APPLE:
Advanced Procedural Programming Language Elements

Christian Heinlein
Dept. of Computer Structures
University of Ulm
Germany
heinlein@informatik.uni-ulm.de

Position Paper for the ECOOP 2004 Workshop on
Object-Oriented Language Engineering for the Post-Java Era

1. Introduction

Today's programming languages, in particular aspect-oriented languages such as AspectJ [8], have received a considerable degree of complexity, making it both hard to learn their "vocabulary" (i. e., simply know all concepts and constructs offered by the language) and to "fluently speak" them (i. e., successfully apply these concepts and constructs in daily programming). In contrast, traditional procedural languages, such as Pascal or C, provided just two basic building blocks: *data structures* (records in particular) and *procedures* operating on them [1]. Modern procedural languages, such as Modula-2 or Ada, added the concept of *modules* to support encapsulation and information hiding [9]. In object-oriented languages such as Eiffel, Java, or C++, these separate and orthogonal entities have been combined into *classes* which offer *subtype polymorphism*, *inheritance* of data structures and procedures (which are usually called *methods* there), and *dynamic binding* of procedures as additional basic concepts.

Even though object-oriented languages support the construction of software that is usually more flexible, extensible, and reusable than traditional "procedural software," it soon turned out that many desirable properties are still missing. For example, *modular extensibility* (i. e., the ability to extend an existing system without modifying or recompiling its source code) is limited to adding new (sub)classes to a class hierarchy, while adding new operations (methods) to existing classes is impossible. Similarly, retroactively extending or modifying the behaviour of operations is infeasible. A great deal of research efforts have been expended in the past years to overcome these limitations by providing even more new concepts, e. g., open classes [3] or advice and inter-type member declarations in aspect-oriented languages [8], to name only a few.

Even though the set of these additional concepts is "sufficient" (in the sense that they indeed solve the problems encountered with object-oriented languages), the question arises whether they are really "necessary" (in the sense that a smaller or simpler set of concepts would not be sufficient). Using AspectJ as an extreme example, this language provides eight more or less different kinds of "procedures," i. e., named blocks of executable code: static methods, instance methods and constructors defined in classes (as in the base language Java), plus instance methods and constructors defined as inter-type members in aspects, plus before, after, and around advice (still neglecting the distinction between "after returning," "after throwing," and general "after" advice).

Figure 1 illustrates this observation graphically: The road leading from procedural languages via object-oriented languages to "conceptually sufficient" aspect-oriented languages climbs up the "hill of complexity" by introducing more and more specialized language constructs in order to "patch" the original deficiencies of procedural and object-oriented languages. This hill of complexity is an undesired burden for language designers and implementors as well as for language users.

