Module Embedding[1]

Atanas Radenski
Computer Science Department
UNC-WSSU, P. O. Box 19479
Winston-Salem, North Carolina 27110, USA
radenski@ga.unc.edu

Summary

This paper proposes a code reuse mechanism called *module embedding* that enables the building of new modules from existing ones through inheritance, overriding of procedures, and overriding of types; the paper also describes an implementation scheme for this mechanism. Module embedding is beneficial when modules and classes are used in combination and need to be extended together, or when modules are more appropriate medium than classes.

Keywords:

modules, object-oriented programming, inheritance, extensibility.

[1] This work is partially supported by NSF grant CCR-9509223 and NASA grant NAG3-2011.

1 Motivation

In modular languages, classes and modules are complimentary constructs that satisfy different needs of programmers. A *class* introduces an abstract data type that can be used to create several objects. A *module* can be employed to

- encapsulate one or more classes;
- define and implement an abstract data structure, i.e., a single entity without an associated type;
- group several related classes and procedures into a subsystem or into a framework;
- encapsulate a library of mathematical functions;
- package a class with related global variables and procedures.

An essential difference between classes and modules is that classes are extensible and their operations are invoked by dynamic binding, while modules are not directly extensible and their operations are invoked by static binding. We propose to eliminate this difference by means of a code reuse mechanism that we call *module embedding*. Module embedding enables the building of new modules from existing ones through inheritance, overriding of procedures, and overriding of types. Module embedding can be beneficial when modules and classes are used in combination and need to be extended together, or when classes are less appropriate than modules or not applicable at all.

We illustrate the essence of module embedding and the potential benefits of its adoption by means of an example. Consider the problem of implementing and using simple bank accounts, such

that:

- each *account* has a certain *balance;*
- a client may *open*, *transact* on, or *close* an account;
- their *total* balance characterizes all open accounts.

In Figure 1, all components of the implementation are encapsulated in a module, *M0*. The class of all accounts is represented as an *extensible record type* (an approach first adopted in Oberon [13, 14] and later used in Ada [10]). Procedure *transact* can be used for deposits *(amt > 0)* or withdrawls *(amt < 0)*; it also updates the *total* balance of all accounts. As in Oberon, the '*' sign is used as mark that designates exported, or public entities [14].

```
module M0;
type account* = record
           balance*: real;
        end;
var total*: real;

procedure open*( var a: account);
begin a.balance := 0 end;

procedure transact*(var a: account; amt: real);
begin a.balance := a.balance + amt;
  total := total + amt
end;

procedure close*(var a: account);
begin transact(a, - a.balance) end;

begin {initialization of M0} total := 0 end.
```

Figure 1. Embeddable module *M0;* exports are marked by '*'.

Consider now the problem of keeping track of the total number of all accounts and assigning account numbers to individual accounts. This can be achieved by embedding module *M0* into a new module, *M1*, which also contains an additional global variable,

numAccnts, as shown in Figure 2. The *embedding* module, *M1*, *inherits* all exports of the *embedded* module *M0*: the type *account*, the global variable *total*, and the procedures *open*, *close*, and *transact*; these entities are re-exported by *M1*.

```
module M1(M0);
type account* = record
           number*: integer;
        end;
var numAccnts*: integer;

procedure open*(var a: account);
begin  ^open(a);
  numAccnts := numAccnts + 1;
  a.number := numAccnts
end;

procedure close*(var a: account);
begin ^close(a);
  numAccnts := numAccnts - 1
end;

begin {initialization of M1}
  numAccnts := 0
end.
```

Figure 2. Embedding module *M0* into module *M1.*

Module *M1 extends* the definition of the inherited type *account* with an additional field to represent the account *number*. The extended type definition comprises two fields: *balance* (inherited) and *number* (extended). In module *M1*, the extended type definition *overrides* the type definition inherited from *M0*. In the embedding module *M1*, any variable (or parameter) of type *account* that is inherited from the embedded module *M0* comprises both fields *balance* and *number*. In *M1*, for example, the parameter *a: account* of procedure *open* has components *a.balance* and *a.number*. As a general rule, a record type definition from an embedded module can be overridden by an extended definition in the embedding module.

