Objects + Scripts = Applications[1]

Oscar Nierstrasz
Dennis Tsichritzis
Vicki de Mey
Marc Stadelmann

University of Geneva[2]

Abstract

We argue that object-oriented *programming* is only half of the story. Flexible, configurable applications can be viewed as collections of reusable objects conforming to standard interfaces together with *scripts* that bind these objects together to perform certain tasks. Scripting encourages a *component-oriented* approach to application development in which frameworks of reusable components (objects and scripts) are carefully engineered in an evolutionary software life-cycle, with the ultimate goal of supporting application construction largely from these interchangeable, prefabricated components. The activity of constructing the running application is supported by a *visual scripting tool* that replaces the textual paradigm of programming with a visual paradigm of direct manipulation and editing of both application and user interface components. We present scripting by means of some simple examples, and we describe a prototype of a visual scripting tool, called *Vista*. We conclude with some observations on the environmental support needed to support a component-oriented software life-cycle, using as a specific example the application development environment of *ITHACA*, a large European project of which Vista is a part.

1. Introduction

Although object-oriented technologies are gradually being recognized as a critical contribution to alleviating the "software crisis", it is also apparent that it can be quite difficult to achieve more than marginal benefits simply by switching to object-oriented programming languages. An arbitrary application may be just as difficult to program with an object-oriented language as with a standard block-structured imperative language—perhaps even more so—and the principal benefits of such an exercise, namely a cleaner and more manageable decomposition of the code, are highly sensitive to the results of the object-oriented analysis and design phases. There is no guarantee that any of the software developed will be truly reusable or that it will survive radical changes in requirements. There is not even any guarantee that an application developer will be able to re-use any non-trivial, previously developed classes in the implementation of a new application.

We claim that this is so because object-oriented languages are still largely perceived as a *programming* technology rather than as a software component production technology. This, in turn, is because we still largely approach software development as a labour-intensive craft rather than as a capital-intensive engineering discipline [5][22][26]. Rather than expending all our effort and creativity on developing individual applications, we should be investing more effort and

1. In *Proceedings, Esprit 1991 Conference*, Kluwer Academic Publishers, 1991, pp. 534-552.

2. Authors' address: Centre Universitaire d'Informatique, 12 rue du Lac, CH-1207 Geneva, Switzerland. E-mail: {oscar,dt,vicki,marc}@cui.unige.ch. Tel: +41 (22) 787.65.80. Fax: +41 (22) 735.39.05.

creativity into the development of standard components, interfaces and tools. The vast majority of applications should be seen as customized assemblages of largely standard parts. Such a scenario, however, requires a shift in our attitudes from short-term, single project software life-cycles to long-term, evolutionary life-cycles that accommodate frameworks that will be used for many projects to come. A software industrial revolution necessarily entails a revolution in the *economics* of software production. We shall not concern ourselves here with the problem of how to bring about such a revolution; we shall confine ourselves to some of the technical problems involved.

We would like to be able to use object-oriented technology as a means to realizing our scenario for component-oriented software development. There are three main technical obstacles:

1. Top-down strategies for requirements analysis, specification and design are unlikely to arrive at any reusable components at the implementation stage.
2. Object-oriented languages typically provide a very limited binding technology for composing software at the level of expressions and statements [5]. In order to compose existing objects one must *program* some new objects.
3. Software composition is made *more* difficult by the ability to define rich interfaces to objects. The most successful examples of reusable components rely on the existence of fairly simple, standard interfaces [27].

The first problem is essentially a methodological one, but is directly addressed by the development of frameworks [27], which package standard software solutions in terms of collections of composable software parts. It is helpful to view application development as a *sales* activity, in which the developer is *negotiating* to sell an available, customizable product, rather than trying to build an entirely new product. Frameworks, then, correspond to product lines, which are developed by R&D departments, not sales departments. By separating these two activities, product reliability can be improved and costs lowered.

