Object-Oriented Software Development Based on Clusters:

Concepts, Consequences and Examples

Wolfgang Pree
Institut für Wirtschaftsinformatik, University of Linz
A-4040 LINZ, Austria, Europe
E-mail: pree@swe.uni-linz.ac.at

Abstract

This paper focuses on object-oriented system development, presenting insights gained from the object-oriented design and implementation of a graphic user interface prototyping tool. The development of this tool corroborates the cluster model of the software life cycle—a model that was first described by Meyer [Meye88, Meye89].

We sketch basic ideas of the cluster model and draw general statements (e.g., how clusters are formed; implications on design methods) from the cluster-oriented design and implementation of a user interface prototyping tool. Examples substantiate that the development of object-oriented systems is characterized best by the cluster model.

Keywords:
object-oriented software life cycle, cluster model, object-oriented design, application frameworks, abstract classes, single inheritance, multiple inheritance

1 Introduction

We presuppose that the reader is familiar with basic object-oriented concepts (independent of a specific language): encapsulation, data abstraction, inheritance, polymorphism and dynamic binding, as well as with principles of graphic user interface application frameworks. (Examples of such frameworks are MacApp [Schm86, Wils90], AppKit [NeXT90] and ET++ [Wein88, Gamm89, Wein89, Gamm90]).

The **D**ynamic **I**nterface **C**reation **E**nvironment (DICE[1]) [Pree90, Pomb91] supports the graphic specification of user interface layouts and offers several ways to enhance their functionality. DICE was realized with the framework ET++, which was implemented in C++ and runs under UNIX and either SunWindows, NeWS, or the X11 window system.

Building DICE on the basis of an application framework is an approach that is often considered to be optimal for object-oriented system development. Further facts that allow the derivation of general implications from the development of DICE are:

- The project size itself (DICE having been developed with an effort of about two person years) provides a sufficient empirical basis for drawing conclusions.
- DICE is a "pure" object-oriented software system consisting only of classes. DICE makes no use of the ability to mix conventional routines that are not members of any class with object-oriented method calls, a typical feature of a hybrid language like C++ .
- DICE is a "typical" application-framework-based software system (i.e., the components of the framework ET++ satisfy the needs of DICE to a high degree).

2 Basic Concepts of the Cluster Model

Meyer [Meye89, Meyer90] coined the term "cluster model" of the software life cycle; a cluster, in this context, is a group of related classes. The development of classes belonging to one cluster constitutes *one* life cycle. The following changes to the usual software life cycle become necessary in this model:

- Activities of the software life cycle are not applied to the system as a whole. (The all-or-nothing approach considers a system as a monolithic entity.) Instead, system development is split into several sub-life cycles overlaping in time.
- The early software life cycle activities (analysis, specification and (re)design) and implementation are merged into one activity—a fact that completely contradicts the usual software life cycle models. The reason why this is possible and makes sense is explained in the next chapter.

[1] This project was supported by Siemens AG Munich

- A new activity—called *generalization*—is introduced in order to produce reusable software components. This activity can be merged with the test activity.

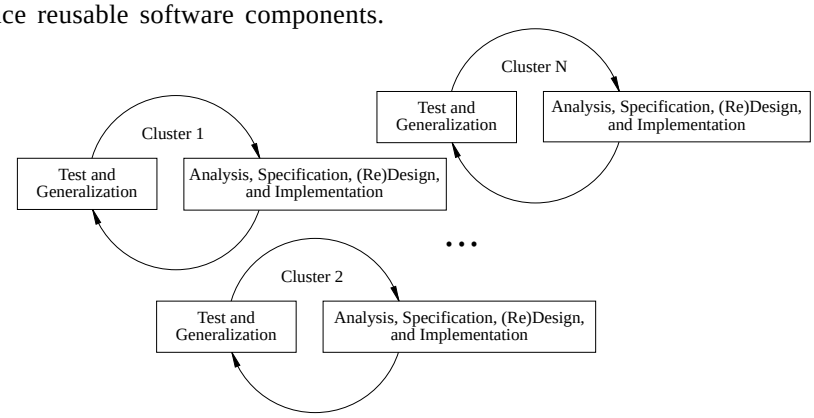

Figure 1: Cluster model of the software life cycle (adapted from [Meye89])

- A new activity—called *generalization*—is introduced in order to produce reusable software components. This activity can be merged with the test activity.

