Object-Oriented Versus Conventional Software Development: A Comparative Case Study

W. Pree, G. Pomberger
Institut für Wirtschaftsinformatik, University of Linz,
A-4040 LINZ, Austria, Europe
E-mail: {pomberger,pree}@swe.uni-linz.ac.at

Abstract

Although object-oriented programming techniques have evolved into an accepted technology with recognized benefits for software development, profound qualitative and quantitative comparisons of conventional (module-oriented) and object-oriented systems are missing. We derive statements about qualitative and quantitative differences between conventional module-oriented and object-oriented software systems from the construction of a prototyping tool which was implemented in Modula-2 as well as in C++ (based on an object-oriented application framework). We also discuss the most striking reusability problems of object-oriented software building blocks.

Keywords:
object-oriented programming, module-oriented programming, reusability, extensibility, type extension, class libraries, application frameworks, C++, Modula-2, Ada, Oberon

Introduction

Unfortunately, most comparisons of conventional and object-oriented programming techniques are based on small projects since the software industry can hardly afford to implement large-scale software systems twice.

This situation served as an incentive for us to compare representative software systems. The conventionally implemented User Interface Construction Tool (UICT[1] [4, 7], implemented in Modula-2) and the object-oriented Dynamic Interface Construction Environment (DICE[1] [8, 9], implemented in C++ with the application framework ET++ [3, 12, 13]) form the basis of such a large-scale case study that helps to evaluate the promises of object-oriented programming as well as its pros and cons compared to module-oriented software system development. Both UICT and DICE serve for the prototyping-oriented development of graphic user interfaces under UNIX and either SunWindows, NeWS, or the X11 window system.

In the following comparison of UICT's and DICE's components, we take the reader's knowledge about module-oriented concepts such as encapsulation and data abstraction for granted. We also presuppose that the reader is familiar with the object-oriented concepts inheritance, polymorphism and dynamic binding, as well as with principles of graphic user interface appli-

[1] This project was supported by Siemens AG Munich

cation frameworks (like MacApp [10, 14], AppKit [6] or ET++ [13]).

Comparison of Tools Implemented with Module-Oriented and Object-Oriented Methods

The comparison of UICT's and DICE's development is based on concepts which the tools have in common. Table 1 shows these concepts and their realization in UICT and DICE.

Prototype Specification

Both UICT and DICE were designed with the goal to raise the abstraction level in order to support the specification of user interface prototypes. UICT employs a special description language (User Interface Specification Language, UISL); DICE provides a graphic-oriented specification formalism. Since the characteristics of these prototype specification formalisms are completely different, we cannot compare these components directly. Thus we discuss the reasons that led to these different approaches and their consequences.

UICT is based on a module-oriented toolbox comparable to the Macintosh Toolbox [2] that provides basic functionality for implementing graphic user interfaces. Because of their low abstraction level such module-oriented toolboxes are not suited for the specification of user interface prototypes. Thus a prototype specification by means of a formal language (with an abstraction level which is much higher than that of a toolbox) was chosen for the module-oriented implementation of the user interface prototyping tool. In case of UICT the module-oriented toolbox is only used for prototype simulation and code generation.

Object-oriented (user interface) application frameworks make the implementation of graphic user interfaces much easier. Furthermore, the abstraction level of prototype specification is considerably raised by means of graphic editors. These were the main reasons why a graphic-oriented specification formalism was chosen for the object-oriented implementation of the user interface prototyping tool.

On the one hand, a module-oriented toolbox influences the specification concept only to a minimal extent. Obviously, user interface elements supported by a toolbox are mirrored in the specification formalism. Nevertheless, the structure of the software component that implements the specification formalism is completely independent of the module structure of the toolbox. Instead, other design decisions (i.e., the decision to use a formal language for prototype specification) influence the architecture and design of the corresponding implementation.

On the other hand, the architecture of an application-framework-based user interface prototyping tool is strongly influenced by the structure of the application framework that is employed. Design and implementation of the prototype specification concept are above all determined by the predefined architecture of the particular application framework.

