(ed.), Manning Publications and Prenctice Hall, 1995, 199-227

Vista – a Visual Software Technique Approach

Stefan Schiffer and Joachim Hans Fröhlich

C. Doppler Laboratorium für Methoden und Werkzeuge des Software Engineering
Johannes Kepler Universität Linz, A-4040 Linz, Austria
Tel.: ++43 (70) 2468-9442, Fax: ++43 (70) 2468-9430
Email: {schiffer,froehlich}@swe.uni-linz.ac.at

March 14, 1994

Abstract

This paper describes Vista, a visual multiparadigm language integrated within a comfortable development environment. Vista supports fundamental software engineering principles during programming such as adequate notation, modularization, and weak coupling. Vista augments the object-oriented programming paradigm by signal-flow and data-flow based programming. It provides capabilities for the construction of event-driven and data-transformation systems. In constructing an application with Vista, visual as well as textual means can be used. Characteristic features of Vista are especially aimed at the combination of high-level and easy-to-use building blocks that are hierarchically organized. We introduce the notion of processors and networks and discuss how a remarkable degree of compositionality and reuse can be achieved by their employment.

Keywords and Phrases*: Vista, Software Engineering, VPL-I: Environments and Tools for VPLs, VPL-II.A.8 Multiparadigm languages, VPL-II.A.9: Object oriented languages, VPL-III.A: Abstraction*

1 Introduction

Within the past few years visual programming (VP) has been increasingly attracting attention, mainly because of its promise to make programming easier, thus allowing not only computer experts but also novices to solve problems with the computer. This revives the expectations of over 30 years ago, when programming languages such as Cobol where supposed to make the programmer obsolete by providing a new natural way for every English-speaking professional to communicate with data processing machines. In fact, programmers did not lose their jobs because of Cobol, and VP will also not leave them unemployed.

However, the market has already discovered the fascination of VP, and various new programming environments have been declared to be “visual”. A closer look at some of those products (e.g., Visual C++, VisualWorks) shows that they typically consist of browsers for manipulating text, combined with a GUI editor and a rudimentary application skeleton generator. Although it may be right to call such environments visual, that is not what we mean by VP. In our understanding, VP has to offer substantially higher expressiveness than conventional textual programming by means of software visualization and a visual programming language. Systems such as LabView [VW86], PARTS [Digitalk 1992], Prograph [Gunakara 1992] and Serius [Serius 1992] fall into this category. (For definitions and taxonomies of VP see [Ambler and Jones 1989;

Chang 1990; Glinert 1992; Myers 1986; Shu 1988; Selker and Koved 1988; Price et al. 1993].)

This paper describes Vista, a visual multiparadigm language integrated within a comfortable development environment. In our opinion, Vista provides substantial expressiveness at the visual layer as well as concepts necessary to build real applications. Vista supports fundamental software engineering principles during programming such as adequate notation, modularization, and weak coupling. Vista augments the object-oriented programming paradigm by signal-flow and data-flow based programming. It provides capabilities for the construction of event-driven and data-transformation systems. Characteristic features of Vista are especially aimed at the combination of high-level and easy-to-use building blocks that are hierarchically organized. In constructing an application with Vista, visual as well as textual means can be used.

1.1 Visual Programming and Software Engineering

VP superbly addresses two key issues of software engineering: mastering complexity and increasing productivity. Complexity can be reduced by offering concepts which raise the abstraction to the visual level. Such high level abstractions reveal semantic relationships among program entities that otherwise would remain hidden. Productivity can be improved by what Shneiderman [Shneiderman 1983] calls the central features that produce enthusiasm in using interactive systems: “visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest”.

With regard to abstraction, building blocks like classes and modules have proven their worth. Well specified, tested components are a prerequisite when manufacturing high-quality applications. But in spite of this indisputable fact, it is a misleading notion that software systems might be built almost solely by assembling building blocks. Captivating, simplistic pictures like Figure 1 are intended to convince the observer how easy programming could be if only the right visualization and assembly tools were available for powerful building blocks. (It is sometimes argued that graphical user interface editors, which are supposed to be usable by nonprogrammers, follow a similar approach. What usually remains concealed is that problems arise as soon as a nonprogrammer has to bring the user interface to life.)

Programming based solely on assembling building blocks is relevant only if static structures with fixed operations are the foundation of the system under construction. This is rarely true and should be called *configuration*, not programming. Software engineers design reactive systems where components are created, changed and destroyed at run time. The specification and implementation of the behavior of such dynamic systems is a difficult task that cannot be handled by merely plugging components together, but requires the definition of the right control structures. Even application

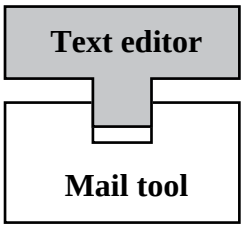

Figure 1. A naive approach: Plugging a text editor into a mail tool.

frameworks [Pirklbauer et al. 1992; Weinand et al. 1989] or design patterns [Gamma 1992] do not make control structures obsolete.

Object-oriented programming (OOP) as the state of the art in software engineering provides good concepts to write high-quality programs. Nevertheless, there is a lack of concepts and tools that help the programmer to cope with the complexity of OOP. A few exceptions exist, e.g., the object-oriented VP environment *Prograph* [Gunakara 1992, Cox and Pietrzykowski 1988], which represents an exemplary step toward promoting OOP by visual means. Prograph is a complete software development tool set based on a visual data-flow language. Though the data-flow paradigm is well suited for transformational systems and a very natural base for VP (by means of lines and boxes), it is not appropriate when designing reactive systems. The modeling process of reactive systems has to cover complex sequences of events, actions, time constraints, etc. which lead to systems with interactive behavior. By contrast, transformational systems convert data by applying functions without the notion of state or time. Although there exist extensions to the pure data-flow model for handling control flow, synchronization or state (e.g., Prograph includes case windows, synchro links, and object attributes), modeling systems solely based on the data-flow paradigm is not always suitable. Such inappropriateness of the modeling language induces what we call a semantic break.

Our efforts focus on how to enrich OOP by means of a visual model without causing a semantic break. In this context we think that it is important to support more than one paradigm. Because we know the potency of OOP, the strength of control-flow programming, as well as the declarative power of data-flow programming, we want to harmoniously integrate them all under a visual layer. Moreover, we are convinced that constructing nontrivial, high quality software requires special knowledge and skills. VP supports, but cannot replace, these human dimensions. Therefore we concentrate our research on how VP techniques can aid software engineers, rather than on the empowerment of lay persons.

2 Vista—a Visual Software Technique Approach

It is our firm conviction that the design of a feasible VP environment should rely on well-founded software engineering concepts and programming paradigms. Unfortunately, too few publications on VP have yet dealt with software engineering principles. The assumption that pictures are natural, so VP is good per se, may be one reason for the lack of research. We are convinced that practical software projects require the application of principles like information hiding and weak coupling for VP as much as for any other programming paradigm; thus strong efforts should be made to incorporate them into VP.

Our approach for a VP environment is to join object-oriented programming based on Objectworks\Smalltalk [ParcPlace 1992a] and programming with signal flow and data flow. We call the model and its programming environment Vista. The name Vista is an acronym for “VIsual Software Technique Approach”, with equal emphasis on “visual” and “software technique”. The model together with the environment

- provides a visual language for defining signal and data networks
- supports the visual design of encapsulated and weakly coupled components
- offers full access to the Smalltalk class library

- supports visual interaction by direct manipulation of components
- avoids visual overload by permitting text input whenever useful

Vista promotes evolutionary prototypical development of object-oriented software systems but does not support the whole software life cycle. It is intended to be used during the design and implementation steps. We tried to rigorously obey software engineering principles for the conception of the model as well as for the implementation of the environment.

In the following we introduce Vista's programming model and explain how this adheres to the basic requirements for the construction of quality software. We do not describe the visual language formally, because in our opinion that is of less value in a truly visual environment such as Vista, where a distinction between the language and the environment is somewhat artificial. Instead, we give a complete example in Section 4, showing how a thermo-alarm system can be build with Vista.

2.1 The Programming Model

Entities called *processors* constitute the central computational components of the Vista programming model. Processors are high-level objects constructed by visual and/or textual means. Textually defined processors (simple processors) are programmed in Smalltalk; they establish the set of Vista primitives. Visually constructed processors (compound processors) are implemented by direct manipulation of visible objects; they make up the programmer-defined library of building blocks.

On the programming surface, visual and tangible representations of processors are exposed and various access is granted by Vista depending on the intended manipulation: if a processor is to be redesigned, the programmer has full access to all aspects of its structure and behavior; if a processor is to be used for construction of another processor, only an external (black box) view of the used processor is given. Interaction between processors is carried out via messages and tokens. As usual in Smalltalk, messages sent to processors (synchronously) activate methods and return values.

An additional communication mechanism is provided by token passing, for which processors have input ports and output ports to receive and send tokens. Processors are visually linked by these ports with *connections*. A collection of interconnected processors is called a *network*. In the following we discuss each of these components in more detail.

2.1.1 Processors

Processors are objects with distinguishing structure and communication facilities. Before explaining the internal structure of a processor, we discuss how processors communicate. This should help to understand what makes them unique compared to ordinary objects.

A processor's operations are usually triggered by tokens. Generally a processor receives tokens at input ports, performs operations bound to these ports, and can emit new tokens via output ports. Emitted tokens are passed over connections to other processors. A token may carry event information (signal token) or data objects (data token). Each transported item is stored in a slot from where it can be retrieved by those components to which the token is passed.

