Import is Not Inheritance
Why We Need Both: Modules and Classes

Clemens A. Szyperski
szyperski@inf.ethz.ch

Institute for Computer Systems
Swiss Federal Institute of Technology (ETH Zurich)
Switzerland

Abstract. The design of many popular object-oriented languages like Smalltalk, Eiffel, or Sather follows a certain trend: The class is the only structuring form. In this paper, the need for having modules besides classes is claimed. Modules stem from a different language family and at first glance it seems that they can easily be unified with classes. Among other things, unifying modules and classes carries the danger of unifying the import and inheritance relationships. Constructs in several languages are discussed that indicate that modules and classes should indeed be kept separate.

1 Introduction and Clarification of Terms

Among other things, the quality of a programming language may be judged by the *number*, *orthogonality*, and *completeness of its concepts*. The first measure is as simple as counting; the second and third are far more difficult to determine. Orthogonality is important to avoid confusion; concept overlaps should at least be kept small. In practice, completeness is a relative measure: It is not expected that any single language will ever fulfill all possible demands. Other important measures like *soundness* (no contradictions in the language definition) or *theoretical completeness* (e.g. Turing completeness) are beyond the scope of this paper.

It is a good rule of thumb to keep the number of concepts in a language small. However, it is always possible to reduce everything to (almost) a single concept. For example, in some functional languages everything is expressed using the single concept of higher-order functions. This tends to make programs hard to grasp. Keeping orthogonality and completeness in mind, it still is often preferable to provide *separate concepts for separate problems*. This leads to more natural ways of expressing solutions in a given language.

This paper concentrates on existing languages, including the most popular object-oriented ones. A single point, that a language should have modules *and* classes (and that import is not inheritance) is claimed and supported. Before going into any details, a clarification of the terms class and module is in order. To avoid early preferences for certain languages, the *terms will be defined the way they should be understood when reading this paper.* Differences to concepts found in existing languages will be discussed in subsequent sections. To avoid confusion with the term *object*, as understood when talking about object-orientation, the term *item* will be used to denote individual things that can be declared and operated on in a given language. Such an item may be a variable, a type, a procedure, a method, a class, or any other entity available in that language.

The rest of the paper is organized as follows. The following subsections cover informal and formal clarifications of the used terms. Section 2 gives two examples revealing certain problems in class-only approaches. Section 3 illustrates how such problems can be dealt with in different languages, also showing the advantages of solutions based on modules *and* classes. (Sections 2 and 3 go into details and may be skipped for a first overview.) Section 4 analyzes modularization aspects in a variety of languages and Section 5 reflects on advantages that can be gained when adding modules to a language. Finally, a summary and conclusions are given.

1.1 Informal Clarification of Terms

A *module* is a capsule containing (definitions of) items. The module draws a strong (syntactic) boundary between items defined inside it and items defined outside in other modules. In principle, items from one module, say *B*, are only visible in a another module, say *A*, if *A imports B*. Thus, considering a set of modules leads to a directed import graph. Often, it is considered good engineering practice to avoid cycles, thus having a directed acyclic graph (*DAG*).

A module may restrict the visibility of the items it contains. It is said that such a module *exports* only a subset of its items. Also, a module may impose usage restrictions on exported items, e.g. by only allowing read-access to an exported variable. However, a module should not restrict the visibility between the items it contains: This may be called the *No Paranoia Rule*. It is very important and its use will be explained in more detail below. (The set of module items exported from a module is often called the module's *interface*, while the complete module is said to be the *implementation* of that interface.)

As a module draws a fence between inside and outside, it also serves as a natural *naming scope*. Symbolic references to items imported from other modules can be qualified using the name of the imported module. A more formal definition of modules is given below.

A *class* is a template for objects: It bundles the (definitions of) items that together implement objects of that class. Objects are said to be *instances* of their class. A class may *inherit* features from one or more other classes, leading to an incorporation of the inherited implementations: Items defined in an inherited class become items of the inheriting class. Combining a set of classes using the inheritance relation leads to an inheritance DAG.

Like a module, a class draws a syntactic boundary between inside and outside. However, it should not be used to control visibility of class details, but to distinguish between those items that do belong to an instance of that class, and those that do not. Additionally, a class is dynamically instantiated, i.e. many objects may belong to the same class, while a module is statically instantiated (only once). In this sense, *a module is a compile-time abstraction* which does not exist at run-time, while *a class is a run-time abstraction* as it defines the structure and behavior of objects at run-time.

