A Poor Man's Approach to Dynamic Invocation of C++ Member Functions

Thomas Kofler, Walter Bischofberger, Bruno Schäffer, André Weinand
Union Bank of Switzerland
UBILAB (Information Technology Laboratory)
Bahnhofstr. 45, CH-8021 Zürich, Switzerland
{kofler,bischofberger,schaeffer,weinand}@ubilab.ubs.ch

Abstract

During the last year we built several solutions for opening our ET++ applications for internal and external scripting. The most annoying part to be coded manually was the code stubs that translate a string based request into the invocation of a member function.

For this reason we built an ET++ specific solution that provides dispatchable member functions in an inexpensive, non-intrusive way. Our solution consists of an extension of the macro generated ET++ run time meta information. To make a member function dispatchable, a developer has to write one macro call. This generates a member function meta object providing information about arguments and a function that serves to invoke the respective member function. These two generated parts work in the context of the dynamic invocation framework, which embodies an architecture that can be customized for varying interfacing needs.

1. Introduction

Our team has been developing interactive standalone applications based on the ET++ application framework for years. During this time the standard application model embodied in ET++ [Wei88, Wei89] fulfilled our requirements well. During 1993 we started to develop distributed applications where several services and tools cooperate to provide a group of users with a certain functionality. A typical example is Beyond-Sniff, a platform for cooperative multi-user development environments. It provides a set of cooperating services and tools. In developing Beyond-Sniff we found that we needed a conceptual framework for interpretively driving an application from within, as well as from other applications. The latter could be patterned, for instance, after AppleScript [App93].

Once we had identified this requirement it was an obvious approach trying to enhance our application framework in order to provide this functionality in a generic way. Our first approach is described in [Kof93]. In this approach we manually wrote stubs to dispatch internal and external dynamic member function invocations. While this proved handy for the first prototypes, it became more and more annoying in large scale applications. For this reason we decided to implement a generic solution that works within our application framework.

A generic solution has to provide a way to invoke an operation whose name is determined at run-time and to provide it with arguments (something straightforward in dynamic languages as Smalltalk or Lisp).

We intended to find a solution that:

- is open and can be used by different kinds of clients, such as embedded interpreters, or by an interapplication scripting mechanism,
- is simple, declarative, and requires little programming from the programmer,
- is type-safe
- is non-intrusive, i.e., the existing code of the application framework must not be changed,
- does not affect portability of the ET++ application framework,
- can be easily implemented and maintained,

- and does not require any additional tools, such as preprocessors or parsers.

The resulting dynamic invocation framework, which we present in this paper, is a typical case of a poor man's approach that works well for its intended application area but also has its limits.

The goal of this paper is to explain the dynamic invocation framework in detail. Section 2 shows how the dynamic invocation framework is used. Section 3 presents the architecture of the framework. Sections 4 and 5 discuss what kind of member functions are supported. Section 6 explains the conversion process between C++ and external data representations. Section 7 compares the dynamic invocation framework with similar approaches, and in Section 8 we draw our conclusions.

2. Using Dynamic Invocation

Dynamic invocation of member functions has many useful applications. In this section, we present concrete examples. We also sketch how the programmer uses our dynamic invocation library without going into details.

Suppose a programmer has written a graphical editor in C++. There is a class, say ‘Editor’, that has a set of member functions, each of them representing a command available to the user. The user invokes command by selecting an item from menus. There are several ways to connect menu items with their respective member functions acting as entry points. Attaching a string containing the name of the respective member function of the class Editor is simple, flexible, and straightforward if member functions can be dynamically invoked. The code in example 2.1 demonstrates the idea. Note that the function ‘CallMemberFunction()’ is a simple interface to the dynamic invocation dispatcher.

```
class Editor : public Tool {
public:
    int CallMemberFunction(const char *cmd);

    virtual int Close(bool askUserIfChanged);                // (1a)
    virtual int Save();                                      // (1b)
};
Msg1(AT(int, Out, --), Editor, Close, AT(bool, In, --));    // (2a)
Msg0(AT(int, Out, --), Editor, Save);                       // (2b)
....
```

```
menu->Add(new MenuItem(cLabelClose, “Close true”));         // (3a)
menu->Add(new MenuItem(cLabelSave, “Save”));                // (3b)
...
editor->CallMemberFunction(menuItem->GetCommand());         // (4)
```