A software life cycle as depicted in Figure 1 results. Note that the activities analysis, (re)design, specification and implementation are merged into one activity.

The recommended order of the cluster life cycle is from the most general clusters (providing some utility features) to the application-specific ones.

The following case study illustrates that the cluster model is applicable for the object-oriented development of whole software systems, especially if these systems are based on powerful class libraries/application frameworks.

3 Cluster-Oriented System Development: A Case Study

This section is divided into two parts: First we draw general statements from the development of DICE that confirm the cluster-oriented software life cycle of object-oriented systems. DICE-specific examples substantiate these statements and further illustrate the cluster-oriented development strategy.

3.1 Derived Results

We derive the following general results from the development of DICE:

(1) Clusters are formed in a natural way (either prescribed by the application framework or by tasks to be accomplished by the system under development). These clusters have independent life cycles, although some classes may be part of different clusters. This is possible because one cluster implies only the development of certain methods of a class. These methods are usually independent of other features of a class. This fact implies that the chronological order of particular sub-life cycles is left to the system developer.

(2) Analysis, specification, design and implementation activities are merged into one activity if object-oriented programming techniques are applied in order to construct a system bottom-up from reusable, predefined software components:

- Analysis, specification, design and implementation are strongly influenced by application framework classes.
- Abstract classes are the main reason why design and implementation activities can be merged: the methods of the abstract classes are developed together with the design/implementation of other classes within a particular cluster. Class design of one class typically evolves with the design/implementation of other classes.

(3) Experience gained during the cluster-oriented development of DICE raises the question whether analysis and design methods can ever be developed that adequately support object-oriented system construction. Many of the “object-oriented” development methodologies (such as those from Booch [Booc86, Booc87]) were structured specifically with Ada in mind [Hend90]. So object-oriented programming techniques like inheritance and dynamic binding are not taken into consideration since Ada is just a module-oriented language. Other methodologies (like that of Abbott [Abbo83]) neglect both the reusability aspect as well as the fact that design and implementation phases are merged by means of abstract classes.

Reusable/extensible classes are not designed a priori. Design and redesign cycles (including a generalization activity) are absolutely necessary in order to yield satisfying solutions for a particular software system, as well as classes that might be reusable in other projects. It is questionable whether the optimum can ever be reached since weaknesses of classes are only discovered when they are used.

3.2 Examples Corroborating Derived Results

In each case we choose an adequate, but not unique example from the development of DICE to illustrate statements made in the previous section. In order make the examples comprehensible, we have to explain a few

details concerning DICE and the employed application framework, ET++:

DICE consists of several logical components that are typical of user interface prototyping tools: Graphic editors form the *prototype specification component*. The *internal prototype representation* plays an important role in each user interface prototyping tool. We designed an abstract class DICEItem (the design process is described below) that factors out common behavior of user interface elements so that other components of DICE (*simulation* and *code generation components*) can be based on that class. Instances of a class ProtoInfoItem represent a DICE-prototype (consisting of several windows, specific attributes, etc.) as a whole.

The ET++ classes Application, Document, View and Window implement as far as possible the *generic design* (the look and feel) of a graphic user interface built with ET++. VObject (visual object) is the abstract superclass for all visual objects that are displayed on the screen. It factors out high-level algorithms (e.g., methods for drawing a graphic object on the screen—which are typically overridden in subclasses) working on any kind of visually represented object. CompositeVObject combines an arbitrary number of VObject instances into a single one. As CompositeVObject is itself a subclass of VObject, instances of CompositeVObject or VObject can easily be nested in a tree-like manner.

The general mechanisms of VObject and CompositeVObject (i.e., event handling and the display or grouping of visual objects) together with classes Group and Expander (subclasses of CompositeVObject that care for the alignment of contained VObject instances) and other concrete or abstract classes (ActionButton, RadioButton, ToggleButton, PopupItem, View, etc.) are used to build every visible component of the user interface with which the end user interacts by means of the keyboard or mouse. Another ET++ class Command supports undoable commands and feedback for operations like dragging visual objects.