Remark. In the recent literature [CHC90], classes are distinguished from types. For this paper, the difference between types and classes is of no importance: The claim that modules are not classes may be translated into another one saying that modules are not types. Formal definitions of types and classes are not given in this paper. However, the formal definition of modules given in the next section may serve as a partial "definition by exclusion".

1.2 Towards a Formal Definitions of Modules and Import Relations

A system (a program) is considered to be composed of a set of items I, where the kinds and semantics of items are defined by some (programming) language L. In a typical programming language such items are constants, types, classes, variables, functions, and the like. Over the set of items I various *refers-to* relations ref_i are defined, where the semantics of these relations are defined by L. For example, a language might introduce relations *reads-from*, *writes-to*, *has-type*, *extends*, or *calls*. What does it mean to *modularize* such a system?

To begin with, a partitioning M over the set of items I is defined. In this paper such partitions are called *modules*. Hence, each item is defined within exactly one module. A module $m \in M$ defines various subsets $exp_i(m) \subseteq m$. A subset $exp_i(m)$ *exports* part of the items contained in m for use in the refers-to relation ref_i. For example, in a language a variable may be exported *read-only*, a type may be exported *leaf* (i.e. under the restriction that it cannot be used to form new subtypes).

Over the set of modules M a relation *imp* is defined: For modules $m, m' \in M$ the term m *imp* m' signifies that m imports m'. Intuitively this means that the export subsets of m' are accessible within m. For example, a module m may call a function exported by another module m' if m *imp* m' holds. From this definition it does not follow that (M, *imp*) forms a DAG. However, as mentioned before, this additional constraint is often considered good style.

The semantics of the partitioning M is based on the set of valid relations ref_i. A refers-to relation ref_i is called *valid under* M if for all items $x, x' \in I$:

- $x\ ref_i\ x'$ is valid in L
- for $x \in m$ and $x' \in m'$, where $m, m' \in M$ are modules:
  $(m = m') \vee (x' \in exp_i(m') \wedge m\ imp\ m')$

That is, the relation ref_i is valid under M if it is valid under the trivial partitioning (i.e. if no module restrictions are in effect) and if all references crossing module boundaries are matched by the import relation and allowed by the corresponding export subsets.

For a language L' to be said to *explicitly support modularization*, L' should be composed of a language L (as used above) augmented with a module construct used to express M, (M, *imp*), and for each $m \in M$ the subsets $exp_i(m)$.

Remark$_1$. Some languages (e.g. Ada [Ada80], or Modula-2 [Wir82]) use textually separate sections to define module *interfaces* and module *implementations*: An interface specifies all the module's exp_i subsets plus the involved items, while the rest is added in the implementation part. Other languages (e.g. Oberon [Wir88b]) use a single textual section to define a module, plus special markings to specify exp_i subsets in-place.

Remark$_2$. Modularization may be applied by convention. This is about what is done in languages like C [KR78], where modules are (by convention) associated with files, and interfaces are expressed by means of header files. However, the intention of this paper is to emphasize the importance of having modules explicitly in the language. To exclude modularization by convention, the formal definition ends in augmenting language L with an explicit module construct.

Remark$_3$. A family of modules may be described by means of a single (generic) parameterized module. Examples are Ada's generic packages or OOZE's generic modules [AG91]. Genericity is an interesting option for modules but is beyond the scope of this paper.

Remark$_4$. Another interesting question regards the introduction of higher-level modules, i.e. modules of modules. There are no new problems if higher-level modules are used to partition M. A particular proposal may be found in [Car89], where higher-level modules are called *systems*. Care must be taken if higher-level modules contain other items than modules on the next lower level (cf. section 4.1). This approach is not followed further in this paper.

2 Pathological Cases – Examples

By now – and ignoring the formal definitions – it may seem that the terms of module and class are very close if not identical. To work out the difference, the following two sub-sections describe two especially severe problems resulting if a language has only classes but no modules. The first problem deals with entities not naturally expressible in terms of classes and the way such entities may be made accessible to some client implementation. The second problem considers invariants spanning multiple cooperating classes. Particular solutions for both problems are discussed in Section 3.

2.1 Feature Import – Pseudo Inheritance versus Pseudo Forwarding.

Many items exist that do not naturally belong to some class. It is artificial to bind *globally used constants or variables* to a class. Likewise, there exist *functions* and *procedures* difficult to associate with a class. For example, functions accepting several parameters of different types would need to emphasize one of them as message receiver.

