be used to collect information from objects of different classes.

   Moreover there are situations where a collection of mutually recursive classes is defined, and then all classes need to be specialized synchronously. For example in the subject-observer example from [GRJV94], a group of observers is assigned to each subject. Later we may wish to provide a specialized subject whose observers must be similarly specialized. It would be most natural to group these classes together in matching pairs rather than have them floating around independently. More generally we believe that programming in the large is supported better when there are non-class-based techniques available for organizing groups of classes which fit together logically either as libraries or because of their mutual references.

3. Tightly coupled classes also often have the need for access to more information about each other than is needed (or desired) for general clients. C++ provides for the specification of "friends" to obtain this access, while Eiffel provides a similar mechanism for revealing designated features to specified external classes. This can result in what Szyperski calls "spaghetti scoping". Instead, we can regain static scoping by placing classes which desire more extended access to each other's features inside the same module, erecting a more strict abstraction barrier to those outside the module boundary. A problem which we address later in this paper is the need for a new class to access hidden features of a previously defined class. Note that C++ and Eiffel would require redefinition of the original class in order to allow "friendly" access by the new class to the hidden features of the original class.

For these reasons we believe that object-oriented languages need a module construct. As noted earlier, Java, Modula-3, Ada 95, and Theta provide such a construct. In section 6 we compare our module design with that provided in these languages. In the next section we describe the module design for $\mathcal{LOOM}$.

4 Modules in $\mathcal{LOOM}$

We have chosen to follow the lead of languages like those in the Modula and Ada families and provide module interfaces which can be completely separate from the module implementation (e.g., the interface can be compiled before the implementation is written). Interfaces are important because they provide a contract between the developer and user of a module. The interface provides a list of items that are available from the module, including their signatures and (hopefully) a specification of their behavior.

A key issue with interfaces is to provide a mechanism for revealing in the interface as much or as little as desired about the details of the implementation of a type. In ADT-style languages it is typical for type definitions in interfaces to consist either of *manifest* types, in which all details of

```
Interface SetOfInt;

type

   IntSetType <# ObjectType
                   add: proc(Integer);
                   remove: proc(Integer);
                   contains: func(Integer):Boolean;
                   intersect: proc(MyType)
                 end;

const

   function newSet(): IntSetType;

end ;
```

Figure 4: Interface for sets

the type definition are revealed, or *opaque* types, in which only the name of the type is revealed. However in ADT-style languages, operations are defined externally from the type itself. Thus an opaque type will be accompanied by a collection of names of constants, functions, and procedures related to it that are to be exported (though their implementations are typically hidden).

In object-oriented languages, however, object types include the names and types of their methods. In ADT-style languages we can choose to include only some operations in the interface, leaving others hidden in the implementation module. We can emulate this in object-oriented languages by providing partial revelations of object types as is done in Modula-3 [CDJ+89]. That is, for each object type we only list those methods which we wish to publicly advertise. This can be expressed using the matching relation from $\mathcal{LOOM}$.

The example in Figure 4 is similar to one presented in [PT93]. In the interface, `SetOfInt`, we reveal that IntSetType has at least methods `add`, `remove`, `contains`, and `intersect` with the appropriate types. It would not be appropriate to replace matching by subtyping (even if it was supported in $\mathcal{LOOM}$) because the fact that `intersect` has a parameter of type `MyType` implies that the given object type has no proper subtypes!

A program or module using `IntSetType` would simply import the module `SetOfInt`:

```
import SetOfInt;
```

The import statement brings all of the definitions in the interface `SetOfInt` into the scope of that program unit, though each of the features imported from `SetOfInt` would have its name prefixed with the module name when used. Thus an importing module would refer to `SetOfInt::IntSetType`. If desired it would be easy to add mechanisms to the language that would allow the programmer to drop such prefixes except where name conflicts exist.

$\mathcal{LOOM}$ allows the programmer to label methods in classes as either `Visible` or `Hidden`, depending on whether or not we wish them to be available to clients (the default is `Visible`). `Visible` and `Hidden` in $\mathcal{LOOM}$ correspond to `public` and `protected` in Java and C++. Thus hidden methods are accessible only inside the class' other methods and its subclasses. An object's instance variables are not accessible outside the object's methods. Why then do we need the extra mechanism

```
Module implements SetOfInt;

   IntSetType = ObjectType
                   add: proc(Integer);
                   remove: proc(Integer);
                   contains: func(Integer):Boolean;
                   intersect: proc(MyType);
                   goFirst: proc();
                   getCurrent: func(): Integer;
                   goNext: proc();
                 end;
    ...
end;  -- SetOfInt implementation
```

Figure 5: Implementation for sets

of partial revelations? The reason is that some objects might need to provide extra methods which are only accessible to a limited number of other types of objects. This is exactly the idea behind C++'s friends and is similar to package default and protected visibility in Java.

