Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Michael Franz

Protocol Extension: A Technique for Structuring Large Extensible Software-Systems

December 1994

226

Protocol Extension: A Technique for Structuring Large Extensible Software-Systems

Michael Franz
Institut für Computersysteme, ETH Zürich, CH-8092 Zürich, Switzerland

Abstract
A technique is described by which dynamically-loadable modules may add methods to existing classes at run-time. This leads to increased flexibility for structuring large extensible software-systems. Through the use of a doubly-indirect dispatch scheme, efficient method activation can be provided without the need to enumerate the set of methods applicable to a particular class at compile time. As a consequence, separately-compiled client modules are not invalidated when methods are added to an imported class. This reduces the number of recompilations and increases development efficiency. The new mechanism has been incorporated into a variant of the Oberon System. The paper proposes a set of syntactic extensions to the Oberon language and discusses implementation options.

Key Words
Software Engineering, Extensible Programming, Programming Language Design, Dynamic Binding, Oberon, Protocol Extension.

1. Introduction

One of the key concepts of object-oriented programming is *dynamic binding*. The term refers to the situation in which the destination address of a procedure call is resolved only at run-time, thereby concealing the exact identity of the callee from the caller. For example, when a message is sent to an object in an object-oriented programming language such as Smalltalk [GR83], the specific effect of the message-send (which is in fact a dynamically-bound procedure call) depends on the type of the receiver object. Hence, sending the *same* message to different objects may result in the activation of *different* procedures.

In the current euphoria about object-oriented technologies, it is often forgotten that dynamic binding is not restricted to languages that offer explicit classes and inheritance. The same effect of indirect procedure activation can also be achieved by way of *procedure variables*. In fact, there are many problems that can be solved elegantly in an object-oriented fashion, but without the use of an object-oriented language.

For example, consider a graphics editor that is able to draw several different kinds of graphical objects, such as circles, rectangles, triangles, etc. Using the programming language Modula-2 [Wir82], which is not usually considered to be object-oriented, this editor could still be programmed in an object-oriented manner by modelling objects as *variant records* and methods as *procedure variables* within individual objects. Each object might have a "method-variable" (record field) *Draw* for displaying it on the screen, but an object describing a circle would contain a reference to a different *Draw* procedure than an object describing a rectangle. To display such

objects, one need only call their *Draw* routines, without having to distinguish between different object-kinds. In this "manual" approach to object-orientation, it is the programmer's responsibility to ensure that the "method-variables" are initialized properly.

The programming language Oberon [Wir88a] is a direct descendant of Modula-2, and "somewhat more object-oriented" than its predecessor, as it provides *data polymorphism* by way of *type extension* [Wir88b]. However, Oberon has no explicit class construct as found in "pure" object-oriented languages. Hence, just as in Modula-2, dynamic binding in Oberon needs to be realized "manually" by the programmer through the use of procedure variables. Unlike Modula-2, Oberon supports *extensibility*. Whereas objects in Modula-2 need to be modelled by variant records, in which the number of variants cannot be modified without changing the source program and recompiling, Oberon's concept of type extension allows *further variants* to be added in *external modules* at some later date. This is useful particularly in conjunction with *dynamic module loading and unloading*, an integral feature of the environment [WG89] in which Oberon evolved.

Dynamic loading allows to add further modules to a running program at any time. It is the key to the unlimited extensibility of Oberon-based systems. While type extension allows to provide more powerful variants of existing object-categories in external modules, dynamic loading makes it possible to put these variants into active use from an existing base-system. Hence, programming in Oberon leads to a new approach of system design, which might be characterized by the term *stepwise extension*.

The aforementioned graphics editor would be programmed in Oberon in such a way that it could be extended by further object categories at run-time. Wirth and Gutknecht [WG92] describe such an editor in detail. The structure of their graphics editor is shown in Figure 1. It places the individual object categories in separate modules that are higher up in the module hierarchy than the editor itself. The editor doesn't "know" what objects it is displaying, but handles them abstractly. In this model, further object-handling modules can be programmed without affecting any of the existing modules, and can then be linked into a running editing session as required by the user. At any particular moment during such a session, only those modules need to be loaded that are required for handling the objects currently on the screen.