Prototype Representation—Employed Data Structures

The qualitative and quantitative comparison of UICT's and DICE's internal prototype representation considers the following aspects:

- We compare employed data structures as well as their management and investigate the influence of different ways of thinking (due to the module-oriented and object-oriented paradigm) on the internal prototype representation.
- The internal prototype representation plays an important role in each user interface prototyping tool. Effects of extensions/modifications of the internal representation are contrasted.

Concepts	Realization in UICT	Realization in DICE
Prototype Specification	User Interface Description Language (Text-Oriented)	Graphic-Oriented Specification Formalism
Prototype Representation (Data Structures)	Intermediate Language Based on Arrays	(Sub)classes of an Application Framework
Prototype Simulation	Interpreter	Simulation Framework
Code Generation	Conventional Generator	Code Generation Framework

Table 1: Common UICT and DICE concepts together with their realization

- The particular software components that implement the internal prototype representation are compared (using lines of code as a metric for a quantitative comparison). This comparison particularly considers the degree of software reuse.

Data Structures and Their Management

UICT and DICE have to store a prototype specification in an appropriate format. This is the precondition of a later simulation and code generation. We induce the following general statements from the comparison of employed data structures:

Data structures employed in the module-oriented implementation are not *active*. This means that modules dedicated to internal prototype representation store only *descriptions* of user interface objects, but not objects in an object-oriented sense. This causes other modules to implement routines that are based on these descriptions—descriptions (= data) and operations (= routines that manipulate and interpret these descriptions) are *separated*. This decreases the extensibility of the data structures.

A typical object-oriented implementation unifies data and operations and makes data structures active. This means that the description of an object's behavior forms a unit (the attributes of an object that constitute its state and all methods that determine its functionality). Furthermore, common behavior is factored out into an abstract class so that other components can be based on that class. Therefore modifications of common behavior are easier to accomplish.

Implication: The employed data structures and module/class architecture primarily result from different ways of thinking in the design of module-oriented and object-oriented software systems.

These general statements are drawn from UICT's and DICE's internal prototype representation:

Data Structures and Their Management in UICT: We first try to explain the main reason for UICT's overall module structure in order to outline the influence of such a modularization on the internal prototype representation: If a system is modularized, only the system's functionality is taken into account in most cases. UICT is a typical example. It consists of modules that *parse* a prototype specification, modules that *handle* tables (=the internal prototype representation), modules that *simulate* a prototype, and modules that *generate* code. Modules that handle an intermediate language stored in arrays (called UI Tables) realize UICT's internal prototype representation. All other modules are built around the table handler modules. (E.g., prototype specifications in UISL are parsed by Translator modules and then transformed and stored in UI Tables.)

A UI Table logically consists of two parts: an *object list* and a *symbol list*. The object list is a set of descriptions of user interface building blocks that are specified for a particular prototype. Thus an object list contains descriptions of user interface elements supported by UICT (e.g., menus, buttons, windows). The symbol list stores all names used in the prototype specification and for each name a reference to its description in the object list. Two modules, called *Table Handler* and *Symbol Table Handler*, manage UICT's prototype representation. This fine-grained modularization is based on the two logical parts of a UI Table.

Data Structures and Their Management in DICE: One design goal of DICE's internal prototype representation was that classes representing user interface elements should not only consist of their attributes described in instance variables but also of methods that determine their functionality, so that data and operations are really unified.

Another important design goal of DICE's internal prototype representation was to define an abstract class that factors out common behavior of user interface elements. Thus classes implementing graphic editor components as well as classes which initiate attribute definition (see below) of a user interface element, simulation of the particular prototype, and code generation can be based on such an abstract class. This is a precondition of a satisfying extensibility of the overall system.

We designed an abstract class called DICEItem for the reasons stated above. Subclasses of this abstract class represent specific user interface elements. Data management (i.e., management of several DICEItem instances) is accomplished by means of already existing ET++ classes for managing object collections.

Extensions of the Prototype Representation

By "extensions of the prototype representation" we mean adding new interface elements that are to be represented (this is the most important extension in connection with user interface prototyping tools). In this section we discuss the impact of such extensions on other components of the particular user interface prototyping tool.