Let us continue with the example above of sets of integers. Suppose our sets are implemented via a binary search tree. This implementation supports an efficient implementation of the remove, contains, and intersect methods. We normally do not wish the user to use the fact that the implementation is as a binary search tree as it may prove convenient to change the implementation later, and, if users can get at the implementation, they may make changes using operations specialized to the implementation which are inconsistent with invariants for the type (e.g., the restriction to disallow duplicate elements).

However, an efficient implementation of the intersect method depends on being able to access the elements of both the receiver and parameter in numeric order. Methods supporting this are typically supplied with binary search trees (inorder traversals), but we do not want these methods to be available to general clients. This is a serious problem in core $\mathcal{LOOM}$ as the use of `MyType` as a parameter type only allows access to the visible (public) methods of the class.

However with modules we can support this by providing in the module implementation a definition as in Figure 5. With this type, the `intersect` method can use the `goFirst`, `getCurrent`, and `goNext` methods to iterate through all of the elements of the receiver of the `intersect` method as well as the parameter. Notice, however, that because of the use of partial revelations, clients outside of the module can only see and use methods `add`, `remove`, `contains`, and `intersect` of `IntSetType`. Thus we have provided access to more of the implementation inside the module than is allowed outside. See Appendix A for the full details of the interface and implementation.

Languages like C++ and Java allow methods of a class special access to non-public methods of parameters of the same class. $\mathcal{LOOM}$ does not provide this access. While this may be seen as a handicap in examples like the above, it has the advantage that we may have different classes which implement objects of type IntSetType (e.g., those based on binary search trees versus ordered lists), and then intersect any two, even if they come from different classes.

In the interface we only reveal the methods `add`, `remove`, `contains`, and `intersect`, so clients outside of the module may not use the other methods. As a result we could later change the implementation to one that no longer supports the `goFirst`, `getCurrent`, or `goNext` methods without breaking any clients. For example, we could change the representation to hash tables without affecting the correctness of clients.

Of course, in general we will have several object types which are defined in the same module implementation, making methods accessible to each other which take advantage of details of their implementations, yet hiding these extra methods via partial revelations in the module interface. In Appendix B we provide a more substantial example of a module. This example is based on the Subject-Observer pattern. Placing several classes in the same module provides the same effect as C++'s friends.

We can also choose to fully reveal a type definition in the interface by providing a complete type definition:

```
revealedType = ObjectType ... end;
```

This technique for providing different levels of information hiding at different levels is also similar to that suggested in [PT93] and [KLM94b], where bounded existential quantifiers based on subtyping are used to perform the information hiding.

Our language now provides two mechanisms for information hiding, one based on labeling methods in classes as either hidden or visible, and the other based on partial revelations in modules. One might ask whether both are really necessary. While there are some advantages to retaining some information hiding mechanisms in classes, if we had to give up either the module-based or class-based mechanisms, we would eliminate the class-based abstraction facilities, as the module-based mechanisms provide better support for programming in the large, which is where the benefits of abstraction are most obvious. Szyperski argues for this point of view, for example, introducing a "No paranoia rule" to eliminate restrictions on visibility within modules. The language Theta [DGLM94] follows a similar rule. While we have not dropped information hiding at the class level in $\mathcal{LOOM}$, it is clear that hiding at this level is less critical than the module level.

5 Opening modules

Different classes of users require different kinds of access to information in modules. Clients will import modules in order to gain access to their functionality, while tool-builders will need access to the details of implementations in order to create more specialized and capable subclasses.

The interface `SetOfInt` in Appendix A is fairly typical of what one might expect in a module providing services for a client. It provides partial revelations of one or more object types, as well as a function which can be used to construct new instances of that type. It would not be desirable to also export the details of the class because the client might then write code depending on those details, limiting the ability to change the representation of the class over time.

However, a tool builder typically needs access to these details in order to write subclasses of a given class. For example, suppose we later wish to define a subclass of `IntSet` that keeps track of the number of elements in the set. If the implementation of the class generating `IntSet` objects, `ListSetClass`, were not hidden in the module implementation, one could simply define the subclass. However, in our design, this class is not available outside of the module implementation.