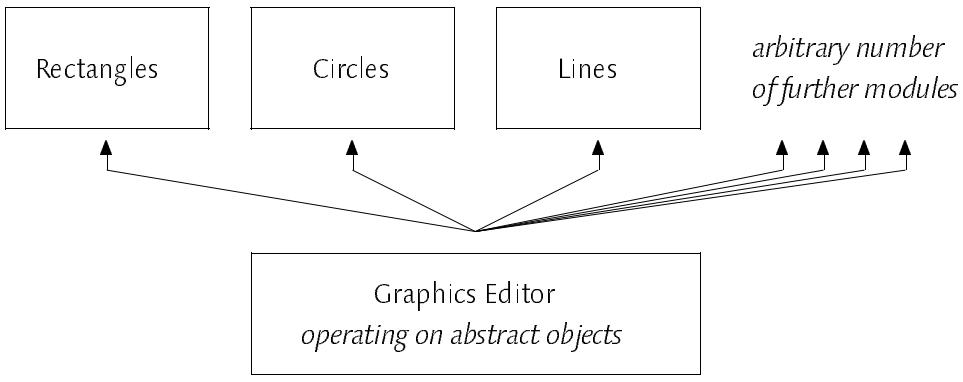

Figure 1: Structure of an Extensible Graphics Editor.

The above is by no means an exotic example. In the world of Oberon, most extensible software packages are structured in a similar manner and rely on a combination of *dynamic loading* and *dynamic binding*. The former allows to add functionality to a basic application at run-time, while the latter enables the construction of an extensible base-system without having to anticipate all possible extensions beforehand.

This paper proposes a new language construct that ties dynamic binding even more intimately to dynamic loading, bringing Oberon closer to programming languages that offer class-based inheritance [GR83, Str87, CDG89], but with fewer limitations. In analogy to the term *type extension*, the new technique has been named *protocol extension*.

2. Protocols

Oberon supports an instance-centered style of object-oriented programming in which *dynamic binding* is achieved under the explicit control of the programmer through the use of *procedure variables* that are installed within individual objects. This is in contrast to the "mainstream" of object-oriented programming languages, in which messages are bound to *procedure constants* that apply simultaneously to all objects of a certain class. As Wirth [Wir89] points out, Oberon's solution can be regarded either as a burden (and a source of mistakes) or as an additional degree of freedom (and power).

The set of methods provided by objects of a particular class is sometimes called its *protocol*. In environments that are based on interpreted execution, such as the one in which the language Smalltalk evolved [Gol84], programmers are free to change class protocols at will, i.e. add and remove methods, or alter their signatures. Unfortunately, this is no longer possible when such languages need to be compiled efficiently. Efficient implementation of class-based inheritance is usually achieved by way of *method tables*, into which *method numbers* serve as indices. This technique requires a unique numbering of methods within each type hierarchy, and the static correspondence of method numbers to method names makes changes in class protocols rather awkward.

For example, adding methods to an existing class *K* requires the renumbering of methods not only in *K* itself, but also in all of its subclasses and wherever any of these methods is invoked, meaning that all direct and indirect clients of *K* need to be recompiled. Such recompilations can cause major disruptions in large systems. In fact, the widespread preference of *interpreters* over *compilers* for prototyping applications may be attributed exactly to the fact that protocol changes are common during experimentation with a prototype, but recompilations costly [WCW90].

Moreover, such recompilations may not even be immediately possible. For example, assume that we are working in an object-oriented environment that provides a number of classes for displaying *graphics* on the screen. Now imagine that new output devices have just been added to the system and we would like to experiment with two new methods *Print* and *Plot* that might eventually be added to the existing graphics classes. However, we probably cannot simply add methods to the root class of the graphics subsystem, because several existing application packages maintained by others would be invalidated by this step. The traditional solution in these cases is to create a *parallel class hierarchy* for the duration of the experiment, starting at the base class in question and re-implementing all classes derived from it.