Example: Splitting DICE's Principal Classes into Clusters

This example substantiates result 1 presented in the previous section: In DICE we discern four principal development clusters:

- *application cluster*, consisting of the ET++ classes Application, Document, and Window, and appropriate DICE-specific subclasses
- *graphic-oriented-editor cluster*, consisting of the ET++ classes View and Command as well as of class DICEItem with all its subclasses
- *simulation cluster*, consisting of DICEItem and ProtoInfoItem
- *code generation cluster*, consisting of DICEItem and ProtoInfoItem

The application and graphic-oriented-editor clusters are prescribed by the application framework ET++. The simulation and code generation clusters result from DICE-specific tasks (prototype simulation and C++ code generation) as well as from the design decision to unify methods for these tasks in a class DICEItem.

The clusters sketched above have independent life cycles, although the classes DICEItem and ProtoInfoItem are part of different clusters. This is possible because the methods accomplishing the simulation in DICEItem and ProtoInfoItem are independent of methods for code generation, which are also implemented in these classes.

Example: Merging Specification, Design and Implementation Activities

The following example illustrates the influence of abstract classes on the software life cycle activities (see result 2 in the previous section).

DICE, for instance, needs a simulation component in order to transform the specification of a prototype into an operational one. Among other tasks the simulation component has to open windows according to the specified windows of a particular prototype. The only thing that cannot be implemented in the simulation component is the window contents to be displayed in these windows. Thus a method

```
virtual VObject *AsSimulationPart();
```

is defined in a class DICEItem for this purpose. The functionality of AsSimulationPart is simply to generate a copy of the particular user interface element and return a pointer to this cloned object. So the simulation component can be based on this method.

The implementation of AsSimulationPart is very simple and prescribed by ET++ because ET++ provides a DeepClone method in its root class Object. This method returns an exact copy (objects referenced by instance variables being copied, too) of an arbitrarily complex object. Thus the method AsSimulationPart can already be implemented in DICEItem. Only a few subclasses (e.g., those which represent view objects) override this method in order to make additional initializations. This is why AsSimulationPart is a dynamically bound method (virtual declaration).

The simulation component therefore constitutes a framework based on the abstract class DICEItem. The methods of DICEItem (in this example only AsSimulationPart) and of the class(es) realizing the prototype simulation evolve together.

The implementation of the appropriate methods in subclasses of DICEItem may also overlap with the design and implementation of the simulation component, which can already be tested, regardless of whether corresponding abstract methods of DICEItem are overridden in subclasses.

Example: (Re)Design and Generalization of Abstract Classes

We discuss this comprehensive example in detail in order to illustrate the cluster-oriented design and development of the abstract class DICEItem. This class plays a central role in three principal development clusters of DICE (i.e., the graphic-oriented-editor cluster, the simulation cluster, and the code generation cluster). The difficulties in the design of DICEItem also corroborate result 3 in the previous section.

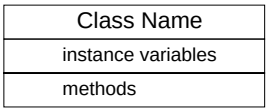

Figure 2: Graphic representation of a class

Figure 2 shows a general scheme of how classes are depicted. For each class the class name, then the relevant instance variables (variable name and type), and finally the definition of relevant methods (i.e., method name and parameters) are given in separate boxes. Depicted instance variables are assumed to be *protected* in the sense of C++, i.e. accessible only in methods of the particular class and its subclasses. Methods are supposed to be *public*, i.e., accessible from the whole system.

Initial Design of Classes Representing User Interface Elements

Our primary design goal is to define an abstract class that factors out common behavior of user interface elements so that classes implementing graphic-oriented-editor components as well as classes which initiate simulation and code generation of the particular prototype, can be based on such an abstract class.

We not only need an abstract class that factors out common behavior of user interface elements; because of the editing requirements placed on DICE's prototype window editor (recursive nesting of VObject instances in Group and Expander objects, editing operations like moving user interface elements within such a tree of user interface elements, etc.), extensions of ET++ classes become necessary, too:

(1) The ET++ classes Group and Expander only provide a method Add that puts a VObject instance as the last element in the group of VObject instances. Due to the required editing operations of DICE's window editor, additional methods are necessary: InsertBefore, which inserts a VObject instance before a VObject instance that is already contained in the group of the particular Group or Expander object, and Remove to remove a VObject instance from the group of VObject instances.

(2) Objects that are stored in a Group or Expander object of a window editor also have to manage their parent information in order to make, for instance, undoable editing operations possible; i.e., objects representing user interface elements must know which Group or Expander object is their parent.

In order to extend the classes Group and Expander, we simply built subclasses of these classes (EditGroup and EditExpander) and added methods as described in (1).