Operations provided by mathematical libraries are another typical example: To what class belongs a set of, say, statistics functions? The functions operate on some argument, say of class *Real*. Consequentially, one could add the functions to *Real*. In turn, this requires adding all functions ever to be applied to real numbers to *Real*! However, for the time being assume that the math library has been packaged into separate classes. Then, how is such a library used?

In a system having only classes there are two possibilities: Inherit the library class or use it by instantiating a dummy object of that class. The former may be called *pseudo inheritance* and the latter *pseudo forwarding*. Table 1 illustrates both possibilities. (The used pseudo-notation is entirely fictitious. Any resemblance to languages, living or dead, is purely accidental.)

```
CLASS StatisticsLib;
  METHOD Gamma(x: REAL): REAL;
  ...
END StatisticsLib;
```

```
CLASS LibUser1;
  INHERITS StatisticsLib;

  METHOD Calc;
    y := self.Gamma(x)
  ...
END LibUser1;
```

```
CLASS LibUser2;
  METHOD Calc;
    stat: StatisticsLib;   (*dummy variable*)
    NEW(stat); y := stat.Gamma(x)
  ...
END LibUser2;
```

Table 1. Feature Import - Pseudo Inheritance vs. Pseudo Forwarding.

Both approaches try to circumvent the missing import construct by somehow accessing the required feature via inheritance. Note that the pseudo inheritance scheme is particularly counter-intuitive: Class *LibUser1* inherits from class *StatisticsLib*, although *LibUser1* is not a specialization of *StatisticsLib*. Neither is it expected to explore polymorphism among these two classes nor is it reasonable to expect overriding of *StatisticsLib* methods in *LibUser1*.

The pseudo forwarding mechanism builds on a variable actually never used as such: The variable *stat* in the example of *LibUser2* is only used as a means of indirection; in the example a simple procedure variable would do. The indirection is actually not wanted in most cases, i.e. the "imported" "class" is always the same. Therefore, the conceptually inferior pseudo inheritance scheme is sometimes preferred to avoid additional indirections.

2.2 System Invariants

Class-centered languages work well as long as important invariants of the constructed systems can be associated with single classes. Such invariants are called *class invariants*. However, a set of cooperating classes may in combination establish and maintain some *co-invariants* (also called *system invariants*). Such co-invariants are expressed over instances of different classes and therefore cannot be established or maintained by a single class.

For example, consider two classes *LinkedList* and *Linkable*. The idea is that *LinkedList* maintains linked lists of objects. These objects must be instances of classes derived from class *Linkable*. Table 2 shows the skeletons of both classes.

```
CLASS Linkable;
  next: Linkable;
  ...
END Linkable;

CLASS LinkedList;
  head: Linkable;   (*ring with head*)

  METHOD Prepend(x: Linkable);
    ASSERT(x.next = NIL);
    x.next := head.next; head.next := x
  ...
END LinkedList;
```

Table 2. Classes LinkedList and Linkable.

A well-known problem arises if a linkable object is inserted into more than one linked list at a time. To detect this kind of problem, *LinkedList* can code into a linkable object whether it is in a list or not. In the example, the list is implemented as a ring. Hence the *next* field can be used to code list membership of a linkable object, say *z*. (It is assumed that *next* is initialized to NIL.)

$$(\forall z : \text{NotInAnyList}(z) = (z.\text{next} = \text{NIL})) \qquad (P)$$

To be sure that no problems arise, it should be guaranteed that a linkable object is always inserted into at most one list. This is equivalent to the invariant that there are never two list heads, say *x.head* and *y.head*, from which the same linkable object, say *z*, is reachable (by traversing the *next* fields).

$$(\forall x, y, z : x \neq y : \text{Reachable}(x.\text{head}, z) \Rightarrow \neg\text{Reachable}(y.\text{head}, z)) \qquad (H)$$

This is a co-invariant of classes *LinkedList* and *Linkable* as it is defined over the instance variable *next* of *Linkable* and *head* of *LinkedList*. If *NotInAnyList*(x) is an established precondition of all list operations that insert a linkable object, then H is invariant. In the example, method *Prepend* asserts $x.next = NIL$, since assuming invariance of P, this guarantees the precondition.

However, this assumes that *next* is not modified by any class other than *LinkedList*. In other words, if *next* is modified outside of *LinkedList*, invariant H is no longer implied by precondition P. On the other hand, if *next* is read-only outside of *Linkable*, *LinkedList* cannot be implemented. Hence, a construct is required that permits free access within a class system (maintaining co-invariants), while restricting access from outside. Such constructs are discussed in section 3.2.