Before a token is handed over to a processor, it is checked by the receiving input port. If the input port awaits event information or data objects, it searches for the items

at the expected slots. If items of the right type are found, no further setup actions are performed; if no suitable data is detected, the port substitutes default values for items with illegal types. The programmer may override the port's default values with an expression. This can be either a simple literal or a sequence of statements.

The fallback scheme for data acquisition (1. token-defined, 2. user-defined, 3. default) greatly contributes to the flexibility of Vista. Due to this modus operandi, arbitrary processors can be connected. An output port is not required to deliver a token satisfying the expectations of the connected input ports as long as either the programmer supplies meaningful data or a suitable default value is available. This feature stresses the special communication mechanism of processors in contrast to common message passing, where the sending object has to know about the services offered by the receiver.

Beyond their special interaction, processors have a distinctive makeup, too. This arise from the hybrid nature of Vista, which integrates a conventual class-based programming system with the interactive manipulation of visible and "living" objects. On the one hand processors are objects in the sense of prototypes, with behavior modifiable on a per object basis. This is true for compound processors, which themselves contain networks (behavior). On the other hand processors share characteristics with objects of class-based, object-oriented languages which behave as defined by the class they belong to. This applies to the method interface of both compound and simple processors. The same ambiguity concerning where behavior is defined applies for a processor's structure, too. Simple processors are objects with instance variables defined by their class. Compound processors consist of inner processors which principally can be added to and removed from each individual processor.

No principle distinction exists between typical objects and processors regarding encapsulation. A processor is divided into an interface part which is accessible from outside and an interior part which can only be accessed by the processor itself. Figure 2 shows the overall architecture of a processor with its interface (ports and dashed boxes) and interior (anything else).

The interface of a processor consists of input and output ports as well as public methods and public processors. Access to a processor is provided only via this interface.[1] Public processors are designated to be replaced by other processors having the same interface. One can image the substitution of public processors like the exchange of a chip on a digital board with a newer, better, or otherwise more suitable version. This feature is extremely important as it allows the construction of frameworks as well as convenient customizing (see also Section 3.2.2, "Substitution").

The interior appearance of a processor depends on whether it is simple or compound. The interior of simple processors consists of instance variables and methods, just as with any class in Smalltalk. This need not be discussed further. Of greater interest are compound processors. Such visually defined processors encapsulate private processors (simple or compound), methods, and networks.

Private processors represent the constituent components of a compound processor and therefore conform to the is-part-of relationship of objects. Installing them inside a processor also manifests this relationship topologically. Private processors may be considered as internal devices used to perform the enclosing processor's task. To hold

[1] It should be noted that the privacy of processors can be violated by low-level programming. Since we did not make any changes to the Smalltalk language, which protects only instance variables from public access, the notion of privacy is not recognized by the compiler. For instance, although hidden by browsers of Vista, private methods are still accessible from Smalltalk code.

atomic attributes such as characters, numbers and color values, private processors have to be used, too. There is no other construct besides processors for this purpose.

Processors are part of inheritance hierarchies. When defining a new kind of compound processor, the programmer has to subclass the general class which is the root of all compound processors. The new subclass provides an empty template for the prototypical instance to be defined in the sequel. Further manipulations (adding new ports, defining networks, etc.) only affect the prototypical processor but not the class. An exception is the definition of methods; they are installed directly in the class. After having finished the prototypical processor, the programmer generates the complete class definition from it. Thereby the processor becomes part of the library of reusable components. The same procedure may applied in order to subclass already existing processors. So specialized subprocessors inherit the properties of their ancestors and represent extensions or adaptations of them as needed.

2.1.2 Methods and Networks

Methods and networks define the behavior of processors. Methods are written in ordinary Smalltalk code by means of text editors, whereas networks are graphically composed in two-dimensional diagrams. As methods in Vista are the same as those of Smalltalk, we do not explain them further but rather focus on the notion of networks.

Networks are ensembles of linked processors or other networks. Vista provides only networks for joining components. Like processors, networks have ports to receive and send tokens. The ports of a network are collected by *terminals* (input and output bars)

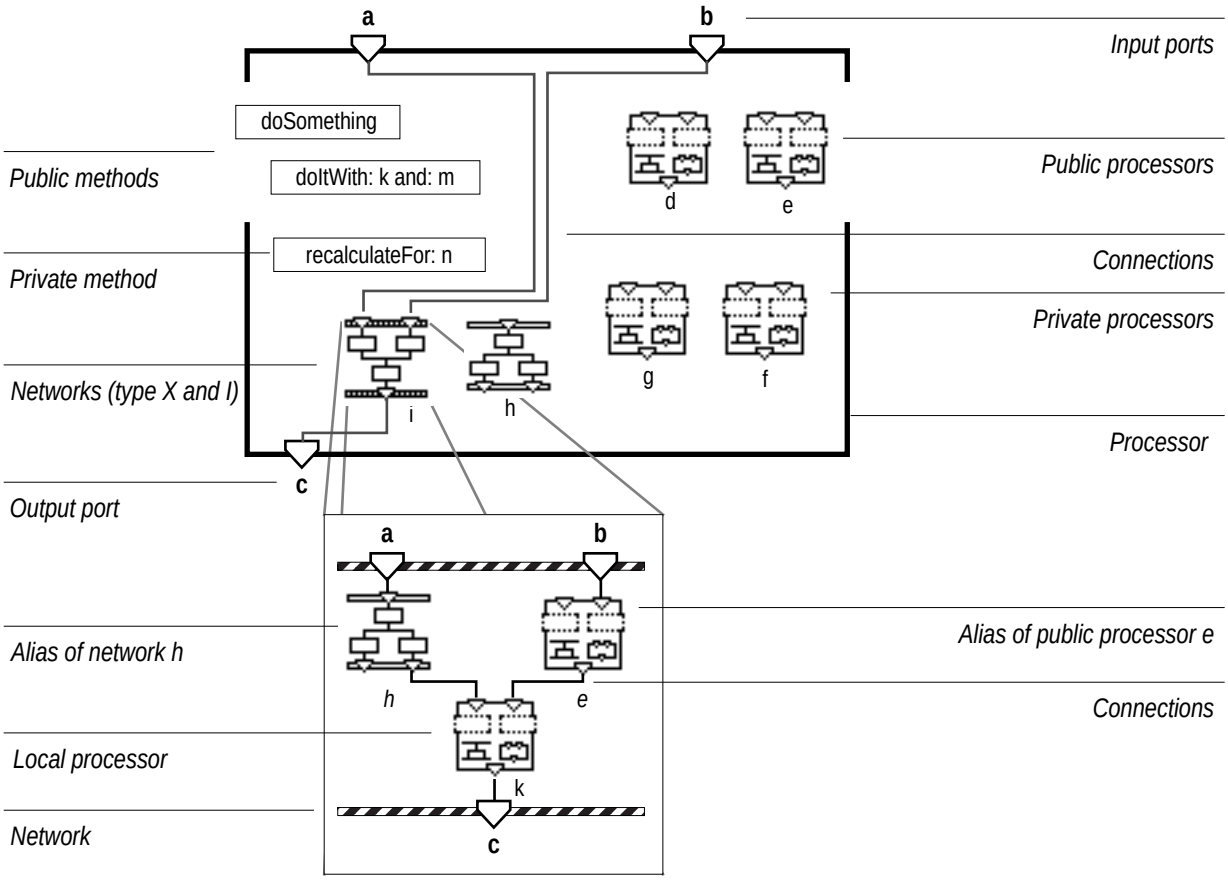

Figure 2. The principle structure of a processor

which constitute the interface of the network. The interior consists of interconnected processors and other networks, which are either locally defined objects or references to objects defined elsewhere (aliases). Local processors and networks are internal to the enclosing network, just like a method's local variables. All components of networks have unique names which are used when sending messages to these components.

The presence of processors inside a network constitutes an important difference between networks and methods. Methods represent *code*. When a method is called, its parameters and local variables are pushed on to the execution stack and removed from there when control returns to the caller. Networks represent interconnected objects, not code. When a token arrives at an input port, these objects are already alive. The sequence by which objects gain control is defined by the way the token moves through the network. When the token eventually leaves the network at an output port, the network's objects remain alive, ready to process the next token.

We distinguish two categories of networks which provide connection points for the enclosing processor's ports and a means of network sharing, respectively:

- *X-networks* link the interior of a processor with its exterior. They either interconnect the processor's input ports and output ports or join internally known entities (sources of control) with the processor's output ports. Terminals of X-networks are dashed.
- *I-networks* are internal to the enclosing processor or network. They can be used within other networks similar to subroutines. Terminals of I-networks are solid.

The distinction between these two sorts of networks is especially important regarding execution. Each processor may contain the source of a control flow. Via X-networks control stemming for an internal source can be passed on to connected processors via a token. Thus an output port of an X-network may trigger without the preceding activation of an input port. For the purpose of network sharing it is not desirable to have sources of control flow inside a dynamically invoked network. The input/output causality would be violated which says that every outgoing token must first fed into a network. For this reason we introduced I-networks which ensure exactly this behavior.

To keep the model simple, Vista currently supports only synchronous forwarding of tokens. If an output port is connected to more than one input port, tokens are transmitted with a depth-first strategy; i.e., a token is passed down along the chain of linked processors rooted at the first connection before the second connection is used. Usually the ordering of connections is defined by the sequence in which the programmer draws them. Explicitly assigned priorities may alter the default order.