In UICT's implementation object descriptions can be accessed from any other module in the system that imports the proper routines and data types from the modules that handle these descriptions. Changes (i.e., new kinds of object descriptions) to modules that handle these descriptions affect the whole software system. (Only changes to data structures that are internally employed by the table handler modules in order to store object descriptions do not affect the overall system.)

In DICE's implementation abstract classes define the common behavior of objects realizing the prototype representation. Other components of a system can be based on these abstract classes. Subclasses of the abstract classes describe specific object behavior. The software system is so flexible because new kinds of user interface elements can be added as subclasses to abstract classes without affecting the other components of the system.

Implications: The concepts inheritance, polymorphism and dynamic binding as offered by object-oriented programming languages are the preconditions for building extensible and reusable software systems: Inheritance opens the possibility of incrementally extending and adapting object descriptions of abstract classes that factor out common behavior. This is done in subclasses without changing the code of the abstract class. Polymorphism and dynamic binding allow the implementation of software components that are based on abstract classes. Thus these components are independent of specific object types.

Extensibility Aspects of UICT's Internal Prototype Representation:

An encapsulation of data structures as done in the Table Handler and Symbol Table Handler modules for an object list and a symbol list is a common used possibility in module-oriented systems to increase the extensibility of a software system. As explained below, the extensibility of such a module suffers from the separation of data and operations and not from the fact that even type extensions are not supported in typical module-oriented programming languages like Modula-2 and Ada [1].

Let us take the Table Handler's definition module as an example in order to illustrate this unsatisfying situation: A type TBHSubwindow (see Figure 1) is defined to describe different subwindows available in UICT. (A UICT prototype window can be divided into several

```
TYPE
        TBHExtension= RECORD
                Width: INTEGER;          (* in pixels *)
                Height: INTEGER          (* in pixels *)
        END;
        TBHSubwType= (Edit, ListMenu, MaskButton);
        TBHLongName= ARRAY [0..200] OF CHAR;
        TBHEditMenu= ...
        ...
        TBHSubwindow= RECORD
                (* attributes common to all subwindows *)
                Ext: TBHExtension;
```

```
            …
            CASE Type: TBHSubwType OF
                Edit:
                    EditMenu: TBHEditMenu;
                    FileName: TBHLongName;
            |   ListMenu: ...
            |   MaskButton: ...
            END (* CASE *)
        END; (* TBHSubwindow *)
```

Figure 1: Definition of type TBHSubwindow in UICT's Table Handler Definition Module

subwindows. UICT supports several types of subwindows, for instance, an Edit Subwindow with the functionality of a text editor, a ListMenu Subwindow that contains a list of selectable textitems, and a Mask Button Subwindow that contains dialog elements like buttons).

This implies that each routine in the implementation module that has a parameter of type TBHSubwindow must be modified if a new subwindow type is added in this type definition. (An implementation detail illustrates the typical functionality of routines that handle subwindows: The Table Handler Module manages separate data structures for each subwindow type in its implementation. The procedure TBHAddSubw(Subw: TBHSubwindow), for instance, has to determine the subwindow type in a CASE statement in order to get the information in which data structure the subwindow has to be placed. A new subwindow type implies the modification of all CASE statements in routines that handle subwindows and the implementation of an additional data structures that manages subwindows of the new type.)

One could argue that the replacement of the general type TBHSubwindow with simple types (e.g., TBHEditSubw, TBHListMenuSubw) would solve this problem. Unfortunately, this solution has similar drawbacks: First, attributes common to all subwindows (like Ext) recur in these simple types. Changes in the common attributes mean that all “simple” types and all routines that handle them must be changed accordingly. Furthermore, the use of the module becomes pretty complicated: routines to insert, remove, etc. subwindows must be implemented for each subwindow type (for example, TBHAdd-EditSubw(...), TBHAddListMenuSubw(...), etc.).