All classes that implement user interface elements in ET++ are subclasses of VObject. Thus class VObject has to be modified to manage parent information: a new instance variable that points to the parent of the respective VObject instance as well as methods SetParent and GetParent to manage this instance variable fulfill the requirements discussed in (2). Such a change in the framework class VObject would be possible (the source code of ET++ is available in the public domain) and the easiest way to add parent information for all subclasses of VObject. Since one of our basic premises was not to change the application framework itself, we refrained from such a modification. This restriction allows us to test the suitability of the object-oriented approach to the extension and modification of given software building blocks. Available object-oriented programming techniques offer two possibilities to extend class VObject without changing its source code:

- One possibility is to use single inheritance by applying the *wrapping concept*: In order to add instance variables/methods to a class (let us take VObject, for example) and its subclasses, we have to build a subclass—e.g., VObjectNode in our example—of the respective class. VObjectNode has at least one instance variable (let us call it wrappedObject) that points to an instance of any subclass of VObject; i.e., wrappedObject is of type "pointer to VObject" and points to an arbitrary VObject instance generated from one of VObject's subclasses. Thus VObjectNode "wraps" a VObject instance by means of its instance variable wrappedObject. VObjectNode redirects all dynamically bound methods of VObject to wrappedObject and adds instance variables and methods that are necessary to manage parent information. Thus instances of VObjectNode unify the behavior of the wrapped object and the additional behavior (managing parent information) implemented in VObjectNode itself.

  Figure 3 sketches class VObjectNode using the general scheme presented in Figure 2. The arrow denotes that VObjectNode is a subclass of VObject (thus we let the class tree "grow" from the top down). Note that VObject * means "pointer to an object of dynamic type VObject".

- Using multiple inheritance would imply a class hierarchy as depicted in Figure 4. Wrapping a VObject instance and redirecting dynamically bound methods to the wrapped object become superfluous. Instead of wrapping VObjectNode instances, we have to build a new subclass that inherits from VObjectNode and the particular VObject subclass to which the behavior of

VObjectNode should be added. Class NodeTextItem in Figure 4, for instance, inherits from class VObjectNode and TextItem, so that instances of NodeTextItem unify the behavior implemented in both classes.

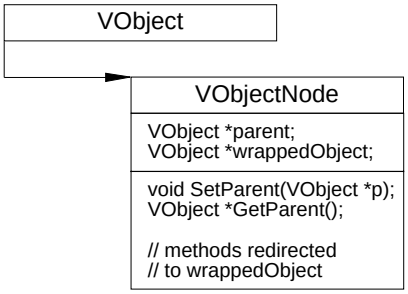

Figure 3: Class VObjectNode

The version of ET++ that was used is incompatible with multiple inheritance (e.g., collection classes cannot be used to store objects which are generated from classes that are derived from more than one base class). Furthermore, the author is convinced that single inheritance produces cleaner class hierarchies. For instance, if multiple inheritance is used as depicted in Figure 4, the class designer has to decide (in case of using C++) if the replicated behavior of VObject in class NodeTextItem (behavior from VObject is inherited from VObjectNode and TextItem) is actually stored once or twice. Furthermore, ambiguous access to base class members must be resolved by qualifying names with the proper class name. (E.g., dynamically bound methods inherited from VObject are ambiguous in NodeTextItem. So all these methods must be overridden in NodeTextItem in order to express explicitly whether the corresponding methods of either TextItem or VObjectNode are to be called.) If these drawbacks of multiple inheritance are taken into consideration, the solution with single inheritance is to be preferred, and not just because ET++ is incompatible with multiple inheritance. Thus the further design of the DICE-specific class hierarchy is exclusively based on the solution with single inheritance.

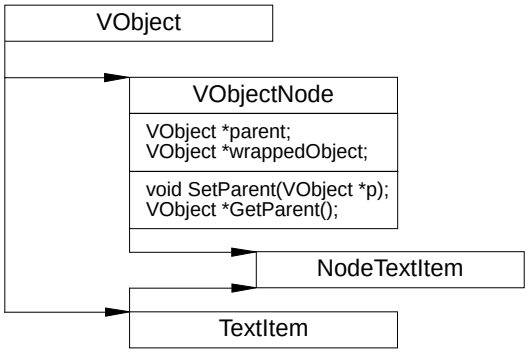

Figure 4: Adding parent information to VObject subclasses with multiple inheritance

The data structure of user interface elements shown in a prototype window editor results from a combination of VObjectNode and EditExpander or of VObjectNode and EditGroup: A prototype window edited consequently contains a group of VObjectNode instances stored in an EditExpander or EditGroup instance. An EditExpander or EditGroup instance is itself wrapped by a VObjectNode instance. Figure 5 outlines this data structure.