2.1.3 Aliases

In Vista, a programming entity is usually defined at the location where it is seen. A programmer looking at an object sees the original. However, sometimes we need to use the same object several times in distinct places. Evident examples include processors shared among different networks or networks used like subroutines within other networks. For these cases Vista provides special components called aliases.

Aliases are surrogates of processors or networks. A token passed to an alias is forwarded to the original, and tokens emitted by the original are sent to its aliases. Aliases

can only reference objects of the same or an outer scope. This rule avoids obscure cross references and enforces proper structuring. Aliases are tightly bound to their originals. Manipulations on the interface of the original object are immediately reflected in its aliases. Deleting the original object removes all its aliases, too.

2.2 Construction of Reactive and Transformational Systems

In the previous section we explained the basic entities of Vista's programming model at a rather abstract level. For a more detailed discussion we now introduce three kinds of processors, namely *signal*, *data* and *coupler processors*. All these processors handle *signal* or *data tokens*, which pass control or data from one processor to the next.

By making a distinction between signal and data processing, Vista provides thought models for the construction of reactive (event-driven) as well as for transformational (data-converting) systems, depending on what is more appropriate for the system under development. Coupler processors combine these two kinds of systems explicitly. This is close to reality, where reactive and transformational systems can often be identified as subsystems of larger systems. They do not exist independently but are coupled at particular points. A radio, e.g., includes the transformational subsystems receiver and amplifier, which make sound out of radio waves; the controls that switch the channels may be regarded as a reactive subsystem since they react to the station selected by the listener. A coupling point might be the electronic circuit which turns the channel selection signal from the controls into a new frequency of the receiver's quartz. Shell languages of UNIX systems which integrate control and data-flow constructs in a really elegant way (ignoring the drawbacks of cryptic textual languages) have also inspired the idea behind the multiparadigm programming aspects of Vista.

Reactive systems are preeminent in Vista, while transformational systems are considered to be part of them. We made this decision because we assume that in most cases looking on the problem domain from a reactive point of view leads to a better model. In our opinion this is apparent when applying object-oriented design methods. We prefer to view an object as a mini-system with an internal state defined by its instance variables and messages to this object as signals or events which change the state seems to be quite right.[1]

2.2.1 Reactive Systems

A reactive system is an event-driven ensemble of cooperating components which respond to a stream of internal and external stimuli. Depending on certain conditions, these stimuli trigger actions which in turn produce cascades of new events. Examples include traffic control systems, computer networks, operating systems and interactive graphical user interfaces of many kinds of modern software. Harel et al. [Harel et al. 1990] state, "Common to all of these is the notion of reactive behavior, whereby the system is not adequately described by a simple relationship that specifies outputs as a function of inputs, but, rather, requires relating outputs to inputs through their allowed combinations in time. Typically, such descriptions involve complex sequences of

[1] We recognize the controversy of such statements. Proponents of data-flow languages might argue that exactly the opposite approach yields more natural models. Maybe this is a philosophical question only; however, we adhere to the notion of messages sent to objects instead of objects being piped through operations (in most cases).

events, actions, conditions and information flow, often with explicit timing constraints, that combine to form the system's overall behavior."

A lot of effort has been invested in developing methods and tools for specifying, designing and implementing such systems, among them module-oriented development environments [Harel et al. 1990] as well as object-oriented approaches [Shlaer and Mellor 1992; Jacobson et al. 1992]. On the one hand, a particular method for the specification and design of reactive systems should allow a very natural mapping of the problem domain to the computer model and implementation; on the other hand, formal means should be used to achieve secured statements about the system's conformance with the requirements and the expected behavior over time.

Vista does not compete with such complex and comprehensive methods and certainly is not intended to cope with real-time embedded systems. Formal methods and explicit modeling of concurrency or distribution were beyond the project's present scope (but see Section 5, "Further Work"). Instead, we designed an environment for the quick assembly of limited reactive software systems by means of encapsulated components. Examples of such components embrace boolean switches, number calculators, alarm clocks, document processors, e-mail servers and operating-system components. We call these building blocks *signal processors*, which, as the name suggest, are intended to react to events, called signals in our terminology.

Signal processors

Signal processors send signal tokens whenever a certain task has been completed or their state changes in a way that may be interesting to the environment. We distinguish active and passive signal processors. Active processors emit signals asynchronously at arbitrary times. Passive processors produce signal tokens only when they receive a message or a signal token from another processor. All activities which are triggered by the flow of signal tokens form a process. The source of a process is always an active signal processor or an external entity like the operating system. Multiple signal flows may be active at the same time; thus processes conceptually execute in parallel. This statement is important when considering networks, as it says that more than one signal token might be on its way through a particular network simultaneously.

2.2.2 Transformational Systems

We outline a transformational system as a data-converting ensemble of components that either generates or transforms data [Harel 1987]. The computational process usually does not depend on particular conditions but only on the available data flowing through the system on predefined paths. Thus the relationship of input to output of such a system is a function with no side effects.

As with reactive systems, we do not claim that Vista's model for transformational systems comprises all aspects of data processing systems. Our model supports pure data-flow semantics with no additional control-flow structures, sequential execution constructs, or whatever augmentations else. These features are covered by the reactive model and thus have been left out of the transformational one. Examples where the application of Vista's data-flow model seems to be well suited are non interactive, batch-oriented problems like data filtering, image processing and format conversion.

Data processors

Data processors form the basis of Vista's transformational model. In contrast to signal processors, data processors do not inform their environment about state changes, but transform input data via internal machinery to output data—a style of computation well known from many data-flow languages. A data processor can execute only if all required data has arrived, i.e., if all its ports hold a token. When this condition is true, the processor performs its *sole* data conversion operation. This behavior is fundamentally different from that of signal processors, where ports are mutually independent and trigger different actions when different signals arrive.

Unlike signal processors, data processors are always passive; i.e., they perform operations only if new data comes in. That is due to the execution model which is solely based on step-by-step forwarding of data tokens. Active sources of data have to be simulated by asynchronously triggered signal/data couplers.

2.2.3 Combining Reactive and Transformational Systems

A network may contain an arbitrary mix of signal and data subnetworks. *Coupler processors* are the designated link points for the joining of signal and data networks. Within Vista two types of coupler processors exist: *signal/data couplers* are triggered by signals and produce data, whereas *data/signal couplers* collect data and send out signals.

Figure 3 shows examples of signal, data and coupler processors as well as diagrams of signal and data networks. The left diagram shows a signal network consisting of a timer and a speaker. When a signal arrives at port *testAlarm*, first the speaker is switched on (connection 1) and second the timer is requested to count down for 10 seconds (connection 2). When the timer expires, it emits a signal at port *expired*, which switches off the speaker.

The right diagram depicts a data network which performs the task of a file chooser. If a user changes the file name pattern in order to get a new filtered list of directory entries, the port *fileNameChanged:* is triggered. The given pattern is then fed into the network, piped through the signal/data coupler *fileName* (which extracts the signals token's item at slot 1), and passed on to the left port of the data processor *matchPatternInList*. This processor checks each element of the list at its right port against the pattern and produces a filtered list of all matching entries. The filtered list is given to the data/signal coupler *newFileList*, which makes a signal token out of it. When invoking file chooser for the first time, signals both on ports *fileNameChanged*: and *dirNameChanged:* are required before *matchPatternInList* can start processing. Later on, each signal token produces a new filtered directory list as old data is remembered by ports of data processors.