Furthermore, module *M1* overrides the bodies of the inherited procedures *open* and *close* (see Figure 2). In *M1*, the overriding body of *open* increments the total number of accounts and assigns an account number to the newly created definition of the account type; the newly defined body of *close* decrements the total number of accounts. Note that the bodies of *open* and *close* inherited from *M0* are still available in *M1* and can be invoked through the designators *^open* and *^close*.

```
module client;
import M1;
var  account: M1.account;
begin {client}
  M1.open(a);  M1.transact(a, 100);
  M1.transact(a, -50);  M1.close(a);
end.
```

Figure 3. Client module.

A *client* can import a module (see Figure 3) and use its exported entities through qualified identifiers; however, the client is not permitted to override them. More differences between module import and module embedding are discussed later in the paper.

In summary, module embedding has the following properties:

- the body of a procedure inherited from an embedded module can be overridden in the embedding module;
- the definition of a record type inherited from an embedded module can be overridden with an extended definition in the embedding module.

We envision several benefits from module embedding. Module embedding is indispensable when modules and classes are used in

combination, and, therefore, should be extended together (as illustrated by the bank account example). In particular, module embedding can be applied to:

- expand the set of global variables related to a class and extend the class itself;
- expand the set of procedures related to a group of classes and extend the classes themselves;
- define and implement extensible typeless entities, such as abstract data structures or libraries of functions.

In the rest of this paper we (1) define module embedding, (2) describe an implementation scheme for embeddable modules, and (3) discuss related work and advantages of module embedding.

2 Embeddable Modules

Definition of Module Embedding

An *embeddable module* (sometimes shortly referred to as *module* in this paper) is a collection of declared constants, types, variables, and procedures/functions (Figure 4). The module can also include a sequence of statements used for initialization. Some of the declared entities are *exported* by the module and can be used by client modules, while the non-exported entities remain *private* in the module. Technically, identifiers of exported entities are marked with a '*' sign. Exported variables are *write-protected* in client modules (i.e., importing modules) but can be updated in embedding modules.

```
module M0;
  const
    ... constant declarations...
  type
    … type declarations...
  var
    exported*: typeIdent;
    private: typeIdent;
  procedure dynamic*;
  begin ... end;
  procedure static;
  begin ... end;
begin
  … module initialization ...
end.
```

Figure 4. Principal parts of an embeddable module.

```
module M1(M0);
  ... new declarations...
  var private : typeIdent;
  procedure dynamic*;
  … overrides M0.dynamic…
  begin ... end;
  procedure static;
  … new private procedure …
  begin ... end;
begin
   ... initialization
end.
```

Figure 5. Module embedding.

Modules are *embeddable*, i.e., one or more existing modules can be embedded in a newly declared module. For example, module *M0* from Figure 4 is embedded in module *M1* in Figure 5. The embedding module *M1 inherits* all components of its embedded module *M0*. Only identifiers that are exported by *M0* are visible in the embedding module *M1*; such identifiers are *re-exported* by *M1*. Besides, the embedding module may declare new identifiers in addition to those inherited from its embedded modules. Newly declared identifiers must be different from identifiers that are exported by embedded

modules but can be the same as their private identifiers.

A procedure identifier exported by an embedded module *M0* can be re-declared in its embedding module *M1*, provided that the procedure heading in *M1* is the same as in *M0*. The newly declared procedure body *overrides* the procedure body inherited from the embedded module. For example, both module *M0* and module *M1* contain private *static* procedures that do not interfere, while the public *dynamic* procedure declared in *M1* overrides the public *dynamic* procedure declared in *M0.* Any call of procedure *dynamic*, including calls from within the embedded module *M0*, will invoke the procedure body declared in *M1*.

```
module M0;
  type class* = record
     a*: integer;
  end;
  var object*: class;
begin
  object.a := 0;
end.
```

```
module M1(M0);
type class* = record
     b*: integer;
  end;
begin
  object.b := 0;
end.
```

Figure 6. An *object* of an extensible type.