To address the second problem, we propose *scripting* as a binding technology for object-oriented languages: a script introduces and binds a set of objects—or, more generally, a set of software components—that will then collaborate to solve a particular problem.

The third problem is addressed by the notion of a *scripting model*, which defines a set of standard interfaces for the components of a particular framework and specifies which components are plug-compatible and how exactly the binding is achieved. For example, a user interface scripting model is typically event-based, and defines what kind of user interface components exist and how they may be attached to applications. A scripting model for Unix commands and files, on the other hand, is stream-based, and defines how sources, filters and sinks may be "piped" together. New components developed for an existing framework must conform to its scripting model or their reusability will be impaired (as is the case with Unix commands that do not use the standard input or output streams). Scripts simply specify the binding between such components.

Just as a successful business requires an infrastructure and a support environment for marketing and sales, component-oriented software development can only succeed if the proper tools

for managing, finding and composing software are available [13]. A *visual scripting tool* is an interactive tool for composing and editing visually presented software components. It supports an application developer in the task of constructing both the internal behaviour and the user interface of a running application from standard software components.

In §2 we shall introduce scripts syntactically and discuss their key properties. In §3 we shall illustrate the paradigm of scripting by means of an example from the domain of office systems. We shall then discuss in more detail the notion of scripting models and how they assign a semantic interpretation to scripts. In §5 we will describe *Vista*, a prototype of a visual scripting tool and in §6 we will briefly discuss some related work. Finally, in §7, we shall discuss the role of Vista within the application development environment of ITHACA, a large Technology Integration Project of the European Community's Esprit programme.

2. Scripts

Scripts provide the "glue" that binds together plug-compatible software components. A software component is any piece of software that imports or exports services. Objects are software components because they both export services—namely their visible operations—and import them, from external "acquaintances" of which they are clients. Classes are also software components, as they export the service of being able to stamp out object instances and they import services from their superclasses. Components are interesting for software composition (1) if the binding of services to clients is delayed, and (2) if the available services are standardized, i.e., if "plug-compatibility" between clients and services is defined. These two properties can then be exploited in a script that specifies bindings between plug-compatible components.

For the moment we can consider scripts as purely syntactic entities. Similarly, we need know nothing more about the services associated to a component than the names by which they may be bound, that is, the component's *ports*. There are *input ports*, each representing a set of available services or properties (i.e., where requests can be received), and *output ports* representing services required from other components (i.e., from which requests will be issued). We will use a graphical notation in which components are represented by rectangles and input and output ports by squares on the perimeter of a component, respectively inside or outside the rectangle (see Figure 1)

Figure 1 A component with one input and two output ports

A script specifies bindings between a set of components. In Figure 2 we see this represented graphically by links between the ports of components. A link between the output port of one component and an input port of another means that the first component can use services provided by the second.

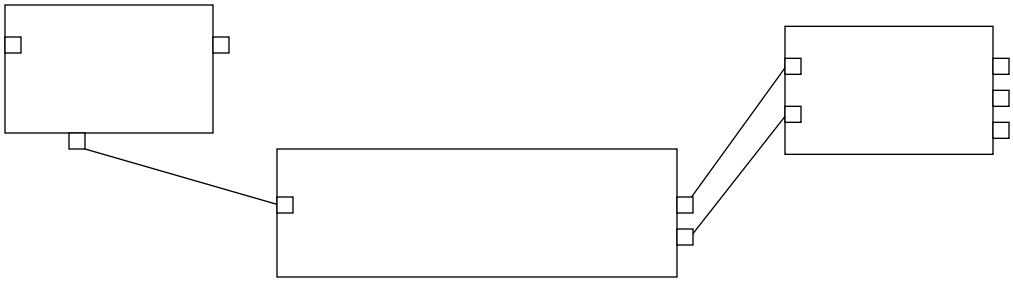


Figure 2 Linking components in a script

Aside from the restriction that output ports can only be connected to input ports, we have not yet introduced any constraints over the syntax of the scripts we can compose in this way. Are there different kinds of ports and links? May we connect multiple inputs to outputs, or vice-versa? Can components have any number of input or output ports? May the graph of a script contain cycles? These questions cannot be answered in absolute terms, but must be considered on the basis of the kinds of ports and links needed for a particular component framework. Scriptable components conform to a *scripting model* (§4) that formalizes both the syntax of scripts and their interpretation. Let us consider the kinds of syntactic rules we would like to impose on scripts:

One may define different *types* of ports, each representing a different set of services. Only *compatible* input and output ports may be linked. Since one type of port may represent a *subset* of services provided by another, standard polymorphism rules may apply.

One may *combine* ports in order to combine sets of services. It is also possible to combine input and output ports, in which case linking would imply a two-way client/server relationship (an example will be given later).

Since input ports represent availability of services, they may normally be linked to multiple output ports, that is, to multiple clients. Ports that permit multiple links are called *multi-ports*. Certain services, however, will only work for a unique client, in which case one may restrict an input port to be *singular*. Since output ports represent the use of service, they will often be singular. Output multi-ports can be very useful, however, for components that iterate over a *set* of service providers.

Binding of ports may be either *optional* or *obligatory*. Output ports typically must be bound to complete a component's behaviour, whereas input ports generally represent availability of services, so may be left unbound. One may also wish to associate *default bindings* for ports.

Certain linking constraints cannot be enforced by considering purely local connections. Such constraints are typically expressed by forbidding certain kinds of cycles in the graph of a script. Such cycles may imply the possibility of a deadlock (e.g., two office procedures competing for the same set of electronic forms), an infinite execution loop (e.g., inside a spreadsheet), or a recursive containment relationship (e.g., a document containing itself). In other cases, however, cycles may be explicitly required, e.g., by a scripting model that supports bidirectional client/server relationships.

There are two further crucial properties of scripts. The first is that *scripts are also components*. One may encapsulate a script as a component in order to reuse it in future scripts (Figure