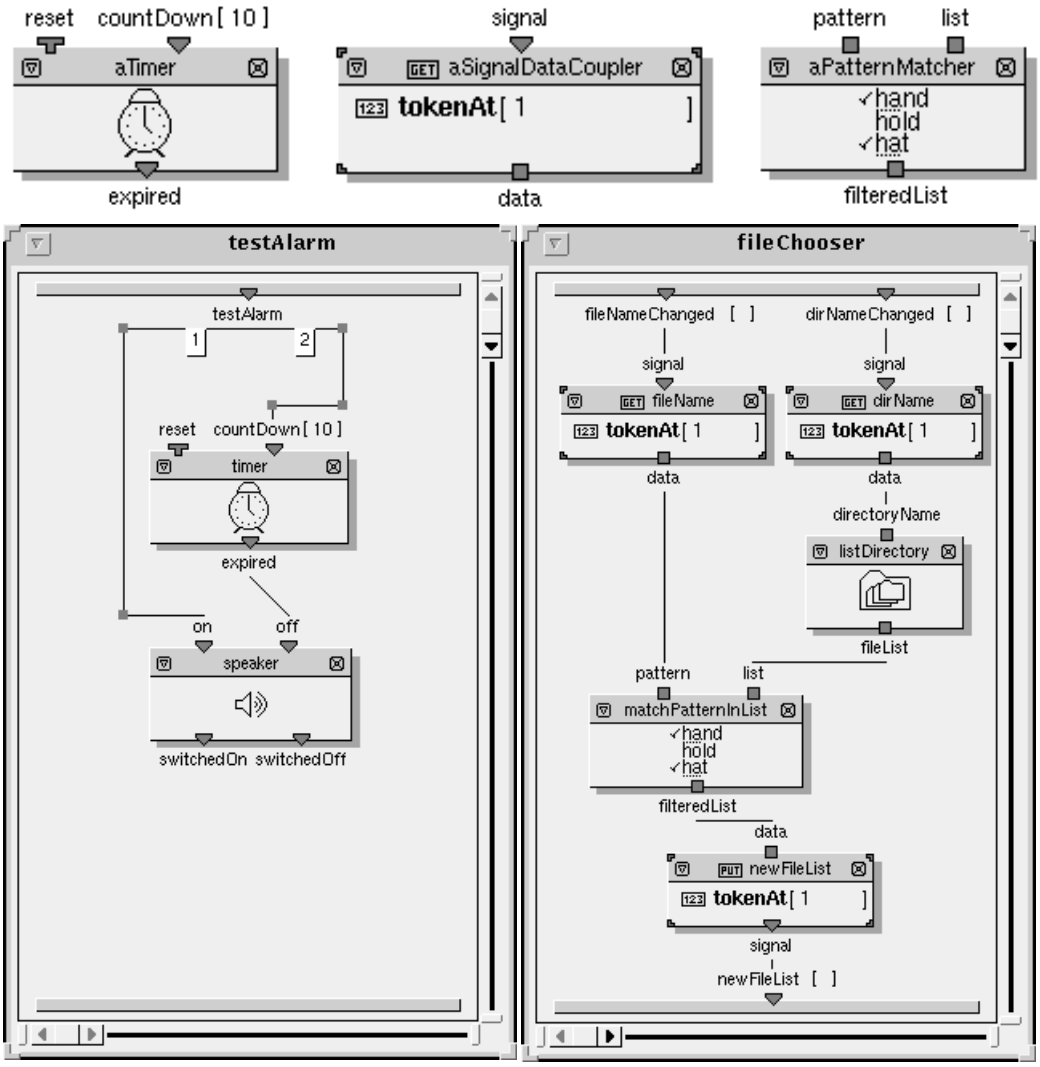

Figure 3. Processors and Networks
Top (l. to r.): Signal processor, signal/data coupler, data processor
Bottom: Signal network, data network

3 Software Engineering Issues

In order to give the programmer the ability to build high-quality software, Vista's concepts are based on a number of fundamental principles of software engineering. Particular determining factors in the design of Vista were the support of:

- *mastery of complexity* by adequate notation and modularization
- *reuse of components* by abstraction and weak coupling
- *safety in programming* by analysis of type information

In the following we give a short overview of the mapping of these principles onto Vista's concepts.

3.1 Adequate Notation

To enhance the comprehension of a complex software system, the right representation of both the system's structure and the execution process is of central importance. Ideally there should be the possibility to look at the system from different points of view; for an outline of the overall design, a coarse-grained view would be quite suitable, whereas detailed insights have to be provided to focus on a single component with all its relationships.

It is not always easy to find the right visualization because different views require different notations. As experience with visual programming has shown, the role of graphics should not be overrated in the search for the right representation. Early approaches in visual programming which totally banned text are considered to have failed because pure pictorial representations lack abstracting power. If the depicted system is not absolutely trivial, it is inevitable that important aspects will remain in the dark; these eventually have to be unveiled by textual means. We are convinced that pictures alone are insufficient to promote understanding of even a moderately complex system.

In Vista the same rank is assigned to graphics and text. Graphics is used for the representation of directly manipulated objects, for overview representations, and for the definition of high-level operations. Text is used for naming objects, for comments, and for writing algorithms.

Objects are depicted either as icons or in an expanded form. Icons show only a typical picture along with the object's name, whereas the expanded representation exposes all accessible properties. Those objects constituting the Vista programming model can be annotated. Annotations have two parts: a brief textual comment and a hypertext document. The hypertext document is a standardized form which contains the specification of the object including examples of how to use it, a dynamically created catalog of current instances with the ability to inspect them, cross-references to other documents, etc.

Lines characterize various kinds of relationships among objects, e.g., hierarchical organization or communication channels. Currently spatial relationships are not used to convey semantics. Thus it is not important whether an object is near another object or if two icons overlap.

Colors and grays display the state of objects (executing = green, blocked = red, disabled = gray). Shadows of different brightness indicate whether an object is the original one (dark shadow) or an alias (light shadow).

Objects are usually kept in lists and registers. From there, objects can be dragged with the mouse to destinations where they are needed. All visible objects are alive and offer a menu containing generic commands such as browse, edit, inspect and annotate, along with special operations depending on the current context.

As a concluding remark on the issue of adequate notation, we once again outline the role of graphics vs. text: although Vista's visual programming language is computationally complete (offering sequential execution, conditional branching, and recursion), it is not intended to be used for writing algorithms. Despite the fact that one could define the faculty function or sorting procedures by interconnecting processors, nobody should do so. Such algorithms can usually be found in the Smalltalk library; if nothing from there seems to be suitable, the programmer should write an ordinary Smalltalk method to obtain what is needed.

The visual capabilities of Vista were designed with the task of system construction in mind. As a rule of thumb, Vista's visual programming capabilities are perfectly suitable if the collaboration of processors has to be specified; e.g., if the sum of an array of integers is to be the result of processor collaboration, the statement *sum := array inject: 0 into: [:s :i | s + i]* will do the job better.

3.2 Abstraction

It seems to be a contraction to try to incorporate abstraction into a visual programming environment which draws its power from the objective to make all objects visible and concrete. Distinguishing between representation of abstraction and abstraction of concepts resolves this discrepancy. Vista supports three kinds of abstraction:

- separation of interface and implementation
- substitution of processors
- inheritance hierarchies

In the following we will explain each of these in more detail.

3.2.1 Separation of interface and implementation

When designing a software component, one has to decide, among other things, what the functional level of the component should be, how much flexibility should be built-in, and how the particular component is to communicate with other components. On the last point, the only way communication within a modular system can take place is via the interfaces of its components. The consequence of this is that using a component requires only knowledge about the specification of the interface; the component's internal structure does not matter or even need not be available when referencing the interface. This fundamental principle is called information hiding.

Vista strictly separates interface and interior at the visual layer. As soon as the interface is defined, its associated component is virtually available: the programmer can see it, touch it, drag it out of the component catalog, and drop it onto the programming area. Of course, nothing will happen if performance of an operation is requested from a component which is not yet implemented, but all communication connections can already be established. When the implementation eventually become available, things start working automatically. This supports the top-down approach of system decomposition and fits into what Richard P. Gabriel [Gabriel 1993] means by abstraction in programming: "An abstraction facilitates separation of concerns: The implementor of

an abstraction can ignore the exact uses or instances of the abstraction, and the user of the abstraction can forget the details of the implementation of the abstraction, so long as the implementation fulfils its intention or specification."

3.2.2 Substitution

Regarding functionality and flexibility, there is a strong correlation between the capabilities of a component and its range of applications in different contexts. In general it can be said that increasing functionality decreases flexibility. That is because a high-level component depends on other high-level components and a number of base modules, with the connections among them mostly hard-wired. Substitution, which also may be called configuration or customizing, can help to overcome this dilemma.

By substitution in the context of Vista, we mean that parts of a processor may be replaced with parts having a compatible interface. It is important to note that some or all of the parts being anticipated for exchange may offer unimplemented operations in their interface. Such parts must be replaced in order to make the containing processor fully functional.

Parts intended for exchange are called public processors as their replacement is performed from the outside. They should not be confused with parameters of methods as they are more than just an object; they are full-fledged processors. Being processors, they have the ability to directly control the flow of tokens inside the processor which they are parts of. Controlling the receiver is not possible for ordinary objects passed to methods as parameters.

Substitution can take place either statically or dynamically. Static substitution happens when a programmer uses a processor in a particular context and customizes this processor in order to achieve the desired behavior. For this purpose one or more of the processor's public processors are superseded. The static replacement of public processors resembles the exchange of chips on a digital board. Dynamic substitution is possible, too. Here a message is sent to the processor whose public processor is to be exchanged. The name of the message is the same as that of the processor to be replaced and the message's sole parameter is the substitute (see also Figure 7)

Because public processors can be replaced by all compatible processors and not only by those of exactly the same type, substitution allows higher abstractions than the simple separation of interface and implementation. Substitution abstracts from the implementation of a group of classes of objects, whereas the separation of interface and implementation hides only the implementation aspects of a single class.

3.2.3 Inheritance hierarchies

Vista is a true object-oriented programming environment; it offers all common features of object-oriented programming languages like inheritance, polymorphism and dynamic binding. We presume that the reader is familiar with this basic concepts of OOP and therefore omit a broader discussion of these terms. Furthermore, we concentrate on Vista's inheritance mechanism, since polymorphism and dynamic binding are essentially a side effect of Vista's Smalltalk-based implementation.

In object-oriented languages like Smalltalk, classes are organized in an inheritance hierarchy. A subclass inherits attributes (instance variables) and methods from its superclass. Processors of Vista are also organized in such a hierarchy. Actually this is a subtree of the Smalltalk class tree rooted at the abstract class *CompoundProcessor* (for visually defined processors). Inheritance of a processor involves all its properties,

namely public and private processors, public and private methods, ports and networks. For all these properties with the exception of networks, Vista does not care about inheritance but relies on the standard mechanism of Smalltalk. Networks are treated specially because they implement instance-based behavior (in contrast to methods, which are classed based). Hence we wrote our own dynamic binding mechanism, which finds the appropriate network when a token arrives at an input port.

Inheritance provides an excellent means for abstraction. It permits the implementation of abstract processors that are semifinished components. An abstract processor defines a general interface of its subprocessors and factors out common behavior into nonempty networks and methods. Empty networks and methods are placeholders for unique behavior and have to be overridden in subprocessors in order to make them usable.