Ex. 2.1 Connecting Menu Items to Member Function by Dynamic Invocation

If the programmer uses our dynamic invocation library, he writes the macro calls as shown in example 2.1 to make ‘Editor::Close(bool)’ and ‘Editor::Save()’ available to dynamic invocation. These declarations turn member functions into *dispatchable member functions*.

Embedding an interpreter for user-level scripting is another interesting area where dynamic invocation is helpful. Manually implementing the code that provides application-specific functions in the interpretive language is tedious and error-prone. Dynamic invocation of member functions may save a lot of time and assures that argument conversion is done automatically and consistently.

If applications can send commands to each other, they can be integrated to any desired degree. Again, an application that can process commands sent via interprocess communication requires code that transforms a command in an external form into an invocation of a function written in the native programming language.

3. Architecture of the Dynamic Invocation Framework

The macro calls as shown in example 2.2 expand to an ordinary static function, called the *callee stub*, and to a statically allocated object, called the *member function meta object*. The primary purpose of the function is to wrap the invocation of the member function, and triggering argument conversion as explained below. The member function meta object, or function meta object for short, describes the argument list of the member function, and also contains a pointer to the callee stub wrapping the member function. The pointer to the function meta object, along with its name, is stored in the class descriptor object. This is shown in figure 3.1.

A *class descriptor* is an object that describes a C++ class. An ET++ object can be asked for its class descriptor by invoking the member function IsA(). The most important use of class descriptors is dynamic type checking. ET++ class descriptors have been used for some time. They are defined by means of a macro much like function meta objects. See [Gam89] for details.