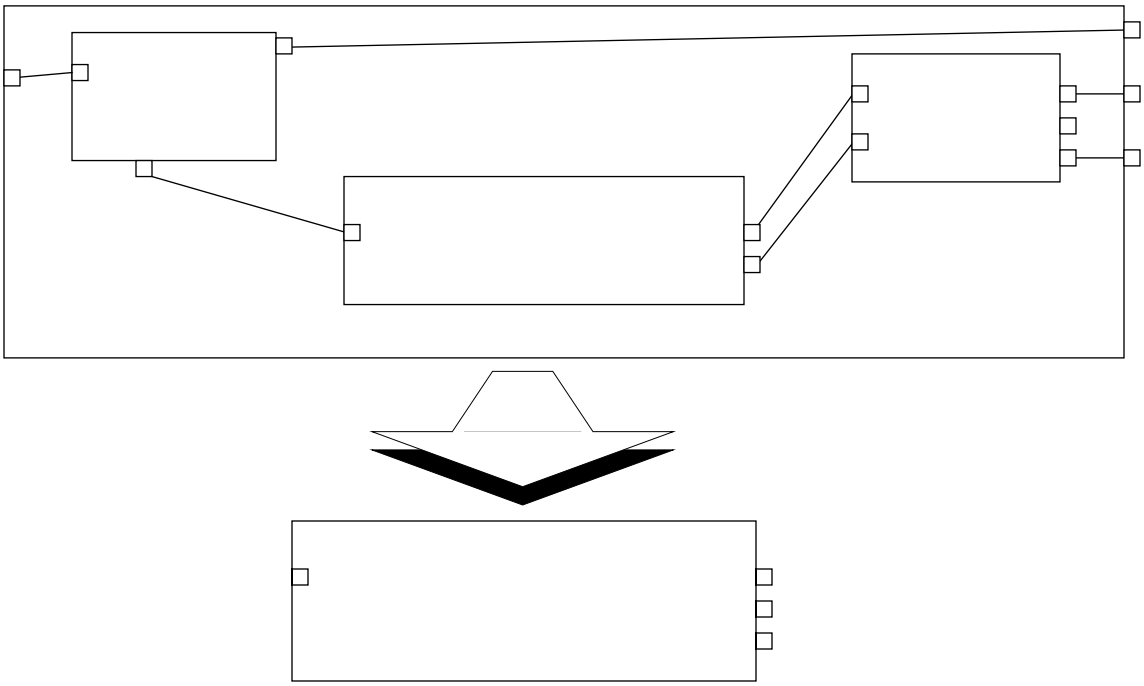


Figure 3 Encapsulating a script as a component

3). A script as a component encapsulates a set of partially bound components and determines which ports will be visible to surrounding scripts. Note that the “import/export” links of an encapsulated script serve only to rename ports, not to bind them—input ports of the script are linked to input ports of its components, and similarly for output ports. A script as component may also contain “holes” representing components to be filled in later (in this case the port associated with such a hole represents *all* the services of the missing component). Container objects are good examples (not just queues, stacks and bags, but also mailboxes, folders and spreadsheets).

The second important property is that scripts (and components) may have *multiple views.* Not only can one choose between the encapsulated and the expanded views of scripts, but one is free to vary the way in which different types of components, ports and links are visually presented. This is the key to a visual scripting tool which supports the interactive specification and interpretation of scripts: components best represented as graphs can be seen as graphs, but components that have a “natural visualization”, such as electronic forms or user interface components, can be viewed as such. Similarly “linking” can be thought of as connecting components with lines or arrows, but it may also be viewed as gluing or positioning components on a surface, plugging together pieces with compatible interfaces or simply selecting appropriate values from a menu or a property sheet. By properly exploiting the potential for multiple views of scripts, we can (1) support the *direct manipulation* of applications, and (2) *control complexity* by reducing the need for textual names and by selectively displaying only those aspects of an application of current interest.

