SOFTWARE ENGINEERING ASPECTS IN LANGUAGE IMPLEMENTATION

Kai Koskimies*
GMD Forschungstelle an der Universität Karlsruhe
Haid-und-Neu Strasse 7, 7500 Karlsruhe, BRD

Abstract

Current implementations of programming languages are modularized according to implementation techniques rather than to the natural units of the language to be implemented. It is argued that such implementations have poor software engineering qualities like reusability and maintainability. A syntax-directed, modular language implementation technique is sketched, based on an object-oriented view of the source language. The technique makes use of the extended types offered by the language Oberon.

1. INTRODUCTION

Typical language implementation software consists of large, complex programs. One could think that traditional software engineering issues like reusability, maintainability, and modularity would play a particularly important role in the developing of such software. Hence it is somewhat surprising to note that practical language implementations are usually bulky, monolithic programs that have relatively weak software engineering qualities. A typical one-pass compiler (see e.g. [WMK80]) consists of a single module taking care of the entire analysis of the source text and synthesis of the target program, assisted by some smaller service modules for scanning, symbol table, etc. The main module may be structured, but it basically follows the programming style of the 60's. This kind of module division is virtually the same for all languages, independently of the size of the language. Hence for sizable languages the main module becomes very large (say, over 10000 lines), exceeding the reasonable limit of a software unit.

The reason for this unsatisfactory situation is probably the fact that programming languages are regarded as complex, indivisable objects. Hence, from the software engineering point of view the language to be implemented is a black box: the implementor tends to think in terms of services needed by the translation process, and does not try to divide the language itself into logical pieces that would be represented by separate modules.

In this paper we try to demonstrate that the implementation of programming languages can be cut into pieces in a sensible and natural way. In particular, we will study how two popular software engineering paradigms could be applied in language implementation, namely *object-oriented programming* and *modular programming*. Both of these techniques contribute to the division of the language implementation into software units that closely correspond to the concepts and structures of the source language as they are presented e.g. in the reference manual. We expect that this will improve especially the maintainability and reuse of language implementation software. Naturally, this topic is not new in the sense that modular and/or

* On leave from University of Helsinki, Finland. This work is in part supported by Deutsche Forschungsgemeinschaft and by the Academy of Finland.

object-oriented languages have been used in language implementation for a long time. The (perhaps) new aspect of this study is the systematic syntax-directed method of modularization. The ideas presented here originate to some extent from the object-oriented language implementation system TOOLS ([KoP87], [KELP88]), and from the experiences in using it [KoM88].

The syntactic structure of a programming language is the most obvious basis for modularizing the implementation. Although many computer philosophers regard the syntax as something irrelevant and superficial, it is nevertheless the only concrete form of manifestation of the language. An older programming paradigm, *structured programming,* has been applied to language implementation in a syntax-directed way, and the result has been one of the most successful language implementation strategies, recursive descent. However, instead of mapping the various parts of a language into procedures we will map them into separately compiled modules. These modules will usually be interpreted as representatives of classes in the sense of object-oriented programming.

We will proceed as follows. Chapter 2 studies our syntax-directed view of the objects constituting a program in detail, with respect to structures found in Pascal-like languages. In Chapter 3 we will present the basic method for modularizing language implementation, and Chapter 4 provides an example of the application of this method. Finally, in Chapter 5 we will discuss the advantages of the method. We assume that the reader is familiar with the concepts of object-oriented programming (e.g. [Weg87]).

2. SYNTAX-DIRECTED OBJECTS

Notion objects

Let us consider a nonterminal symbol of a context-free grammar as the name of a class. An instance of such a class represents a realization of a particular language notion; we call these instances *notion objects.*

How well do the notion objects cover the natural objects that constitute a program? First, we note that many of the notion objects may be needed only for syntactic analysis (possibly required by a particular parsing method), or for some trivial information passing. The other, more essential notion objects constitute something which is near to an abstract representation of the source text. However, here we will not make any distinction between "essential" and "non-essential" notion objects (or nonterminal classes).