The remainder of this paper describes a new dynamic-binding mechanism, by which class protocols can be augmented in external modules without affecting any existing clients. The additions become active when the module containing them is *loaded dynamically*, and they can be undone by *unloading* the module again, just like other kinds of extensions in the original Oberon system. During experimentation with a prototype it is therefore possible to dynamically change the behaviour of a live object configuration. The flexibility thereby gained is similar to that of interpreted systems, but with the efficiency of compiled code.

3. Protocol Extension

Protocol extension is based on two ideas. The first of these is to separate the *method* abstraction from both the class to which it applies (or rather, the *data type*, since there are no explicit classes in Oberon, upon which this work is based), and from the procedures that are executed upon method invocation. In the following, these stand-alone method-designators will be referred to as *messages*, indicating that they can be sent to objects even in the absence of a corresponding implementation, although this may not be very sensible. (Note, however, that in the Oberon System [WG89] it is very common to send messages to objects that don't understand them, in which case they are simply ignored.)

The second idea of protocol extension is to utilize *dynamic loading and unloading of modules* for changing protocols at run-time, delegating the assignment of method-table slots to the *linking loader*. This is a natural evolution of the dynamic-extension mechanism already present in the Oberon System, which allows to change the set of available *Commands* under user (or program) control by loading and unloading modules dynamically.

Below, the individual elements of the protocol extension mechanism are introduced, and, informally by way of examples, a corresponding notation for the programming language *Oberon* that extends the present language definition [Wir88a] without affecting any existing syntax.

3.1. Messages

A *message* is a named entity that abstractly describes the *behaviour* of an object and the minimal type requirements that an object must fulfil so that the message is applicable to it. It is represented by a triplet consisting of the *message name*, the *message base type* and a list of *message parameters*. A message is said to be *based* on its message base type, which must be a *record* (or *pointer to record*), and may only be applied to objects of this type or extensions thereof. A compiler can determine whether these requirements are met.

Messages belong to the scope of the declaring module and may be exported by it. If a message is exported, then all *conforming implementations* (see section 3.2) should be exported by their respective modules as well. This facilitates the static detection, solely from interface definitions, of implementation inconsistencies that might occur when a message is implemented more than once for the same type in any desired module configuration.

Example: The following module

```
MODULE DisplayOps;
    IMPORT Display;
    MESSAGE Display.Object!Draw(scale: INTEGER);
END DisplayOps.
```

introduces a message *DisplayOps.Draw* based on the type *Object* imported from module *Display*. Note that it is entirely possible to declare further messages based on *Display.Object* in other modules, and that these may also be called *Draw*.

3.2. Message Implementations

A *message implementation* describes the actions that are performed when the associated message is received by an object of a certain type. Unlike *methods* in traditional class-centered models, however, message implementations are not required to lie in the same scope as the message definition. The dynamic binding of messages to appropriate message implementations is not governed by scoping rules that statically link a receiver's type with a specific method implementation, but is controlled *by the run-time presence of modules* that contain message implementations for selected types.

Message implementations are discriminated by *type qualifiers* that describe the data type for which each particular implementation of a message is valid. They need to *conform* to the message they implement. A message implementation is said to *conform* to a message if its name and parameter list (i.e., its *signature*) are identical to that of the message, and if its type qualifier is equal to or an extension of the message base type. A message that has no conforming implementation for type *T* is called *weak in T*. The opposite is called *strong*.

Example: In a module that imports both *Display* and *DisplayOps*, the declaration

```
PROCEDURE (self: Display.Object)!DisplayOps.Draw(scale: INTEGER);
BEGIN  ...
END Draw;
```

defines a procedure that *implements* the message *Draw* defined in module *DisplayOps* for objects of type *Display.Object*. The type qualifier directly following the *PROCEDURE* reserved word distinguishes a message implementation syntactically from an ordinary procedure declaration. It also defines a local name for the self-parameter that is passed implicitly to the procedure upon activation.

3.3. Message Designators

Messages are activated by the use of a *message designator*. Activating a message implies the *dynamic binding* of the abstract message to a concrete implementation based on the *run-time type* of the

distinguished *receiver* argument. Activating a message for a receiver of type *T* in which the message is weak leads to the execution of the implementation that belongs to the largest base type of *T* in which the message is strong (i.e. the implementation is *inherited* from the corresponding base type). If the message has not been implemented for any base type of *T*, a run-time error is generated (this corresponds to the call of an *abstract method* in other programming languages).