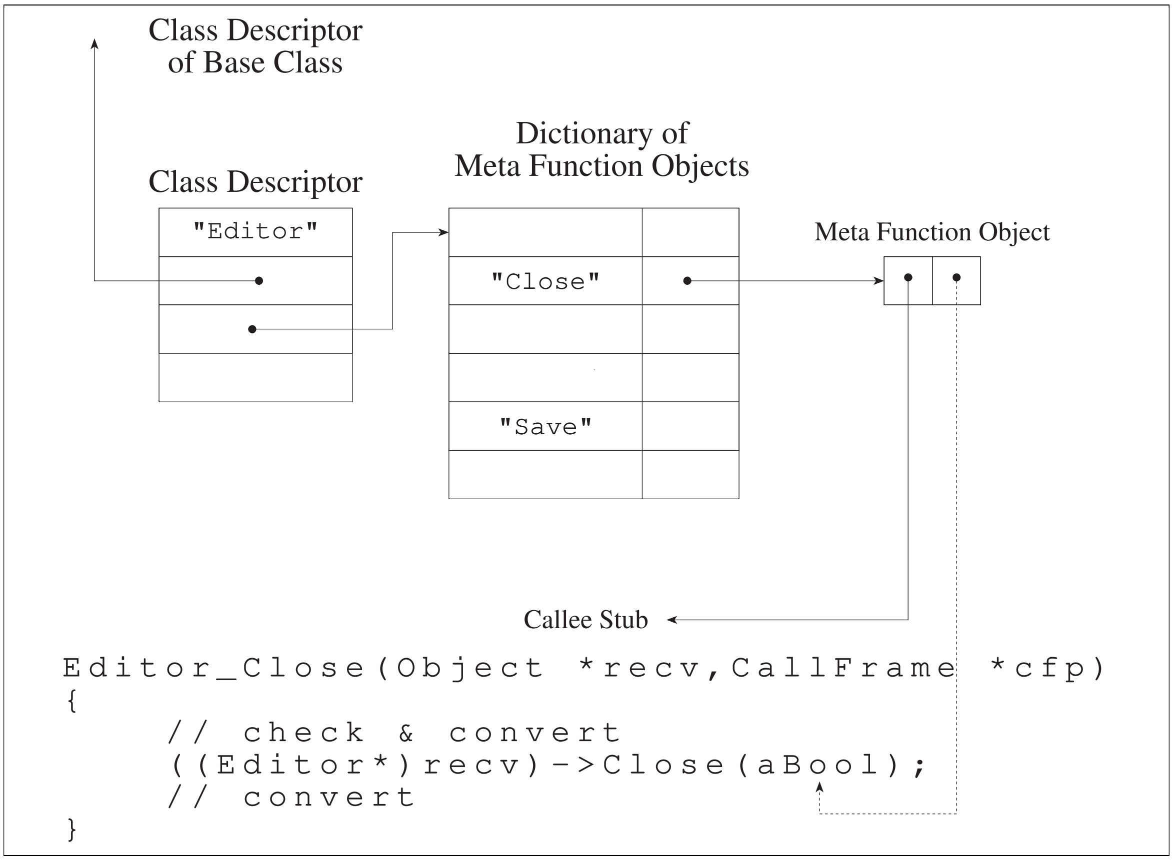

Fig. 3.1 Structure of the Meta Information for Dispatchable Member Functions

The callee stub also passes the member functions its arguments. The conversion is done by an object known as *call frame*. A call frame object knows the meta function object and the external representation of the invocation, e.g. a string. The call frame interprets the description of the argument list as provided by the function meta object and converts, driven by the argument list description, the arguments from the external representation. After the member function has returned, the return value is converted into a form as required by the particular external representation, and memory allocated by the call frame before the invocation is deallocated.

Two components remain to be introduced. The *dispatcher* is responsible for looking up the meta function object and for creating a call frame that can deal with the particular external representation used. The dispatcher then invokes the callee stub passing it a pointer to the object the member function is invoked for, and an appropriate call frame. If a member function involves pointers to ET++ objects as an argument or as return value, then the call frame contacts the *object name mapper*. The object name mapper maintains a map that associates names with object pointers.

Since we build on the existing meta information provided by ET++ our approach works only for instances of classes derived from the ET++ class Object. By porting or reimplementing the meta information infrastructure this restriction could be eliminated. Second, the kind of arguments that can be passed is restricted. This topic is discussed in depth in Section 4 and 5.

4. Supported Signatures

In this section, we give an overview of what our dynamic invocation framework can cope with. We cover the fundamental and actual limitations, and we discuss why these limitations do not pose problems as long we do not leave the realm of the ET++ class library.

We have two dimensions regarding the range of member functions that can be made dispatchable. One of them is the range of applicable types. The other is the range of supported calling semantics. In this section, we concentrate ourselves on applicable types. For a complete understanding, section 4 and 5 must be viewed as a unit.

Overview

We start by introducing some technical terms. In C or C++, the type of an argument or a return value consists of a *base type*, and optionally, of a *type constructor*. A type constructor is a *type expression* using *type operators*. Type operators are the dereferencing operator ('*'), the address operator ("&"), and the subscript operator that is denoted as pairs of brackets with or without an integral constant enclosed[1]. Defining a new C++ class means to introduce a new base type whereas expressing a type by means of an existing base type and a type expression does not introduce a fundamentally new type. Notice that a typedef name is not a base type in our sense, but an alias for a type expression.

The applicable set of base types and type constructors is constrained for our dynamic invocation framework. Base types have to satisfy some conditions discussed below whereas the set of applicable type constructors is small and fixed. The limitations regarding base types and type constructors are of different nature. The set of applicable type constructors is fixed insofar that we have to adapt the dynamic invocation framework if a new type constructor is to be supported.