3 Pathological Cases – Solutions

3.1 Importing Features – Inheritance Abuse versus Module Import

Section 2.1 indicated that traditional procedures and functions as well as globally used constants and variables do not relate well to classes. However, one wants to avoid having procedures or variables existing outside of *any* structure. Thus many languages force their declarations to belong to the only structuring form available, e.g. to some class. This leads to complex or weird language constructs:

In C++ [Str86], feature import is solved by redefining the required items in *declarations*. It is the obligation of the linker to check whether such references can be resolved by some matching *definitions*. As in C, modularization is available on the level of *files*. While this mechanism is not quite a language level module concept, it at least avoids inheritance abuse in order to implement imports. Global variables and proper procedures are available. Visibility of class and non-class entities is controlled using totally different constructs.

Smalltalk-80 [GR83], Eiffel [Mey88] and its descendant Sather [Omo91] have no module concept. The problem is attacked by automatically importing all other classes currently in the system. In other words, it is possible to directly refer to the names of all other classes within a class. The only means of accessing class features are inheritance and message-sends. For example, in Eiffel the inheritance concept is routinely abused to import standard input/output procedures. Additionally, Sather allows for accessing arbitrary items of other classes. To overcome the lack of proper procedures, Sather allows to call a method directly (passing a void receiver object for self). In Smalltalk-80 *Meta Classes* are used mainly to compensate for missing global variables and procedures. Eiffel incorporates the notion of *Once Procedures* to work around missing global variables. Sather instead adds *Shared Instance Variables* (i.e. class variables) to solve this problem. Finally, Eiffel and Sather allow for declaration of *Private Features* to support information hiding on the class level.

Having modules in the repertoire of a language it is not necessary to artificially bind items like global variables or procedures to classes. Hence, the example of section 2.1 can be solved by defining functions like *Gamma* within some library module and therefore by using import instead of inheritance. (This does not solve the problem if one really wants to add a new feature to an existing class. For this kind of incremental modification a different concept, like the Capsules concept of Fresco [Wil91], is required. Cf. section 4.)

Modula-90 [Ode90] mixes the concepts of modules and classes by having a single construct for both, but providing for two different relations among such class-modules:

Import and inheritance. While this solves the problems mentioned above, it adds a certain amount of confusion by mixing two otherwise orthogonal concepts. More on this in section 4.

3.2 Guaranteeing System Invariants – Spaghetti Scoping versus Modular Scoping

In C++ cooperation among classes is typically tackled using *friend functions*, a mechanism allowing access to inner parts of a class (called *private* in C++) from within another class. In some sense, the C++ friend mechanism is a "scope go-to" as it interrelates classes defined somewhere in the system by adding *ad-hoc* privileges. On the other hand, to maintain principles of information hiding, C++ expects friend relations to be explicitly (and statically) declared in all affected classes. Therefore, the friend mechanism has no principal advantages over static bundling of classes into modules. However, it does allow *spaghetti scoping* between arbitrary classes.

Eiffel and Sather have the same kind of problem due to their lack of modules. Here, referring to the example of section 2.2, one would export the field *next* from class *Linkable* to class *LinkedList* and its subclasses, only. This limited export to named classes has the same structural weakness (*spaghetti scoping*) that the friend mechanism of C++ has.

However, if modules are part of a language, the solution is simply to use read-only export of *next*. Within a module the direct access to the implementation of a class is accessible to other classes in the same module (No Paranoia Rule). Hence, *next* can be manipulated within *LinkedList*. See Table 3; names marked with an asterisk (*) or a dash (-) are exported by the module, where the dash indicates read-only export. (Instead of using read-only export, *next* could be kept private and a *Next* method could be added to *Linkable*.)

```
MODULE Lists;
  CLASS Linkable*;
    next-: Linkable;
    ...
  END Linkable;

  CLASS LinkedList*;
    head: Linkable;   (*ring with head*)

    METHOD Prepend*(x: Linkable);
      ASSERT(x.next = NIL);
      x.next := head.next; head.next := x
    ...
  END LinkedList;
END Lists.
```

Table 3. Bundling LinkedList and Linkable classes into a module.

Remark. Apparently, modular scoping is more limited than spaghetti scoping. For example, a set of classes A, B, and C may cooperate in a way that the pairs A-B and A-C each need to be in a common module. Then, modular scoping forces all three classes into a single module, although classes B and C may have no direct relations at all. This is considered a feature rather than a weakness of the proposed module scheme: It forces tightly related parts of a system into common modules, leading to a clean modular structure.