Though the concept of type extension (as supported by Oberon [16]) ameliorates this problem to some extent, it offers no satisfying solution to the extensibility problem: Let us again take the situation outlined above as an example. If we had type extension at hand, we could define a general type TBHSubwindow that contains only attributes that are common to all subwindows. Specific subwindow types (e.g., TBHEditSubw, TBHListMenuSubw, TBHMask-ButtonSubw, TBHEmptySubw) are extensions of this general type TBHSubwindow. Thus, for example, one routine that adds a subwindow to a table is sufficient: the subwindow to be added is passed as a parameter of type TBHSubwindow. New subwindow types that extend this general type can be handled by such a routine without modifications of that routine. Nevertheless, new user interface elements cause changes in other modules of UICT (in the simulation, and code generation components), regardless of whether type extension is used in the modules that handle UI Tables or not. For example, routines that implement the simulation of a subwindow have to know the particular subwindow type, of course, and thus are affected by such extensions.

The crucial point is that the data stored in Table Handler and Symbol Table Handler Modules are not *active*: These modules store only *descriptions* of objects. (They hide the internal representation of their object list and symbol list, and export only certain data structures and routines to access these lists.) Thus descriptions can be accessed from any module in the system that imports the proper routines and data types from the Table Handler and Symbol Table Handler Modules. Changes to Table Handler and Symbol Table Handler Modules (i.e., new kinds of object descriptions) affect the whole software system.

Extensibility Aspects of DICE's Internal Prototype Representation:

We illustrate the extensibility of DICE's components that are based on DICE's internal prototype representation (class DICEItem). We choose the attribute definition of user interface elements, which is part of DICE's specification component, as our example.

Attributes that describe a user interface element's behavior are defined in appropriate dialog boxes. Let us sketch the attribute definition of user interface elements from the user's point of view as a precondition of understanding its object-oriented implementation described below: In order to set attributes of user interface elements, one selects the particular user interface element and chooses the menu item "Item Attributes..." from a DICE-specific menu. A dialog box is opened where element-specific attributes are manipulated.

The implementation of attribute definition is based on the abstract class DICEItem: The crucial point is that classes comprising DICE's specification component should only implement a *framework* for attribute definition, i.e., handle mouse events, menu selections and the opening/closing of a dialog box in which attributes are manipulated. Only the window contents (i.e., the specific attributes) of the dialog box must be provided by the particular user interface element. The update of specific attributes which may be changed in the dialog box has to be accomplished by the particular user interface element, too, since attributes are instance variables of each user interface element. So the abstract class DICEItem defines two (abstract) methods for attribute definition called GetAttributesDialog and UpdateAttributes.

Arbitrary new interface elements can be added as subclasses of DICEItem. They just have to override the element-specific methods for attribute definition GetAttributesDialog and Update-Attributes. Furthermore, instance variables representing their attributes must be added. Algorithms for attribute definition as implemented in the specification component still work since they are based on the abstract class DICEItem.

Thus typical object-oriented programming techniques (inheritance, dynamic binding, and polymorphism) are the precondition for the implementation of a framework for attribute definition.

Methods of DICEItem for simulation and code generation are designed in an analogous way. So simulation and code generation frameworks can be implemented based on DICEItem. They need not be changed if new user interface elements are added as subclasses of DICEItem.

Software Reuse

The comparison between UICT and DICE components that realize the particular internal prototype representation points out significant differences between module-oriented and object-oriented systems regarding reuse of already existing software components.

We use the written lines of code and the average number of lines of code per routine/method to directly compare UICT's and DICE's components (see Table 2).

The amount of code that has to be written in UICT's Table Handler Module is about seven times as much as in DICE's component for internal prototype representation. The complexity of routines can be cut back to about one third in the object-oriented implementation. The main reason for such code and complexity reductions is the high reusability of application framework components.

The module-oriented user interface prototyping tool does not even reuse existing modules for managing data structures like lists, etc. Everything is implemented from scratch. This is done in such a way that modules implemented for that purpose in UICT are again very specific.

The object-oriented realization of the internal prototype representation reuses as much code as possible from the application framework it is based on. In case of DICE's prototype representation component, we can also calculate the ratio of reused and newly written code (see Table 3).

This ratio (newly written lines of code : reused lines of code = 1052 : 1307 = 1 : 1,24) strongly depends on the application framework that is used and the originality of the problem at hand.

