In Proceedings 1994 IEEE Symposium on Visual Languages, Oct. 4-7, 1994, St. Louis, MO, USA, p. 40-47

Concepts and Architecture of Vista - a Multiparadigm Programming Environment

Stefan Schiffer and Joachim Hans Fröhlich

C. Doppler Laboratory for Software Engineering
Johannes Kepler Universität Linz, A-4040 Linz, Austria
Email: {schiffer,froehlich}@swe.uni-linz.ac.at

Abstract

This paper describes Vista, a visual multiparadigm programming environment. We introduce the notion of processors and networks and discuss their application in the construction of event-driven and data-transformation systems. Further, we give an overview of Vista's object-oriented architecture.

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 laity to encroach on the domain of computer experts to solve problems with the computer. 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 some might call such environments visual, that is not what we mean by VP. 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 [7], PARTS [1], Prograph [9] and Serius [11] fall into this category.

Our approach for a VP environment is to join object-oriented programming based on Objectworks\Smalltalk [8] and programming with signal flow and data flow. We call the model and its programming environment Vista, an acronym for "VIsual Software Technique Approach", with equal emphasis on "visual" and "software technique". We tried to rigorously uphold software engineering principles for the conception of the model as well as for the implementation of the environment. Together with the environment, the model:

- provides a visual language for defining signal and data networks for the construction of reactive and transformational systems
- 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 strives to provide substantial expressiveness in the visual layer as well as concepts necessary to build real applications. 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.

Vista promotes evolutionary prototypical development of object-oriented software systems during design and implementation.

In the following we introduce Vista's programming model and give a rough sketch of its internal architecture.

2 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, we have visual and tangible representations of both simple and compound processors and various access is granted depending on the intended manipulation: if a processor is to be redesigned, the programmer has full access to all aspects of its structure