Thus a VObjectNode instance always wraps an object that is created from a subclass of VObject. Figure 6 depicts the structure of one VObjectNode.

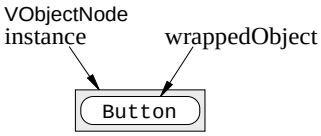

Figure 6: Data structure of one VObjectNode

Next we discuss a design decision that turned out to be wrong. Unfortunatley, we lacked sufficient experience during the design process to choose a better solution a priori, a situation that is typical in the design of classes.

Let us first sketch the reason for our wrong design: Up to now we had designed classes (EditExpander, EditGroup, and VObjectNode) that implemented behavior described in (1) and (2) at the beginning of this example, i.e., classes that help to implement editing functions. What we still needed to design was an abstract class that factors out common behavior of user interface elements. Let us call this class DICEItem. All user interface elements supported by DICE had to be subclasses of DICEItem and override the proper element-specific methods.

As a consequence, VObjectNode instances should not wrap arbitrary VObject instances, but only DICEItem objects. Thus the type of wrappedObject has to be DICEItem * instead of VObject *. The resulting class hierarchy is depicted in Figure 7.

Redesign of the Class Hierarchy

Only with the use of these classes in the graphic-oriented-editor, simulation and code generation clusters and by considering the resulting complex data structure of user interface elements (double encapsulation per user interface element—see Figure 8) did it become clear that the classes VObjectNode and DICEItem could be merged into one class: DICEItem can also implement the behavior of VObjectNode (i.e., instance variables parent and wrappedObject as well as the methods SetParent, GetParent, and all dynamically bound methods of VObject that must be redirected to wrappedObject). Thus the class VObjectNode becomes superfluous. The instance variable wrappedObject in this modified class DICEItem is again of type VObject *.

The unification of VObjectNode and DICEItem to the class DICEItem implies a data structure of user interface

elements that is again as simple as depicted in Figures 5 and 6. The only difference is that objects are not of type VObjectNode, but of type DICEItem. The depicted objects (static text, action buttons, and labeled buttons) are instances of subclasses of DICEItem (DICETextItem,

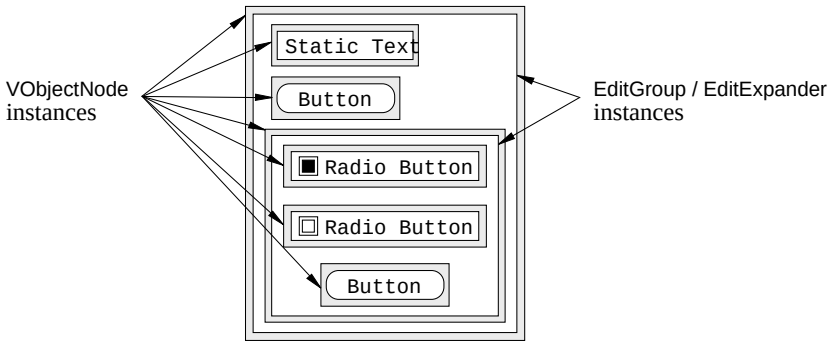

Figure 5: Data structure of user interface elements

DICEActionButton, DICELabeledButton) and thus also have the type DICEItem due to polymorphism.