A message *M* that is weak in a type *T* can be made strong by dynamically loading a module that implements *M* for *T*. Loading a module containing a message implementation that is already strong in the system is considered a load error, analogous to a *module key mismatch* in systems that support dynamic module loading. Unloading a module weakens all message implementations originating in that module.

Example: Sending the message *DisplayOps.Draw(100)* to a variable *grafobj* of static type *Display.Object* (or an extension thereof) is expressed by

```
grafobj!DisplayOps.Draw(100)
```

Message designators are distinguished by an exclamation mark following a record or pointer-to-record designator. The variable designator in front of the exclamation mark fulfills a dual role. It is used to select the actual procedure that implements the message for the dynamic type of the designated variable, and it also designates the self-parameter that is passed to that procedure. (The use of exclamation marks in the notation was inspired by Hoare [Hoa78].)

3.4. Reflection

Message designators may also be used in relational expressions. Two message designators may be compared with each other, implying a test whether the corresponding messages are implemented by the same procedure at that moment. They may also be compared to *NIL*, which represents an *abstract message* for which no implementation exists at all in the system.

Example: Suppose that the run-time type of *grafobj* is unknown and that the message *DisplayOps.Draw* has no default implementation (a *default implementation* denotes an implementation that is guaranteed to be strong for the message base type, for example because it is implemented in the module containing the message declaration). By the test

```
grafobj!DisplayOps.Draw # NIL
```

one can determine at run-time whether an implementation of *DisplayOps.Draw* is currently defined for the run-time type of *grafobj*.

3.5. Superclass Delegation

The implementation of a message *M* for a type *T* may delegate control to the implementation of the same message for *T*'s direct base type *BT*. The self-object passed to the base type's message implementation retains its type identity and is not reduced to *BT*. This mechanism may be used only within the implementation of a message, and only for activating the implementation of the identical message for the direct base type. The restriction is necessary so that invariants guaranteed by objects in "closed" modules cannot be circumvented.

Example: The following implementation of *Init* for type *T* activates another procedure that currently implements *Init* for *T*'s direct base type *BT*.

```
PROCEDURE (obj: T)!Init;
BEGIN  obj!(BT)Init; ...
END Draw;
```

Syntactically, superclass calls are denoted by inserting the name of the supertype in parentheses before the message name (this is the same syntax as for Oberon's type guards). Although the compiler allows only the name of the receiving object's direct base type between the parentheses, this syntax is useful as it reminds the programmer of the specific message implementation that is called.

4. Implementation

This section presents a set of data structures and algorithms that can be used for implementing the language constructs introduced above. A first implementation exists that is based on a version [Fra93] of the Oberon system [WG89] for the Apple Macintosh [App85]. The algorithms used in the existing prototype are more complex and less elegant than the one shown here, but have the advantage of avoiding the use of recursion when message implementations are propagated through a type hierarchy.

Before going any further, however, some additional terms need to be defined. In the following, the term *root type* will denote any type that is not an extension of another type. A root type along with all of its extensions (i.e., also the extensions of its direct extensions, etc.) is called a *type family*. Note that each message affects exactly one type family, although not necessarily at its root type. Moreover, the *depth of a type hierarchy* is measured by saying that the root type has *extension level* 0, its direct extensions have extension level 1, and so on.

Now to the implementation. In the implemented system, objects contain an invisible *type tag*, which is a pointer to a *type descriptor* characterizing the object's data type. Type descriptors have the following basic internal structure:

```
TYPE
    TypeTag = POINTER TO TypeDescr;
    TypeDescr = RECORD
        nofMsg: INTEGER;                    number of messages (used only in root types)
        exts: TypeTag;                      list of types that are extensions of this one
        link: TypeTag;                      siblings on the same extension level
        base: ARRAY MaxLev OF TypeTag;      table of base-type relationships
        impl: ARRAY MaxMsg OF PROCEDURE     table of procedures that implement messages
    END;
```

The individual fields of this type descriptor have the following functions:

- The *nofMsg* field counts the number of messages that are currently in use in a type family. One counter is used per type family and located in the descriptor of the family's root type.