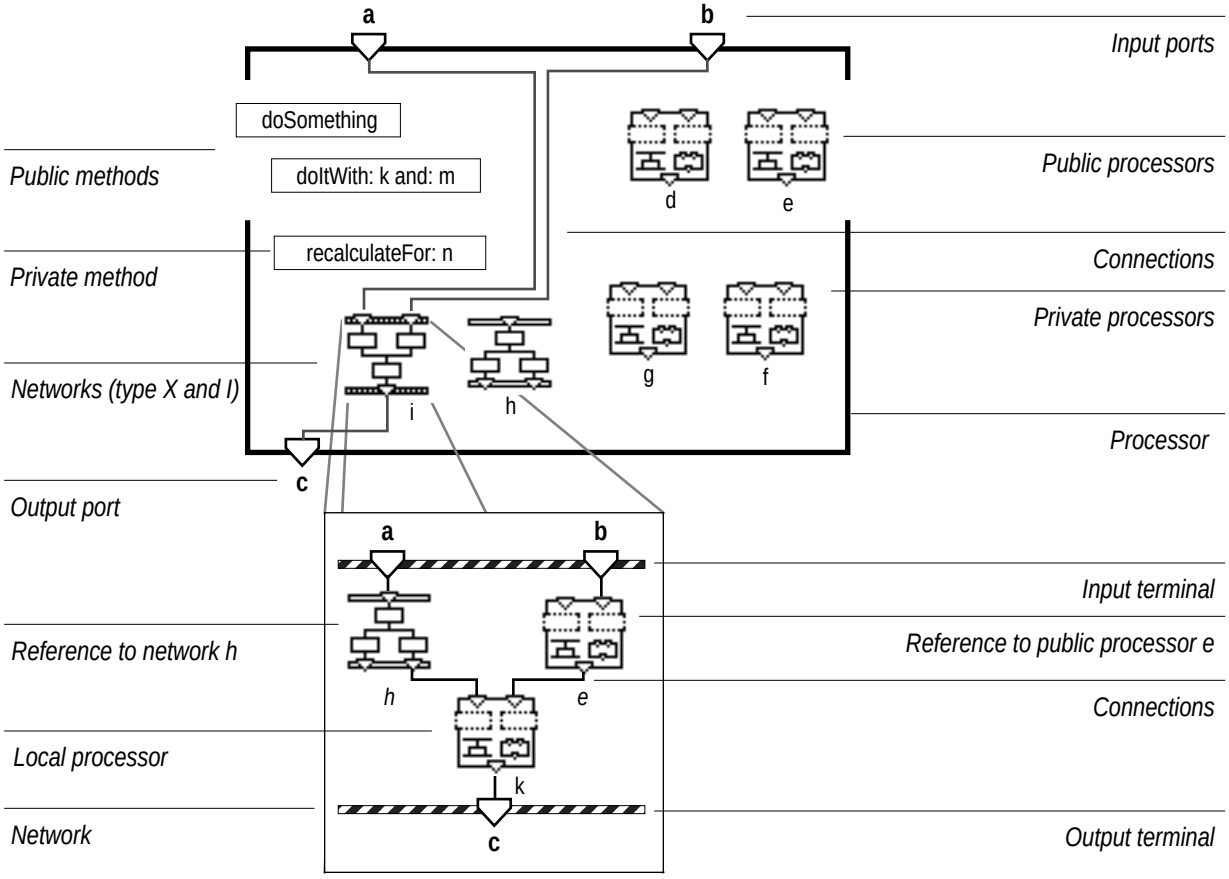

Figure 1. The fundamental structure of a processor

and behavior; if a processor is to be used to construct another processor, only an external (black box) view of the processor is given. Interaction between processors is carried out via messages and tokens. Message synchronously activate methods and return values. For token passing, 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 Processors

A processor is consists of an interface part which is accessible from outside and an internal part which can only be accessed by the processor itself. Figure 1 shows the overall architecture of a processor with its interface (ports and objects within the 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 its interface. Public processors are designated to be replaced by other processors having the same interface. Imagine 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 allows the construction of frameworks as well as convenient customizing.

The internal appearance of a processor depends on whether it is simple or compound. The interior of simple, textually defined processors consists of instance variables and methods, just as with any class in Smalltalk. This need not be discussed further. Of greater interest are visually constructed compound processors. They encapsulate private processors (simple or compound), private methods, and networks. Private processors represent the constituent components of a compound processor and therefore conform to the is-part-of relationship. They may be considered as internal devices used to perform the enclosing processor's task.

For the benefit of uniformity there is no other construct besides processors for the visual construction of compound processors. This applies even to atomic attributes such as characters and numbers.

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 a prototypical instance to be defined in the sequel. After having finished the prototypical processor, the programmer generates a class definition from it. Thereby the processor becomes part of the library of reusable components. The same procedure may be applied in order to subclass already existing processors.

2.2 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. Vista's method concept is identical to that of Smalltalk. Therefore we do not explain it further but rather focus on the notion of networks.

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. Whenever a token arrives at an input port, it is passed to one or more networks.

Networks are ensembles of linked processors or other networks. Like processors, networks have ports to receive and send tokens. The ports of a network are collected by *input* and *output terminals* 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. 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 onto the execution stack and removed from there when control returns to the caller. Networks, by contrast, represent not code, but objects that are already alive when a token arrives at an input port. 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, providing connection points for the enclosing processor's ports and a means of network sharing, respectively:

- *X-networks,* which either interconnect the processor's input ports and output ports or join internally known entities with the processor's output ports
- *I-networks,* which can be used within other networks similar to subroutines

Networks support synchronous and asynchronous forwarding of tokens. In synchronous mode, tokens are transmitted with a depth-first strategy. That is, if an output port is connected to more than one input port, 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. In asynchronous mode, each connection leading away from an output port creates a new concurrent token forwarding process. Later on, these processes may deliver tokens to an input port simultaneously. Semaphores are provided for situations were mutual exclusive token handling is required.

3 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: *signal*, *data* and *coupler processors*. All these processors handle *signal* or *data tokens* which pass control or data respectively, 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. 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.