Furthermore, a record type identifier exported by an embedded module *M0* can be re-declared in its embedding module *M1*. A record type definition in *M1* extends the definition inherited from *M0*; the extended definition comprises all fields originally specified in *M0* and, in addition, the fields specified in *M1*. The extended type definition *overrides* the type definition inherited from *M0.* Consider, for example, a *class* declared in *M0* and extended in *M1*, and an *object* exported by *M0* (Figure 6). Although the *object* is originally declared in *M0*, when inherited by *M1* it contains all fields that belong to the extended *class*.

A newly declared module can embed one, two, three, or more existing modules (provided the exported name spaces of the embedded modules are disjoint). In other words, embedding is a form of *multiple inheritance* that applies to modules. Such form of multiple inheritance can be implemented in a relatively straightforward manner because modules, in contrast to classes, are typeless single instances that do not require run-time dispatch tables. As far as record type extension is concerned, we prefer to limit it to single record types, i.e., to *single inheritance*, for the sake of conceptual simplicity and efficient implementation.

Module Compilation, Execution, and Import

Embeddable modules are *separate compilation* units and *separate execution* units. The implementation converts a correct module 1) into a symbol file that represents the module's interface and 2) into an object file. At run time, the implementation executes a module by loading its object file and then executing the module body. For example, module *M0* (Figure 4) can be compiled and then executed. A module can be directly executed for either testing the module, or because that module is intended to be used as a *main module*, or *main program* in traditional terms. This approach eliminates the need for a special linguistic construct for main programs. A major benefit is that a main module (i.e., a main program) is as extensible and adaptable as any other module.

A client module specifies a list of imported modules, as for example *module C0; import M0; ... end.* The client, *C0*, can refer to

exported entities only through qualified identifiers, such as *M0.exported* and *M0.dynamic*. At run time, the implementation executes imported modules before the execution of the client. By definition, imported modules are *shared* between client modules. At run time, the implementation loads only one instance of each imported module and this copy is shared by all of its clients.

The execution of a module body is preceded by the execution of the bodies of its base modules, if any. This rule ensures proper module initialization.

Comparison of Module Import and Module Embedding

The principal difference between module import and module embedding is that *module import* implements *shares-a* relationship between modules while *module embedding* implements *contains-a* relationship.

Consider for example an *application* in which modules *M1* and *M2* (Figure 7) import *M0*. The imported module *M0* is shared between modules *M1* and *M2*. Any change of *M0* by, say, *M1* is visible for *M2* as well.

Alternatively, if module *M0* is embedded into modules *M1* and *M2*, each of these modules will have *M0* as its proper part (Figure 8). Now, *M1* and *M2* incorporate separate instances of *M0*. Therefore, *M1* may change components inherited from *M0*, but these changes do not affect the same components inherited by *M2*. Besides, *M1* and *M2* can make different extensions of the same record type inherited from *M0*. Likewise, *M1* and *M2* can have different procedures override the same procedure inherited from *M0*.

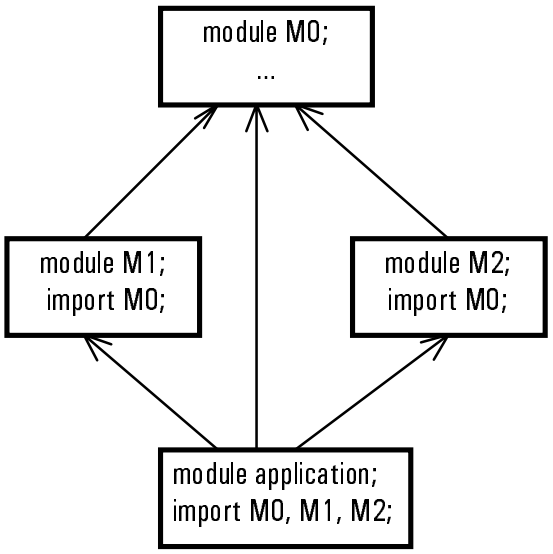

Figure 7. Import implements *shares-a* relationship (graphically represented by arrows).