	UICT's Prototype Representation Component	DICE's Prototype Representation Component
written lines of code	7600	1052
routines/methods	159	62
lines per routine/method	47,8	17,0

Table 2: Comparison of UICT's and DICE's prototype representation components based on common metrics

	DICE's Prototype Representation Component
newly written lines of code	1052
reused from ET++ classes	1307

Table 3: Ratio of reused and newly written code

Simulation and Code Generation

Design and implementation of these components are influenced by the internal prototype representation which is in turn determined by different ways of thinking in module-oriented and object-oriented systems, as discussed above.

Since a detailed description of these components would yield no further insights we just compare these components as shown in Tables 4 and 5 using again the overall lines of code and the average lines of code per routine/method as metrics:

The direct comparison again confirms that the amount of code that is to be written in UICT's components as well as the code's complexity is significantly higher than in the corresponding components of DICE.

UICT's components are almost implemented from scratch and only reuse routines of a module-oriented toolbox for the implementation of window system specific tasks. DICE's components again use some ET++ mechanisms/classes so that the calculated reusability factor is similar to the internal prototype representation component (i.e., the number of newly written lines of code is less than half compared to the number of reused lines of code).

As already described in context with the internal prototype representation the components with a module-oriented implementation are not open for extensions (i.e., adding new user interface elements means that also the simulation and code generation components have to be changed). On the other hand, the object-oriented solution is optimal: due to abstract classes, no modification of these components is necessary: user interfaces elements are simply added as subclasses of DICEItem.

Summarizing Remarks

The comparison between module-oriented and object-oriented software development impressively demonstrates the benefits of the object-oriented paradigm. The key point is that the object-oriented paradigm (encapsulation, inheritance, polymorphism and dynamic binding) encourages the building of extensible systems and software reuse.

The comparative case study confirms that concepts introduced by object-oriented programming languages are quite different from already existing ones and, furthermore, necessary in order to improve software quality in general. Reusability contributes a lot to improve software quality: Obviously, the quality of reusable building blocks determines the quality of the software system which is based on them to a high degree.

The module-oriented paradigm is not suited to building software components that are even remotely as reusable as object-oriented components. Thus the design and architecure of existing modules has almost no impact on the overall quality of new systems.

	UICT's Prototype Simulation Component	DICE's Prototype Simulation Component
written lines of code	5926	1155
routines/methods	58	81
lines per routine/method	102,2	14,3

Table 4: Comparison of UICT's and DICE's prototype simulation components

	UICT's Code Generation Component	DICE's Code Generation Component
written lines of code	7976	1079
routines/methods	96	28
lines per routine/method	83,1	38,5

Table 5: Comparison of UICT's and DICE's code generation components

Due to the new dimension of reusability of object-oriented software, the implementation effort can be drastically reduced, as the quantitative comparisons demonstrate.

To sum up, the object-oriented paradigm makes qualitative and quantitative improvements of software development possible. Nevertheless, these improvements do not happen by accident. As sufficiently known, the object-oriented programming paradigm requires rethinking of software development in general. Unfortunately the programmer is also faced with a lot of problems during object-oriented system development. The following section focuses on reusability problems that are still inherent in object-oriented software building blocks.

Reusability Problems of Object-Oriented Software Building Blocks

Efficient object-oriented system development strongly depends on the extensibility and reusability of available software building blocks. During the development of DICE we identified some problems that make the reuse of software components (though implemented with object-oriented programming techniques) rather difficult:

Inefficient Reuse

It is often unclear how specific behavior can be added in an optimal way to predefined software components:

- by the extension of classes (creating subclasses),
- by the implementation of new classes, or
- by using mechanisms (e.g., change propagation as described in [5]) that are supported by the particular class library/application framework.

The root of this problem is the high complexity of class libraries and especially application frameworks. In order to decide how specific behavior can best be implemented on the basis of a class library/application framework, a high degree of familiarity with the respective class library/application framework is mandatory. The user of classes must know exactly which features the individual classes provide and even sometimes how the behavior of individual classes is implemented.