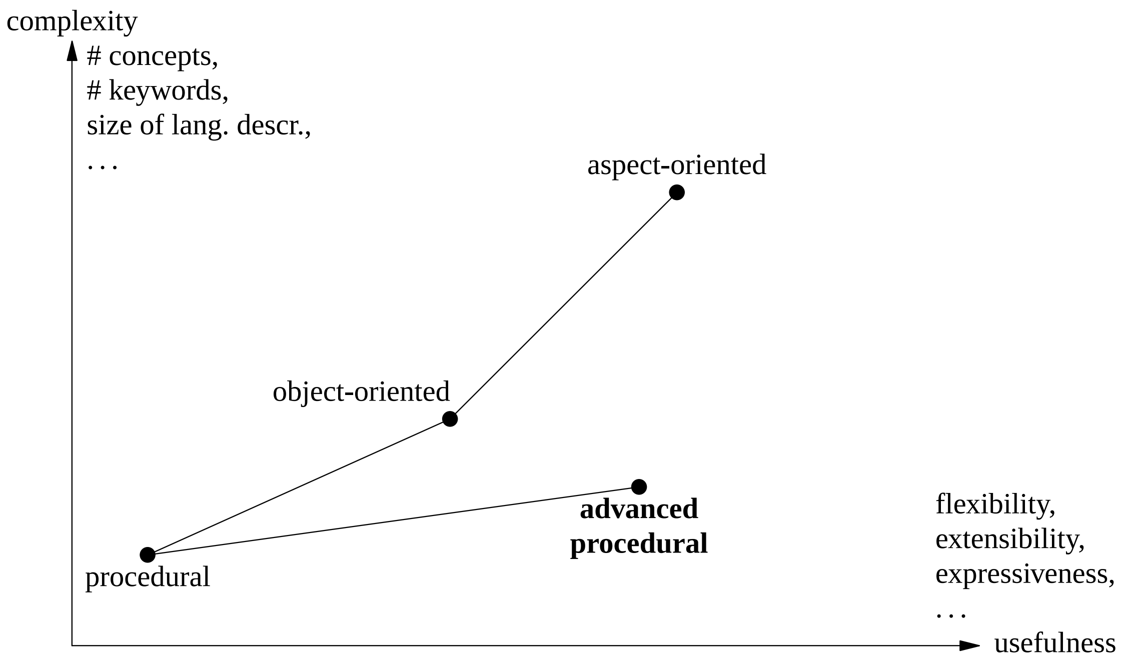

Figure 1: Hill of complexity

From a strictly conceptual point of view, this manifold of procedures is highly redundant: methods and constructors defined in classes are dispensable because they could always be defined in aspects; method and constructor bodies are dispensable because their code could always be defined as advice; before and after advice (the latter in its three variants) are dispensable because they are just special cases of around advice (calling `proceed` at the end resp. beginning of the advice, appropriately embedded in a `try`/`catch` block if necessary to distinguish the three variants of after advice). After these conceptual reductions, around advice – i. e., the possibility to freely override an existing "procedure," either with completely new code (that does not call `proceed`) or with code that augments the original code (by calling the latter via `proceed`) – remains as one of the *essential* (i. e., really necessary) procedure categories. (This goes in line with the statement that "dynamically scoped functions are the essence of AOP" [4].)

It turns out, however, that the potential for conceptual reductions is still not exhausted: By employing dynamic type tests (`instanceof` operator) in an around advice, a programmer is able to emulate the standard dynamic method dispatch provided by the base language Java (or any other dispatch strategy he likes) by executing the advice code only if the dynamic type of the current object is a particular subtype of its static type (or if some other arbitrary condition is satisfied) and simply call `proceed` otherwise. This means in consequence that the specialized built-in dispatch strategy for methods is dispensable from a purely conceptual point of view, thus removing the essential difference between statically and dynamically bound methods, i. e., between static and instance methods.

Similar considerations can be applied to data structures: data fields in classes are dispensable because they are just a special case of inter-type field declarations in aspects. Taken to the extreme, classes can always be declared with empty bodies, because their data fields, constructors, and methods can be declared more modularly and flexibly in aspects.

2. Suggestion

Given these observations, the basic suggestion of this paper is to go back to the starting point of procedural programming languages and extend them into a different direction in order to create *advanced procedural languages* which are significantly simpler than aspect-oriented languages while offering comparable expressiveness and flexibility (cf. Fig. 1).

In particular, replacing simple, statically bound procedures with arbitrarily overridable *dynamic procedures* (roughly comparable to around advice) covers (with some additional syntactic sugar which is not essential) the whole range of dynamic dispatch strategies usually found in object-oriented languages (single, multiple, and even predicate dispatch [2, 5]) plus the additional concept of advice (before, after, and around) introduced by aspect-oriented languages. Nevertheless, dynamic procedures remain a single, well-defined concept which is in no way entangled with data structures, class hierarchies, and the like and therefore is hardly more complex than traditional procedures.

Similarly, replacing simple record types having a fixed set of fields with modularly extensible *open types* and *attributes* (roughly comparable to empty classes extended by inter-type field declarations) covers classes and interfaces, field declarations in classes and aspects, multiple inheritance and subtype polymorphism, plus inter-type parent declarations and advice based on `get` and `set` pointcuts (since reading and writing attributes of open types is implicitly done via overridable dynamic procedures). Again, open types constitute a single, well-defined concept which is little more complex than traditional record types.

Finally, preserving resp. (re-)introducing the *module* concept of modern procedural languages with clearly defined import/export interfaces and a strict separation of module definitions and implementations [11], provides perfect support for encapsulation and information hiding, even for applications where sophisticated concepts such as nested or friend classes are needed in today's languages [6, 10].

3. An Example of Open Types and Dynamic Procedures

This section presents a brief example of open types and dynamic procedures in "Advanced C." A little software library for the representation and evaluation of arithmetic expressions shall be developed.