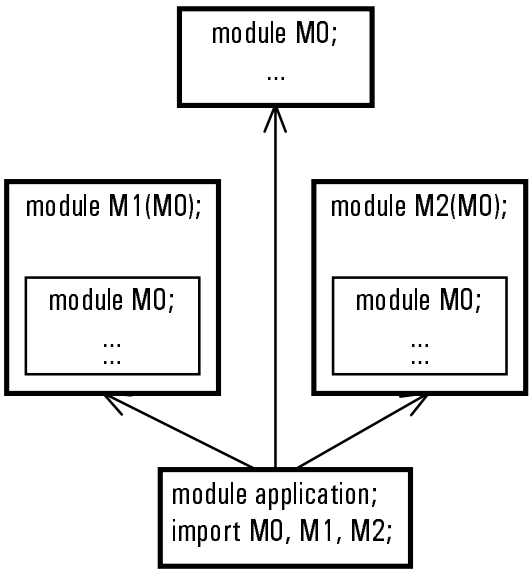

Figure 8. Embedding implements *contains-a* relationship (graphically represented by nesting).

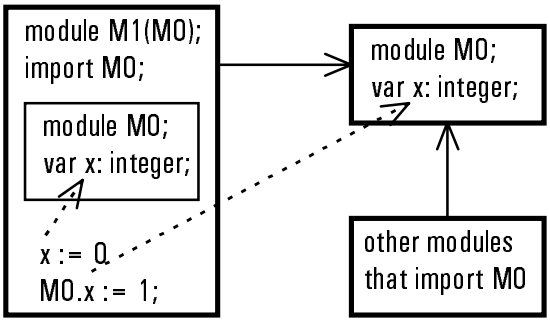

Figure 9. Mixing *contains-a* and *shares-a* relationships.

Note that module import and embedding can be mixed, if necessary in order to implement shares-a and contains-a relationships. For example, a module *M1* can import module *M0* and, at the same, embed module *M0* (Figure 9). In this case, *M1* embeds a separate instance of module *M0* and also refers to the shared instance of the imported module *M0*. Note that the qualified identifier *M0.x* stands for an imported component, while the identifier *x* always stands for a component that is inherited from an embedded module.

3 Implementation Issues

A Language with Modules Embedding and its Implementation

We have incorporated embeddable modules in an experimental object-parallel language. Our embeddable modules allow gluing *together* sequential domain-specific code with ready-to-use generic parallel algorithms, in order to effectively build parallel applications. Technically, a generic parallel algorithm is specified as an embeddable module which implements a common synchronization and communications structure, such as a pipeline, a grid, a master-server structure, etc. This module can be *embedded* into domain-specific modules that contain only sequential code. Thus, a concrete parallel application is derived from the paradigm by embedding the paradigm module into a module with domain-specific sequential code.

Generic parallel algorithms, also called *parallel paradigms*, were introduced and studied by Brinch Hansen [5] in terms of the

structured language SuperPascal [4]. This type-safe parallel language is based on some widely accepted parallel programming means, such as *send, receive, for-all*, and *parallel* statements, and *channel* types. In SuperPascal, a paradigm can only be implemented as a concrete main program using concrete constants, types, and procedures. In order to mix a paradigm with user supplied sequential code, the user must have the source code for the paradigm and make textual modifications, such as adding new program components and modifying existing ones. The *Paradigm/SP* language that we have developed enhances the parallel-programming features of SuperPascal with embeddable parameterized modules. We have studied known parallel paradigms and developed new ones, and our conclusion is that embeddable modules provide easy to use support for parallel paradigms, while traditional classes may lead to representations that are unnecessarily complex. Furthermore, module embedding, and procedure and type overriding make it possible for a general 'paradigm' module to be easily mixed with sequential problem-specific code, i.e., adapted to a specific application.