- In the field *exts*, the descriptor for each type maintains a pointer to a linear list of type descriptors relating to the type's extensions. The elements of this list are linked together (in an undetermined order) via the *link* field.

- The table *base* describes base-type relationships, which are used for implementing type tests in constant time as suggested by Cohen [Coh91]. The table contains references to the ancestors of the described type *T* at different levels of the type extension hierarchy, i.e. *base[0]* contains the tag of *T*'s root type, *base[1]* contains the tag of the root type's direct extension along the path to *T*, and *base[extension level of T]* contains the tag of *T* itself. All other entries are set to *NIL*.

- The table *impl* stores the addresses of message-implementation procedures for the type in question. Some of these implementations may be specific to this type and its descendants (i.e., *strong*), others may have been inherited (*weak inherited message*), and some messages may not have been implemented at all, in which case the table entry is equal to *NIL* (*weak abstract message*).

Associated with each *message* in the module of its declaration is an invisible variable initialized by the loader that identifies the *message slot* of the affected type family in which the addresses of the corresponding message implementations are stored.

```
TYPE
    Message = INTEGER;
```

Possible implementations of the functions that have to be performed in order to support protocol extension are described below. No distinction is made between types and their associated type descriptors, since the latter at run-time correspond to the former at the language level. Moreover, the following abbreviations will be used to denote values that can be calculated statically for any type *T*:

T.LEV the extension level of *T* (0 = root type, 1 = direct extension of root type, ...)
T.BASE the direct base type of *T* (corresponding to T.base[T.LEV–1] in the type descriptor)
T.ROOT the root type of *T*'s type family (corresponding to T.base[0])

4.1. Finding a Message Implementation

Recall that a *message* represents an index into the slot table of the type family it is associated with, and that every object *obj* contains an invisible tag that is a pointer to its type descriptor. The message designator

```
obj!Mod.Msg
```

represents the address of the implementation of *Mod.Msg* that currently applies to the (run-time) type of *obj*. Hence, it can be computed by evaluating

```
obj.tag↑.impl[Mod.Msg]
```

In comparison to message activation in traditional compiled class-based languages, the proposed mechanism requires one further indirection, since the index of the applicable message slot cannot be determined at compile time. However, due to cache effects, on modern processors, the cost of this extra memory lookup should be negligible.

4.2. Initializing a Type Descriptor

Type descriptors are initialized when the module defining the corresponding types is loaded. Since type-extension hierarchies contain no cycles, the compiler can prepare the list of type descriptors to be initialized in such an order that base types appear always before their extensions. The initialization procedure can then be described as follows:

```
PROCEDURE InitTypeDescr(T: TypeTag)
BEGIN
    IF  T.LEV = 0   THEN                        (* is it a root type? *)
        T.link := NIL;                          (* root types have no siblings *)
        T.base[1..MaxLev] := NIL;               (* initialize base-type table *)
        T.impl[0..MaxMsg] := NIL;               (* invalidate all implementation slots *)
        T.nofMsg := 0                           (* set message counter of this family to zero *)
    ELSE   (* T.BASE has been inited already *)
        T.link := T.BASE.exts; T.BASE.exts := T;  (* insert at head of base-type's extension list *)
        T.base := T.BASE.base;                  (* copy base-type table from direct base-type *)
        T.impl := T.BASE.impl                   (* inherit all implementations from direct base-type *)
    END;
    T.exts := NIL;                              (* new types have no extensions yet *)
    T.base[T.LEV] := T                          (* append own tag to copied base-type table *)
END InitTypeDescr;
```

4.3. Adding a Message by Dynamically Loading its Module

Messages are added to the system when a module *Mod* is loaded dynamically that contains *MESSAGE* declarations. For each message *Msg* based on a type *T*, the loader has to perform the following actions:

```
INC(T.ROOT.nofMsg);                 new message for this type family
Mod.Msg := T.ROOT.nofMsg            assign slot number to message variable
```

Note that this assignment of message slots in strictly ascending order leads to "holes" in the implementation table when modules containing messages are unloaded. When experimenting with changing message protocols during prototyping, this may eventually lead to a state in which no more messages can be added to the system although there are many unused message slots. In practice, one therefore replaces the simple *nofMsg* counter by a *message allocation map* that keeps track of the slots in use within each type family. A slot can then be re-used as soon as the message previously associated with it is unloaded.