Second, sometimes there is no clear one-to-one correspondence between syntactic notion objects and logical language objects. First, there may be "syntactic discontinuity" so that a logical language object is represented by disconnected pieces of text. In such a situation it may be difficult to assign naturally a particular nonterminal for the class of the objects, although the objects have clearly a syntactic representation. A typical example is the grouping of variable declarations that have the same type:

```
VariableDeclaration ::= 'VAR' IdList ':' TypeId
IdList ::= IdList ',' Identifier | Identifier
```

A single variable declaration (representing a variable object) consists of an identifier and of the type denotation following the list of identifiers; between these pieces of text there may be an unbounded number of other identifiers. Note that in this case a piece of text (the type denotation) is shared by several objects.

In some languages (e.g. Ada) the above form of variable declarations is viewed as a short-hand notation for a sequence of declarations of single variables. The expanded form would of course eliminate the problem. In general, short-hand notations may cause problems in finding the syntactic counterparts for the classes of the language definition.

Third, in some cases a nonterminal instance actually represents a set of objects. This situation arises when the nonterminal acts either in the role of a *metaclass* or in the role of a *collective class*.

An instance of a metaclass nonterminal corresponds to a class definition so that an unlimited number of objects may be generated during run-time as instances of the nonterminal instance. A trivial case of a metaclass is a nonterminal which produces something that corresponds to a class in the source language, e.g. class definitions in Simula. In general, if a language concept has the property that it can be instantiated, the corresponding nonterminal belongs to the category of metaclasses. Such concepts include e.g. type and procedure definitions in Pascal-like languages.

Metaclass nonterminals are usually associated with other nonterminals that produce the instantiation structure. For example, in Pascal variable declarations and calls of the standard procedure "new" are used to create instances of types, and procedure calls are used to instantiate procedures. The notion objects created in response to these nonterminals are "generative" in the sense that the activation of some of their methods will give rise to an instance of a class defined by an instance of a "metaclass nonterminal".

An instance of a collective nonterminal class represents invariably a fixed group of objects. The most frequent example is a list structure (declaration list, statement list etc.); in this case the objects belonging to the group (i.e. the list elements) are represented by other notion objects. However, in some cases the objects belonging to the group are not syntactically represented. Such a situation may arise e.g. in the declaration of a structured variable, in which the components are not represented by distinct pieces of text.

Note the difference between collective classes and metaclasses: an instance of a collective class is a set of objects, possibly of different classes, while an instance of a metaclass is a class that may or may not be later used for creating instances of this class.

Finally, in some cases a natural object is not at all represented by a notion object, not even via a metaclass nonterminal. A trivial example is the standard environment in Pascal (i.e. standard constants, types, etc.). However, in this case the additional syntactic structures can be easily introduced, generating an empty string at the beginning of the program. If a class has no association with syntax, we call it a *semantic class*. In Pascal-like languages the existence of such classes is not quite obvious, although a particular implementation technique may require them.

Class hierarchies

Although the hierarchic relations implied by the syntactic production rules provide a seemingly obvious basis for establishing a class hierarchy for nonterminal classes, this idea makes sense only in certain special cases. The fact alone that production rules are usually recursive makes it impossible to build a general class hierarchy on this basis. However, sometimes a production rule exhibits a classification of nonterminals that can be understood as an introduction of a superclass. We will call these *superclass productions.* In standard Pascal (BSI[82]), such productions are e.g.:

```
new-type::= simple-type | structured-type | pointer-type
simple-type::= ordinal-type | real-type
ordinal-type::= enumerated-type | subrange-type | ...
structured-type::= ["packed"] unpacked-structured-type |
structured-type-identifier
unpacked-structured-type::= array-type | record-type | ...
```

These productions actually define the hierarchic type system in Pascal: "simple-type", "structured-type", and "pointer-type" are subclasses of "(new-)type"; "ordinal-type" and "real-type" are subclasses of "simple-type" etc.