The set of supported C++ type constructors contains pointers or one-dimensional vectors, both involving up to three levels of indirection. There is a well-known twist with pointers and vectors in C++ since there is no syntactical difference between them. In section 4, we will explain how we handle type constructors in detail.

We support the following base types: all primitive C++ types (char, int, double, etc.), classes like Point and Rectangle viewed as though they were *built-in* into C++, and the universal class ‘Object'. The class Object is the root of the ET++ class library. Supporting the class Object automatically supports all its derived classes. A built-in class is a class whose objects are normally passed by value. Objects of built-in classes do

---

[1] The function call operator '()' is also a type operator in our sense, but pointer to functions are not supported anyway.

not have a real identity, and they are normally defined as automatic object variables. In contrast, ET++ objects are never passed by value, and they are always allocated from the free store.

Introducing a new base type playing the role of a new built-in does not pose difficulties. The same is true if we introduce a new ET++ class. In both cases, a programmer defines a simple meta object, or a class meta object. The definition of a meta object describing a built-in or an ET++ class is a simple and inexpensive task. The presence of the meta object suffices to use the base type in the signature of a dispatchable function.

Typedef names are not base types in our understanding. A *trivial typedef* name is an alias for another base type. A *non-trivial typedef* name is an alias for a general type expression. We cannot support any type expression that takes a non-trivial typedef name for a base type because we cannot resolve, unlike the C++ compiler, the typedef name.

Enumeration types currently are not supported. This is a limitation that could be alleviated. What we have to do is to extend the infrastructure for defining enumeration meta objects.

ET++ Signatures Use Only Few Types

We will now discuss why our dynamic invocation framework, despite its limitations, proves to be useful. An examination of the signatures of the member functions found in ET++ classes shows that we have only a small number of base types, and a small number of type constructors. Since almost all ET++ classes are derived from the root class Object, supporting the class Object automatically includes the support of all its derived classes.

All member functions in ET++ have comparably simple signatures. The fundamental reason behind this fact is that most concepts are encapsulated in ET++ classes. For example, complex data structures are usually constructed with container objects. Thus, a simple pointer to the anchor of a complex data structure suffices to refer to the data structure. As a consequence, member functions never take arguments with a complicated structure. Argument types are rather simple, like pointers to objects of any complexity, integers, or pointers to integers, etc.[2]

The range of supported base types and type constructors was derived from inspecting the signatures of the public ET++ member functions. Even if the protocols of ET++ classes would radically change in the future, we do not expect any changes regarding the set of supported base types and type constructors.

5. Member Function Meta Objects

A member function meta object describes its member function. Function meta objects are stored on per class basis in a dictionary that enables for lookup by the name of the respective member function. A function meta object contains:

- a reference to its corresponding class meta object
- the external name of the member function
- a description of the member function's signature
- a function pointer to the callee stub.

Defining Function Meta Objects

Function meta objects are denoted by means of macros resolved by the C++ preprocessor. Similar macros are used for denoting class meta objects. There is conceptually one macro named 'Msg' to define a function meta object. Since the preprocessor does not support macros with variable argument lists, we have a family of macros with the same name, but with different suffixes designating the number of arguments.

---

[2] Interestingly, this is also true for the types of data members. Otherwise, the meta object approach as described in [Gam89] would fail for the same reason.

The macro call denoting the function meta object supplies all needed information: the class the function is member of, the name of the member function, and a description for each argument including the return value. The template in example 5.1 shows the syntax.

```
MsgN(ReturnValueDescription, Class, MemberFunction, ArgumentDescription1, ...);
```

Ex. 5.1 Syntax Template of the Msg*N* Macro

An argument or the return value is described using a macro again. The name of this macro encodes the type constructor. The first argument is the base type to which the type constructor is applied. Its second argument specifies the direction of the information flow (also known as mode), and the last argument describes who owns the memory involved with an argument. Some of the argument description macros take four arguments. The additional argument to the macro describes what arguments of the member function contains size information about vectors of variable size. We will show concrete examples covering this case later in this section. The template in example 5.2 shows the syntax.

```
TypeConstructor(BaseType, Mode, OwnershipDescription, SizeInformation);
```