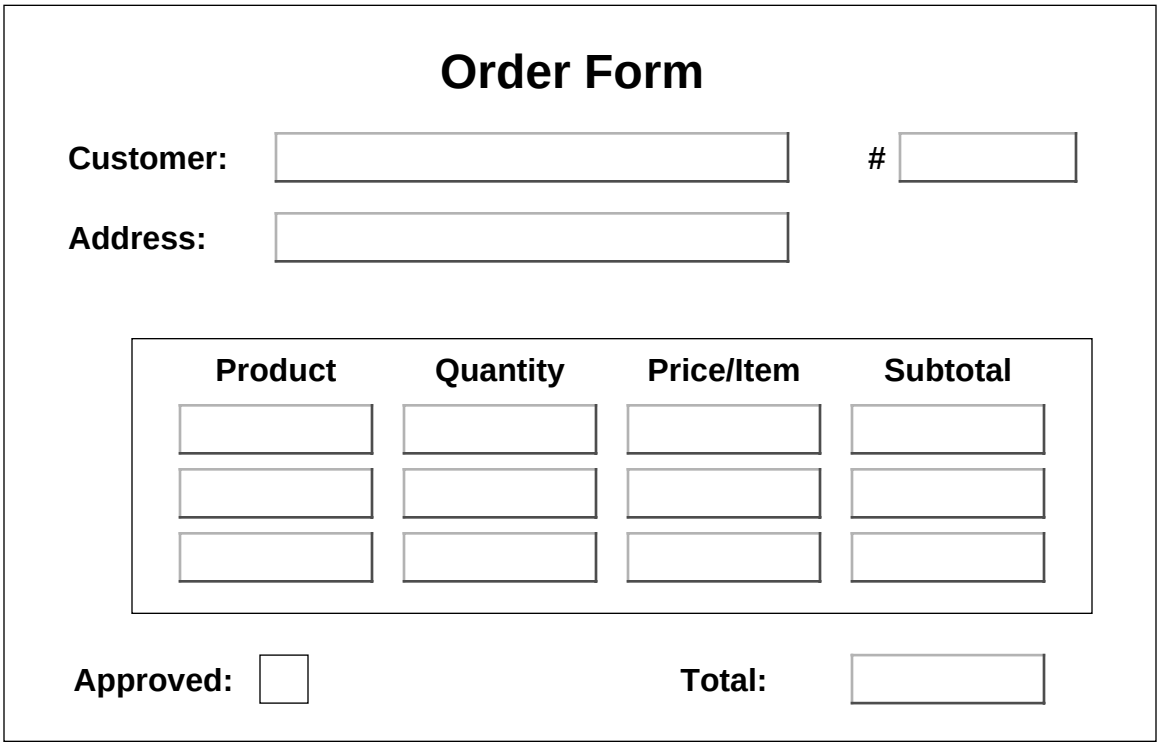


Figure 4 An active form

3. Scripting Active Forms

In a sense, all programming languages obey some scripting model, but typically the level of scripting is very low: only expressions and statements can be plugged together to form new expressions and statements. Scripting, as opposed to programming, becomes interesting the moment one identifies a well-understood application domain in which many applications have similar characteristics that can be factored out as software components. The area of Office Information Systems contains many suitable examples [18]. In fact, electronic spreadsheets, hypertext systems and Fourth Generation Languages can all be seen as examples of scripting tools for specialized application domains.

A prime example of a suitable scripting domain is that of electronic forms. Forms are pervasive in office systems, are well-understood, and are readily decomposed into standard components. As an example, we shall look at how one may script *active forms*. A passive form differs from an active form in that the former functions purely as medium for storing structured information whereas the latter has an associated *behaviour*. For example, by filling in a field of an active form, one may cause a side effect in another field. We make this distinction because we wish to emphasize that scripting is useful not only for building user interfaces, such as the appearance of passive forms, but for constructing the underlying applications as well, such as the associated behaviour of an active form.

Consider the order form shown in Figure 4. At the very least we would like to be able to script the appearance of such a form. At this level we can identify three different kinds of components: form *surfaces*, constant *labels* and modifiable *fields*. Scripting in this sense is essential-

ly a matter of gluing labels and fields to a surface. A form is a surface that *owns* a number of labels and fields, each of which has some associated position on the surface. The scripting model is very simple so far, in that the *owns* relationship is the only kind of link defined, and implies nothing more than the ability of the surface to display its parts. (The actual positioning of the parts on the surface may be static, or may be dynamically determined by means of graphical constraints.) Fields come with their own behaviour that allows users to interact with them.

There are two behavioural aspects of active forms that we would like to script in addition to their appearance. First, fields have an associated behaviour which can be quite rich [12] [24]. Second, forms themselves may support different views depending either on their internal state or on the context in which they appear (e.g., explicit user requests to present a different view, or implicit constraints determined by the capabilities of the current user).

Let us first consider the various kinds of fields present: the # field is a unique identifier for the form which is initialized at form creation time but which may not be modified. The **customer** and **address** fields are plain, modifiable text fields. (Though in a more realistic example, they would likely serve as an interface to a database of customer records.) The **product** and **quantity** fields are also modifiable text and integer fields, but they should generate (1) a query that retrieves the price per item and (2) a computation that produces the subtotal. Presumably one may order more than just three items, which implies that the list of products ordered is actually a *subcomponent* of the form, a container for a sequence of product/quantity/price/subtotal entries, each of which can be seen as a subform. The **total** field is an unmodifiable integer field whose computation is triggered by modifications in the list of order entries. Finally, checking the **approval** box means that the form can no longer be modified, that is, it implies a side-effect on the form causing it to become a “frozen” view of its contents.