3.1 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.

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 [3] and object-oriented approaches [4,12]. 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 are beyond the project's present scope (but see Section 5, "Further Work"). Instead, we designed an environment for the rapid 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 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.

3.2 Transformational systems

We outline a transformational system as a data-converting ensemble of components that either generates or transforms data [2]. 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 in 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-

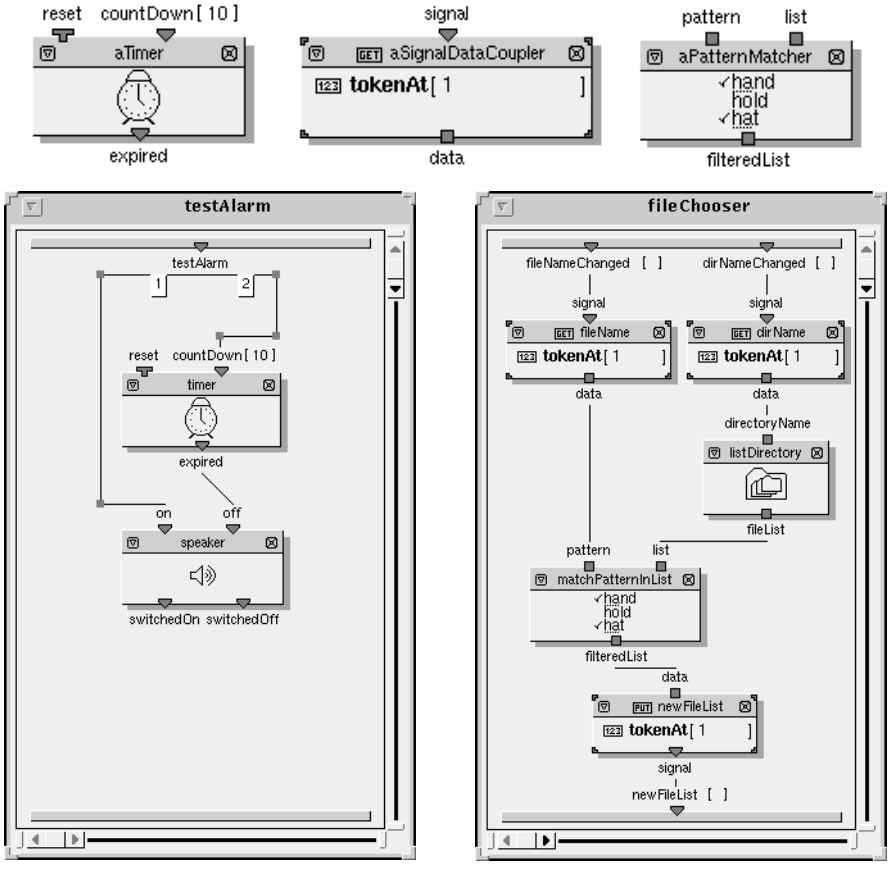

Figure 2. Processors and networks
Top (l. to r.): Signal processor, signal/data coupler, data processor
Bottom: Signal network, data network

flow semantics with no additional control-flow structures, sequential execution constructs, or other augmentations. 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 noninteractive, batch-oriented problems like data filtering, image processing and format conversion.

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. This 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.

3.3 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.