4 Discussion of Module and Class Constructs in Different Languages

In this section a closer look at modules and classes in existing languages is taken. An early treatment of the module concept may be found in [Par72]. First implementations of modules may be found in Mesa [MMS79] and UCSD Pascal [Bow80]. The class concept originated in Simula-67 [DMN68], was later used in Smalltalk-80 as the only structuring means, and recently found its way back to traditional languages, forming Object-Pascal [Tes85], C++, or Eiffel.

The primary family of modular languages stems from Modula-2 and has direct successors in Modula-2+ [RLW85], Modula-3 [CDG*88], and Modula-90, of which the latter two also add object-oriented concepts. Ada has a modularization concept called *packages*. Languages like C++, Eiffel, or Sather concentrate on the class construct. Finally, this paper concentrates on languages with static checking: Languages like Self [US87] or CLOS [DG87] delegate checks to run-time and are not considered. However, even for Smalltalk, an untyped language geared towards rapid prototyping, the attempt has been made to add modules [WW88] or other additional structuring means [Wil91]. Still, the message that modules are an orthogonal and quite useful construct in an object-oriented language apparently has not yet reached sufficient audience.

4.1 Modules

Modula-2 is perhaps the most popular language (almost *by name*) with a fully developed modularization concept. However, Modula-2 took the module concept too far by allowing modules to be nested inside scopes with dynamic extent, i.e. procedures. This leads to some confusion, as the variables defined in such modules behave very much like local variables of a procedure. Hence, one might argue that the Modula-2 kind of module contributed to blurring the distinction of modules and types or classes. For example, in [Car89] one can read *"[Modules] are very similar to abstract types, but add the notion of imported identifiers [...] thereby evading the strict block-structure of statically scoped languages."* This opinion lead to generalizations of the module concept to a class concept in Modula-90 by explicitly introducing a notion of *module types*. This completes confusion: A record structured type can be described using both, the RECORD or the MODULE TYPE constructor.

C++ has a poor man's modularization concept stemming from its predecessor C. It is realized by means of files and partial redefinitions of these (usually supported using so-called *include files*). This mechanism simulates modules by files which are not part of the language itself. Problems of inconsistencies among various versions are not handled by the compiler but delegated to the file system and various auxiliary tools, e.g. *make* or *sccs*.

Ada's package concept gets very close to the module concept. However, Ada has a complicated form of expressing the import relation and export subsets, making it a complicated language to use and to read. For example, the use–clause (forming the import relation among packages) can perform overloading of already visible names, thus combining two separate concepts into one.

Oberon [Wir88b], a successor of Modula-2, on the other hand eliminated local modules. As an important result, the language became cleaner and the notions of module and type well separated and orthogonal. In Oberon-2 [MW91] a step has been taken that is close to the decisions taken for Object-Pascal or C++: Instead of extending the module concept to form classes, the record construct has been extended to allow binding of procedures (methods) to record types. Class inheritance is then based on the *type extension* concept

already present in Oberon [Wir88a], and no new constructor OBJECT (à la Object-Pascal) or CLASS (à la C++) is required.

The Fresco system [Wil91] defines an interesting variation of modules, called *capsules*. Fresco capsules are in a sense more orthogonal to the rest of the language (in this case Smalltalk) than the modules proposed in this paper: A class may take its definition from multiple capsules. This has the advantage of simplifying modifications of a system performed as an afterthought. However, it is not clear how capsules relate to the partitioning of a system's design. In fact, Fresco encourages the incremental addition of capsules spanning multiple levels of abstractions at a time. This is motivated by the way Smalltalk is typically extended, i.e. by means of small additions made to existing classes. In a certain sense, capsules are even orthogonal to modules, as they propose a partitioning into incremental, history-driven *deltas* applied to a system, while modules are most useful when partitioning a system into levels of abstraction or units of service.

4.2 Classes

Many pure object-oriented languages like Smalltalk, Eiffel, and Sather directly use classes for modularization. To allow for feature import, inheritance and message-sends are used. In Eiffel inheritance is frequently used to simulate imports. To remove clutter again, Eiffel as well as Sather have features for *undefining* inherited features. (In the Sather report it is even stated that *"elimination of a parent's object or shared attributes can save space when they are not needed"* ...). This in turn adds the problems of having features visible in a superclass view of an object that are actually deleted from the feature list of the object's class. It seems that the lack of a proper import relation is partially compensated for by allowing to *undefine* unwanted side-effects of the abused inheritance relation.