Inheritance together with substitution provide an excellent means to build object-oriented frameworks. This is a superior technique that codes common patterns of a certain problem domain into highly reusable abstractions [Stritzinger 1991]. Vista's support for the construction of frameworks is perhaps its greatest strength besides the visual facilities.

3.3 Weakest Coupling

Bertrand Meyer [Meyer 1988] defines the *weak coupling* or *small interfaces principle* as follows: "If any of two modules communicate at all, they should exchange as little information as possible."

He justifies this principle in particular by the criterion of continuity and protection, which demands that changing a small part of a module's interface should result in little modification of other parts of the system. But that is not the only advantage. Weak coupling also ensures the composability of software components, because a small communication channel increases the probability of finding components that are able to participate in communication. An example is the toolbox of UNIX commands. A lot of UNIX commands can work on a stream of bytes; they read bytes from a default input channel, process them, and write them to a default output channel. These channels form a very narrow interface. Consequently UNIX commands adhering to this standard can be combined in thousands of ways.

The UNIX toolbox example shows that there is something beyond weak coupling, which we call weakest coupling. Using the terminology of Vista, we outline the *weakest coupling principle* as follows: "For a signal processor to inform any other processors about anything, it should not address these processors explicitly but broadcast a signal together with a minimum amount of related data".

With this principle we expect to obtain maximum composability even for building blocks that were never planned to work together. There is a good chance that unforeseen combinations of building blocks are possible if 1) data received or transmitted is simple and 2) sender or receiver of information need not be known in order to process data.

The elegance of data-flow languages is founded on point two of this observation. The implementor of a functional block for a data-flow language has to think only about the input/output channels, whereas the user of such blocks may combine them in arbitrary ways, provided that data types match.

It was an incentive for us to apply the principle of weakest coupling for an object-oriented visual language that encourages the combination of high-level building blocks

instead of focusing on algorithmic aspects. The search for an appropriate construct eventually led to Vista's signal processor, an object that mainly communicates with the environment through signal tokens instead of sending and receiving messages (see Section 2.2.1, "Reactive Systems").

Observance of weakest coupling leads to one-way communication relationships where the sending processor need not care about the receiver and the receiver may react to the signal without being aware of where it comes from. Vista is not the originator of such interobject communication. The change-and-update mechanism of Smalltalk [Goldberg and Robson 1983] is based on a similar concept. One difference between Smalltalk's mechanism and Vista's signal metaphor is that the change-and-update mechanism is used for certain design patterns only, e.g., within the MVC architecture, whereas in Vista one would not program without signals. Another difference is that change-and-update relationships are invisible, whereas in Vista signal connections are concrete and visible objects.

We cannot predict how much productivity can be increased using weakest component coupling or what the penalty will be in terms of performance or efficient use of resources. Empirical results will emerge over time and with the construction of larger, nontrivial software.

3.4 Analysis of Type Information

Preferences for static or dynamic typing fuel a never ending dispute among committed proponents of the respective points of view. In object-oriented languages like C++ and Eiffel [Ellis and Stroustrup 1990; Meyer 1988], the compiler can detect whether a message sent to an object is recognized by this object (i.e., if the object is of the right type), whereas in languages like Smalltalk and Self [Ungar and Smith 1987] this can be determined only at run time.

Both kinds of type checking have their advantages and disadvantages. Dynamic type checking offers a maximum of flexibility and simple declaration of variables, with the drawback of some uncertainty, unreadability and inefficiency of programs. Static type checking restricts polymorphism and may force the programmer to perform obscure type casts. Nevertheless, static typing supports documentation and maintenance, offers safety guarantees, and contributes to efficiency of message passing.

Vista follows a best-effort strategy with regard to type checking according to the motto: "Never let the user provide information the system already knows." It uses all available information to detect and prevent actions which would obviously lead to type incompatibilities, but does not guarantee that every operation specified is type clean. The advantages of static type checking are reflected in Vista as follows.

- *Readability*. Since every object of Vista's programming model is visible and alive, the programmer can always recognize the type of a certain object by either looking at its representation (layout, icon, ...) or by inspection.
- *Efficiency*. Message passing in Vista is done exclusively by the underlying, dynamically typed Smalltalk system. Here efficiency is not improved.
- *Safety*. Actions by the programmer concerning substitution of public processors with other processors are type safe. When such an operation takes place, Vista automatically checks whether the surrogate processor is com-

patible with the public processor. If this is true, the substitution is carried out; otherwise an error message is reported.

Although Vista's support of type checking is more ad hoc than rigorous, it helps the programmer to avoid unintentional errors without any modification of the Smalltalk language.

4 A Complete Example

Explaining and understanding visual programming without pictures is almost impossible. In this section we give a complete example of a simple application built with Vista. We want to demonstrate how the various concepts fit together at the visual layer and reveal how Vista cooperates with a third-party user interface builder.

4.1 Task

A thermo-alarm processor (TA) is to be built that receives temperature values, compares them with a given lower and upper limit and triggers an alarm device if the temperature is out of range. The limits specifying the temperature range can be changed. TA is to maintain the constraint that the lower limit never goes beyond upper limit. The alarm device has to be substitutable. TA triggers whenever the temperature goes out of range and when temperature is normal again. The left side of Figure 4 shows TA's interface with public processors *max*, *min* and *alarmDev* for upper and lower limit, and the alarm device. Below this picture TA's iconic representation is shown.

For testing purposes we need a user interface that looks like the prototype at the right side of Figure 4. The sliders *Temp*, *High* and *Low* modify the corresponding values the TA should watch. Temperature values are displayed numerically below the

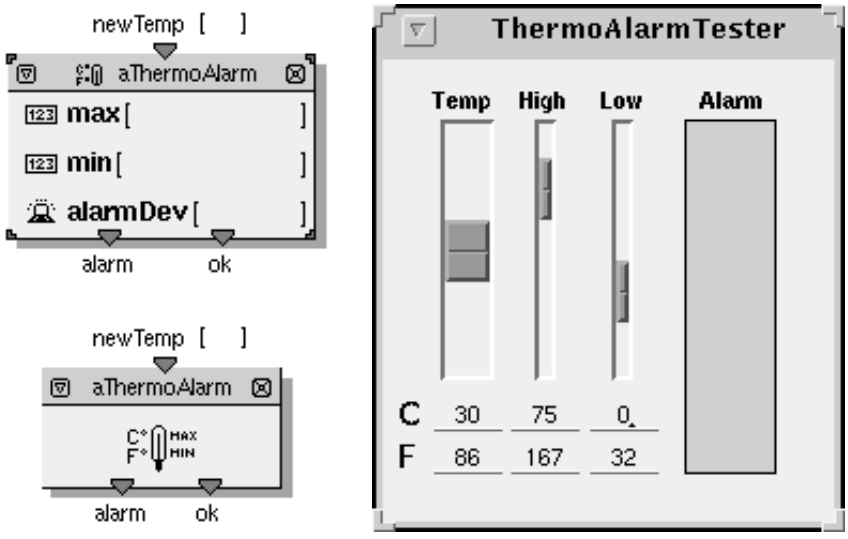

Figure 4. Thermo-alarm processor (left) and user interface prototype (right)

sliders (in degrees Celsius and Fahrenheit). The area labeled *Alarm* is a colored region that is green if the temperature is within the limits, else blinking red. This region is independent of the exchangeable alarm device, which actually is an invisible speaker.

4.2 Building the User Interface

Vista does not provide a user interface builder itself. We use VisualWorks [ParcPlace 1992b] for building user interfaces of Vista programs. VisualWorks is built on Objectworks\Smalltalk. It provides an integrated environment for operating system independent application development. A VisualWorks application consists of

- a *user interface* built with a set of common widgets like windows, sliders, push buttons and list panes
- an *application model*, which coordinates the effect of user input and links the user interface to the underlying domain model
- a *domain model*, which defines the essential structure and behavior of the application (in our case, this is the TA processor)

Figure 5 shows this layered architecture applied to our thermo-alarm system. The application model is between the user interface on top and the domain model at bottom.