We have developed a prototype implementation of the *Paradigm/SP* language. The prototype implementation consists of (1) a compiler that generates abstract code and (2) a loader and an interpreter for this abstract code. The implementation is an enhancement of the SuperPascal compiler [5] and its predecessor [3] with separate compilation, dynamic binding for methods and classes, and dynamic loading. The implementation reuses some algorithms from the Oberon compiler [15]. The compiler incorporates two independent components, the parser and the assembler. The parser performs

traditional recursive descent syntax and semantic checks, outputs a symbol file, and generates intermediate code. This intermediate code is then processed by the assembler, which performs optimization and generates an abstract object file. Finally, abstract object files are dynamically loaded and then executed by the interpreter. A working version of the prototype implementation is available in [18].

The abstract code of a module consists of two parts, *initialization* code and *proper* code. The proper code is a direct compilation from the module statements. The initialization code is executed by the dynamic loader and is used to set-up several types of descriptors associated with each loaded module. *The import descriptor* associates module numbers with the effective base addresses of imported and base modules. The *procedure descriptors* and the *type descriptors* are discussed in the next two sections.

The initialization code of all modules from an application is executed prior the execution of any proper code. Besides, code of an embedded module is executed prior code of its embedding modules; code of an imported module is executed prior code of its clients.

Type Overriding

An extensible record type is bound at run time to a particular record definition (i.e., to a particular set of fields). The length of the type can be different in different applications. Even in the same application, an extensible record type can have different definitions with different lengths. Assume, for

example, that a record type *T* is declared in module *M0* (Figure 8) and re-defined in modules *M1* and *M2*. In the application shown in Figure 8, the record types *M0.T*, *M1.T*, and *M2.T* will be bound to three different type definitions in modules *M0, M1,* and *M2* correspondingly.

A *dynamic type* is either an extensible record type, or a type with one or more components of a dynamic type. Objects of dynamic types are implemented as pointers and are automatically allocated and deallocated heap memory.

For each dynamic type, the compiler generates a *type descriptor* that belongs to its declaring module and that binds the type to a particular type definition. The type descriptor contains the type length and references to the type descriptors of all dynamic components of that type. For example, the principal components of an extensible record type descriptor are:

Record-Type-Descriptor = Record-Type-Tag Record-Length Dynamic-Field-Descriptor-References

The displacement of this type descriptor regarding the base address of the declaring module is kept in the symbol file.

The initialization code of the declaring module contains instructions that assign into the type descriptor the record length and references to the dynamic field descriptors. When an extended module re-defines the type, its initialization code updates the type descriptor with the length of the extended type definition and with references to the newly added dynamic record fields.

For each whole dynamic variable (or value parameter), the compiler generates

- an instruction to allocate a memory bock for the whole variable and of its dynamic components when the variable scope is entered;
- an instruction to deallocate that memory when the scope of the variable is exited.

All deallocated blocks of the same type are kept in a list of free blocks and are re-used for next memory allocations.

A type that is local for a module is not visible in its embedding modules; by definition, such a local type is not extensible. However, a local type can still contain dynamic components and is, therefore, implemented as a dynamic type.

Procedure Overriding

The implementation separates user declared procedures in two categories. Procedures that are marked for export are *external* because they can be called in client modules and can be re-defined in embedding modules. All other procedures are *local* for their declaring module.

An external procedure is bound at run time to a particular procedure body (i.e., particular implementation). For each originally declared procedure, the compiler generates a *procedure descriptor* that belongs to the declaring module and that binds the procedure to a particular procedure body. The procedure descriptor contains the items that are needed for an external procedure call, namely a reference to the procedure body and the base address of the module containing that body:

Procedure-Descriptor = Module-Base Procedure-Body-Reference

The initialization code of the declaring module contains an instruction that assigns the module base and a reference to the procedure body into the procedure descriptor. When an embedding module redefines the procedure, its initialization code updates the procedure descriptor with the base address of the embedding module and a reference to the newly declared procedure body. Since an embedded module is always loaded before its embedded modules, the procedure descriptor refers to the very last implementation of the procedure. A call of the procedure from any module invokes that last version.