To achieve the structuring power of modules, Smalltalk, Eiffel and Sather would need nested classes, where classes nested inside the same host class would have cross-access to their implementations. Such a nesting of classes is rather problematic, as it is no longer clear what instantiation means: Are instances of a nested class members of the same class, even if they have been instantiated from within different instances of the host class? (Interestingly enough, C++ *has* nested classes but treats them as if they where declared in a non-nested fashion!)

Simula supports nested classes, and uses this construct to mimic modules by means of classes. A typical example is the Simula standard class *Simset* [BDM*73] which uses a doubly linked ring to implement a set abstraction. *Simset* defines three nested auxiliary classes (*Linkage*, *Link* and *Head*). Hence, class *Simset* serves as a kind of module packaging the nested classes. To use classes defined in *Simset* within a statement block, the block needs to be "prefixed" with the name of *Simset*. This is similar to an import construct and adds a certain confusion: The "imported" class is more used as a module than as a class.

BETA [KMM*87] solves the problem by two apparently overlapping concepts. Here, nested classes are realized using so called *patterns*, possibly with exactly one instance, may be used to tackle the problem of cooperating classes. Also, *fragments* (close to modules, but additionally supporting multiple versions under control of the language), may be used. From the language report [KMM*90] it is not too clear when to use which concept, but the intention apparently is to use patterns to model concepts and fragments to provide implementations.

5 Effects of Modularization on System Structure

General Software Engineering aspects of modularization are well-known since Parnas first pleaded for introduction of a module concept. This section reflects on some aspects that may be reconsidered when adding modules to a language.

A good motivation on the necessity of having modules in a language to be used for large projects may be found in [Car89]. Cardelli states explicitly that *"a surprisingly common mistake consists in designing languages under the assumption that only small programs will be written; for example languages without [...] modules or type systems. If widely used, such languages eventually suffer conceptual collapses under the weight of ad hoc extensions."*

5.1 Information Hiding – Efficiency versus Encapsulation

Traditionally, adding full encapsulation to an implementation of some abstraction increases costs. The information hiding paradigm does not allow for direct access to implementation or representation details, requesting that all interactions being performed via abstract operations (methods or procedures). While recent work [HCU91] indicates that fully abstract interfaces need not be a hurdle when aiming at efficiency, current language implementations perform poorly when abstractions are taken too far: Nearly every single operation is then implemented as a procedure or method invocation. As a result, significantly more complex compilation techniques (like compilation at run-time, or global program analysis) are required to regain efficiency. Often, such techniques require simultaneous availability of all source code, colliding with practical requirements (and perhaps legal issues) in large projects.

Using modules, this problem can often be solved. The *No Paranoia Rule* stated above allows information hiding from module clients, while at the same time imposing no restrictions on the interrelations within a module. Hence, if it is possible to implement closely interacting parts within a single module, efficiency problems due to information hiding are not an issue. In practice, this strategy works if object-orientation is only taken to medium granularity, i.e. if objects represent larger abstractions than, say, integers or characters. This is especially the case if supportive classes (or simply procedures) are used to implement a class: Such auxiliary classes need not even be exported by a module.

5.2 Separate Compilation and Dynamic Loading

Modules are the natural *unit of compilation*. They are small enough to allow for fast recompilation, and in many cases their interfaces can be kept stable enough to reduce cost-intensive recompilations of clients. If modules are part of the language (as opposed to being simulated by files à la C), the compiler can maintain version information. Then, recompilation of clients of a module which changed its interface can be enforced, unnecessary recompilations can be avoided, and incompatibilities can be detected at compile-, link-, or load–time.

Also, a module is a useful unit of *dynamic loading*. If this is done, the time of loading is not under control of the programmer and it becomes hard to establish initial invariants of a module. To cope with this situation, an initialization body should be attached to a module. (Without dynamic loading of modules the module body can be seen as an option for the language designer: The programmer could as well call initialization procedures.) A module body contains code executed immediately after loading the corresponding module (and after loading and initializing all imported modules).

5.3 Support for Hybrid Object-Oriented Language Concepts

In traditional procedural languages many different concepts are present. Types, constants, variables and procedures are the most important ones. Pure object-oriented languages like Smalltalk have a tendency to get rid of such constructs in favor of a few new, more general ones, i.e. classes and objects. A less radical approach is followed by hybrid languages that maintain the proven concepts of older languages and (more or less smoothly) add new object-oriented concepts. C++ does this for C, Oberon-2 for Oberon, and Modula-90 for Modula-2.

In a hybrid object-oriented language it is not expected that everything is expressed in terms of classes and objects. Hence, it is important to have some kind of bundling concept: A clean module construct is just perfect for this. For example, it is possible to implement a module which exports classes for its extensible abstractions, while at the same time details within the module are implemented in a conventional fashion wherever appropriate. This way, readability and efficiency can be improved by using the most natural constructs for the problems at hand.