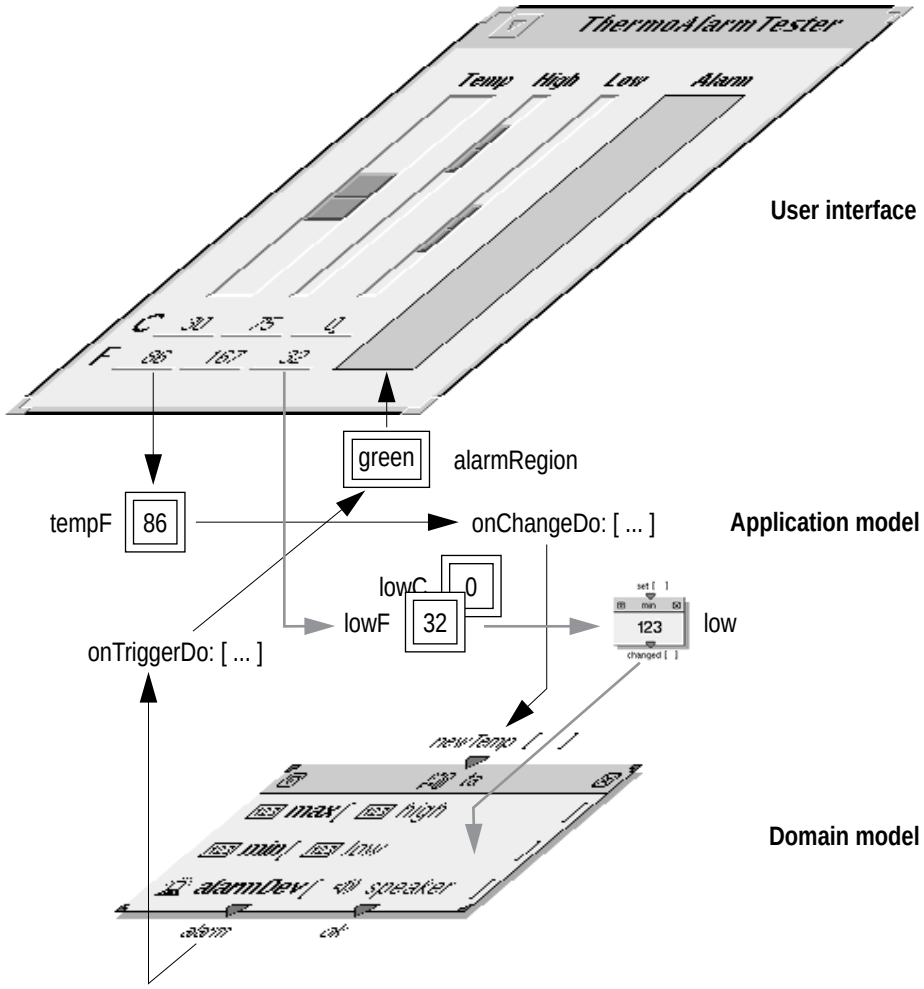

Figure 5. The layered architecture of VisualWorks

It consists of a number of elements that are either value holding objects for the user interface widgets (*tempF*, *lowF*, *lowC*, *alarmRegion*) or connectors to the domain model (*onChangedDo*, *low*, *onTriggerDo*). The application model is sketched only partially in order to avoid clutter. Before explaining the construction of a TA processor, we show how a particular TA is connected to the application model.

The light gray arrows in Figure 5 depict the principle flow of control when the user moves the slider for adjusting TA's minimum temperature limit. Any movement of the slider changes *lowF* which stores the current lower limit. This value holder forms the glue between two more objects, namely its (unnamed) visual display at the user interface and *lowC* that holds the limit converted to Celsius. These two objects depend on *lowF*. So, whenever *lowF* changes, they are updated, too. In the case of *lowC*, a conversion function is triggered that assigns the result of the formula *(lowF-32)*5/9* to *lowC*. This in turn updates *lowC*'s visual representation on the user interface.

Although we have not engaged Vista so far, the thermo-alarm system is already partially functional. Moving the sliders updates the number displays accordingly, and editing any of the numbers updates the corresponding slider as well as their Celsius and Fahrenheit display.

The alarm region is still dead, as the necessary logic is not connected yet. This leads us to Vista and to the embedding of TA. First, we create a new TA from class *ThermoAlarm* (see 4.3.1, "Defining the Interface"). Then we link TA's public processors with the related objects of the application model. As TA expects processors at its interface, we have to convert the value holders *lowF* and *highF* into number processors. We do this by sending them the message *asProcessor*. Figure 6 shows the situation after the message has been sent to *lowF* which currently holds the value 32.

The underlying mechanism that creates the object relations shown in Figure 6 does the following: First, it instantiates the number processor *low* from class *NumberProcessor*, which belongs to the set of Vista's primitives. This processor does not directly store the associated number, but maintains a reference to the value holder *lowF*. Second, the created processor is added to the list of objects dependent on *lowF*. Thus the value holder *lowF* automatically informs the processor *low* about changes.

The number processors *high* and *low* are plugged into the corresponding slots *max* and *min* of TA. In addition TA's abstract alarm device is replaced by a concrete speaker object.

Now we send the value holder *tempF* the message *onChangeDo:*, thus connecting TA's input port *newTemp* to *tempF*. The block parameter supplied by the message specifies that whenever *tempF* changes the current temperature is passed to *newTemp*. Linking TA's output ports with the alarm region is the final action in order to fully connect TA to the application model. To achieve this, the message *onTriggerDo:* with a block parameter is sent to ports *alarm* and *ok*. Each time one of these ports triggers, the statements inside the blocks are executed, thus updating the alarm region. Figure 7 out-

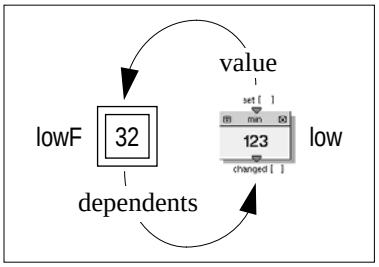

Figure 6. Interconnection of a value holder and a number processor

```
ApplicationModel subclass: #TATester

initialize

ta := ThermoAlarm new
        min: (self lowF asProcessor);
        max: (self highF asProcessor);
        alarmDev: (Speaker new).
tempF onChangeDo: [
        ta newTemp: self tempF value].
ta alarm onTriggerDo: [
        self alarmRegion startBlinking.
        self alarmRegion color: #red].
ta ok onTriggerDo: [
        self alarmRegion stopBlinking.
        self alarmRegion color: #green].
```

Figure 7. Initialization of the thermo-alarm processor

lines the method *initialize* of class *TATester* that is necessary for proper initialization of TA.

Now the user interface has been built and the code to couple it with TA has been written. The next step is to implement TA itself by means of the visual part of Vista.

4.3 Building the Thermo Alarm Processor

To ease the description of the construction process, let us build TA from scratch, thus assuming that it has nothing in common with any existing processor. When building a new Processor, at first we get two templates, one for the processor's interface, another for its implementation. These templates are to be filled with ports, processors, methods and networks.

4.3.1 Defining the Interface

The interface of a processor consists of input and output ports, public processors and public methods. For TA we need an input port and two output ports which we name *newTemp*, *alarm* and *ok*. After naming and typing, the ports immediately show up in the template. Next we select the required public processors from Vista's processor register, where all processors are organized in categories. We do so by dragging two number processors and one alarm device to the section *Public Processors* of the interface template. Afterwards we name them *min*, *max* and *alarmDev*. We do not need any public methods and therefore leave the corresponding section blank.

The interface is now complete and we are able to catalog (install) the TA processor in Vista's register. By that the interface is turned into the Smalltalk class *ThermoAlarm*. The new processor is yet not functional but ready to be used anywhere. The top of Figure 8 depicts TA's complete interface and the processor register.

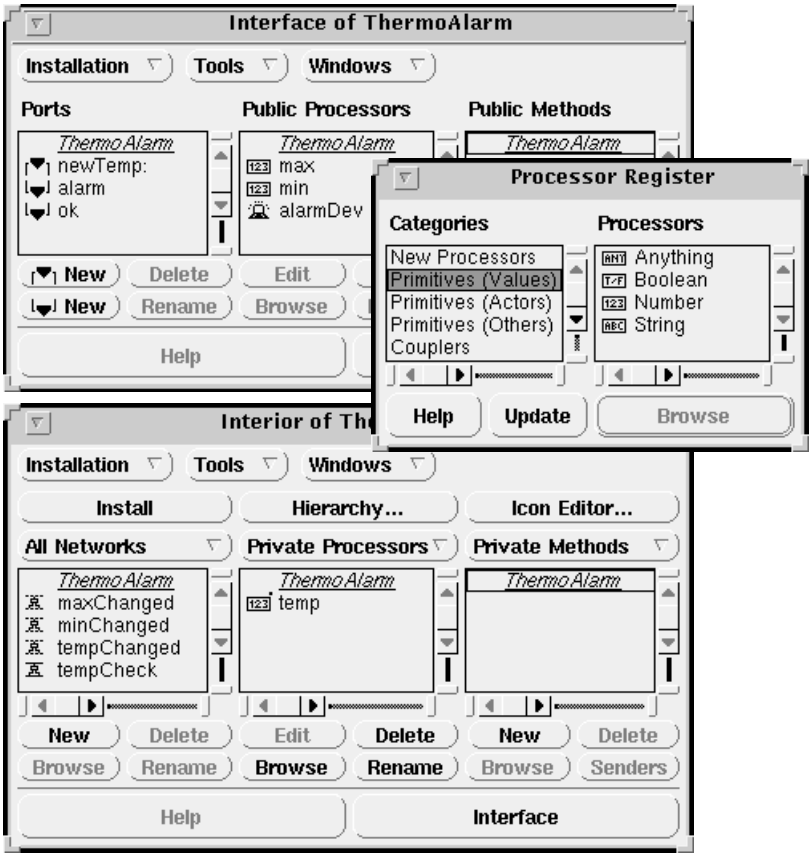

Figure 8. Interface and interior of the thermo-alarm processor, register of processors

4.3.2 Defining the Interior

Considering TA's task, the interior of TA can be kept quite simple. A private processor named *temp* holds the last reported temperature. Additionally we need three X-networks, namely *tempChanged*, *minChanged* and *maxChanged*, which handle changes of the temperature, and the lower and upper limits. A fourth internal network comprises the common part of all X-networks. It is used to test if a temperature value violates the limits. We name that network *tempCheck*. Figure 8 depicts the interior template after creation of processor *temp* and all networks. Let us now consider each particular network.

The network *tempChanged* (see Figure 9) just contains aliases of the private processor *temp* and the internal network *tempCheck*. The brackets attached to the input port *newTemp* indicate that signals arriving at this ports carry one data item (i.e., the new temperature value). After storing the new temperature in *temp* (connection 1) *tempCheck* tests if the temperature exceeds the limits (connection 2).