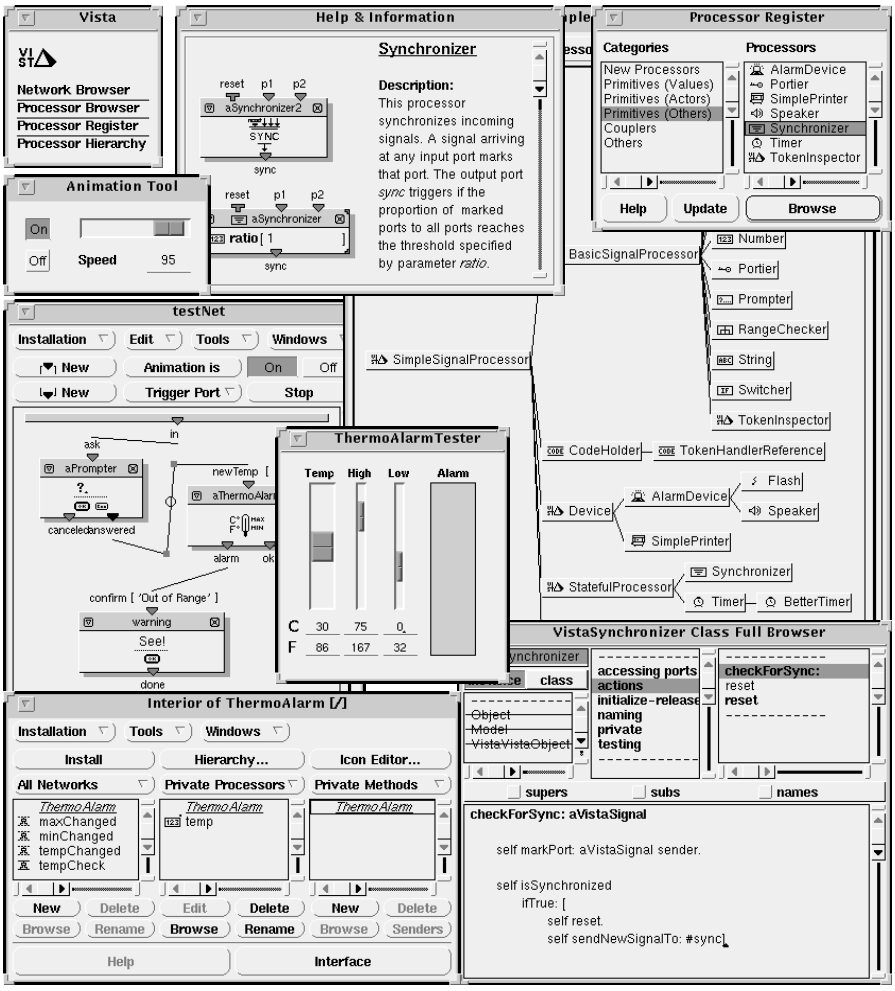

Figure 3. Snapshot of the Vista programming environment

Figure 2 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. A signal arriving at port *testAlarm*, first switches on the speaker (connection 1) and then starts a 10-second timer countdown (connection 2). When the timer expires, it emits a signal at port *expired*, which switches off the speaker.

The right diagram of Figure 2 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 port *pattern* of the data processor *matchPatternInList*. This processor checks each element of the sequence at its port *list* 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 the file chooser for the first time, signals on both ports *fileNameChanged*: and *dirNameChanged:* are required before *matchPatternInList* can start processing. Later on, each signal token produces a new filtered directory list as ports of data processor save old tokens.

Figure 3 shows a snapshot of the Vista programming environment during the construction of a thermo-alarm processor. 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 information to any visible 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.

4 Architectural overview

To date little literature has been published on the internal architecture of visual programming systems. This may be due to the more traditional approaches where the visual programming language has almost nothing in common with its implementation language. In Vista, however, the visual programming system and the underlying internal architecture are closely related: both strictly obey object-oriented principles; even more, they heavily rely on the Smalltalk programming environment.

