Object-Oriented Development of a Data Flow Visual Language System

Alex S. Fukunaga†, Takayuki D. Kimura††, Wolfgang Pree††

†Harvard University
Cambridge, Massachusetts 02138

††Department of Computer Science
Washington University
St. Louis, Missouri 63130

Abstract

This paper describes the object-oriented development of ProtoHyperflow, a data flow visual language. We demonstrate how object-oriented software construction principles can be used for the development of extensible and reusable building blocks for the development of data flow based visual languages.

1 Introduction

Object-oriented software development techniques make it possible to develop generic applications for specific domains. These application frameworks consist of a class library forming a frame that has to be customized for a specific application. User interface application frameworks like ET++ [13], MacApp [14], or AppKit [9], for example, provide a reusable, blank application that implements much of a given user interface look-and-feel standard. The programmer can concentrate on implementing the application-specific parts. Application frameworks are not limited to the construction of interactive, graphic-oriented user interfaces, and can be applied to any area of software systems.

Recently there has been a trend to use data flow as the underlying computational model for visual language implementations. As a result, many recent languages share common features, and visual programming languages seem to be reaching the point where they build upon previously existing languages. For example, several languages build on the concepts introduced by Show and Tell [5], including Data-Vis [4], Extended Show and Tell (ESTL) [8], and Hyperflow [6].

This trend towards a standardization of a class of visual languages motivated us to apply the application framework idea to this domain. With visual language design becoming an evolutionary process, the availability of an appropriate object-oriented application framework can tremendously reduce the implementation effort of such languages by eliminating the need to reinvent the wheel for each new languge. In this paper, we describe our first efforts in the endeavor to develop such a framework.

2 The language ProtoHyperflow (PHF)

ProtoHyperflow (PHF) is a data flow visual programming language which is a derivative subset of Hyperflow. While Hyperflow is designed as a visual language for a pen-based multimedia system, PHF is implemented on a traditional, mouse/CRT-based system using C++ and the user interface application framework ET++ [12, 3, 13, 1].

PHF is a general purpose visual language system consisting of an integrated editor and a data driven interpreter system. The following is an informal description of the PHF language. Most of the language constructs available in PHF appear in other data flow visual languages, thus forming the basis for factoring out commonalities into an application framework.

2.1 PHF syntax

The syntax of PHF—like that of many other data flow visual languages—consists of boxes and arrows, a box representing a process, and an arrow representing a data flow between processes. Boxes are called *vips* (visually interactive processes) in PHF, and arrows are called *connectors*.

Computation in PHF is carried out by a homogenous community of vips communicating with each other. Vips can be recursively nested.

The vip is the only unit of system decomposition in PHF, paralleling the design of LISP, in which lists are

1049-2615/93 $03.00 © 1993 IEEE

the only structure. This gives PHF the ability to treat programs as data objects, as described in Section 3.6. A vip consists of a *mailbox*, a *body*, and an optional *name*. A mailbox holds a discrete data object, such as an integer or string. The body is the semantic content (the implementation of the semantics) of a vip. The body of a vip can be a system defined PHF primitive (e.g., a '+' primitive which returns the sum of a vip's inputs), a reference to another vip (a call to a user-defined function), a nested ensemble of vips, or it may be empty. A vip may also have a name, which appears on the top left corner of the vip. Names are necessary when defining functions (see Section 2.5). A connector establishes dataflow between the two vips that it connects. A connector may also have a label. A PHF program is a directed acyclic graph, where the nodes are vips and the edges are connectors.

It is necessary to introduce some shorthand terminology here, in order to facilitate a more detailed discussion of the constructs used in PHF. We shall define 'vip X' to mean 'the vip with the name X, and 'connector X' to mean 'the connector with the label X'. Also, an empty vip shall be called a variable vip.