Because this is a fairly common need, we remedy this by allowing users to access the details of a module implementation by opening a module. The difference between importing a module and opening a module is that importing a module makes accessible all of the features in the interface, while opening a module makes available all of the features from both the interface and implementation. The compiler does not need access to bodies of classes from the opened module, but only the names of features and their associated types. This is equivalent to exporting all of the names defined in the implementation (as well as those types given complete definitions in the interface).

An interesting consequence of allowing one module to be opened in the implementation of another is that new types can be defined which match partially revealed types from other modules. For example, suppose interface `A` reveals that `T <# T'` for some object type `T'`. Then if the implementation of interface `B` opens `A` (*i.e.*, it opens `A`'s implementation), it may define a type `U` which matches `A::T` because it now has access to the complete definition of `T`. As in `A`, where only some of the methods of `A::T` were revealed, we may wish to reveal only some of the methods of `U`. We can express this in $\mathcal{LOOM}$ by including multiple bounds on `U` in interface `B`. Thus interface `B` might contain:

```
U <# A::T, ObjectType{m':u';...};
```

while the implementation of `B` would include the complete definition of `U`. This notation is also useful in the case where a type `U` matches multiple previously defined types. Similarly we generalize the `include` feature in type definitions so that several types can be included in a new type definition as long as there are no conflicts in method names.

If the interface declares a type to be an extension of a partially revealed type from another module, the implementation module holding the definition of that partially revealed type will need to be opened in the implementation module of the new type so that the new definition can be checked for correctness. It would not be permitted to simply "include" the partially revealed type in the new definition without complete knowledge of that type as there might otherwise be name conflicts for methods. Since classes generating objects of the new type will often be defined by inheritance from those generating objects of the partially revealed type, the programmer will generally want to "open" the containing module anyway.

While opening a module might seem to be contrary to the whole reason for introducing modules, we see this as simply another example of the open-closed principle. That is, at the time a module is defined it is closed and hence ready to be used by clients. However, later we may wish to extend its functionality in ways that require access to its implementation. Failure to allow access would require us to either go back and modify the original module, rewrite most of the code from that module, or export all of the details of the class definition to all clients.

6 Modules in other object-oriented languages

The module design in Modula-3 was very influential in our design. Partial revelations are lifted directly from Modula-3, for instance. There are, however, several differences between their design and ours. A major difference is that in Modula-3 a class definition may be spread over many files, each of which reveals different features of an object type. Thus clients can import only those aspects of the class that they need.

We omitted this feature because we believe that in most cases only two interfaces are relevant: the interface presented to clients and the interface which is of use in defining subclasses. This latter interface provides access to implementation details that clients need not and should not see. We present these two interfaces by allowing modules to either import or open other modules. We find this less confusing than having to search over a large number of modules for all of the pieces of a class definition. A slight disadvantage of our approach is that we do not provide an explicit interface for modules opening a module. Thus we do not present the same sort of explicit contract between the two modules as we do between a module and its client.

The language Rapide [KLM94a] also supports partial revelations using subtypes. Rapide makes the interesting decision to merge types and interfaces into one construct, which they also denote as an interface. Similarly they use the keyword module to designate all implementations of their interfaces. Thus a module may play the role of a class or of our module implementations. Along with partial revelations, they (like Ada) also support "private" interfaces, though it is not clear how useful such a feature is in a language in which all objects are references. One interface can "include" another in the same way that our object types can include another, but can also be used in interfaces in the same way that our "import" works. Rapide also allows selective import and renaming upon import. Rapide does not seem to support any sort of inheritance with modules (either in the form of regular inheritance of classes or our "open" for modules).

Ada [Int95] supports hierarchies of packages (modules). That is if `A` is a package, then we can define a package `A.child`. The interface of `A.child` inherits the public interface of `A` while its implementation inherits all of the private features of its ancestors. Our proposal is more flexible in that a module implementation may open another module and use its features without exporting them through the interface. Moreover a module may open many modules, rather than being limited to a single parent.

Java's packages perform many of the roles of modules. Packages, however, consist of collections of separately compilable classes. Classes may be added to an existing package, though restrictions in many implementations require the compiled code for all classes from the same package to reside in the same directory. Methods declared to be protected or to have default package visibility are accessible to all classes in the same package. Interfaces are not supported well by packages as all methods in interfaces are public. It is not possible to define an interface with methods that are only visible within the package. This provision of different levels of visibility within and outside of the package is an important feature of our modules.