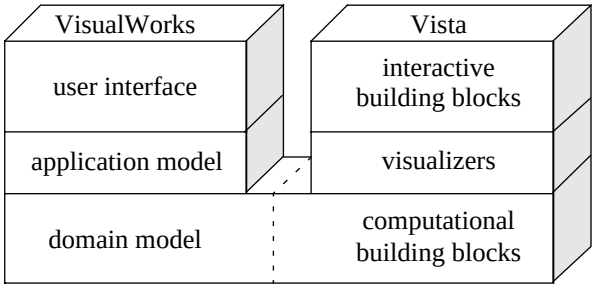

Figure 4. Overall architecture

A simple object-oriented design would provide capabilities for objects of these classes to directly represent themselves at the graphical user interface level, thus tightly coupling the user interface and the conceptional model. One may argue that such a design is preferable because immediate feedback on user interaction can be provided by the computational units without the need for frequent callbacks. This approach presumes that the user interface is as important as the underlying computational layer [5]. We contradict this assumption in most cases. Actually the reverse (and traditional) approach, namely the separation of user interface components and computational units, leads to much more flexible systems which allow dynamic exchange or removal of the visual layer without affecting the logic behind it. This is a highly desirable property if, e.g., a visual program is to be compiled into a stand-alone application without the overhead of the visual layer. Figure 4 pictures Vista's overall architecture, which gives Vista its flexibility.

Vista does not provide a user interface builder itself. We use VisualWorks [13] for building user interfaces of Vista programs. VisualWorks provides an integrated environ-

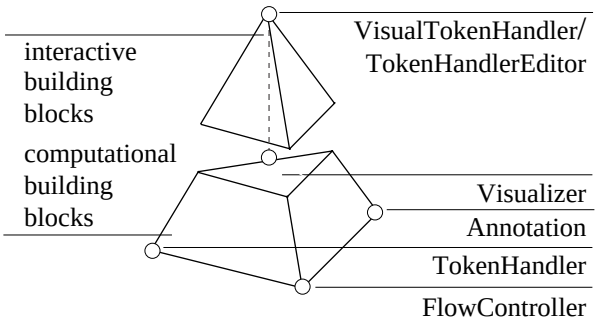

Figure 5. The Vista Pyramid

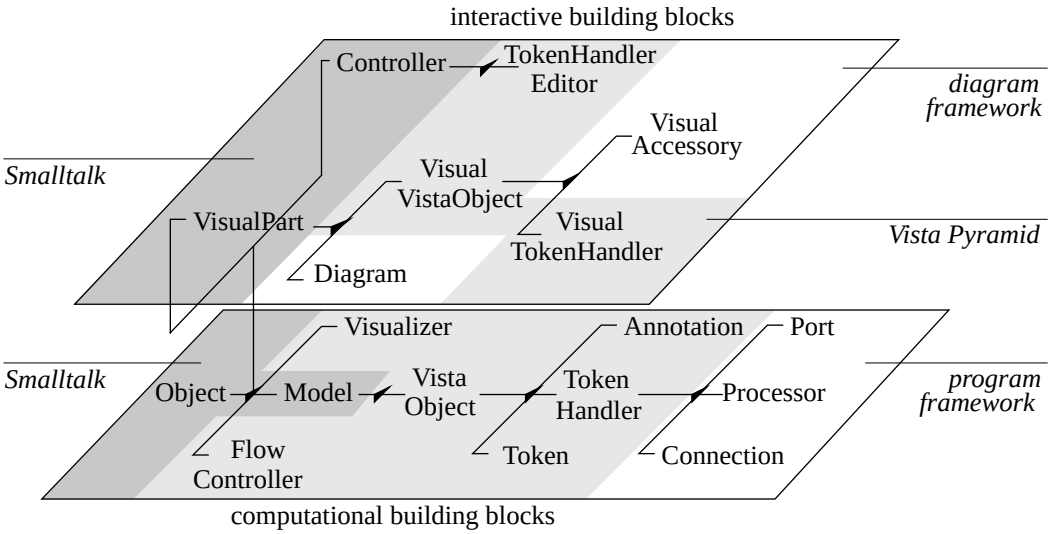

Figure 6. Class categories and layers

ment 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-button 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

A Vista application is internally composed of:

- *interactive building blocks*, which interact with the user during the construction process according to the semantics of their computational building blocks
- *visualizers*, which construct, animate, and analyze the interactive building blocks
- *computational building blocks*, which influence the control flow through a Vista application

From VisualWorks's point of view, Vista's computational building blocks constitute the domain model. A detailed description of the coupling of Vista to VisualWorks is given in [10].