2.2 Data objects in PHF

Data objects which flow from box to box are another common property of data flow visual languages. The following data objects are currently implemented in PHF: integers, strings, signals, and vips. Strings in PHF are prefixed with a quote (') in order to prevent them from being evaluated as references to other vips. Signals are an enumerated data type which is either valid or invalid, and are used to denote the result of a predicate (such as = ,<>). The use of vips as data objects is sketched in Section 3.6. All data objects can be transmitted via mail (see below) and can be displayed in a variable (empty) vip.

2.3 Communication between vips

There are two modes of communication in PHF: *mailing* and *broadcasting*.

Mailing is communication of discrete data objects by dataflow across connectors. In mailing, the contents of the mailbox of the source vip is copied to the mailbox of the destination vip. This is the standard mode of communication between vips.

Broadcasting is a special mode of communication which involves no connectors. A broadcasting vip, which is denoted as a vip with a dotted border (see Figure 5) transmits the contents of its mailbox to all of the children of its parent -- its sibling vips. (See the description of the factorial function in Section 2.5 for an example of the usage of the broadcasting mechanism.)

2.4 PHF execution protocol

A PHF program is executed from its outermost vip. The execution mode implemented by PHF is, as with most current data flow visual languages, data driven (all vips which have input will execute).

A vip is executable exactly once, and will execute when it has the minimum number of valid inputs (input connectors on which the source's mailbox is ready to be transferred). The number of minimum inputs is a semantic property of a vip. For example, a variable vip (an empty vip) will execute as soon as it has one valid input (all other inputs will be ignored), while a + primitive (summation) will not execute until all of its inputs are valid. If a variable vip X has two input connectors with sources at vip Y and Z, then this results in a nondeterministic behavior, where the value transferred to X is the value of the source which is ready first (If Y is ready to transmit first, then X receives the value of Y, but if Z is ready to transmit first, then X receives the value of Z).

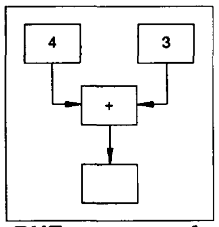

Figure 1: PHF program for addition

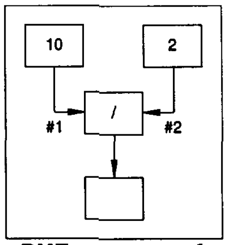

Figure 2: PHF program for division

2.5 PHF programming constructs

The following sections describe key programming constructs in PHF.

Primitive and user defined functions, binding rules

A vip may invoke a system defined primitive or a user defined function.

Figure 1 shows a PHF program which calculates the sum of 4 and 3. The + vip is a system defined primitive which returns the sum of all of its inputs.

Another type of function is one which involves parameter binding. For example, for a binary division operation (a division with two inputs), it is necessary to distinguish which of the operands is divided by the other. Thus, binding rules are necessary.

The binding rules in PHF are name based. In the case of system defined primitives, the parameters which need to be bound to input values are defined by the system as #1, #2, ... #n, where #1 is the first argument, #2 is the second argument, and so on. Input values are bound to these parameters by labeling the connectors which connect the input values and the primitive. Thus, in a binary division, the dividend must be bound with system parameter #1, and the divisor must be bound with system parameter #2, and the '/' primitive will return the value of #1/#2. Figure 2 shows a PHF program which calculates 10/2.

User defined functions are implemented by naming a vip, and then referencing that vip from another vip. Figure 3 shows the definition of Increment, a vip which takes one input parameter in vip X, and returns X+1. The @ vip is a system defined primitive which transfers the contents of its mailbox to its parent. Figure 4 shows CalcInc, a PHF program which calls the Increment function with input value 5. When CalcInc is executed, the 5 is mailed to the vip which calls Increment. Then a copy of the function Increment is created, with 5 bound to the vip X. The copy of the function vip is then executed. The result of the addition, 6, is sent out of the Increment function to its parent, which is the vip which calls Increment in CalcInc, and the result then flows to the variable vip at the bottom of the CalcInc vip.

T he binding of input values to unbound variable vip in the function is established by associating the labels of the connectors entering the function call vip with the names of the parameter vips of the function. Thus, in the example above, the empty variable vip X in Figure 3 is associated with the connector labeled X in Figure 4.

PHF functions have only one return value, which is mailed out via the @ vip. Note that it is not possible to have side effects in PHF, because of its data flow based nature.

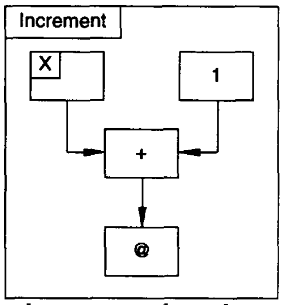

Figure 3: Increment function definition

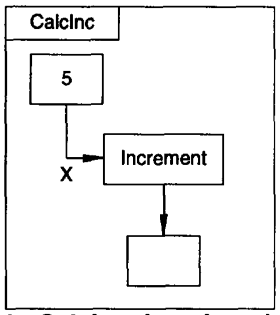

Figure 4: CalcInc function definition

Conditionals

Conditionals are implemented in PHF using the broadcasting mechanism (Section 2.3) and the signal data type (Section 2.2). If a vip receives an invalid signal, it is inactivated so that it does not execute. A broadcasting vip can prevent all of its sibling vips from executing by broadcasting an invalid signal. Thus, conditionals can be implemented by having multiple vips, among which only one is selected by invalidating all of the others.

The recursive implementation of the factorial function (!) demonstrates the usage of conditionals. As shown in Figure 5 this function takes one integer input, X and processes it as follows: if X=0 then return 1 else return X * (X -1)!. (X is assumed to be a positive integer).

The = and <> are PHF primitives which return valid or invalid depending on whether the 'equal' and 'not equal' predicate holds true for their inputs. If X=0, then the = predicate returns a valid signal, while the <> predicate returns an invalid signal, and a 1 is returned. However, if X <> 0, then X * (X -1)! is returned.

Note that PHF uses an asynchronous, parallel execution model, so a vip executes as soon as its inputs are ready. Thus, when making conditional statements, the conditionals must be mutually exclusive, or the results will be unpredictable (as mentioned above, the program

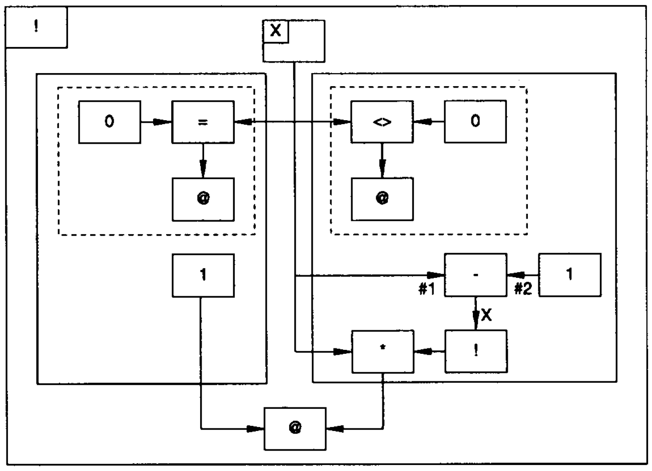

Figure 5: Factorial Function

will still be syntactically correct, but it will be nondeterministic).

3 PHF's object-oriented implementation

The principal software components of the PHF system are a direct-manipulation GUI (graphic user interface) editor with an integrated data driven interpreter system.

The graphical components of the PHF system have been developed in C++ using the application framework ET++ [12, 3, 13, 1]. Some details concerning the implementation effort of this component are presented in the Section 5. Since there are numerous publications on how to implement GUIs based on appropriate object-oriented application frameworks, we do not describe that PHF system component, and instead concentrate on the interpreter component.

In a manner similar to that with which ET++ provides software concepts for implementing GUIs, we developed classes that form a basic application framework for a data driven interpreter system for data flow visual languages.

We will first describe the principles of an application framework (short: framework) and then sketch the interpreter framework which resulted from the object-oriented PHF development. We take the reader's knowledge about the object-oriented concepts of inheritance, polymorphism and dynamic binding for granted.

3.1 Application frameworks

We first discuss some aspects of class libraries before defining the term application framework. Compared to conventional routine libraries, class libraries are hierarchical, with the most general class at the top of the hierarchy tree (if single inheritance is used). This hierarchical organization helps to reduce the complexity of a library. An important principle behind the design of a class hierarchy is that the common behavior of classes is factored out into their superclasses.

Classes which factor out common behavior of other classes typically contain some methods that cannot be implemented. Any class that contains one or more "empty" methods (i.e., methods with some kind of dummy implementation) is termed *abstract class*. It does not make sense to generate instances of them. Nevertheless abstract classes may also contain methods that can already be implemented in advance for all subclasses.

The most important aspect of abstract classes is that they form the basis of *extensible* and *reusable* software systems: it is possible to realize whole software systems using only abstract classes, i.e., the protocol supported by them (we define the term "*protocol* of a class" as all the methods and instance variables provided by a class). If subclasses of abstract classes are added to the class library, these software systems need not be changed. They also work with the objects of new subclasses of these abstract classes (on which other software systems are based), since these objects support at least the protocol (though implemented in a specific manner) defined in their (abstract) superclasses. The methods of abstract classes are dynamically bound, so that the corresponding methods of the objects which are instances of the new classes are called at run time.

For instance, a visual language interpreter component which can be implemented based on the protocol provided by the abstract classes VIPShape and DataObj will work with any objects generated out of subclasses of these abstract classes.

New subclasses of abstract classes can reuse all the code that was already implemented in their superclasses. Class libraries are called *application frameworks* if they apply the ideas presented above in order to provide a software system which is a generic application for a specific domain. Classes comprising the interpreter, together with all the abstract classes these components rely on, form a visual language interpreter application framework. Applications based on such an application framework are built by customizing its abstract and concrete classes. Thus, a given framework already anticipates much of an application's design which is reused in all applications based on the classes of that application framework. This implies not only a code reduction but also a standardization of that domain. As we have remarked above, the domain of data flow visual languages is becoming standardized, so we believe that the application of a framework to this domain is timely.

3.2 PHF's class library

A PHF program can be modeled abstractly as a community of vips interacting with each other by sending messages via the mailing and broadcasting mechanisms. This classifies objects in PHF into two categories: 1) the visually interacting processes, and 2) the data objects which are being passed from vip to vip. Another way to classify objects in PHF is to differentiate between visual objects and internal objects. The visual objects are the visual component of the vips themselves (the boxes) and the connectors. The internal objects are the implementations of the semantic content of the vips and the data objects.

Thus, we designed the abstract classes VIPShape, VIPContent, and DataObj, which represent the visual content, the semantic content, and the data objects. Each of these abstract classes and the interpreter system based on them shall be described in detail.

VIPShape is the abstract class for the visual objects in PHF (see Figure 6). Its concrete subclasses are the boxes (instances of the classes VIPRectShape and BroadcastVIPRectShape) and the connectors (instances of the class VIPConnector).

The abstract class VIPShape itself is a subclass of ET++'s abstract class VObject (visual object), which describes properties common to objects which have a visual representation (e.g., rendering, event handling, resizing, moving). VObject, together with other abstract and concrete classes of the GUI application framework ET++ saved a significant amount of work in implementing PHF's visual components, i.e., the language editor and the visual aspects of the interpreter.

The VIPRectShape subclass represents a vip's visual component. Its members include its name, lists of input and output connectors, the mailbox which contains a data object, and a semantic content (a subclass of VIPContent). In addition, each vip has a parent vip and a list of contained vips as instance variables, reflecting the recursively nested nature of the vips. Methods defined for VIPRectShape include methods to manipulate the members listed above, as well as methods which handle the visual aspects of the vip.

VIPConnector represents connectors between vips. The connector has as its members the start and end vips of the connector, and its name.

```
VIPShape
 |
 |-- VIPRectShape
 |     └-- BroadcastVIPRectShape
 └-- VIPConnector
```

Figure 6: VIPShape class hierarchy

3.3 Data objects

The abstract class DataObj (see Figure 7) represents a data object in PHF. These data objects are the message packets which are sent from vip to vip via mail/broadcast. DataObj's protocol provides especially the following two methods: GetValue allows the actual data (data of type int, string, enumerated type Signal, or a pointer to a VIPRectShape) to be extracted from the packet, and HFDataObjToStr returns a standard representation of the data (a string) which can be displayed in the visual environment.

The concrete subclasses DataObj_Int, DataObj_Signal, and DataObj_String contain instances of the actual data objects and override the GetValue and HFDataObjToStr methods.

New data types can be easily added by defining a new concrete subclass of DataObj and overriding the GetValue and HFDataObjToStr methods (see Section 3.6).

```
DataObj
 |
 |-- DataObj_Int
 |-- DataObj_Signal
 └-- DataObj_String
```

Figure 7: DataObj class hierarchy

3.4 Semantic components

The abstract class VIPContent represents the semantic content of a vip. It has a member pointer to the VIPRectShape it belongs to, and defines the dynamically bound method Execute, which every subclass of VIPContent must override.

The subclasses of VIPContent mirror the language definition for the semantic content of a vip. The subclasses include: VarVIP (variable vip), EnsembleVIP (nested ensemble of vips), FuncVIP (a reference to another vip - a function call), and PrimOpVIP (PHF primitive).

PrimOpVIP is an abstract class encompassing all system defined primitives. It defines generalized methods for obtaining message packets from input connectors (CheckInputs, GetMsg) and updating the value in the vip's mailbox (SetResult). Subclasses of PrimOpVIP must override the ProcessInput method, which processes the inputs and returns as a result a DataObj instance. For example, the ProcessInput method for the '-' primitive matches the two operands with the input for the connector labeled with #1 and #2, and returns a DataObj_Int whose value is #1 - #2.

Extensions to the language semantics are easily accomplished by adding concrete subclasses of VIPContent for major language constructs (e.g.. iteration) and of PrimOpVIP for primitive operations (e.g., square root) .

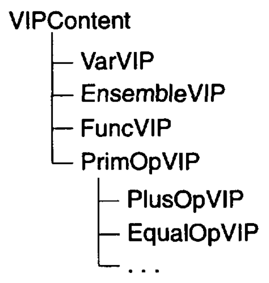

Figure 8: VIPContent class hierarchy

3.5 Interpreter execution protocol

A PHF program is executed by 1) making a copy of the outermost vip of the program by using the DeepClone method of ET++'s root class Object (makes an identical copy of an object, including all objects referenced as instance variables), 2) opening a new window for the copy (the Execution Window) and 3) invoking the Execute method of the copy.

The execution mode implemented by PHF is, as with most current data flow visual languages, data driven, in which any vip which has input is executed (to our knowledge, only VPL [7] has implemented demand driven execution).

An ensemble of vips is executed in the following way: A vip is executable exactly once, and will execute when it has the minimum number of valid inputs (input connectors on which the source's mailbox is ready to be transferred). The number of minimum inputs is a semantic property of a vip. For example, a variable vip will execute as soon as it has one valid input (all other inputs will be ignored), while a + primitive (summation) will not execute until all of its inputs are valid.

This is implemented as follows: A queue of all vips whose valid flags are set is constructed (the valid flag may be reset if the vip has received an invalid message). Empty VarVIPs (for which user input is required) are moved to the front of the queue, followed by vips with broadcast borders. If a vip can not execute because the minimum number of its valid inputs are not ready, then it is inserted at the end of the queue.

3.6 Extensibility of PHF's core framework: adding higher order functions

The design of the PHF interpreter around the abstract classes VIPContent, and DataObj made possible the construction of a core framework which could be easily extended. New data types are added by adding subclasses of DataObj, and semantic extenstions to the language are made by adding subclasses of VIPContent. Since the core language operates on the abstract classes VIPContent and PHFDataObj, it is possible to make significant language extensions without modifying the core of the language. The following example demonstrates this point.

To the basic PHF language as described above, we added the capability to handle higher order functions and an extensive set of primitives to manipulate functions as data objects, similar to the generality with which LISP treats program and data. A full description of these features is beyond the scope of this paper (see [2]), but their implementation demonstrates the power of object oriented design.

Using these language constructs, for example, we can define a higher order function ReverseOp which takes a function F and three vips X, Y, and Z as input (we do not describe ReverseOp's realization). A use of that function might be to reverse two arbitrary connections in a self modifying PHF program (see Figure 9).

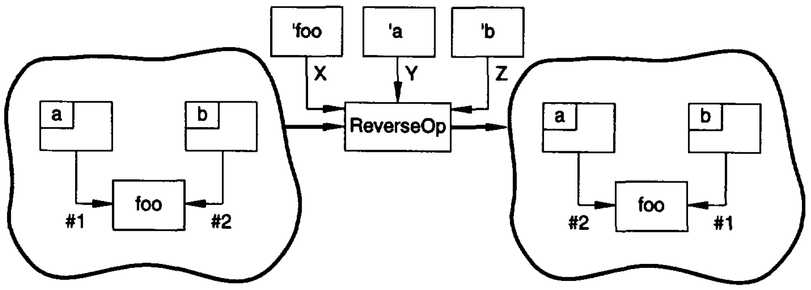

Figure 9: Usage of ReverseOp

In order to implement the manipulation of vips as data objects, we first have to add a new data type for quoted vips (vips which are treated as data). Thus, we create DataObj_VIP, a subclass of the abstract class DataObj.

- DataObj
  - DataObj_Int
  - DataObj_Signal
  - DataObj_String
  - **DataObj_VIP**

Figure 10: Extended DataObj class hierarchy

A new construct, Apply, which takes a quoted vip as input, executes it, and returns the result, is implemented as a subclass of the abstract class VIPContent.

- VIPContent
  - VarVIP
  - EnsembleVIP
  - FuncVIP
  - **Apply**
  - PrimOpVIP
    - PlusOpVIP
    - EqualOpVIP
    - **InsertVIPOp**
    - **ConnectVIPOp**
    - **ExtractVIPOp**
    - ...

Figure 11: Extended VIPContent class hierarchy

The complement of this construct, Quote, which transforms a vip into a data object, is implemented by adding a subclass to VIPShape, i.e., QuotedVIPRectShape, which inherits all of the properties of VIPRectShape, but has a thickened border to denote that it contains a quoted vip (Figure 11).

- VIPShape
  - VIPRectShape
    - BroadcastVIPRectShape
    - **QuotedVIPRectShape**
  - VIPConnector

Figure 12: Extended VIPShape class hierarchy

Finally, the primitives necessary to fully manipulate the vip data structure, including primitives to insert, connect, extract, and disconnect vips are implemented as subclasses of the abstract class PrimOpVIP (Figure 11).

Although this is certainly a nontrivial extension to the language, the only modifications necessary are as described above. No modifications were necessary to the core application framework. It should be noted that these extensions were not envisioned during the original design of the PHF class library. The fact that we were able to easily modify the language to incorporate these extensions is a testament to the effectiveness of the object-oriented design methodology.

4 Implementation Details

The current version of the PHF system supports all of the language features described above (including support for higher order functions) in an integrated editor/interpreter environment. It has a Motif-GUI interface provided by the ET++ framework and runs under UNIX on several hardware platforms (e.g., Sun SPARCstation, IBM RS6000, HP 9000/700, DEC RISC, i486). Since the GUI component of PHF only uses the protocols provided by ET++, it is a trivial matter to port PHF to any platform supported by ET++.

The size of the C++ code that had to be written (i.e., excluding the classes reused from the ET++ application framework) breaks down roughly as follows:

	Lines of Code
PHF-Editor	1900
PHF-Interpreter	2000
Total	**3900**

It is important to note that constructing the portable GUI components, which is usually considered the primary bottleneck in developing a visual language, took a relatively short time (approximately 2 person weeks), since ET++ provides extensive support for developing these GUI components independent of the hardware platform. This number does not include the time it took to become familiar with object-oriented programming and the ET++ class library.

5 Conclusion

We have described the implementation of the simple data flow based visual programming language ProtoHyperflow, focusing on the developed application framework components for data flow visual languages. The object-oriented paradigm (encapsulation, inheritance, polymorphism and dynamic binding) encourages the developing of extensible systems and software reuse. A precondition to being able to apply the object-oriented paradigm is to find good abstractions (abstract classes) that form the basis of building application frameworks for specific domains. To this end, we developed PHF, a minimal data flow based visual programming language.

PHF is an application framework which provides a minimal set of features associated with a data flow based visual programming language. We have shown that by building on this framework, it is possible to develop more complex languages, as we demonstrated by the addition of higher order functions.

Our experience with building PHF on top of the ET++ framework has also shown that it is possible to significantly decrease development time by taking advantage of a previously existing general purpose application framework. In addition, PHF has the additional advantage that it is portable across all platforms that ET++ is compatible with, since PHF only uses the protocols provided by ET++, and does not interact directly with the underlying windowing system (i.e. SunWindows, X Windows). This makes PHF considerably independent of its implementation platform.

The PHF application framework is considered to be a starting point for future research in the area of domain specific application frameworks for visual langauges. We expect that despite of known problems of object-oriented software development (as described, for instance, in [10] and [11]) qualitative and quantitative improvements in the development of visual language systems are possible by applying the application framework approach.

References

[1] Eggenschwiler T, Gamma E. ET++ Swaps Manager: Using Object Technology in the Financial Engineering Domain; OOPSLA'92, Special Issue of SIGPLAN Notices, Vol. 27, No. 10, 1992.

[2] Fukunaga A, Pree W, Kimura TD. Functions as Data Objects in a Data Flow Based Visual Programming Language; in Proceedings of the ACM Computer Science Conference, Indianapolis, IN, 1993.

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

[4] Hils D. A Visual Programming Language For Visualization of Scientific Data. Ph.D. Thesis, Dept. of Computer Science, University of Illinois, Urbana, 1992

[5] Kimura TD, Choi JW, Mack JM. A Visual Language for Keyboardless Programming; Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, 1986.

[6] Kimura TD. Hyperflow: A Visual Programming Language for Pen Computers; in Proceedings of IEEE Workshop on Visual Languages, Seattle, Washington, 1992.

[7] Lau-Kee D, Billyard A, Faichney R, Kozato Y, Otto P, Smith M, Wilkinson I. VPL: An Active, Declarative Visual Programming System; in Proceedings of IEEE Workshop on Visual Languages, Kobe, Japan, 1991.

[8] Najork M, Golin E. Enhancing Show-and-Tell with a polymorphic type system and higher order functions; in Proceedings of IEEE Workshop on Visual Languages, Skokie, Illinois, 1990.

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

[10] Pree W, Pomberger G. Object-Oriented Versus Conventional Software Development: A Comparative Case Study; EuroMicro '92 Conference, Paris, France, 1992.

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

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

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

[14] Wilson DA , Rosenstein LS, Shafer D. Programming with MacApp; Addison-Wesley, 1990.