Figure 9 also depicts network *tempCheck*. This network is shared among the other networks. The input and output bars are solid, indicating its special purpose. Inside the network we see a local processor named *rangeChecker* and the public processor *alarmDev*. If port *check* of processor *rangeChecker* triggers, a signal is emitted at one of the output ports *tooLow*, *tooHigh*, or *ok*. Which port is actually triggered depends on whether the current temperature violates the limits. A signal emitted by any of the ports *tooLow* or *tooHigh* switches on the alarm device and is further routed to the net-

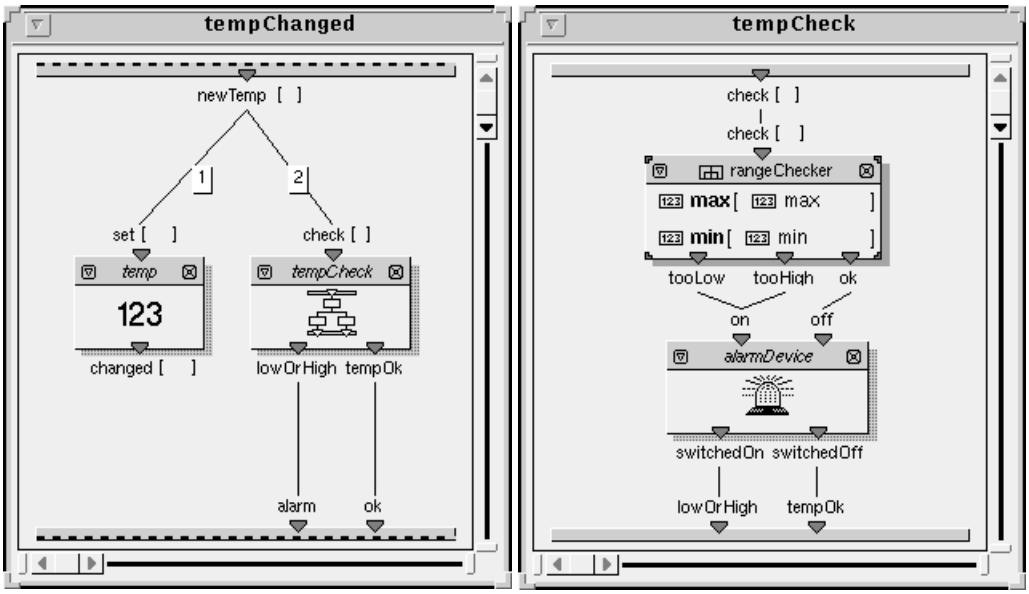

Figure 9. Networks *tempChanged* and *tempCheck*

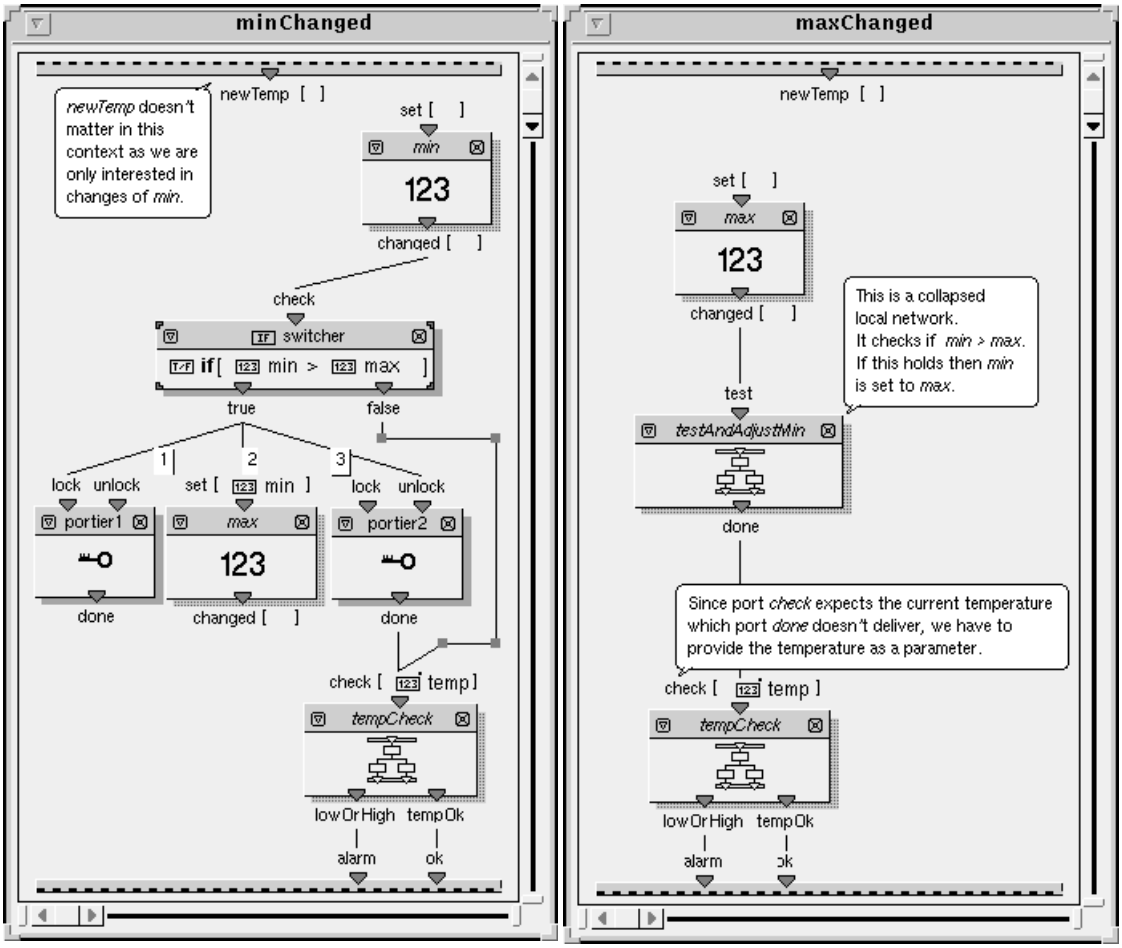

Figure 10. Networks *minChanged* and *maxChanged*

work's port *lowOrHigh*. A signal emitted by port *ok* turns off the alarm and eventually triggers port *tempOk*.

Figure 10 shows the networks *minChanged* and *maxChanged*. These networks are activated whenever the value of *min* or *max* changes. Network *minChanged* is fully exposed, whereas parts of *maxChanged* have been collapsed into the local network *testAndAdjustMin*. Network *maxChanged* works analogously to *minChanged*; hence we explain only the expanded network.

The switcher within network *minChanged* first checks whether *min* > *max*. If so, *max* is set to *min* in order to ensure the integrity of the temperature range. Afterwards *tempCheck* is activated just as within the network *tempChanged*. The two processors *portier1* and *portier2* are of particular interest. These processors temporarily prevent port *changed* of processor *max* from sending out a signal. This is necessary because otherwise the operation that sets *max* to *min* would have a side effect on network *maxChanged*.

Now the TA processor is finished. Installing the interior in class *ThermoAlarm* is the final step required in order to turn the networks into methods of this class.

Figure 11 shows a snapshot of the Vista programming environment during the construction of TA. The Vista workbench consists of a variety of tools which are designed to offer visual interaction at a level suitable for the supported task. Currently available are

- registers for organizing and storing objects
- browsers, editors and inspectors for exploring and manipulating objects and relationships among them
- animators and debuggers for the interactive execution of signal and data networks
- on-line help for most parts of the environment
- an annotation facility, which lets the user attach arbitrary information to any object

We believe that these tools strike the right balance between graphical and textual representation, hence saving screen space and avoiding visual overload. A preliminary version of the Vista programming environment is available on SparcStations, on the Macintosh, and under MS-Windows.