4.4. Adding and Removing Message Implementations

A module *Mod* may contain several implementations of messages *Msg0*, *Msg1*, ..., *MsgN* for types *T0*, *T1*, ..., *TM* (*M* <= *N*). When such a module is loaded, the corresponding messages are strengthened in their respective types. Consider the following auxiliary procedure:

```
PROCEDURE Bequeath(t: TypeTag; msg: Message; old, new: PROCEDURE);
   VAR subt: TypeTag;
BEGIN
   IF  t.impl[msg] = old  THEN
      t.impl[msg] := new;                  change behaviour of message
      subt := t.exts;
      WHILE  subt # NIL  DO
         Bequeath(subt, msg, old, new);    propagate to all subtypes
         subt := subt.link
      END
   END
END Bequeath;
```

The procedure *Bequeath* propagates an implementation change to a certain type and all of its affected descendants. Descendants are affected only when their implementation of the message in question is weak, i.e., identical to that of the direct base type. With the aid of *Bequeath*, the following steps need to be taken for each newly added implementation *proc* of message *msg* based on type *t*:

```
PROCEDURE AddMsgImpl(t: TypeTag; msg: Message; proc: PROCEDURE);
BEGIN
    IF  (t.impl[msg] = NIL)                      message currently not implemented
        OR (t.impl[msg] = t.BASE.impl[msg])      message currently weak (inherited)
    THEN  Bequeath(t, msg, t.impl[msg], proc)    propagate new implementation
    ELSE  LoadError()                            message already strong in this type
    END
END AddMsgImpl;
```

If a module contains several implementations of the same message for different types, then the order in which these are added is irrelevant. However, a clever compiler can reduce the workload required for inheritance propagation at loading time by ordering the implementations by type in descending order, largest type first.

When a module is unloaded, all message implementations originating in it must be weakened. This is done by propagating the message implementation applying to the base type, or *NIL* if dealing with the root type.

```
PROCEDURE RemoveMsgImpl(t: TypeTag; msg: Message);
BEGIN
    IF  t.LEV = 0  THEN
        Bequeath(t, msg, t.impl[msg], NIL)       make message abstract
    ELSE
        Bequeath(t, msg, t.impl[msg], t.BASE.impl[msg])
    END
END RemoveMsgImpl;
```

Note that unloading a module creates the same dynamic message implementation context that would have existed if that module had never been loaded at all.

5. Discussion

Protocol extension opens up novel ways of structuring large software-systems. Recall the example of the graphics editor from the introduction. This had been partitioned in such a way that each object category was implemented in a separate module. What if, however, these object-handling modules themselves got unwieldily large? Protocol extension makes it possible to organize such applications in a *two-dimensional* matrix, subdividing functionality not only by object category, but also splitting up message protocols into several modules that can be loaded dynamically as required.

As an illustration, study the module organization depicted in Figure 2. Now consider what modules are required for *printing* a document containing only *circles*. Only four modules need to be loaded for this task, namely the core of the graphics editor, and three extension modules: *Circles* (defining the circle object-type), *Printing* (defining the printing-related message protocol), and *Printing Circles* (implementing the messages of the latter for the former). Note that the object-category modules (along the vertical axis in the diagram) and the message-category modules (along the horizontal axis) are mutually disjoint, while each message-implementation module

imports one object-category module (the one to its left) and one message-category module (the one below it).