If modules are not available, but conventional constructs like statically bound procedures are still desired, rather weird concepts may result. For example, in Sather it is allowed to directly access parts (called *features*) of another class by simply naming them: *y := MATHCONSTS::pi* accesses the constant *pi* defined in class *MATHCONSTS*, or *OUT::s("Hello")* calls the "method" *s* of class *OUT* to output a string. In the latter case the method is called with a void receiver (i.e. self = void), which is fatal for true methods. Hence, such "methods" need be written like proper procedures, and the additional self-parameter is superfluous.

Summary and Conclusions

Adding a module concept to object-oriented languages seems unnecessary, as classes are often thought of being just like modules, plus inheritance, plus dynamic binding, and plus multiple instantiation. In this paper, the claim has been made that a language should provide for both, modules and classes.

To shortly summarize, modules are compile-time abstractions that provide for
- syntactical structure in a large system
- orderly scoping of names and interrelations using the import graph
- natural units of information hiding, separate compilation, and dynamic loading
- orderly use of global variables and procedures
  (comparable to class variables and class methods).

Modules need not be nestable and definitely should not be nested within other structures like classes. It should be emphasized that modules are understood as a purely syntactical construct used to add structure and visibility borders to a system. Modules have no special run-time semantics (perhaps with the exception of module initialization bodies, cf. Section 5.2).

Classes on the other hand are run-time abstractions that provide for
- object templates defining object structure and behavior
- multiple instantiation (object construction)
- orderly inclusion and extension of superclass implementations
  using the inheritance graph.

In a language that does not separate classes from types, classes may also provide for typing

purposes. It is important to note that the separate role of modules remains even if a language distinguishes between classes and types.

Having only classes, the various roles of modules need to be added to the class construct. This leads to solutions that are neither intuitive (e.g. inheritance of an I/O library class), nor conceptually clean (e.g. *spaghetti scoping*). Such approaches are present in many of the current object-oriented languages like C++ or Eiffel.

Several languages exist that provide for both constructs. For example, Oberon-2 defines a combination that closely follows the suggested module semantics; other languages, like Modula-3, come close. The potential of information hiding, separate compilation, and dynamic loading has been fully utilized in the Oberon system [WG89]. Likewise, the BETA system [KMM*90] uses fragments to support modularization and separate compilation.

The module concept is particularly lightweight (e.g. [Gri91], [WG92]): Adding modules to a language requires only small syntactical efforts and the implementation overhead is small. Furthermore, there are no run-time costs besides those possibly introduced by separate compilation.

Modules and classes serve different purposes when structuring a system. In this paper, several indications have been given where to put what. For module-only as well as for class-only approaches, guiding heuristics exist (for example [Par72], [Mey88]). Future work may be directed at giving good criteria for designing object-oriented systems by decomposing into modules *and* classes.

Adding a modular structuring concept to object-oriented languages has the same principal problems as adding types has. In both cases, with types as well as with modules, the programmer is forced to introduce structure that does not directly solve any problem. This could be a hindrance when prototyping rapidly in new application areas. For better maintainability of larger systems (e.g. larger prototypes) it is strongly felt that structure becomes more important than ad-hoc coding potential.

Remark. This paper concentrates on working out the characteristics of modularization in an object-oriented setting, and does not cover experiences on application of such a scheme. However, it might be worth mentioning that the proposed scheme has been applied using the language Oberon-2 in actual projects of non-toy size (among them an object-oriented operating system).

Acknowledgements

I would like to thank H. Mössenböck, R. Griesemer, S. Lalis, C. Pfister, J. Templ, and W. Weck for carefully reading earlier versions of this paper, as well as for many fruitful discussions on the topic presented. Further, I am grateful to the anonymous referees for providing many helpful comments. Remaining faults, especially regarding details of one of the many cited programming languages, are of course mine.

References

[Ada80] Reference Manual for the Ada Programming Language – Proposed Standard Document, United States Dep. of Defense. July 1980.

[AG91] A.J. Alencar, J.A. Goguen. OOZE: An Object-Oriented Z Environment. Conf. Proc. ECOOP '91, Geneva, Switzerland. Lecture Notes in Computer Science 512, Springer-Verlag, Berlin. July 1991.

[BDM*73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, K. Nygaard. *SIMULA BEGIN* (second edition). Van Nostrand Reinhold, New York, NY. 1979. First edition: 1973.