Thus applications based on powerful, complex class libraries/application frameworks will sometimes be inefficient: application programmers who reuse software components often do not (and should not) have intimate knowledge about implementation aspects of particular features provided by reusable components. Although one can assume that particular services of software components rely on efficient implementation techniques, it is difficult to assess whether certain combinations are still efficient.

Our experience has shown that application-specific solutions are often chosen that are too cumbersome, even after a designer has accumulated extensive experience with a particular application framework. For example, in the prototype specification component of DICE ET++'s change propagation mechanism was used to recompute the layout after making size changes in user interface elements. After consulting one of the developers of ET++, it was determined that recomputation of the layout could be invoked by directly calling a particular method (at the right time), which is much more efficient than using change propagation.

Difficult Maintenance of Reusable Software Components

If classes are reused in other projects, the need for more general versions becomes apparent. To address this issue, Winblad et al. [15] suggest dividing programmers into two groups: *class programmers* and *application programmers*. Class programmers are preferably highly qualified software engineers who develop classes and enhance them for reusability in future projects. Application programmers produce applications as quickly as possible by means of reusable components and give the class programmers necessary feedback to make these components more reusable.

Nevertheless, the evolution of class hierarchies by the class programmers can also have subtle side effects. The "advance warning" of application programmers by class programmers about changes in reusable software components is almost impossible since the effects of (even slight) modifications in class libraries/application frameworks often cannot be predicted. A report of all changes is also unpracticable because application programmers would have to understand many implementation details in order to comprehend all changes and their effect on applications based on these modified classes.

For example, DICE was always ported to more recent versions of ET++ as soon as these versions were available in order to profit from improvements. The migration to more recent ET++ versions was usually no problem, although class interfaces often changed radically. The benefits (unification of concepts in new classes, better performance, etc.) outweighed the effort (in average 2-4 person days). But one example demonstrates that even almost negligible changes can have enormous effects on applications that are based on a class library/application framework: ET++, for example, implements a hook method

```
virtual bool IsEqual(Object *op);
```

in class Object, the root of ET++'s class hierarchy. IsEqual is used in several other classes—especially in collection classes like SortedObjList and Set—in order to compare two objects. The implementation of this method in class Object can only check pointer identity. Subclasses (e.g., TextItem) have to override this method accordingly.

All editing operations of DICE's prototype specification component are based on pointer identity of DICEItem instances. (DICEItem is derived from the ET++ class VisualObject which is in turn a sublass of ET++'s root class Object.) Since the desired behavior of IsEqual is already implemented in Object, it was not necessary to override this method in DICEItem. After DICE was ported to an improved version of ET++, some of the editing functions did not work properly (user interface elements that were moved within a window specification editor were removed together with others, etc.). Only a careful examination of the source code of ET++'s collection classes made it obvious that comparison of objects by means of IsEqual was wrong: This method of class Object was not overridden in VisualObject in earlier versions of ET++. The new ET++ version implements a method IsEqual in VisualObject. So DICEItem inherits a behavior of IsEqual from VisualObject which is different from pointer identity. The elimination of such errors is easy (in this example a proper IsEqual method in DICEItem), but their detection can sometimes mean hard work that requires knowledge of implementation details of reusable classes.

Restricted Extensibility of C++ Classes

C++ requires that the programmer explicitly declare methods to be dynamically bound. Furthermore, instance variables of a class can be protected so that subclasses have no access to them. These characteristics of C++ imply that C++ classes can often only be reused and extended without problems if the designer has forseen the wishes of future users. In many cases reuse is only possible if the source code of the class to be modified is available. These facts directly contradict the promises of object-oriented programming.

Let us illustrate this statement. Class behavior can be modified by overriding dynamically linked methods (called hook methods) in subclasses.

This procedure is possible only at locations where the class designer provided hooks. Application frameworks like ET++ in particular provide a structure in their “empty” application which is to be extended accordingly for an application that is based on it. Certain extensions are necessary for almost every application. These extensions are prescribed by the framework (by means of hook methods) and forseen by the designer of the framework classes. In this case it is easy to reuse and extend those building blocks. For extensions that were not foreseen, the source code of the respective superclass method(s) must be copied and complemented by the desired statements. This procedure is only possible if

- the source code of the class to be modified is available; and
- the instance variables used in the respective method(s) can be accessed in their subclasses.