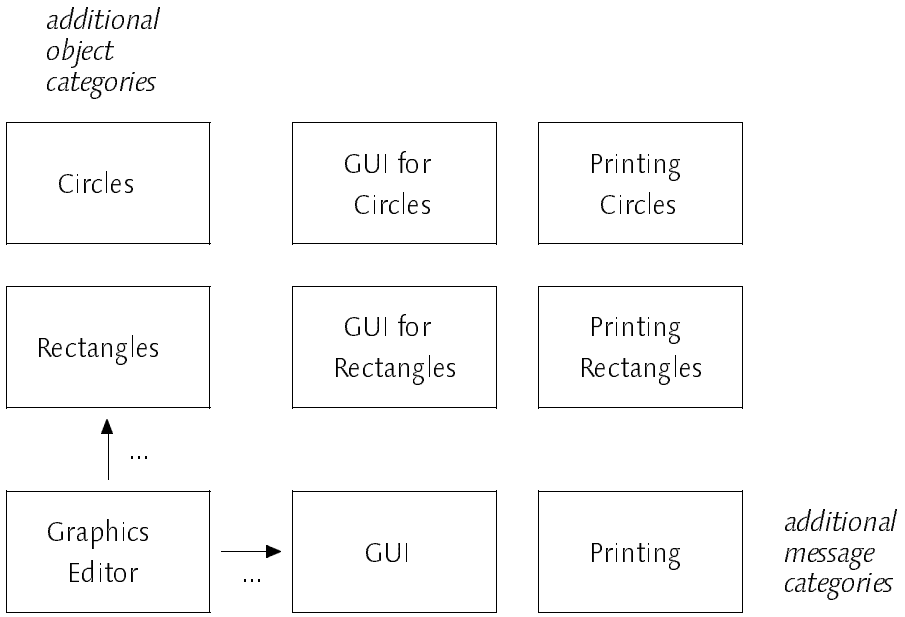

Figure 2: Two-Dimensional Extensibility of a Graphics Editor.

Protocol extension enables the construction of conceptually very large systems that require a much smaller memory space. But even for smaller systems, it might be sensible to place seldom-used messages and their implementations in separate modules that are loaded dynamically on demand. The module organization illustrated above also makes it easier to divide implementation work among several programmers. A further advantage is the possibility to change the behaviour of objects on-the-fly, specifically the facility to attach methods to a live data structure for debugging purposes.

Protocol extension also overcomes an unnatural restriction of compilers for class-oriented programming languages that penalizes the addition of functionality to existing classes. When they are implemented using static compilation and table-based message dispatch, current languages effectively force the designers of class hierarchies to anticipate all future uses of a class and create *abstract methods* that may be overwritten by actual implementations later. This is a grave constraint, considering that the process of software development is characterized by experimentation. Systems that offer *dynamic compilation* [DS84, CUL89] have no such limitations, but suffer from recompilation delays when class protocols are changed and provide less efficient message invocation. The use of protocol extension with explicit messages eliminates both the need for recompilations and the need for abstract methods, as messages may be added externally when the need arises. Nevertheless, it is still possible to determine the protocol accepted by a certain class in a specific module context by the help of appropriate browsing tools.

Lastly but just as importantly, protocol extension can be applied to *static objects* on the stack as well as to dynamic objects in the heap, allowing to use the object metaphor even for transient objects without sacrificing efficiency.

6. Related Work

In most object-oriented programming languages, *classes* are the primary construct while *methods* are secondary and subordinate to them. A markedly different approach has been taken in the *Common Lisp Object System* (CLOS) [DG87]. In CLOS, methods are grouped together not by the classes to which they apply, but by the operations they perform. A group of methods that implement a certain behaviour for different types of receiver is called a *generic function*. In order to activate a desired behaviour, one invokes the corresponding generic function, which chooses one of its constituent methods based on the types of the specified arguments.

The concept of protocol extension is able to partly duplicate the generic function metaphor, in that it allows the construction of a *module* that adds a new method across an existing type hierarchy, grouping all of the message implementations together. The generic functions of CLOS are more general than that, as they support multiple inheritance and a type-dispatch mechanism taking into account more than just one of the message's arguments. However, this generality also implies that they cannot be implemented as efficiently as protocol extension.