In Theta [DGLM94], specifications play the role of our interfaces. Class declarations in modules include the name of the type being implemented, though no explicit import is required to gain access to names declared in interfaces. Those methods listed in associated type are the public methods of the class. Other methods may be made available to subclasses by listing them in a "provides" clause. Instance variables are not available to subclasses. Details of classes (including instance variables) are visible throughout their enclosing modules. Modules only indicate the routine implementations that they are exporting, type implementations are exported if one of the exported routines returns an object of that implementation. Theta contains no explicit import statements, and there is no support for "opening" one module implementation in another.

7 Conclusion

The module design presented here allows the user to group together tightly coupled systems of classes. This is advantageous because it

1. allows the programmer to restrict visibility of features of classes outside of the module while allowing classes within the module full access to details of implementations,

2. allows finer control over the name space in a program, and

3. supports separate compilation.

Modules also provide natural locations to store definitions of functions, procedures, and constant definitions that would otherwise have to be supported in an artificial way within classes.

Module interfaces, as well as narrowing the interdependencies between implementations of different groups of classes, also provide a clear contract between the module implemented and client as to the behavior of the constructs exported. While only the type information is actually checked by the system, we expect that non-machine checked specifications will also be included in these contracts.

Our module design for $\mathcal{LOOM}$ allows the programmer to import a module's interface or to open the implementation in another module. Clients typically import interfaces, while tool builders typically will need to open implementations in order to refine class definitions from the other module or to gain access to features not available to clients. Partial revelations allow the user to distinguish between methods available within the module implementation and those available to clients external to the module, providing tighter coupling to classes in the same module. Like Theta [DGLM94], we see no need to support parameterized modules when we already support the provision of type parameters for type and class definitions.

We have implemented an interpreter for $\mathcal{LOOM}$ with the module features described here. Our interpreter does not support separate compilation, though the type-checker is designed to use only information that would be available in separately compiled units. A complete description of the type-checking rules and natural semantics is available in [Pet96] (with later modifications as given in [Van97]).

We intend to examine issues involved in the support of subclassing mutually recursive types and classes in a way similar to that proposed in [BOW98]. That paper proposed using inner interfaces and classes as a scoping mechanism for mutually recursive constructs. However, for reasons similar to those noted in section 4, classes aren't an ideal choice for grouping such constructs. Modules seem like a much better choice.

Future work also includes building a full-blown compiler for $\mathcal{LOOM}$ in order to provide a more usable implementation and allow better testing of the language's support of programming in the large.

References

[AG96] Ken Arnold and James Gosling. *Java*. Addison Wesley, 1996.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *to appear in Proceedings of ECOOP '98*, 1998.

[CDG+88] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC-31, DEC systems Research Center, 1988.

[CDJ+89] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Sixteenth ACM Symp. Principles of Programming Languages*, pages 202–212, 1989.

[DGLM94] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Abstraction mechanisms in Theta. Technical report, MIT Laboratory for Computer Science, 1994.

[GRJV94] E. Gamma, R.Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and reuse of object-oriented designs.* Addison-Wesley, 1994.

[Int95] Intermetrics. *Ada 95 Reference Manual, version 6.0.* 1995.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *23rd ACM Symp. Principles of Programming Languages*, pages 68–78, 1996.

[KLM94a] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *21st ACM Symp. Principles of Programming Languages*, pages 138–150, 1994.

[KLM94b] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Conference Record of POPL '94: 21st ACM SIGPLAN–SIGACT Symposium of Principles of Programming Languages, Portland, Oregon*, pages 138–150. Association for Computing Machinery, January 1994.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. M.I.T. Press, Cambridge, MA, 1987.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[MMMP90] O. Madsen, B. Magnusson, and B. Moller-Pedersen. Strong typing of object-oriented languages revisited. In *OOPSLA-ECOOP '90 Proceedings*, pages 140–150. ACM SIGPLAN Notices,25(10), October 1990.

[MMP89] O. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA '89 Proceedings*, pages 397–406. ACM SIGPLAN Notices,24(10), October 1989.

[Pet96] Leaf Petersen. *A module system for LOOM.* Williams College Senior Honors Thesis, 1996.

[PT93] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, 1993.

[Szy92] Clemens A. Szyperski. Import is not inheritance; why we need both: Modules and classes. In *Proceedings of ECOOP '92*, LNCS 615, pages 19–32, 1992.

[Szy96] Clemens A. Szyperski. Independently extensible systems - software engineering potential and challenges. In *Proceedings, 19th Australian Computer Science Conference, Melbourne, Australia*, pages 19–32, 1996. Appeared as special edition of the Australian Computer Science Communications, 18(1), pp. 203-212, February 1996.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP '97*, pages 444–471. LNCS 1241, Springer-Verlag, 1997.

[Van97] Joe Vanderwaart. The new guide to LOOM (draft). Technical report, Williams College, 1997.