A recurring object structure makes up the internal backbone of every Vista application. This object arrangement consists of a VisualTokenHandler, a Visualizer, a TokenHandler, a Processor, and an Annotation. The reference relations among these key classes look like a pyramid if they are arranged inside a three-dimensional space, so that classes having more to do with graphical representation tasks lie above classes that are more involved in computation tasks (see Figure 5). Since this pyramid combines Vista's most reused, abstract classes and therefore is an important part of the overall architecture, we will refer to this pyramid as the Vista Pyramid.

Class TokenHandler is the common part (here in the sense of superclass) of classes that model essential concepts of Vista's programming model, like processor, port, connection, and network. During the computation process token handlers pass tokens to each other. A token causes an operation to be performed in the receiving token handler according to a strategy specific to the receiver's state and behavior. These strategies are encoded in objects of class FlowController. Usually token handlers come in pairs with flow controllers. One can think of token handlers as syntactic elements, whereas flow controllers encapsulate the semantic, possibly context-sensitive operations of processors. These tightly coupled objects are called *program unit* for short if their separation is of no interest.

A token handler does not know how to visualize itself. This knowledge must be encoded by visualizers, which bridge program units and their interface building blocks. Invisible visualizers work at the interface between graphical and computational building blocks. Visualizers are responsible for constructing structured visual objects. For a token handler to have its state reflected at the user interface (e.g., become disabled), it requests that its visualizer appropriately update the graphical representation. A visualizer also initiates the animation of the token flow at the user interface. User interactions are routed to program units via objects of class TokenHandlerEditor. Actually, this class is just a specialized controller in the sense of the usual MVC concept [6]. Figure 6 reveals Vista's internal architecture in more detail.

The introduction of visualizers and flow controllers leads to a complete separation of the graphical layer from

the computational layer. This strict separation has several consequences:

- Different visualizers display visual token handlers with different geometric properties. Therefore, multiple, possibly different graphical representations, e.g., two-dimensional and three-dimensional diagram elements, may exist for one program unit.
- The graphical layer of the program itself is of no interest to the users of applications built with Vista. The separation enables the generation of program versions that are stripped of the interactive building blocks.
- By means of visualizers, different flow controllers can interpret visual token handlers and relations between them in different ways. Therefore, one program can react differently with regard to different program states or environments.

Currently, flow controllers and visualizers cannot be programmed visually. Until now, this design decision has not appeared to be a drawback. When, for instance, different logic is needed for a class modeling an essential concept of Vista's programming model, a new flow controller must be programmed textually. Considering that Vista supports a style of visual programming relying basically on assembly of well-understood simple processors by visual means in order to obtain processors at higher abstraction levels, it becomes obvious that new behavior for these basic elements rarely needs to be defined.

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 is 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 is needed in order to obtain better performance.

6 Summary

This paper has outlined concepts and architectural 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. Key concepts of Vista's internal architecture, especially the pyramid classes and the strict separation of the interactive and computational building blocks, have proven to be highly reusable.

References

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

[2] D. Harel, "Statecharts. A Visual Formalism for Complex Systems," in *Science of Computer Programming*, Vol. 8, No. 3, Jun. 1987, pp. 231-274.

[3] 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.

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

[5] T. D. Kimura. "Hyperflow: A Visual Programming Language for Pen Computers," in *Proceedings of the 1992 IEEE-Workshop on Visual Languages*, pages 125-139, 1992.

[6] G. E. Krasner, S. T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk 80," in *JOOP*, Vol.1, No.3., 1988.

[7] "LabVIEW: Laboratory Virtual Instrument Engineering Workbench," in *Byte*, Sept. 1986, pp. 84-92.

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

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

[10] S. Schiffer and J.H. Fröhlich, "Visual Programming and Software Engineering with Vista," to appear in M. Burnett, A. Goldberg, T. Lewis (eds), *Visual Object-Oriented Programming: Concepts and Environments*, Manning/Prentice-Hall, 1994.

[11] *Serius Programmer User's Guide*, Serius Corporation, 1992.

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

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