We start by defining some open types with associated attributes.

```
// General expression.
type Expr;

// Constant expression.
type Const;

// Const is convertable to Expr, i. e., a subtype of Expr.
conv Const -> Expr;

// The value of a constant expression is an attribute of Const.
attr val : Const -> int;

// Binary expression.
type Binary;
conv Binary -> Expr; // Binary is a subtype of Expr, too.
attr op : Binary -> char; // Operator,
attr left : Binary -> Expr; // left and
attr right : Binary -> Expr; // right operand of binary expression.
```

Then, a dynamic procedure (or *global virtual function* in a C/C++-like nomenclature) called `eval` is defined to compute the value of an expression.

```
// Evaluate constant expression.
virtual int eval (Expr x : Const) {
    return x@val; // @ is the attribute access operator
        // that is very similar to the dot operator in other languages.
}

// Evaluate binary expression.
virtual int eval (Expr x : Binary) {
    switch (x@op) {
    case '+': return eval(x@left) + eval(x@right);
    case '-': return eval(x@left) - eval(x@right);
    case '*': return eval(x@left) * eval(x@right);
    case '/': return eval(x@left) / eval(x@right);
    }
}
```

In a later stage of the development, we detect that we have forgotten to implement the remainder operator `%`. We fix this in a completely modular way (i. e., without the need to touch or recompile the above code) by adding another "branch" of `eval` overriding the previous one if the condition `x@op == '%'` is satisfied.

```
// Evaluate remainder expression.
virtual int eval (Expr x : Binary) if (x@op == '%') {
    return eval(x@left) % eval(x@right);
}
```

For a particular application of the library, we want divisions by zero to return a special null value (represented by the smallest available integer value) that propagates through all arithmetic operations. This can be achieved, again in a completely modular way, by introducing the following additional branches of `eval`.

```
// Special null value.
const int null = INT_MIN;

// Catch divisions by zero.
virtual int eval (Expr x : Binary) if (x@op == '/' || x@op == '%') {
    if (eval(x@right) == 0) return null;
    else return virtual(); // Call previous branch.
}

// Catch null-valued operands.
virtual int eval (Expr x : Binary) {
    if (eval(x@left) == null || eval(x@right) == null) return null;
    else return virtual(); // Call previous branch.
}
```

4. Dynamicity of Open Type Instances

Instances of open types differ from instances of classes in object-oriented languages such as Java or C++ in two ways.

First, their set of attributes is dynamic, again in two ways: Because the attributes of a particular type can be defined in different modules, the set of *all* attributes is unknown when compiling a single module. Furthermore, since modules containing attribute definitions might be loaded dynamically at run time, the set of all attributes belonging to a type is even unknown at link or program start time. To support this kind of dynamicity, objects of an open type might be represented as dynamic lists of attribute/value pairs. (However, it is possible to devise more time- and space-efficient representations, too.) If an attribute is accessed that is not present yet, a well-defined null value is returned for read accesses, while a new attribute/value pair is added to the list for write accesses.

Second, the dynamic type of an object, which is equal to its static type immediately after creation, can be changed at run time. For example, an object initially created as a person might later be specialized to a student or an employee, without losing its unique identity or its current attribute values. It is even possible for an object to possess multiple dynamic types, e. g., for a person to become a student *and* an employee at the same time, even if no corresponding static type representing an employed student has been defined. Because procedures are not directly associated with types or objects, such manipulations cannot lead to "message not understood" or other run time type errors, i. e., the system remains statically type-safe.

References

[1] A. V. Aho, J. E. Hopcroft: *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] C. Chambers, W. Chen: "Efficient Multiple and Predicate Dispatching." In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '1999)* (Denver, CO, November 1999). *ACM SIGPLAN Notices* 34 (10) October 1999, 238−255.

[3] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein: "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java." In: *Proc. 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000). *ACM SIGPLAN Notices* 35 (10) October 2000, 130−145.

[4] P. Costanza: "Dynamically Scoped Functions as the Essence of AOP." *ACM SIGPLAN Notices* 38 (8) August 2003, 29−36.

[5] M. Ernst, C. Kaplan, C. Chambers: "Predicate Dispatching: A Unified Theory of Dispatch." In: E. Jul (ed.): *ECOOP'98 − Object-Oriented Programming* (12th European Conference; Brussels, Belgium, July 1998; Proceedings). Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin, 1998, 186−211.

[6] J. Gosling, B. Joy, G. Steele: *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[7] C. Heinlein: *Vertical, Horizontal, and Behavioural Extensibility of Software Systems*. Nr. 2003-06, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, July 2003.
http://www.informatik.uni-ulm.de/pw/berichte/

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 − Object-Oriented Programming* (15th European

Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327−353.

[9] D. L. Parnas: “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15 (12) December 1972, 1053−1058.

[10] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[11] N. Wirth: *Programming in Modula-2* (Fourth Edition). Springer-Verlag, Berlin, 1988.