An external procedure call is compiled into an instruction with two arguments. The first argument is the *number* of the declaring module (i.e., the module that contains the procedure descriptor). The second argument is the *offset* of the procedure's descriptor regarding the base address of its declaring module:

Procedure-Call = Instruction-Code Declaring- Module-Number Descriptor-Offset

At run time, the *entry point* of an external procedure is calculated by fetching the *base address* of the declaring module from the import descriptor and then fetching the contents of the procedure descriptor. The procedure descriptor is then used to establish the entry point of the procedure body and the base address of the module that contains that body. Such an invocation is nearly as efficient as a procedure call through a pointer:

Declaring-Module-Base-Address = Import-Descriptor [Instruction . Declaring-Module-Number]
Procedure-Descriptor-Address = Declaring-Module-Base-Address + Instruction . Descriptor-Offset
Procedure-Entry-Point = Procedure-Descriptor . Procedure-Body-Reference

Module-Base-of-Procedure-Body = Procedure-Descriptor . Module-Base

Local procedures are bound to their declaring module at compile time. A local procedure call is compiled into an instruction that contains the displacement of the procedure entry address regarding the location of the calling instruction itself.

4 Related Work and Conclusions

Embeddable modules are useful for the implementation of typeless structures, such as, for example, common libraries of procedures or generic algorithms. Alternatively, *classes* are needed when it comes to the implementation of multiple entities of the same abstract type. In modular object-oriented languages, such as Oberon-2 [6], classes are represented by means of extensible record types [13].

```
MODULE M0;
  TYPE class* = RECORD
    ... data fields ...
  END;
  PROCEDURE (VAR self: class) method*(…);
  BEGIN ... implementation ... END;
END.

MODULE M1;
IMPORT M0;
  TYPE subClass* = RECORD(M0.class)
    ... new data fields ...
  END;
  PROCEDURE (VAR self: subClass) method*(…);
  BEGIN ... type guards/tests ... END;
END.
```

Figure 10. Methods as type-bound procedures.

As illustrated in Figure 10, methods are represented as type-bound procedures that are syntactically connected to an extensible record type, i.e., a class [6]. Other modular languages, such as Ada-95

[10, 11] and Modula-3 [2], follow similar approaches to classes and methods.

```
module M0;
  type class* = record
    ... data fields ...
  end;
  procedure method*(var self: class; ...);
  begin ... implementation ... end;
end.

module M1(M0);
  type class* = record
    ... new data fields ...
  end;
  procedure method*(var self: class; ... );
  begin ... end;
end.
```

Figure 11. Methods as module-bound procedures.

Similarly to known modular languages, we represent classes as extensible record types. What is different in our approach is that record extension overrides an existing type and does not introduce a new type. Furthermore, we represent methods as module-associated procedures rather than type-bound ones; module-associated procedures can be overridden similarly to type-bound procedures, as specified earlier in this paper. Module-associated procedures can be related to extensible record types, i.e., to classes, through regular parameters and therefore, used as methods for those classes. A module *M0* (Figure 11) can implement a *class* as an extensible record type and a *method* as a procedure. Furthermore, a module *M1* that embeds *M0* can extend the *class* and override the inherited *method*. Objects that belong to the *class* can be passes as parameters to the *method*. Such extended objects do not require type test and guards because they are not polymorphic, in contrast to

extensible objects in Oberon ([8] offers a detailed discussion of type tests and guards). A disadvantage of our proposed extensible records is that they do not support heterogeneous data structures.

The purpose of the module system of the Cecil language [17] is to support encapsulation and static type checking for multimethods. Cecil supports explicit module import that obeys visibility rules similar to those adopted in C++ subclassing. As analyzed in [17], the standard visibility rules of module imports can make static subchecking of multimethods impossible and force dynamic typechecking. In order to ensure static subchecking of multimethods, a form of module import called module extension is proposed in [17]. In contrast to our proposed concept of module embedding, in Cecil extended modules are shared, just like imported modules. Furthermore, extended or imported modules in Cecil allow standard subtyping while our embeddable modules allow type redefinition.