In general, a superclass production is of the form

```
S ::= C1 | C2 | ... | Cn
```

where S is the *superclass nonterminal* and C1, C2, ... , and Cn are *subclass nonterminals.* Typical superclass productions can be found easily in Pascal:

```
simple-statement::= empty-statement | assignment-statement | ...
structured-statement::= compound-statement |
conditional-statement | ...
conditional-statement::= if-statement | case-statement
```

Except for types and statements, superclass productions can be found e.g. for operators and elementary value accessing:

```
multiplying-operator::= "*" | "/" | "div" | "mod" | "and"
factor::= variable-access | unsigned-constant | ...
```

In the first case subclasses are represented directly by terminal symbols. In the second case the subclasses are intuitively not so clear, partly because the naming conventions of various factors do not emphasize the common aspect.

In Pascal declarations must be given in a fixed order, implying that there is no superclass production for declarations. However, if such a restriction were not imposed by the syntax, a superclass production would appear in the form:

```
declaration::= constant-declaration | type-declaration | ...
```

Note that superclass nonterminals should not give rise to separate notion objects: the logical language concept is represented by the most accurate nonterminal. Hence, a chain of superclass nonterminal instances in the syntax tree should be regarded as a single instance of

the lowest-level class; the instances of the superclass nonterminals then represent only different levels of the object.

Attributes and methods

The attributes (local data) and methods of notion objects vary considerably depending on the nature of the language concept represented by the nonterminal. First we note that the attributes are usually divided into two categories: those contributing to the structural relations of the objects (structural attributes), and those describing some non-structural properties of the objects. The first category essentially establishes an abstract tree structure of the source text. Note that the attributes do not merely provide local connections, but also non-local. For example, a nonterminal instance denoting a variable reference in an expression probably has an attribute referring to the variable object, which might be represented by a nonterminal instance far from the expression. However, we consider such attributes non-structural: structural attributes are related only to the substructures.

Maybe the most notable special feature of notion objects (when compared to the conventional objects) is the creation operation: instead of an explicit operation these objects are assumed to be created implicitly on the basis of some pattern matching process applied to an input text. Normally this is done through parsing. In the top-down approach discussed in the following section the creation operation can be directly implemented as the parsing procedure of the nonterminal in question.

A typical method possessed by many notion objects is a check on the values of the non-structural attributes (or sometimes on the non-structural attributes at the objects referenced by other attributes). Such a check is usually required by the "static semantics" of the language, and can be carried out immediately after the creation of the object. These checks concern e.g. various conditions imposed by the type system, like type compatibility rules. Since the static checks can be carried out at creation time, it is natural to consider them as part of the creation operation. Some checks cannot be done until at run-time because they depend on dynamic information; these checks can be usually regarded as part of the method describing the dynamic behaviour of the object in question. For example, a range check may be necessary in connection with the execution of an assignment statement in Pascal.

Besides the creation operation and the possible associated checking operation, notion objects very seldom seem to have more than one method. Normally each language concept has a single role or activity throughout the interpretation of the source text; indeed, this should probably be a basic language design principle. Typical methods are e.g. the evaluation of an expression or some part of it, or the execution of a statement. In a compiling implementation the corresponding methods would produce the code for executing these actions.

3. MODULAR LANGUAGE IMPLEMENTATION METHOD

In a conventional recursive descent compiler the main module consists of a set of procedures, one for each nonterminal of the source language. The main module is assisted by several "implemetation service" modules for scanning, symbol tables etc. Our basic idea is to reverse

this scheme: we want to have a separate module for each nonterminal, and to provide all the implementation services in one module (see Fig. 1 and 2). The latter module is assumed to be the same for all implemeted languages; hence it can be viewed as a kind of a language implementation system, although in a slightly different sense than the conventional metacompilers.