The work of Harrison and Ossher [HO90] on *subdivided procedures* is also related to protocol extension. Their system provides *functional extension* by the addition of *alternate procedure bodies* to a procedure, which are selected based on criteria specified by the programmer. Protocol extension can be viewed as a specialization of the subdivided procedure mechanism with regard to the subdivision criterion (dispatch on type only), which can, however, be implemented more efficiently and allows not only the *addition* of alternatives, but also their *dynamic removal*.

In the context of Oberon, the programming language Oberon-2 [MW91] needs to be mentioned also. Oberon-2 is an extension of Oberon that offers *type-bound procedures* and a type-dispatch mechanism resembling that of class-based programming languages. Hence, Oberon-2 has the same static correspondence of method numbers to method names as class-based languages, ruling out protocol-changes without the invalidation of clients.

Acknowledgement

The author is indebted to Ch. Denzler for his collaboration in implementing the prototype described here. Many thanks also go to M. Brandis, R. Crelier, Th. Gross, J. Gutknecht, H. Mössenböck, M. Reiser, J. Templ, and N. Wirth for commenting on earlier drafts of this manuscript.

References

[App85] Apple Computer, Inc.; *Inside Macintosh*; Addison-Wesley; 1985ff.

[CDG89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow and G. Nelson. *Modula-3 Report*; Report #52, Systems Research Center, Digital Equipment Corporation, Palo Alto; 1989.

[Coh91] N. H. Cohen; Type-Extension Type Tests Can Be Performed In Constant Time; *ACM TOPLAS*, 13:4, 626–629; 1991.

[CUL89] C. Chambers, D. Ungar and E. Lee; An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes; *OOPSLA '89 Conference Proceedings*, published as *Sigplan Notices*, 24:10, 49–70; 1989.

[DG87] L. G. DeMichiel and R. P. Gabriel; The Common Lisp Object System: An Overview; *Proc. ECOOP '87*, published as *Springer LNCS*, 276, 151–170; 1987.

[DS84] L. P. Deutsch and A. M. Schiffmann; Efficient Implementation of the Smalltalk-80 System; *Conf. Record 11th Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, Utah, 297–302; 1984.

[Fra93] M. Franz; Emulating an Operating System on Top of Another; *Software–Practice and Experience*, 23:6, 677–692; 1993.

[Gol84] A. Goldberg; *Smalltalk-80: The Interactive Programming Environment*; Addison-Wesley; 1984.

[GR83] A. Goldberg and D. Robson; *Smalltalk-80: The Language and its Implementation*; Addison-Wesley; 1983.

[HO90] W. Harrison and H. Ossher; Subdivided Procedures: A Language Extension Supporting Extensible Programming; *Proc. 1990 International Conf. on Computer Languages*, IEEE Computer Society Press, 190–197; 1990.

[Hoa78] C. A. R. Hoare; Communicating Sequential Processes; *Communications of the ACM*, 21:8, 666–677; 1978.

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

[MW91] H. Mössenböck and N. Wirth; The Programming Language Oberon-2; *Structured Programming*, 12:4, 179–195; 1991.

[Str87] B. Stroustrup; *The C++ Programming Language*; Addison-Wesley; 1987.

[WCW90] J. C. Wileden, L. A. Clarke and A. L. Wolf; A Comparative Evaluation of Object Definition Techniques for Large Prototype Systems; *ACM TOPLAS*, 12:4, 670–699; 1990.

[WG89] N. Wirth and J. Gutknecht; The Oberon System; *Software–Practice and Experience*, 19:9, 857–893; 1989.

[WG92] N. Wirth and J. Gutknecht; *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley; 1992.

[Wir82] N. Wirth; Programming in Modula-2; Springer; 1982.

[Wir88a] N. Wirth; The Programming Language Oberon; *Software–Practice and Experience*, 18:7, 671–690; 1988.

[Wir88b] N. Wirth; Type Extensions; *ACM TOPLAS*, 10:2, 204–214; 1988.

[Wir89] N. Wirth; Modula-2 and Object-Oriented Programming; *Proceedings of the First International Modula-2 Conference*, Bled, Yugoslavia, 7–13; also published as Report No. 117, Departement Informatik, ETH Zurich; 1989.