It has been widely recognized [9, 7, 1, 6] both modules and classes support necessary abstractions, which should be, used as complementary media. What we propose is to *shift power* from classes to modules by introducing a form of inheritance that applies to modules. The object-orientation of *Paradigm/SP* is founded on enhancements of traditional concepts, such as module, record type, and procedure. In this, we agree with authors who prefer using widespread and historically established terms with object-enhanced syntax and semantics [12].

Embeddable modules seem to be conceptually simpler than classes and, therefore, are easier to use. In particular, an

embeddable module can be a good replacement for a single-entity class, or a singleton pattern [16]. Because of its relative simplicity, module embedding is easier to implement than sub-classing. For example, multiple inheritance for modules is easier to implement efficiently than for classes because modules are not first-class objects (there are no module types and module references). In particular, invoking a procedure that is inherited from one of several embedded modules is as efficient as calling a procedure through a pointer.

A disadvantage of embeddable modules as compared to classes is that modules do not introduce types and, therefore, cannot be used to create multiple instances. Furthermore, dynamic types impose run-time overhead on the implementation. Finally, a drawback of module embedding is that it complicates the semantic and the syntax of the underlying modular language. Nevertheless, we believe that the introduction of module embedding is justified, because known code reuse mechanisms, such as parametric polymorphism (i.e., generics), interfaces, procedure parameters or pointers cannot completely achieve the effects of module embedding.

The existence of two principal object-oriented styles has been distinguished in the recent years. The traditional and older style is centered on classes as abstractions, and on objects as concrete instances of such abstractions. Such a *class-based* object style is supported by a number of pure or hybrid object-oriented languages, such as Smalltalk, Simula, C++, Java, and Oberon-2. The alternative object style is centered around concrete objects and on the possibility to use such objects as prototypes from which other

objects can be derived. Such a *prototype-based* object style is supported by newer languages, such as Self and NewtonScript. One of the strengths of the *Paradigm/SP* language is that it supports to some extent both class-based and pattern-based object-styles. Indeed, embeddable modules are relevant medium for pattern-based OOP, while extensible record types support class-based OOP. Therefore, *Paradigm/SP* is a hybrid object language, which, by the way, is a relevant tool for the traditional structured programming style as well.

5 References

1. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Technical report UUCS-92-007, University of Uta, 1992.
2. L. Cardelli, J. Donahue, L. Glasman, M. Jordan, B. Kalsow and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27, No8, 1992, 15-42.
3. B. Hansen. *Brinch Hansen on Pascal Compilers*, Prentice Hall, 1985.
4. B. Hansen. SuperPascal - a publication language for parallel scientific computing, *Concurrency - Practice and Experience*, 6, No 5, 1994, 461-483.
5. B. Hansen. *Studies in Computational Science: Parallel Programming Paradigms*. Prentice Hall, 1995.
6. H. Moessenboeck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.

7. C. Lucas and P. Steyaert. Modular inheritance of objects through mixin-methods. In *Advances in Modular Languages,* University of Ulm, 1994, 273-282.

8. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. ACM Press, 1992.

9. C. Szyperski. Why we need both: Modules and classes, in *Proceedings of OOPSLA*, 1992, 19-32.

10. S.-T. Taft. Ada 9X: A technical summary, *Communications of the ACM*, 35, No 11, 77-84.

11. T.-S. Taft. Ada 9X: From abstraction-oriented to object-oriented, *Proceedings of OOPSLA*, 1993, 127-135.

12. J. Winkler. Objectivism: "Class" considered harmful, *Communications of the ACM*, 35, No 8, 1992, 128-130.

13. N. Wirth. Type extensions, *ACM Transactions on Programming Languages and Systems*, 10, 1987, 204-214.

14. N. Wirth. The programming language Oberon, *Software - Practice and Experience*, 18, No 7, 1988, 671-690.

15. N. Wirth and J. Gutknecht. *Project Oberon*, ACM Press, 1992.

16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley, 1995.

17. C. Chambers and G. Leavens. Typechecking and Modules for Multimethods, ACM Transactions on Programming Languages and Systems, 17, No 6, 1995, 805-843.

18. A. Radenski (1998). Prototype Implementation of Paradigm/SP,

http://www.rtpnet.org/~radenski/research/language.html.