[Bow80] K.L. Bowles. *Beginner's Guide for the UCSD Pascal System.* Byte Books, Peterborough, NH. 1980.

[Car89] L. Cardelli. Typeful Programming. (Quest Language). *Research Report* 45, DEC Systems Research Center, Palo Alto, CA. May 1989.

[CDG*88] L. Cardelli, J. Donahue, L. Glassmann, M. Jordan, B. Kalsow, G. Nelson. Modula-3 Report. *Research Report* 31, DEC Systems Research Center, Palo Alto, CA. August 1988.

[CHC90] W. Cook, W. Hill, P. Canning. Inheritance is not subtyping. *Proc. POPL '90.* ACM Press, January 1990. Also: *ACM Trans. on Progr. Lang. and Systems.* 125-135, 1990.

[DG87] L.G. DeMichiel, R.P. Gabriel. The Common Lisp Object System: An Overview. (CLOS). *Conf. Proc. ECOOP '87*, Paris. Lecture Notes in Computer Science 276, Springer-Verlag, Berlin. June 1987.

[DMN68] O.-J. Dahl, B. Myrhaug, K. Nygaard. SIMULA 67 Common Base. Norwegian Computing Center, Oslo. 1968.

[GR83] A. Goldberg, D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA. 1983.

[Gri91] R. Griesemer. On the Linearization of Graphs and Writing Symbol Files. *Technical Report* 156, Institute for Computer Systems, ETH Zurich, Switzerland. March 1991.

[HCU91] U. Hölzle, C. Chambers, D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. Conf. Proc. ECOOP '91, Geneva, Switzerland. Lecture Notes in Computer Science 512, Springer-Verlag, Berlin. July 1991.

[KMM*87] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard. The BETA Programming Language. In: B.D. Shriver, P. Wegner (Eds.), *Research Directions in Object-Oriented Programming.* MIT Press. 1987.

[KMM*90] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Draft of an upcoming book. October 1990.

[KR78] B.W. Kernighan, D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ. 1978.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* (Eiffel Language). Prentice-Hall, Englewood Cliffs, NJ. 1988.

[MMS79] J.C. Mitchell, W. Mayburry, R. Sweet. Mesa language manual. *Technical Report* CSL-79-3, Xerox PARC, Palo Alto, CA. April 1979.

[MW91] H. Mössenböck, N. Wirth. The Programming Language Oberon-2. *Structured Programming*, 12:4, 1991.

[Ode89] M. Odersky. Extending Modula-2 for Object-Oriented Programming. (Modula-90 Language). *Proc. First Int. Modula-2 Conf.*, Bled, Yugoslavia. October 1989.

[Omo91] S.M. Omohundro. The Sather Language. *Technical Report* TR-91-34, International Computer Science Institute, Berkeley, CA. June 1991.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15:12, 1053-1058. December 1972.

[RLW85] P. Rovner, R. Levin, J. Wick. On extending Modula-2 for building large, integrated systems. (Modula-2+ Language). *Research Report* 3, DEC Systems Research Center, Palo Alto, CA. January 1985.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA. 1986.

[Tes85] L. Tesler. Object-Pascal Report. *Structured Programming* (was: *Structured Language World*), 9:3, 10-17. 1985.

[US87] D. Ungar, R.B. Smith. Self: The Power of Simplicity. *Conf. Proc. OOPSLA '87*, Orlando, FL. October 1987.

[WG89] N. Wirth, J. Gutknecht. The Oberon System. *Software - Practice and Experience*, 19:9. September 1989.

[WG92] N. Wirth, J. Gutknecht. *The Oberon System.* Addison–Wesley, Reading, MA. 1992.

[Wil91] A. Wills. Capsules and Types in Fresco – Program Verification in Smalltalk. *Conf. Proc. ECOOP '91*, Geneva, Switzerland. Lecture Notes in Computer Science 512, Springer-Verlag, Berlin. July 1991.

[Wir82] N. Wirth. *Programming in Modula-2* (fourth edition). Texts and Monographs in Computer Science. Springer-Verlag, Berlin. 1988. First edition: 1982.

[Wir88a] N. Wirth. Type Extensions. *ACM Trans. Programming Languages and Systems*, 10:2, 204-214. July 1988.

[Wir88b] N. Wirth. The Programming Language Oberon. *Software - Practice and Experience*, 18:7, 671-690. July 1988.

[WW88] A. Wirfs-Brock, B. Wilkerson. An Overview of Modular Smalltalk. *Conf. Proc. OOPSLA '88*, San Diego, CA. September 1988.