We show how this behaviour might be scripted in Figure 5. The order form script consists of two surfaces visible to the end user representing the unfrozen and frozen views of the form, a contents surface where the field behaviour is scripted, and two external components, the “factory” for unique identifiers, and the product entry script. The fields of the two visible surfaces are essentially views on the underlying contents fields (links are implicit). The view fields of the first surface will forward modifications to the underlying contents, but those of the second surface are all “frozen” and ignore attempts to modify values. The approval field triggers the transition between from the unfrozen surface to the frozen one. The behaviour of order entries is separately scripted and is linked to the container for order entries in the contents surface.

Although we have not explained in detail the kinds of ports, links and components of this informal example, the principle ideas of scripting should be apparent: visualization and direct manipulation of components, scripting of both user interface and behaviour, standard kinds of ports and links, multiple views, and using scripts as components.

4. Scripting Models

So far we have concentrated on essentially *syntactic* aspects of scripts: their visual representations and rules for composing them. The interpretation of scripts, that is their *semantics*, is equal-

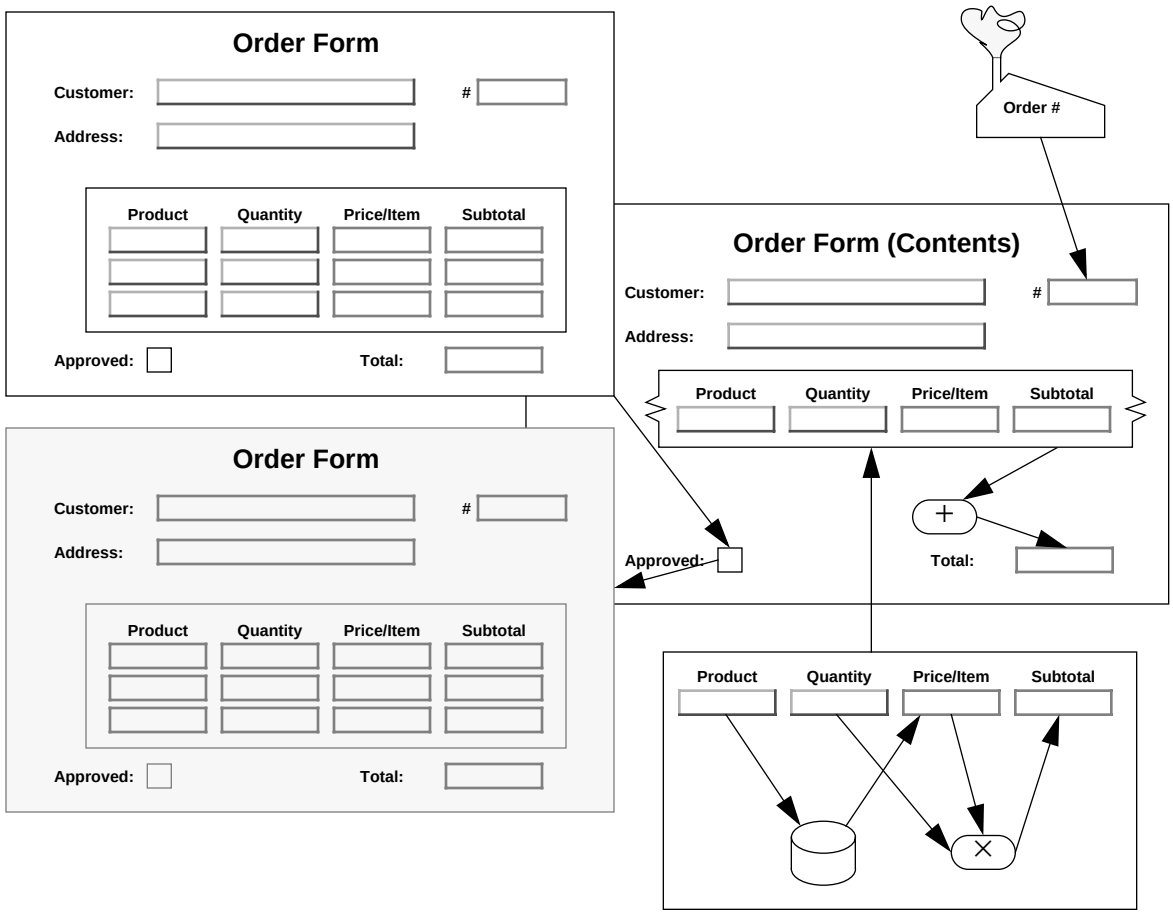


Figure 5 Scripting an active form

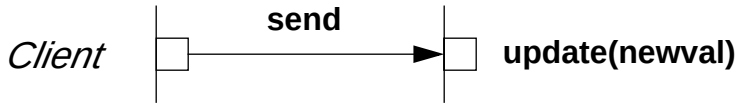
ly important, as this is where the connection between scripts and objects is established. Both syntax and semantics must be formalized in the scripting model for a component framework.

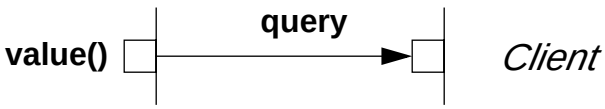
The scripting model also acts as a bridge between the application developer and the component developer. Links have, in effect, *two* semantics: one for the interpretation of the script, and another for the components that make use of these links. Consider, for example, the link between a view field on a visible surface of an active form and the corresponding text field of the form contents. To interpret such a link, it suffices to "introduce" the view field to the text field that it is connected to. There are many possible ways such a connection may be established, one of which is that the view field be an object with a "connect" operation that can be called when the connection is made. A scripting tool can guarantee