Ex. 5.2 Syntax Template of an Argument or Return Value Description

Conveying Structural Information about Pointers and Vectors

The type constructor encoded in the macro’s name gives more information than its corresponding C++ notation. Using different type constructors for syntactically equivalent notations conveys structural information. This information answers the question what 'Object **' means: Is it a pointer to a pointer, or is it a vector of pointers, or is it a vector of vectors of Objects? Except for the first case, we also need to know how many slots the vectors on the second and third levels have.

Consider the code in example 5.3. The different type constructors used in the example tell us how a function interprets arguments involving pointers or vectors. For instance, the type constructor of U’s second argument describes it as a pointer to a pointer (=PP), i.e., that the argument conceptually is a pointer passed by reference. For the second example V, we have a vector of pointers (=ZP). The vector is null-terminated. Finally, W also takes a vector of pointers, but this time its size is passed in the argument named size. The reader might have noticed that the macro AVP takes four arguments. The fourth argument specifies what arguments of the member functions carry size information for the argument being described. The actual argument for Size(), here the number one, specifies the relative position of the argument containing size information. Thus, the third argument of W contains the actual size of the vector.

```
void Foo::U(int i, Object **);                 // (1)
void Foo::V(int i, Object **);                 // (2)
void Foo::W(int i, Object **, int size);       // (3)

Msg1(Void, Foo, U, AT(int, In, --), APP(Object, InOut, --));                      // (1)
Msg1(Void, Foo, V, AT(int, In, --), AZP(Object, In), ()));                        // (2)
Msg2(Void, Foo, W, AT(int, In, --), AVP(Object, In, --, Size(1)), AT(int, In, --)); // (3)
```

Ex. 5.3 Same C++ Type with Different Meanings

Argument Modes

The second argument to a type constructor macro describes the *mode* of a member function's argument. The mode argument can take the following values: In, Out, or InOut. These values indicate the direction of the information flow.

In the case of an argument with In mode, information flows into the function, but not back. In the case of an Out argument, information flows out of the function. We assume that the data structure referenced by an In argument is fully initialized before calling the member function, whereas the data structure referenced by an Out argument is not initialized at all. The data structure of an InOut argument also is fully initialized, but we are interested in the values stored in the structure after the call, too.

Ownership Description

The mode of an argument does not convey enough information to determine what memory has to be allocated before the invocation, and what memory has to be deallocated after invocation. The third argument to a type constructor macro call carries information such that we can deduce, within the bounds of our approach, the ownership of memory. The argument is called the ownership description.

Consider the signature 'char *Foo::Name()'. Obviously, the return value of Foo::Name is an Out argument. The question is whether the string has been allocated on behalf of the caller, or whether the string is still used by the callee after the invocation. Since the latter is a common case, additional information about ownership allows to cover a larger range of calling semantics.

The ownership description is based on the levels of indirection involved, and on the silent assumption that memory for a level belongs to the caller if the opposite is not stated. For instance, the string referenced by an argument of type ‘char *’ is on the first level of indirection, the strings referenced by an argument of type ‘char **’ are on the second level, and so forth. If memory referenced, say, on level 1 belongs to the callee, we describe this fact by writing 'Mine(1)'. Ownership descriptions can be combined by concatenating them by means of the dot. Example 5.4 shows two typical examples where memory referenced by a return value is not allocated on behalf of the caller.

```
char *Foo::Name();
char **Foo::GetArgv();

Msg0(AZ(char, Out, Mine(1)), Foo, Name);
Msg0(AZZ(char, Out, Mine(1).Mine(2)), Foo, GetArgv);
```

Ex. 5.4 Describing Ownership of Memory

For ET++ objects, ownership descriptions are meaningless. If a pointer to an ET++ object is expected as part of an In argument, we map the object name to the object pointer. If a pointer to an ET++ object is referenced by an Out argument, then the call frame tells the object name mapper to generate a new name if necessary.