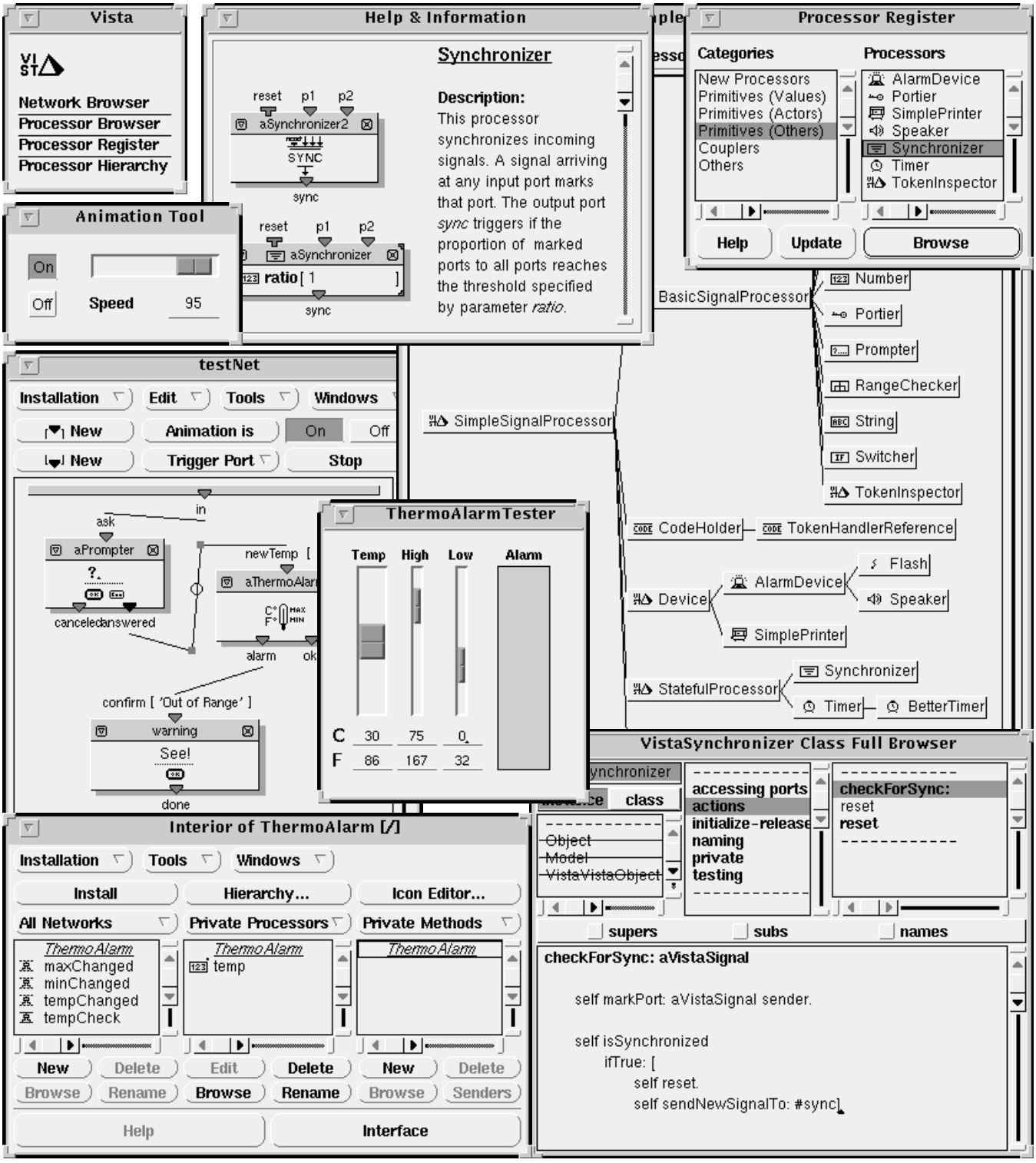

Figure 11. Snapshot of the Vista programming environment

5 Further Work

Several redesign cycles were needed to simplify and clarify the key concepts of Vista, but so far the final stage has not been reached. Our current work on Vista covers concurrency, distribution, and performance issues.

A problem which remains conceptually unsolved concerns the indeterministic behavior of Vista programs if asynchronous forwarding of tokens is allowed. In this case one token may pass another, thus causing unexpected and undesirable side effects. We are currently exploring several formalism and strategies to cope with this problem. One of the decisions we will have to make will be whether mechanisms which deal with these aspect of concurrency should be built into Vista or provided by special processors.

In addition, we are facing problems concerning execution speed and memory requirements. Vista's communication mechanism, based on token passing, is sluggish even compared to the execution of Smalltalk methods. We feel that the use of optimization heuristics as well as compiler techniques are needed in order to obtain better performance. With respect to memory requirements the prototypical characteristics of processors leads to heavy objects. For instance, every processor has to instantiate each of its own networks even if there is no structural difference among networks belonging to the same class of processors. Despite the advantage of this approach, which allows self-modifying objects (processors modifying their subprocessors and networks dynamically), in many cases such capabilities are not needed but introduce a remarkable overhead. Whether the interesting possibilities offered by prototypes is worth the additional complexity needs careful justification.

Improving Vista's performance is the most important next step, as this is the precondition for programming real-world applications with Vista. Gathering experience with the implementation of large systems will help to advance with the conceptional work. Additionally, this work will indicate whether Vista has the potential for significant rationalization of the software development process — the global goal of software engineering.

Summary

This paper has outlined concepts and application issues of Vista – a multiparadigm system that supports the construction of semifinished components in the realm of reactive and transformational systems.

Various kinds of processors (signal processors, data processors, and couplers) are central concepts to Vista. Usually processors communicate with each other by means of tokens that are either signals or data items. Token routing networks combine processors. This communication mechanism leads to weakly coupled, high-level building blocks.

Within the Vista project we have strived to master complexity of larger visual programs by obeying such important software engineering principles as

- weak coupling of building blocks
- functional and data abstraction
- security by introducing an elementary type system
- component reuse
- direct integration of documentation parts in visual programs

Visual programming has become increasingly attractive in recent years. Now an engaged discussion on how to incorporate software engineering principles has to take place to make visual programming relevant for professional software production. We are convinced that combining visual with object-oriented programming is a step in the right direction. Vista follows that approach. It will hopefully make programming less frustrating and more productive.

References

[Ambler and Burnett 1989] A. L. Ambler and M. M. Burnett, "Influence of Visual Technology on the Evolution of Language Environments," *Computer*, Vol. 22, No. 10, Oct. 1989, pp. 9-22.

[Chang 1990] S.K. Chang (ed), *Principles of Visual Programming Systems*, Prentice Hall, New Jersey, 1990.

[Cox and Pietrzykowski 1988] P.T. Cox, T. Pietrzykowski, "Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism," *Proceedings of the International Computer Science Conference*, 1988, pp. 695-704.

[Digitalk 92] *PARTS Workbench User's Guide*, Digitalk Inc., 1992.

[Ellis and Stroustrup 1990] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, 1990.

[Gabriel 1993] Richard P. Gabriel, "Abstraction descant, part I," *Journal of Object-Oriented Programming*, Vol. 6, No. 1, Mar./Apr. 1993, pp. 10-14.

[Gamma 1992] E. Gamma, *Objektorientierte Software-Entwicklung am Beispiel von ET++ Design-Muster, Klassenbibliothek, Werkzeuge* (in German), Springer, Berlin, 1992.

[Glinert 1992] E. P. Glinert, "Visual Programming Environments and Graphical Interfaces: Where We Are Now, Where We're Headed," *Tutorial notes at the 1992 IEEE Workshop on Visual Languages*, Seattle, Washington, 1992.

[Goldberg and Robson 1983] A. Goldberg and D. Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, Reading, 1983.

[Harel 1987] D. Harel, "Statecharts. A visual formalism for complex system," *Science of Computer Programming*, Vol. 8, No. 3, Jun. 1987, pp. 231-274.

[Harel et al. 1990] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," in *Software State-of-the-Art: Selected Papers* (Tom DeMarco and Timothy Lister ed.), Dorset House Publishing, New York, 1990, pp. 322-338.

[Jacobson et al. 1992] I. Jacobson, M. Christerson, P. Jonsson, and G. Oevergaard, *Object-Oriented Software Engineering—A Use Case Driven Approach*, Addison-Wesley, Wokingham, 1992.

[Meyer 1988] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, New York, 1988.

[Myers 1986] B. A. Myers, "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy," *Conference Proceedings, CHI'86: Human Factors in Computing Systems*, Boston, Apr. 13-17, pp. 59-66.

[ParcPlace 1992a] *Objectworks\Smalltalk Release 4.1, User's Guide*, ParcPlace Systems Inc., 1992.

[ParcPlace 1992b] *VisualWorks Release 1.0, User's Guide*, ParcPlace Systems Inc., 1992.

[Price et al. 1993] B. A. Price, Ronald M. Baecker, and Ian S. Small, "A principled taxonomy of software visualization," *Journal of Visual Languages and Computing*, Vol. 4, No. 3, Sep. 1993, pp. 211-266.

[Pirklbauer et al. 1992] K. Pirklbauer, R. Plösch, and R. Weinreich. "ProcessTalk: An Object-Oriented Framework for Distributed Automation Software," *Proceedings of the 4th International Symposium on Systems Analysis and Simulation*, Elsevier, 1992, pp. 363-368.

[Selker and Koved 1988] T. Selker and L. Koved, “Elements of Visual Lamguages,” Proceedings of the 1988 Workshop on Visual Languages, Pittsburgh, Pennsylvania, 1988, pp. 38-44.

[Serius 92] *Serius Programmer User’s Guide*, Serius Corporation, 1992.

[Shneiderman 1983] Ben Shneiderman, “Direct Manipulation: A Step Beyond Programming Language,” *Computer*, Aug. 1983, pp. 57-69.

[Shu88] N. C. Shu, *Visual Programming*, Van Nostrand, 1988.

[Stritzinger 1991] A. Stritzinger, “Reusable Software Components and Application Frameworks, Concepts, Design Principles and Implications”, PhD thesis, University of Linz, Austria, 1991.

[Shlaer and Mellor 1992] S. Shlaer and S. J. Mellor, *Object Lifecycles—Modelling the World in States*, Yourdon Press Prentice Hall, Englewood Cliffs, 1992.

[Gunakara 1992] *Prograph Reference Manual*, The Gunakara Sun Systems Ltd., 1992.

[Ungar and Smith 1987] D. Ungar, R. Smith, “Self: The Power of Simplicity,” *Proceedings of the OOPSLA 87 conference*, Paris, Vol. 22, No. 12, Dec. 1987, pp. 227-242.

[Weinand et al. 1989] A. Weinand, E. Gamma, and R. Marty, “Design and Implementation of ET++, a Seamless Object-Oriented Application Framework,” *Structured Programming*, Vol. 10, No. 2, 1989, pp. 63-87.