A Sample LOOM Program for Integer Sets Using Modules

The $\mathcal{LOOM}$ interface and implementation for module `SetOfInt` illustrates how partial revelations can restrict access to methods outside the module while allowing full access within the module. The implementation module imports the interface of `IntOrdList` in order to implement sets as ordered lists. The intersection method of SetOfInt is destructive – the receiver of the message is updated to hold the value of the intersection.

In the listing, a row of "-"s indicates a module boundary.

```
Interface IntOrdList;

type

   OrdListType = ObjectType
      first: proc();
      next: proc();
      off: func():Boolean; -- is current elt off end of list?
      add: proc(Integer);
      deleteCur: proc();  -- current is next elt after deleteCur
      contains: func(Integer):Boolean;
      getCur: func():Integer
   end;

   OrdListClassType = ClassType ... end;
      -- class types include instance variable and method types
const

   OrdListClass: OrdListClassType;

end;

----------------------------------------------------

Interface SetOfInt;

type

   IntSetType <# ObjectType
      add: proc(Integer);
      remove: proc(Integer);
      contains: func(Integer):Boolean;
      intersect: proc(MyType)
   end;

const

   function newSet(): IntSetType;
```

```
end; -- Interface SetOfInt

----------------------------------------------------

Module implements SetOfInt import IntOrdList;
-- SetOfInt is implemented as a specialized ordered list in order
-- to make the methods find and intersect more efficient.

type

   IntSetType = ObjectType include IntOrdList::OrdListType
      remove :proc(Integer);
      intersect: proc(MyType)
   end;

   ListSetClassType = ClassType
          include IntOrdList::OrdListClassType;
      methods visible
         remove :proc(Integer);
         intersect: proc(MyType)
      end;

const

   ListSetClass = class inherit IntOrdList::OrdListClass
      methods visible
         procedure remove(elt:Integer) is
            begin
               if find(elt) then deleteCur()
            end;

         procedure intersect(other:MyType) is
                                 -- destructive intersection
            begin
               first();
               other.first();
               while (not off()) and (not other.off) do
                  if getCur() < other.getCur() then
                     deleteCur()
                  else if getCur() > other.getCur then
                         other.next()
                       else
                         next();
                         other.next()
                       end -- else if
                  end -- if
               end -- while
```

```
            while not off() do
               deleteCur()
            end -- while
         end -- function
      end;  -- class

   function newSet():IntSetType;
      begin
         return new(ListSetClass)
      end;

end -- Module
```

B The Subject-Observer Pattern in a Module

In this section we present a more extensive example with multiple types and classes in the same module. This example constitutes a simple implementation of the Subject-Observer pattern.

The interface provides only those methods which need to be accessible outside of the module. Though we haven't shown particular examples here, methods in each of the three classes (`Observer`, `Subject`, and `Event`) may take advantage of methods of the others which are not mentioned in the module interface.

```
Interface SubObs;

   type
      ObserverType <# ObjectType
         notify: proc(SubjectType, EventType);
      end;

      SubjectType <# ObjectType
         addObserver: proc(ObserverType);
      end;

      EventType <# ObjectType
         ...
      end;

   const
      createObserver: func(...):ObserverType;

      createSubject: func(...):SubjectType;

      createEvent: func(...):EventType;
end;

--------------------------- compilation boundary -----------------
```

```
Module implements Subobs;

   type
      ObserverType = ObjectType
         notify: proc(SubjectType, EventType);
         ...
      end;

      SubjectType = ObjectType
         addObserver: proc(ObserverType);
         notifyObservers: proc(EventType);
         ...
      end;

      EventType = ObjectType
         ...
      end;

   const

      class Observer(...)
      var
         ...

      methods
         procedure notify (subject: SubjectType, event: EventType)
            begin ... end;
         ...
      end;

      class Subject(maxObservers: integer)
      var
         numObservers = 0: integer;
         observers = new(arrayclass(ObserverType, nil, 1, maxObservers))
                                                  : arraytype[ObserverType];
             -- array of maxObserver elts of type ObserverType - set to nil
      methods
         procedure addObserver(observer: ObserverType)
         begin
            if numObservers < maxObservers then
               numObservers := numObservers + 1;
               observers.set(numObservers, observer);
            else ...
            end;
         end;

         procedure notifyObservers(EventType e)
```

```
      var
         i: Integer;
      begin
         for i := 1 to numObservers do
            observers.get(i).notify(self,e);
         end;
      end;
   end;  -- class Subject

   class Event ...
end; -- Module
```