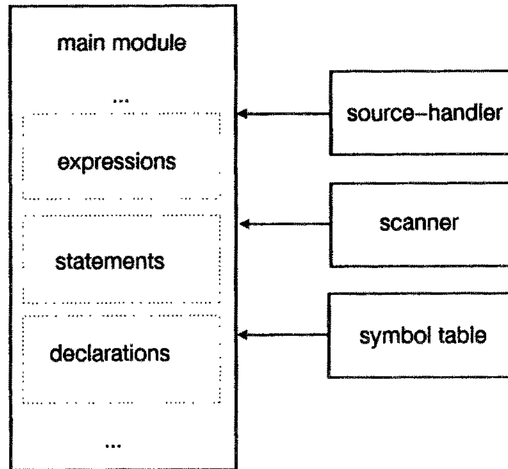

Fig. 1. Conventional module division

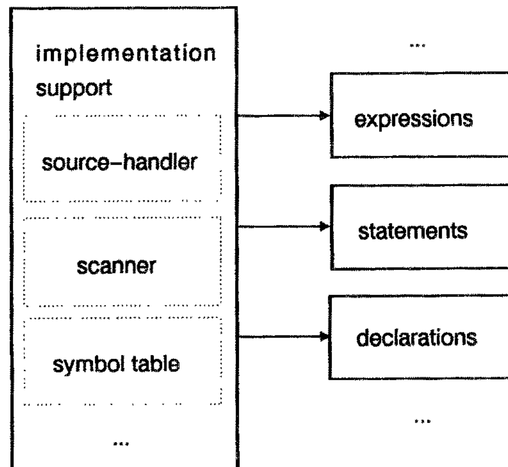

Fig. 2. Source language oriented module division.

The nonterminal modules are viewed as the specifications of the corresponding nonterminal classes. Each such module will therefore define in its visible part the class type, which will always be a pointer to a record consisting of the attributes of the notion objects, and the

methods of the class. One of these methods must be the creation operation, but otherwise they can be freely chosen.

Implementation language

We will present the method using Wirth's new language Oberon [Wir88a]. It turned out that the type extension mechanism provided by this language [Wir88b] fits very nicely with the requirements of our task. Two problems arise when a conventional modular language (like Modula–2 or Ada) is used. First, structural attributes cannot be handled naturally because of the rigidity of the type system. If we want that the class type is defined by each module, then the class type definition of module A needs the class type of module B, if A–objects need to refer to the B–objects. Since the class type must be given in the definition part, this means that the definition part of A must import B. However, since nonterminals may be recursive, it may be that B also needs to refer to A–objects (possibly indirectly through a chain), and therefore the definition part of B must also import A. This leads to forbidden circular importing. In Ada the problem is particularly difficult because even the opaque types must be completely defined in the definition part (in the private section). In Modula–2 the problem can be solved by making the class type an opaque type, and by defining the attributes in the implementation part. However, this would mean that *all* attributes are invisible for the outsiders, which in turn leads to the introduction of additional methods only for reading the values of some attributes.

In Oberon a visible record type, defined in the definition part of a module, can be extended with a non–visible part in the implementation module. This is exactly what we want: the structural attributes can be conveniently represented in the non–visible extension. Note that the structural attributes should indeed be invisible; they are needed only for implementing some of the methods of the class, and should not be touched by the outsiders.

The other problem concerns the construction of class hierarchies. In Ada or Modula–2 such hiearchies must be presented using variant records in a rather clumsy way. This can be done e.g. by defining all the subclasses in the uppermost level (that is, in the root class) as a variant record, and by importing this type to all the subclasses. Such a solution is however very unnatural, and it in fact breaks the modularization by introducing implicit dependencies between the modules.

In Oberon, the superclass module must also be imported by the subclass module, but the subclass can be defined simply as an extension of the superclass. Hence the attributes of each class level can be naturally defined in the right place. There are no hidden dependencies between modules: the superclass module need not know anything about the subclasses.