6. Call Frames and Callee Stubs

This section discusses call frames and how they relate to callee stubs. The Msg*N* macros discussed in section 5 not only expand to code that defines the statically allocated function meta object, but also code for the callee stub. A callee stub always has the signature as shown in example 6.1.

```
int SomeClass_Fct(SomeClass *receiver, CallFrame *ffp);
```

Ex. 6.1 Signature of Callee Stub

The callee stub is invoked by a dispatcher which is also responsible for providing the pointer to the receiver object[3], and an appropriate call frame object. The call frame object is used in the callee stub to provide the

[3] the ‘this’ pointer

member function call with arguments. The call frame has the actual arguments at its disposal in form of the external representation, e.g., a string with a particular syntax.

To invoke the member function, conversions are needed to transform values from the external representation to the C++ representation, and from the C++ representation to the external representation. A call frame object is responsible to do all these conversions. For accomplishing the conversions, the callee stub calls some member functions of the call frame to obtain the actual value of an argument. Of course, the macros presented in section 5 and the call frame's class interface are designed to smoothly work together.

A call frame needs the information contained in the member function meta object. The reason is to minimise the number of access functions we need to provide in the call frame's public interface. Selection of the appropriate conversion function is not done by name, but by exploiting the C++ matching mechanism for overloaded functions. This means, as the most important implication, that we need exactly one conversion function for all pointer to ET++ objects. There is no need to change the set of conversion functions if a new ET++ class is introduced. If we had to do so, the whole approach would be of little use.

The callee stub is coded such that its wrapped member function is not invoked if one of the argument conversions fails. Conditions of this kind are signalled to the dispatcher by the return value of the callee stub. In any case, the call frame is given the chance to do the necessary cleanup, i.e., to possibly deallocate memory belonging to the caller.

It is worthwhile to notice that type errors in the code generated by the function meta object definition macro are detected at compile time. Wrong specifications of mode, ownership, or size, cannot not be detected, of course.

7. Comparison with Similar Systems

In this section we compare our approach with CORBA [OMG91, OMG92] and MetaFlex [Joh93]. We chose CORBA because it is an emerging standard for distributed objects. This comparison is similar to comparing an aircraft carrier with a little boat, but it reveals interesting insights. MetaFlex, on the other hand, was developed with goals similar to ours but its developers ended up with a completely different approach.

CORBA

CORBA was designed with the idea of creating a standard for interacting with distributed objects. One of our goals was to find an inexpensive non-intrusive solution for (remotely) interacting with ET++ objects.

CORBA provides a traditional object model that can be mapped to a wide number of programming languages which do not even have to be object-oriented. Our object model is defined in C++, and it is restricted by the way C++ is used in ET++.

CORBA defines two ways how clients can interface with the Object Request Broker. Compiled clients use generated code stubs for invoking member functions of distributed objects. Interpreted clients use the dynamic invocation interface to pass an invocation request to the Object Request Broker. In our solution we cannot provide for interfacing via client stubs. Our intended clients are interpreters that use a dynamic invocation interface.

Every CORBA object request broker provides one single space for all its clients and makes it transparent in which server a certain object resides. In our solution every server has its own object name mapper. Object names are therefore only valid in the context of a server.

CORBA was designed is intended to provide for interoperability over different platforms of different vendors. Our solution is intended for scripting ET++ applications running on different platforms.

MetaFlex

MetaFlex was developed to accelerate the implementation of AppleScript support for applications implemented with Aldus' application framework. MetaFlex has therefore almost the same purpose as our dynamic invocation framework.

MetaFlex parses C++ class definitions to generate the meta information available at run time. The output of MetaFlex is C++ code which is compiled and linked with the code of the classes it describes. In order to reduce the amount of code that is generated MetaFlex provides a language that serves for specifying the kind and amount of meta information to be provided.