Generalization Activity

DICEItem itself will not be reusable in other applications since it implements an abstract protocol that is specific for requirements of the user interface prototyping tool DICE (e.g., simulation and code generation). Thus a generalization activity is missing in one of the clusters DICEItem is part of: DICEItem, for example, could be split into two classes (e.g., VObjectNode and DICEItem). These classes should not be siblings in the class hierarchy (as depicted in Figure 7), but DICEItem should inherit from VObjectNode. VObjectNode just manages

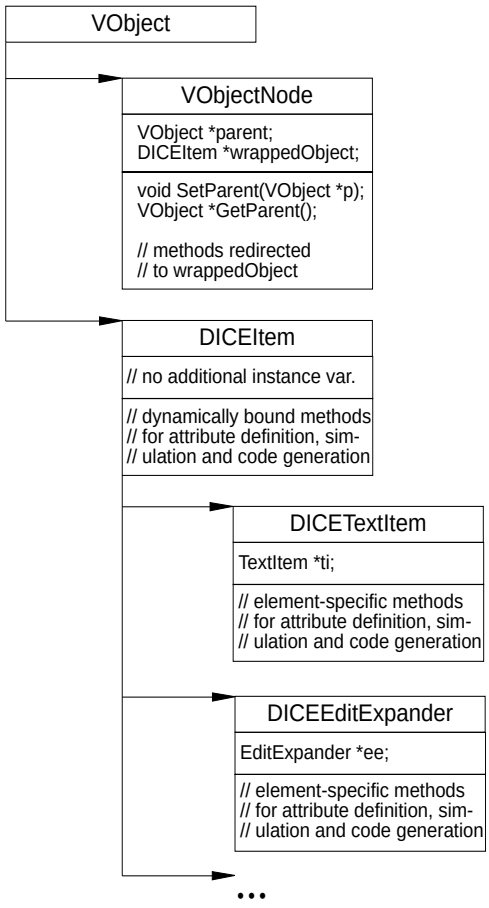

Figure 7: Extended class hierarchy

parent information for a VObject instance. Since DICEItem needs this behavior, it inherits it from VObjectNode. DICEItem implements only DICE-specific bahavior (simulation, code generation, etc.). This redesign factors out general, (re)usable behavior from the former class DICEItem into a class VObjectNode, which will be reusable in other applications that need parent information for VObject instances.

This detailed example was presented in order to discuss some aspects of class design in a cluster-oriented

software life cycle: initial design, redesign and generalization of DICEItem have been sketched. The described design/redesign cycles of DICEItem also illustrate that reusable/ extensible classes are not designed a priori.

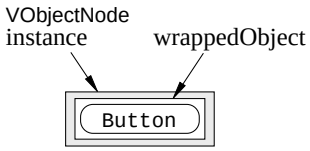

Figure 8: Double encapsulation per user interface element

4 Summarizing Remarks

To sum up, the development of DICE is characterized best by a cluster-oriented life cycle: Classes belonging to a cluster are refined, the refinement of different clusters overlaps, and the system grows step by step to the final application. The presented examples give a glimpse of how object-oriented software systems are developed based on the cluster model of the software life cycle.

References

[Abbo83] Abbott R.: Program Design by Informal English Descriptions; in Communications of the ACM, Vol. 26, No. 11, 1983.

[Booc86] Booch G.: Object-Oriented Development; IEEE Transactions on Software Engineering, 12, Feb., 1986.

[Booc87] Booch G.: Software Engineering with Ada, Benjamin/Cummings, California, 1987.

[Gamm89] Gamma E., Weinand A., Marty R.: Integration of a Programming Environment into ET++: A Case Study; Proceedings of the 1989 ECOOP, July 1989.

[Gamm90] Gamma E., Weinand A.: ET++ Introduction and Installation; UBILAB Union Bank of Switzerland, 1990.

[Hend90] Henderson-Sellers B., Edwards J.M.: The Object-Oriented Systems Life Cycle; in Communications of the ACM, Vol. 33, No. 9, Sept. 1990.

[Meye88] Meyer B.: Object-Oriented Software Construction; Prentice Hall, 1988.

[Meye89] Meyer B.: The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design; in Proceedings of Tools ‘89 (Paris, France), 1989.

[NeXT90] NeXT, Inc.: 1.0 Technical Documentation: Concepts; NeXT, Inc., Redwood City, CA, 1990.

[Pomb91] 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.

[Pree90] Pree W.: DICE—An Object-Oriented Tool for Rapid Prototyping; in Proceedings of

Tools Pacific ‘90 (Sydney, Australia, 1990), pp. 357-365.

[Schm86] Schmucker K.: Object-Oriented Programming for the Macintosh; Hayden, Hasbrouck Heights, New Jersey, 1986.

[Wein88] 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.

[Wein89] 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.

[Wils90] Wilson D.A., Rosenstein L.S., Shafer D.: Programming with MacApp; Addison-Wesley, 1990.

Trademarks: