A Component-Based Modeling Approach

Alois Stritzinger
Institut f. Wirtschaftsinformatik
C. Doppler Laboratory f. Software Engineering
J. Kepler University of Linz, Austria

Abstract
This paper starts with a consolidated description of widely applied reuse techniques and discusses their drawbacks. As one of the most promising techniques, component-based software development is presented in more detail. Our modeling approach defines three representation techniques for capturing and documenting the essence of component systems. Each one can be represented in two forms, graphically or textually. To further ease modeling complex software systems, we present a general architecture model which structures a system in layers with well-defined tasks. Last, we discuss our view on the task of modeling in contrast to designing and present a list of modeling process activities which can be performed in any order.

Keywords: analysis, design, modeling, components, software architecture

1 Background

Reuse of prefabricated hardware building blocks has proven to be an extremely useful production technique in numerous industrial domains. Electric switches, controllers, vents, motors, gears, pumps, etc. are usually not designed for a single application, but for many potential uses. Therefore, the desire to assemble or compose software products from prefabricated, reusable parts is also natural and rather old. The invention of object-oriented programming was an important step in the direction of greater reusability. The simplest and most often practiced reuse in object-oriented programming is reuse of concrete classes. This kind of reuse presupposes concrete implementations of well-defined, elaborated concepts. Few or no assumptions are made about the availability of other classes. Typical examples are container classes. Inheritance even allows extension or modification of reused classes which do not fit exactly. This kind of reuse is well established, for instance, by Smalltalk class libraries [Goldberg93]. In Smalltalk systems even basic data types and control structures are provided by library classes. Although this kind of reuse was and is quite successful, it has serious limits. Above all, prefabricated code can only be used by new code and not vice versa. But the latter would be very useful in order to make larger structural concepts or frameworks reusable.

To circumvent the problem, abstract classes which contain, at least partially, abstract methods were introduced. Abstract classes or methods serve to define an interface in a clear syntactical and semantical form. To be able to create instances which conform to these interfaces, abstract classes have to be subclassed with concrete classes. Thus it is possible to design complex structures consisting of objects of several collaborating classes, where some of them are abstract. Whenever such a structure is reused, all abstract classes have to be extended by application-specific concrete classes.

Reusable groups of well-coordinated abstract classes and concrete classes which define the standard behavior and core structure of all potential applications are called *application frameworks* [Weinand88]. The purpose of frameworks is to factor out as much common code as possible into reusable classes. Application frameworks often suffer from inherent complexity which has to be mastered by the developers [Stritzinger91]. On the other hand, even very sophisticated and elaborate frameworks became intolerably inflexible when the requirements on applications grow over time. No framework can serve an unlimited period of time. This problem led to another reuse technique: components.

2 Reusable Software Components

In recent years component-based software development has established itself as an extremely successful reuse technology, even in commercial terms. Component-based software development is based on the concept of composing or assembling systems out of independent but pluggable parts. Since it is rather unrealistic to make all components with similar interfaces (like Lego bricks), components have a configurable adapter by which they can be coupled. This capability for configuration or adaptation is the central characteristic of components. Beyond that, it is hardly possible to specify more common features or attributes of components, since different products employ different approaches. Nevertheless, we investigate component technology further on a common basis, although a certain feature may not be available in a concrete component system.

At run time, components have similarities with objects or object complexes. They also store a state in attributes which may (but need not) be encapsulated and provide operations for testing and manipulating the state. Furthermore, components define and trigger events of internal processes for an interested environment. Which events a component defines depends strongly on its functionality. Typically, events correspond to substantial state changes. Putting two components together means establishing a relation between an event on the one side and an action on the other side. At run time, triggering an event causes the activation of all associated operations.

Components can be described by means of one or more classes or by a conventional module. Thus object-oriented programming is no prerequisite for components. Though components and objects are orthogonal concepts, the advantages of object-oriented programming can also be useful within components. Usually components have a static nature; they begin to exist at program start and live until program end. Assembling and configuring of components is often done in composition environments by means of scripting and direct linking.

In addition to the already mentioned properties, components are often used for visualization of program data or control elements; i.e., they are parts of the user interface. Thus a composition environment provides functionality for arranging and directly manipulating user interface building blocks. Furthermore, all commercially successful systems provide a rich and powerful component library ([Delphi95, VisualAge95, VisualBasic95, VisualSmalltalk95]). But component-based software development requires neither a composition environment nor a component library: a regular programming environment and appropriate conventions would suffice. But a comfortable composition environment together with a powerful component library allows working at a very high abstraction level. Thus even less-trained developers can produce acceptable results. The price, on the other hand, is a high demand on memory capacity and processor speed. But exchanging intellectual power for raw machine power seems to be the way to success. The more serious problem with componentware for the moment is that systems consisting of more than a few dozens parts quickly become incomprehensible. To master this problem, either very strict nesting and encapsulation techniques have to be imposed or more coarse-grained components have to be programmed.

Although component technology has its drawbacks, if used as the one and only implementation technique, it seems to be very attractive for designing and modeling complex interactive systems.

3 Component-Based Modeling

Before we present our modeling approach, let us say a few words about our view on the task of modeling. In contrast to design, which usually denotes a well-defined phase in the software life cycle between requirements engineering and implementation, modeling covers all activities ranging from analytical studies of an application domain to testing of system parts. Our experience has shown that these activities require a lot of knowledge and creativity. Thus it is not purposeful to structure modeling activities by a strict method. The process of modeling is not predictable and cannot be controlled in a generic way.

Important properties of design or modeling techniques are the various visual representation forms (diagrams and descriptions) of several system aspects. In fact, aside from diagramming techniques, design methods often provide little guidance toward reaching well structured systems. We are not surprised by this focusing on design description and representation, because our view on modeling agrees that it is simply impossible to find adequate solutions just by following a strict method. Consequently, we also focus first on the description and representation of the system architecture.

3.1 Style of Representation

The main purpose of the representation of the architecture is the documentation of (intermediate) results of the modeling process to communicate the concepts among developers, managers, programmers and anyone else. For obvious reasons, the description technique should be simple and easily comprehensible. On the one hand, it should be relatively (programming) language-independent, but, on the other hand, the implementation should be rather straightforward. Due to the importance of reusability, prefabricated components should be easily embedable. Additionally, it would be nice to somehow support the mental process of creating a conceptual model.

Most representation techniques use primarily diagrams and charts to capture the essential model concepts. The reason is the more direct transformation of mental models into graphic images and their better intuitive comprehensibility. In order to reduce the variety of boxes, arrows and symbols, textual annotations are used. But graphics have their drawbacks: They tend to require enormous drawing space which is usually just not available or useful. Furthermore, transforming a graphic diagram into an executable program is often a nontrivial, error-prone task. In this respect, textual representations are superior. Unfortunately, they are less intuitive and prove inferior as thinking tools or vehicles. So our solution encompasses both, a graphical and a textual representation. Depending on personal preferences, project guidelines or other influences, either notation or both can be used. In each case we capture the very same content, but leave some freedom for the concrete representation.

3.1.1 Composition Representation

The first kind of diagram or description depicts the decomposition of (part of) the system into components and how these components are coupled. Each building block has a unique name and a number of essential attributes. The attributes serve to describe the main characteristics of their components. Many attributes hold references to other components. We distinguish between direct and indirect references. Direct coupling expresses that one component uses another directly for some reason and that the former knows the actions of the latter. Hence all actions defined by the used part can be invoked. In some cases it is impossible or undesirable to know exactly which or how many components are referenced by an attribute. For such situations, we offer the possibility of indirect coupling. Indirect coupling means that the first component can only use a (typically small) set of standardized actions which are forwarded by the mediating component. Mediating components which do not offer additional functionality are represented in a collapsed form by a small circle.

Figure 1 depicts a component structure model consisting of three coupled components. The second attribute of *component2* holds a string value, which can be written directly behind the attribute's name whenever no other component refers to the string.

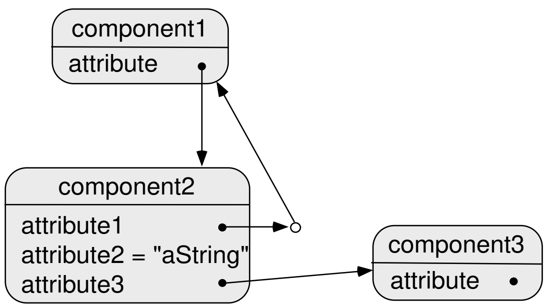

Figure 1: Component structure model

The following textual description represents similar structure in a textual form.

```
components
    component1: {attribute}
    component2: {attribute1, attribute2 = ''aStringValue'', attribute3}
    component3: {attribute}
    comp4, comp5, comp6: {...}

couplings
    component1.attribute -> component2
    component2.attribute1 ->-> component1
    component2.attribute3 -> component3
```

The textual description gives a better chance to annotate attributes and couplings. Components of the same kind, but with different couplings, can be described collectively in a list as shown for *comp4* , *comp5, comp6*. Indirect coupling is indicated by a symbolized double arrow (->->).

The purpose of the component representation is to describe the static structure of the system. To complement this picture, we need a description of the dynamic behavior, which is introduced in the next section.

3.1.2 Collaboration Representation

For understanding a system, the explanation of the dynamic interactions of its components is as important as the description of its static structure. In fact, well-known object-oriented design methods (e.g., [Wirfs-Brock90]) focus primarily on the responsibilities/behavior of objects instead of their relationships. In our opinion, preferring responsibilities over relationships is neither right nor wrong because it is a kind of a chicken-and-egg problem. In other words, it is not possible to give one aspect absolute precedence over the other; both the composition and the collaboration model have to be developed in parallel.

For equal rights reasons, we now start with a textual representation. It first describes the components by relevant actions and events which can and should be used by other components. Each action is introduced by a leading ”>” symbol; the events are marked by an appended ”>” symbol. Next, we describe all collaborations with causality relationships. A causality relationship associates an event of an originator component with an action of a receiver component. The meaning is obvious: Whenever the originator component triggers the event (because of a state change during an action execution), the associated action is invoked at the specified receiver component. Optionally, a collaboration relationship may specify a list of arguments which serve as parameters for the action. The collaboration relations of a component have to be deposited in the adapter of the component. This adapter can be implemented by a generic event dictionary where the event names serve as keys and the action commands can be registered appropriately. When an event is triggered in a specific component at run time, a dispatching mechanism looks up in the event dictionary and activates one by one the action commands that are registered for the event.

```
components
    component1: {
        > action;
        event >}

    component2: {
        >action1;
        >action2;
        >action3;
        >action4;
        event>}

    component3: {
        event>}
```

```
collaborations
    WHEN component1 TRIGGERS event SEND action1Command TO component2
        [WITH arguments].
    WHEN components2 TRIGGERS event SEND actionCommand TO component1.
    WHEN component3 TRIGGERS event SEND action4Command TO component2.
```

As in the composition model, equal components can be described collectively by arranging them in a list. Figure 2 depicts the dynamic collaboration model in a graphic form.

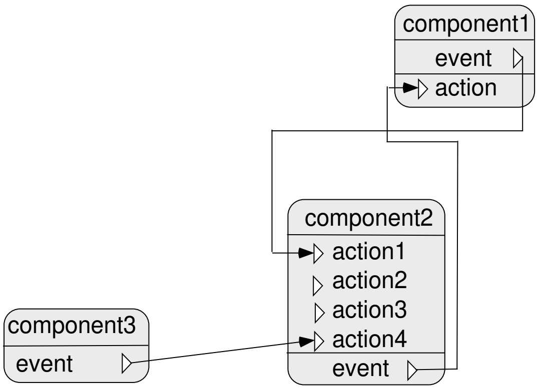

Figure 2: Component collaboration model

It is important to understand that (dynamic) collaboration between two components does not require static coupling of the participants. Static coupling implies that one component statically (in source code) knows the other. But our technique is based upon the requirement to let independent parts work together. Thus a collaboration relationship specifies both the originator of the event and the receiver of action command. Coupled components can communicate either by dynamic invocation (*collaboration*) or by static action calls (*cooperation*). At the current development stage of our modeling technique, we do not explicitly describe component cooperation because it is of secondary importance.

Thus far our models have concentrated on concrete instances of components: we described real world entities (things, persons, artifacts, phenomena, etc.). But sophisticated programming techniques usually deal with abstractions of these entities (concepts, types, classes, etc.). Therefore, let us shift our attention from conceptual modeling to programming and introduce a description form which may serve as a good starting point for implementation activities.

3.1.3 Class Interface Description

Although object-oriented techniques are not prerequisites for the implementation of components, we prefer this style of programming. Components can be implemented relatively straightforwardly by classes and the additional concept of event propagation. We use a form of class diagrams which describe variables, methods and events of each class. A component type corresponds closely to a class or a set of tightly cooperating classes. In the latter case, one class serves to define the component interface, while the rest of the group provides the implementation.

In the graphical representation shown in Figure 3, each feature of a class is listed textually in a short form (i.e., name). The relationship of a class to the superclass(es) is expressed graphically by a line containing a triangle or textually by associated superclass names. Relationships with other classes by (uses, creates, holds, etc.) attributes are indicated by directed edges or by associated class type names behind the attributes name. But nevertheless, defining class interfaces out of our component models is more than a trivial task. In general, creating a reusable component definition for a number of potential instances for numerous applications requires a lot of experience.

As with the component models, we can use a textual representation for class description as well. In fact, for class interface descriptions the textual form offers a number of advantages: It can serve directly as a program skeleton or class definition. It allows a more detailed description of method headers with arguments and return values, including type specifications and semantic descriptions.

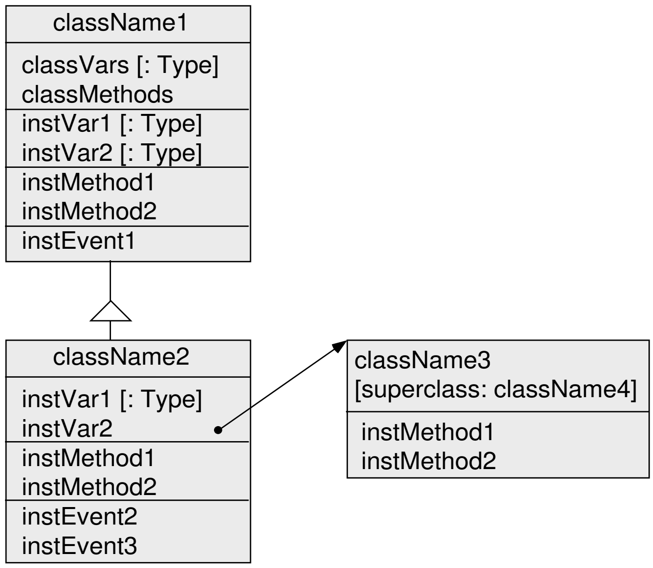

Figure 3: Class diagram

One might argue that instance variables should be encapsulated and accessed or manipulated only by public methods and thus they should not be present in class interface definitions. Of course, we do advocate for encapsulation, but hiding instance variables completely makes the comprehension of the task and purpose of a class much harder. Especially the semantic description of the methods is much easier and more compact, if one can refer to instance variables. But this does not mean that a user of a class can rely on the presence or types of instance variables or even directly access them.

class: ***class name***
superclass: *superclass name*

class variables:
...

class methods:
...

instance variables:
instVar1: type1
instVar2, instVar3: type2,

instance methods:
methodHeader1
semantic description

methodHeader2
...

events:
event1
semantic description

event2
semantic description

In order to allow easy adaptation of class or module descriptions to the implementation language, we do not define a particular method header syntax. But we encourage everybody to use a consistent notation. Our preferred method header syntax is a slight extension of the proposed standard Smalltalk syntax [IBM92], which defines three forms corresponding to three kinds of messages. Since Smalltalk does not define types for arguments or a return value, we extend the Smalltalk headers by an optional definition of a return value, which is introduced by the special symbol „^:“, and the concept of *nameWithType*. A *nameWithType* denotes either one word which includes type information (e.g., *contentRectangle*) or an argument name followed by a type (e.g., *contRect <Rectangle>*).

```
simpleMethodName  [ ^: nameWithType*]
operatorSymbol nameWithType  [ ^: nameWithType]
methodKeyword: nameWithType  { methodKeyword: nameWithType } [ ^: nameWithType]
```

* The specification of the return type is necessary only when an object different from the receiver is returned.

The semantic description of each method should specify the functionality of the operation from an outside view. Special attention should be given to describe relationships between methods and events which are triggered by the operation. Vice versa, the semantic description of events should specify under which circumstances and by which operations the event is triggered.

3.2 Layered Application Architecture

Composing applications out of components is not as simple as building toy models out of Lego bricks; therefore we suggest a layered architecture for highly interactive applications, with well defined functionality at each layer. For typical information system applications, our architecture model consists of four layers, as shown in Figure 4.

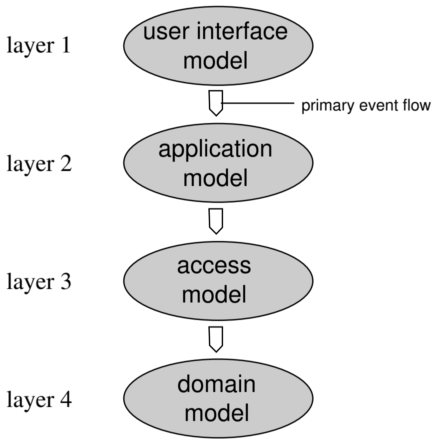

Figure 4.: Layer Architecture

Layer 1: User Interface Model

The user interface layer consists entirely of the user interface building blocks, i.e., components which have a run-time representation on the screen. Obviously, the task of these components is visualizing of application data and control devices as well as capturing user input and triggering events for invoking the associated commands. Very often, user interface components can be changed either by direct manipulation by the user (e.g., selecting a list item) or by changing program states (e.g., updating a displayed list). Since interaction components cannot assume any functionality of the functional components, the collaboration can only be configured dynamically in one direction. Communicating changed data or program states back to the user interface can be done either by standardized (update) actions or by dynamically established collaboration. The user interface building blocks are arranged in a contained hierarchy where the window components form the roots.

Layer 2: Application Model

The application model consists of components which form the functional core of the application. It has to build up and open the user interface at program start. During program execution the application components have to maintain the user interface according to the program state. Actuating a control element at the user interface usually results in invoking a method of the application model. Since the application model knows the user interface model as well as the lower-level model, it establishes the dynamic collaboration relationships among all components. It is purposeful to decompose a complex application model into several independent components which either work together via dynamic collaboration or even better are just synchronized by the underlying layer.

Layer 3: Access Model

The access layer serves as mediator between the user interface or the application model on the one side and the domain model on the other side. It has two important tasks: first, synchronizing accesses to and updates of the data (i.e., broadcasting changed events to all dependent view or application components) and second, supplying view components with their data when required. Ideally, the access layer should completely hide and encapsulate the underlying data, but provide controlled access. Therefore, only values and copies of data objects should be passed through to the higher layers. If, for efficiency reasons, copying huge structures is not feasible, the access layer has to be informed about each update of domain objects. Access model components which implement a general change propagation mechanism and a configurable accessor mechanism can be prefabricated and reused. Application-specific access model components can be viewed as the respective handling instances (authorities) for the domain components.

Layer 4: Domain Model

The primary purpose of the domain model is to store the application data during program execution. We prefer to use a structural object model, which is an object-oriented data model of the domain. The domain model provides all necessary operations for querying and manipulating the data in a basic, application-independent form. Furthermore, the domain model is responsible for checking all integrity conditions and ensuring other semantic characteristics of the data model. The domain model should not make any assumption about the above layers. If separated domain models might influence each other, all manipulations should be done via the access layer. In such cases, only dynamic collaboration can be used.

Variants

For simple applications we can omit the access layer and integrate its functionality into the domain layer. For applications which implement numerous undoable commands, we can insert a special command layer between the application and the access layer. Database applications should, at least in theory, just implement the domain layer with a persistent data model.

3.3 Modeling Process Activities

According to Wirfs-Brook et al. [Wirfs-Brock90], software modeling has much in common with developing a mental model of some problem domain. We believe that acquiring a mental (conceptual) model is an important part of the entire software modeling process. As already stated earlier, we do not believe that the modeling process can be structured or controlled in a general form. Nevertheless, it is possible and useful to identify and describe common activities.

Domain Investigation

Having a sound knowledge on the problem domain is an absolute requirement for being able to develop systems that solve real problems. For acquiring problem domain knowledge, a number of techniques have been discovered and proven purposeful, including interviews, questionnaires and accumulation of artifacts (forms, reports). Additionally, use cases and scenarios can be used for describing business process activities. These utilities are documented extensively in the requirements engineering literature.

Conceptual Modeling

The results of investigation and recognition of domain concepts and phenomena have to be described in application domain terms. The used terminology should be defined in a kind of reference glossary. According to Kristensen and Østerbye [Kristensen94] "conceptual modeling denotes an understanding and construction process with the purpose of forming a model of some part of the world expressed by concepts and phenomena, i.e. a concept formation process. The purpose of the model may either be the understanding of some existing system or the design of some new model system.“ The resulting model describes or represents concepts and phenomena of the problem domain and their relationships in an informal way, so that the concepts and phenomena can be communicated among all involved persons.

User Interface Modeling (Prototyping)

A crucial step in modeling is the development of a user view of the system. This can be done best by involving the experts (i.e., end users) as important participants in the process. What we denote as user interface modeling has already been studied and described by a number of scientists (e.g., [Bischofberger92, Budde84]).

Data Modeling

Especially (but not exclusively) in information system development, data structures form a decisive factor in the development process. The data engineering community has created a rich repertoire for many aspects of data modeling.

Behavior Modeling

What we term behavior modeling is a consolidation and simplification of dynamic and state modeling as described by Rumbaugh et al. [Rumbaugh91]. Here we view the system model as a state automaton where each system state or mode is expressed in domain subject terms. The behavior model is formed by a list of transactions. Each transaction is described by a user command (input), a current state, the resulting visual response (output) and the resulting new or old state. Additionally, each transaction may specify its implications on the data model. It is important to understand that our behavior model views the system as a black box with a known internal state. The behavior model should closely correspond to the conceptual model of the end user of the system.

Component Modeling

Component modeling encompasses the activities of finding and identifying all components, defining attributes, defining actions, defining events, and defining couplings and collaborations of the proposed composition and collaboration model.

Components are usually already indicated by nouns of the glossary. They often correspond to nouns, but in contrast to object modeling, closely related things which cannot exist independently are grouped together within one component. Adjectives which can be associated with component nouns are often candidates for attributes and verbs for actions. Since no component is an island, some attributes may refer to other components, and/or events can be identified which may be of interest to the environment.

Interface Definition Modeling

We assume that a number of components can be reused from a library of prefabricated parts. Only components which are not already available have to be implemented by means of a program. For this purpose we suggest deriving class interface definitions out of the component model. Developing the class definitions requires some experience to find good compromises between optimal suitability for the system at hand and enough flexibility for future changes or reuse engagements for the component.

Implementation and Test

Perhaps surprisingly, these activities belong to the modeling activities, too. But creating an executable system in parallel with other modeling activities has the advantage that many hindsights gained from programming can improve the overall process and the product. Additionally, feasibility or efficiency problems of any kind arise much earlier, and developers are motivated by an already working subsystem.

4 Conclusion

Component-based software development has emerged as one of the most promising reuse technologies in recent years. So it is likely that component-based implementation techniques will play a major role, especially in commercial applications software development. Working with components requires special consideration in modeling or designing in order to create flexible, reusable, but still practical parts. We have presented an approach for modeling component-based systems which borrows some ideas from object-oriented techniques and adds component-specific representations. Furthermore, we have suggested a general architecture as a guideline and presented a flexible process model to ease the overall modeling task. Thus far, we have applied our approach primarily in smaller-scale projects using Smalltalk systems for hand-made component implementation and assembly. But our experiences are promising. We are currently starting to evaluate our technique on a larger scale using a component composition environment.

References

[Bischofberger92] Bischofberger W, Pomberger G; Prototyping-Oriented Software Development; Springer 1992

[Budde84] Budde R, Kuhlenkamp K, Mathiassen L, Züllighoven H; Approaches to Prototyping; Springer; 1984

[Delphi95] Borland; Product Documentation

[Goldberg83] Goldberg A, Robson D; Smalltalk-80: The Language and its Implementation; Addison-Wesley; 1983

[IBM92] IBM; Smalltalk Portability: A Common Base; Redbook No. GG24-3903,

[Kristensen94] Kristensen B B, Østerbye K; Conceptual Modeling and Programing Languages; ACM Sigplan Notices, Vol 29, No 9, Sept 1994

[Rumbaugh91] Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W; Object-Oriented Modeling and Design; Prentice-Hall; 1991

[Stritzinger91] Stritzinger A; Reusable Software Components and Application Frameworks; VWGÖ, Wien 1991

[VisualAge95] IBM; Visual Age for Smalltalk 3.0 Product Documenation

[VisualBasic95] Microsoft; Visual Basic Product Documenation

[VisualSmalltalk95] ParcPlace-Digitalk; Visual Smalltalk 3.0 Product Documentation

[Weinand88] Weinand A, Gamma E, Marty R; ET++—An Object-Oriented Application Framework in C++; OOPSLA 88 Proceedings, Special Issue of SIGPLAN Notices, Vol 23, No 11; 1988

[Wirfs-Brock90] Wirfs-Brock R, Wilkerson B, Wiener L; Designing Object-Oriented Software; Prentice-Hall; 1990