We will assume the existence of a module which provides all the necessary language–independent implementation services; this module is called "Support". These services include at least support for scanning, for deterministic modular recursive descent parsing (as proposed in [Kos88]), a general object retrieval mechanism, and a collection of useful data types (e.g. lists). In fact, some of the facilities of this module can be viewed as "system" classes, so that certain nonterminals will be their subclasses. For example, a declaration can be viewed as a subclass of "the class of objects to be found on the basis of an identifier"; the latter class is provided as a type of "Support". Similarly, a statement list structure can be presented as a subclass of a general list.

Again, the presenting of such system classes becomes very natural in Oberon. For example, the class record of a declaration can be defined as an extension of an element type (exported from "Support"), allowing these declarations to be inserted into a general-purpose object-base with a fast search mechanism.

Construction of modules

For each nonterminal, the definition module is constructed as follows:

1) Import list:

   If the nonterminal is a subclass nonterminal, import the module of the superclass nonterminal. If the nonterminal is a subclass for a system class, import "Support".

2) Data part:

   Define the following type:

   type Attr = **record** (S)
   A_1: T_1;
   ...
   A_n: T_n;
   end;

   where S is either the attribute record type of the form X.Attr of the superclass nonterminal X, or some system class of the form Support.C, and $A_1...A_n$ are the visible attributes of the instances of the class. This type describes all the data contained by the objects. The actual class type is defined as:

   type Class = **pointer to** Attr;

   so that objects will always be represented by dynamic variables.

3) Methods:

   Define the heading of the creation operation as

   procedure Create(): Class;

   Define the headings of other methods, possibly with parameters of type "Class".

For each nonterminal, the implementation module is constructed as follows:

1) Import list:

   Import all the modules of the syntactic substructures of the nonterminal. If the syntactic structure contains terminal symbols, or if the methods otherwise make use of "Support", import it.

2) Data part:

   If the notion objects have structural attributes, define an extension of Attr:

```
type Attr = record (S)
    A1: T1;
    ...
    An: Tn;
    X1: U1;
    ...
    Xk: Uk;
end;
```

where $X_1,\ldots,X_k$ are the structural attributes of the notion objects. Types $U_1,\ldots,U_k$ are of the form W.Class, where W is an imported module.

3) Methods

Implement the creation method in the style of a parser routine of a recursive descent parser, by calling the creation methods of the constituent structures in appropriate places. In the case of a superclass nonterminal, return the value returned by the creation method of one of the substructures (subclasses) as such. Otherwise create a dynamic variable representing the object, and initialize the attributes. Implement other methods as required.

Comments

The above rules are the basic guidelines that should be followed in the design of the modules. Sometimes additional importing is necessary, for example when an object has a non–structural attribute that refers to another object. Then the class type of some other module has to be imported into the definition module. Additional types required for defining the "Attr" type may also be necessary in the definition module.

A particular problem arises in parsing, because in principle a module should not know anything about the other modules, except what is available through the interface. Hence a nonterminal module should not make use of any information concerning the syntactic structure of other nonterminals, either. This requirement rules out conventional recursive descent parsing, where the starter and follower tokens are assumed to be known globally for the entire grammar. The problem can be solved by collecting necessary parsing information dynamically, during parsing [Kos88]. We will not discuss this method here, but we merely assume that deterministic parsing is in some way possible.

We have not paid any attention to metaclass nonterminals. The simplest way to present them is to let the generating notion object have an attribute that refers to an instance of the metaclass nonterminal. A method of the generating notion object (e.g. variable declaration) can then make use of this reference, and create an instance according to the referenced object (type definition).

4. AN EXAMPLE

As an example of the application of the modular language implementation method, we will examine the developing of an interpreter for the following toy language:

```
Prog ::= DeclList StatList
DeclList ::= (Decl ';')*
Decl ::= TypeDecl | ConstDecl | VarDecl
TypeDecl ::= 'TYPE' id '=' number '..' number
ConstDecl ::= 'CONST' id '=' number
VarDecl ::= 'VAR' id ':' id
StatList ::= Stat (';' Stat)*
Stat ::= AssStat | LoopStat | IfStat
AssStat ::= id ':=' Expr
LoopStat ::= WhileStat | RepeatStat
WhileStat ::= 'WHILE' Expr 'DO' StatList 'END'
RepeatStat ::= 'REPEAT' StatList 'UNTIL' Expr
IfStat ::= 'IF' Expr 'THEN' StatList ElsePart 'END'
Expr ::= Term (Oper Term)*
Term ::= number | id | SubExp
SubExp ::= '(' Expr ')'
Oper ::= '+' | '–'
```

For simplicity, only integers and their subranges are allowed types; logical values are assumed to be represented by integers (0: false, <>0: true). Hence type checking consists only of run–time range checks. Standard identifier "INTEGER" denotes the type of integers provided by the underlying system.

In the sequel we will examine some of the modules required for this implementation; a complete listing of the modules can be found in Appendix.

Let us first study the modules for declarations. A declaration can be viewed as a subclass of "identifiable" objects provided by the support module. Hence we write the definition part as follows:

```
definition Decl;
   import Support;
   type Attr = record (Support.EntryType)
                     name: String;
               end;
        Class = pointer to Attr;
   –– methods:
   procedure Create(): Class;
   procedure Find(Key: String): Class;
end Decl.
```

We assumed here a general string type "String". This specification implies that the common properties of declarations are 1) that they are identifiable objects, 2) that they have a string–valued name for identification, and 3) that function "Find" can be used to access an object through its name.

A type declaration is a subtype of a declaration:

```
definition TypeDecl;
   import Decl;
   type Attr = record (Decl.Attr)
                     lo: Integer;
                     hi: Integer;
               end;
        Class = pointer to Attr;
```

```
    --methods:
    procedure Create(): Class;
end TypeDecl.
```

This states only that each type has two visible attributes, the lower and the upper bound of the range. Type declarations (or types) have further a subtype, namely (the declaration of) the standard type:

```
definition StandTypeDecl;
    import TypeDecl;
    type Attr = record (TypeDecl.Attr) end;
         Class = pointer to Attr;
    --methods:
    procedure Create(): Class;
end StandTypeDecl.
```

The attributes of the standard type will be the same as for other types; hence the definition module is practically identical to type declaration. The essential difference appears only in the implementation part, because the creation method of standard types is implemented differently (i.e., without parsing).

Let us examine another class chain. A statement has the property that it belongs to a list of sequentially executable statements. Hence a statement is a subclass of list elements provided by the support module. A common method to all the statements is the execution of the statement:

```
definition Stat;
    import Support;
    type Attr = record (Support.ListElemType) end;
         Class = pointer to Attr;
    -- methods:
    procedure Create(): Class;
    procedure Execute(S: Class);
end Stat.
```

An assignment statement is further a subclass of a statement. An assignment statement has no visible attributes, but two structural hidden attributes referring to the left-hand side and to the right-hand side of the assignment.

```
definition AssStat;
    import Stat;
    type Attr = record (Stat.Attr) end;
         Class = pointer to Attr;
    -- method:
    procedure Create(): Class;
    procedure Execute(AS: Class);
end AssStat.
```

Note that in the "Stat" module the execution method only selects the appropriate lower class method on the basis of the actual class of the parameter object. This arrangement is necessary in the absence of virtual methods. Let us look at the implementation part:

```
module AssStat;
   import Support, VarDecl, Decl, Expr;
   type Attr = record
          lhs: VarDecl.Class;
          rhs: Expr.Class;
          end;
   procedure Create(): Class;
          var AS: Class;
              D: Decl.Class;
   begin  -- AssStat ::= id ':=' Expr
          New(AS);
          D:= Decl.Find(Support.ScanTC(Support.identifier));
          Support.ScanSymbol(":=");
          if (D is VarDecl.Class) and (D<>nil)
                 then AS.lhs:= D(VarDecl.Class);
                 else Support.Error("Declaration error");
          end;
          AS.rhs:= Expr.Create();
          return AS;
   end Create;
   procedure Execute(AS: Class);
   begin
          Expr.Evaluate(AS.rhs);
          VarDecl.Assign(AS.lhs, AS.rhs.val);
   end Execute;
end AssStat.
```

In this case we have a taken short-cut: "lhs" is not really a structural attribute but a direct reference to the left-hand side variable; in this way we avoid the introduction of an additional nonterminal on the left-hand side. For this reason we have to import "VarDecl". Note also that we have to import "Decl" because we apply the searching mechanism defined for declarations. "Expr" has to be imported because the creation of an assignment requires the creation of an expression.

Note that this kind of modularization turns the language structures into replacable, separately compiled units. For example, if we wish to add a new statement into the language, we only need to modify the implementation part of "Stat", and recompile it together with the new module for the additional statement. If we want to change the form of a while statement, or its implementation, it is enough to modify and recompile the implementation part of "WhileStat". The definition parts of modules are affected only if there are some changes in the abstract, visible properties of the language structures, and even then the recompilation is reduced to a minimum.

5. DISCUSSION

One of the most crucial part of software engineering is our ability to make use of the existing software for developing new systems. This ability is known to be very low, e.g. a study [Jon84] indicates that only about 15% of written code is original, the rest being some kind of adaptation or copying of existing software. We feel that in language implementation software the situation is at least as bad. For example, in each new language implementation the concept of an arithmetic expression is re-implemented, although this part of the software is practically identical to

numerous other implementations. One could imagine that in many cases some general form of an expression would be quite sufficient. Similar examples can be found easily.

Our study suggests that programming languages are perhaps not so indivisable beasts as people tend to think. Rather, it seems that the possible non-modular features of present programming languages are a consequence of the monolithic implementation paradigm. If programming languages had evolved with the support of truly modular implementation paradigms, they might look very different from what they are today. It could be even possible that the concept of a programming "language" would gradually vanish in such development, in favour of a programming "system" consisting of relatively loosely coupled, replaceable parts. In the author's view, this kind of development would be profitable to software engineering in general: various kinds of differently oriented programming systems could be rapidly developed, and the programmers would be easily adapted to using these systems because they consist of familiar, generally known components.

REFERENCES

[BSI82] British Standards Institution: Specification for Computer Programnming Language Pascal. BSI6192, BSI 1982.

[Jon84] Jones C.: Reusability in Programming: A Survey of the State of the Art. IEEE Transactions on Software Engineering, SE-10,5 (Sept 1984), 488-493.

[Kos88] Koskimies K.: Modular Recursive Descent Parsing. Unpublished manuscript, September 1988.

[KELP88] Koskimies K., Elomaa T., Lehtonen T., Paakki J.: TOOLS/HLP84 Report and User Manual. Report A-1988-2, Department of Computer Science, University of Helsinki, 1988.

[KoM88] Koskimies K., Meriste M.: Experiences with Class-Based Implementation of Programming Languages. Technical Report, Department of Computer Science, University of Helsinki, 1988.

[KoP87] Koskimies K., Paakki J.: TOOLS - A Unifying Approach to Object-Oriented Language Interpretation. In: Proc. of ACM Sigplan '87 Symposium on Interpreters and Interpretive Techniques, Sigplan Notices 22,7 (June 1987), 153-164.

[Weg87] Wegner P.: Dimensions of Object-Oriented Languages. In: OOPSLA '87, Sigplan Notices 22,12 (Dec 1987), 168-182.

[WMK80] Welsh J., McKeag M.: Structured System Programming. Prentice-Hall International, 1980.

[Wir88a] Wirth N.: The Programming Language Oberon. Software Practice and Experience 18,7 (July 1988), 671-690.

[Wir88b] Wirth N.: Type Extensions. ACM TOPLAS 10,2 (April 1988), 204-214.