Another design principle of application frameworks restricts its extensibility independent of the particular implementation language. State-of-the-art application frameworks usually adhere to the idea of a narrow inheritance interface [13]: Behavior that is spread over several methods in a class should be based on a minimal set of methods that have to be overridden in subclasses. Thus a client deriving subclasses from an existing class has to override just a few methods in order to adapt its behavior. Not adhering to this narrow inheritance principle often means that too many methods have to be overridden, resulting in ugly and bulky code.

As a consequence modifications not forseen by a class designer imply a reimplementation of considerable parts of the respective class.

Source Code Required for Modifications

Experience has proven that almost no software component is free of errors. This is also true of components that are intended to be reused. In order to correct errors or to eliminate restrictions in the library classes by creating subclasses, the source code is often necessary to understand class behavior and to find the methods that have to be modified.

The comprehension of a class to be modified can become more difficult due to the fact that objects often send messages to themselves, which may cause the execution of methods up and down the class hierarchy [11].

Conclusion

The comparison of module-oriented and object-oriented system development demonstrates that object-oriented programming techniques are important techniques to produce software components that are open for extensions and thus reusable. Thus software quality can be raised and the amount of code to be written can be reduced.

However, if one considers application frameworks as an effort to apply the object-oriented programming paradigm as cleanly as possible, then one must conclude that this paradigm is not sufficient for the achievement of systems that are extensible/reusable without considerable problems. Extensibility/reusability still has limits that cannot satisfy a system developer.

References

[1] ANSI and AJPO: Military Standard: Ada Programming Language; American National Standards Institute and United States Government Department of Defense, Ada Joint Program Office, ANSI/MIL-STD-1815A-1983, 1983.

[2] Apple Computer: Inside Macintosh, Vol. I-V; Addison-Wesley, 1985-1988.

[3] Gamma E., Weinand A., Marty R.: Integration of a Programming Environment into ET++: A Case Study; Proceedings of the 1989 ECOOP, July 1989.

[4] Keller R.: Prototyping-Oriented System Specification—Concepts, Methods, Tools and Implications (in German); Verlag Dr. Kovac, 1989.

[5] Krasner G.E., Pope S.T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80; Journal of Object-Oriented Programming 1, 3 (Aug./Sept. 1988).

[6] NeXT, Inc.: 1.0 Technical Documentation: Concepts; NeXT, Inc., Redwood City, CA, 1990.

[7] Pomberger G., Bischofberger W.R., Keller R., Schmidt D.: TOPOS—A Toolset for Prototyping-oriented Software Development; in Actes de la 4ème Conferénce de Génie Logiciel, AFCET, Paris, Oct. 1988.

[8] Pomberger G., Bischofberger W., Kolb D., Pree W., Schlemm H.: Prototyping-Oriented Software Development, Concepts and Tools; in Structured Programming Vol.12, No.1, Springer 1991.

[9] Pree W.: DICE—An Object-Oriented Tool for Rapid Prototyping; in Proceedings of Tools Pacific '90 (Sydney, Australia, 1990).

[10] Schmucker K.: Object-Oriented Programming for the Macintosh; Hayden, Hasbrouck Heights, New Jersey, 1986.

[11] Taenzer D., Ganti M., Podar S.: Problems in Object-Oriented Software Reuse, Proceedings of the 1989 ECOOP, July 1989.

[12] Weinand A., Gamma E., Marty R.: ET++ - An Object-Oriented Application Framework in C++; OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988.

[13] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework; in Structured Programming Vol.10, No.2, Springer 1989.

[14] Wilson D.A., Rosenstein L.S., Shafer D.: Programming with MacApp; Addison-Wesley, 1990.

[15] Winblad A.L., Edwards S.D., King D.R.: Object-Oriented Software, Addison-Wesley, 1990.

[16] Wirth N.: The Programming Language Oberon; Software Practice and Experience, 18, 7.