Compared to MetaFlex our approach is lightweight and inexpensive. MetaFlex is, however, a much more powerful system because it does not need to restrict the kind of C++ code that can be processed. MetaFlex allows a developer to control what kind of meta information is needed. Our approach merely permits a developer to specify which member functions are dispatchable.

The generality of MetaFlex has its price. Its implementation was no trivial endeavour whereas implementing our dynamic invocation framework took a relatively small effort. We could also profit from previous work such as the infrastructure for ET++ class meta objects [Gam89]. Johnson et alii also reported that they had to adapt the parser because it could not deal with code not compliant with the standard. We avoided such problems by letting the C++ compiler do the work.

MetaFlex as described in [Joh93] is neither portable, nor is the mechanism for dynamic invocation type-safe. Our dynamic invocation framework is type-safe and portable. The mentioned deficiencies are not inherent to the MetaFlex approach, however.

8. Conclusions

In this paper, we presented our dynamic invocation framework for string based dispatching of C++ member functions, and how we applied the framework when we embedded the scripting language Tcl into ET++. Our approach has proved powerful enough to endow the application framework ET++ with an infrastructure that supports scripting in general. It makes the tedious and error-prone work of writing callee stubs superfluous. The solution we described virtually eliminates all manual work.

There are different ways how the functionality for dynamically invoking C++ member functions can be achieved. We decided to sacrifice generality in favour of an inexpensive, but limited solution. Our approach works well only for a single-rooted class library like ET++. A solution like MetaFlex, which is based on a C++ parser, is general, but expensive in terms of development time.

A developer using an application framework with scripting support profits in two respects. First, he or she does not need to start from scratch to make an application scriptable. The basic functionality is rather inherited. As a welcome side effect, the scripting interface to the generic components of the application framework automatically remains the same. Second, the application framework offers a set of useful components that reduce the amount of work to make the specific functionality available for scripting.

Scripting on the interapplication level becomes increasingly important. If we use C++ to implement many applications, we need to overcome the static nature of that language. Supporting dynamic invocation of member functions is the key to solve the problem. As long as we use the same application framework for developing a number of applications, a specific approach like ours is feasible, sufficient and economically interesting.

A. References

[Appl93] Apple Computer, Inc.: AppleScript Developer's Toolkit Version 1.0 (Part Number 030-3994-A). Cupertino (CA), 1993.

[Bec88] Kent Beck, William Cunningham: A Laboratory For Teaching Object-Oriented Thinking. In: OOPSLA’89 Proceedings, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 372-377.

[Gam89] Erich Gamma, André Weinand, Rudolf Marty: Integration of a Programming Environment into ET++ - a Case Study. In: Proceedings ECOOP’89, Cambridge University Press, Cambridge (Nottingham, UK), 1989, pp. 283-297.

[Kof93] Thomas Kofler: Integrating Interpreters in the Application Framework ET++. Student Term Project in Computer Science, Institut für Informatik, Universität Zürich, 1993.

[Joh93] Richard Johnson, Murugappan Palaniappan: MetaFlex: A Flexible Metaclass Generator. In: Proceedings ECOOP'93, Springer-Verlag, July 1993, pp. 502-527.

[OMG91] Object Management Group: The Common Object Request Broker: Architecture and Specification. Revision 1.1 (OMG Document 91.12.1), 1991.

[OMG92] Object Management Group: Object Management Architecture Guide. 2nd Edition (OMG Document 92.11.1), Framingham (MA), 1992.

[Ost90] John K. Osterhout: Tcl: An Embeddable Command Language. In: Proceedings of the USENIX Winter Conference, Jan 1990, pp. 133-146.

[Wei88] André Weinand, Erich Gamma, Rudolf Marty: ET++ - an Object-Oriented Application Framework in C++. In: OOPSLA'88 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 168-182.

[Wei89] André Weinand, Erich Gamma, Rudolf Marty: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. In: Structured Programming, Vol. 10, No. 2, June 1989, pp. 63-87.