that this connection only be made between plug-compatible fields. In this case, presumably, the view field expects the component to which it is connected to have operations that allow its value to be retrieved and to be updated. Note, however, that the interpretation of the script is strictly limited to managing the connections. It is the business of the component itself to decide when and how to make use of the connection. A scripting tool, then, functions like a corporate lawyer setting up standard contracts between standard kinds of customers and clients. It oversees and authorizes the signing of the contracts, but it is not concerned with their execution.

Scripts may have multiple interpretations depending on how they are to be used. Although the interpretation of a script is limited to establishing connections, this may be done in many dif-

ferent ways. For example, if an active form designed by scripting is to be directly interpreted, this implies that the binding between components be very loose. If, on the other hand, we wish to generate an object class from an active form script, then we might "compile away" the bindings to obtain a more efficient implementation. Differences in interpretation may also be due to different visualization requirements, for example, whether the script is to be executed as an end-user would see it, or in a special "debug" mode that exposes its implementation.

A basic scripting model for active forms is quite simple, requiring only two kinds of links in addition to the *owns* link: a *send* link, which allows one field to update another, and a *query* link, which allows a field to obtain the current value associated with another. We can visualize these links as follows:

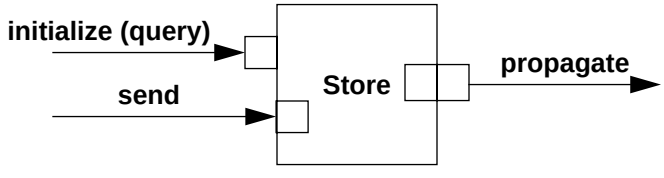




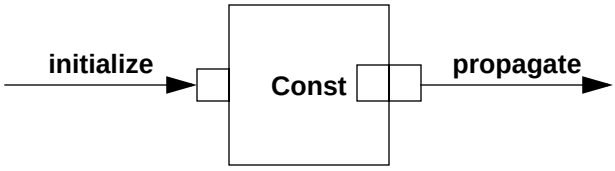
In both of these figures the *client* is the component with the output port and the service is provided by the component with the input port. The services provided are the **update()** and **value()** operations. The *direction* of the links is intended to indicate the direction of information flow, not which component initiates the exchange of information.

The *meaning* of these connections is that the client component will be able to use the indicated operations of the other component it is connected to. Note that in both cases the client is free to use the connection at any time. The scripting model only states what a connection entails, not when it will be used.

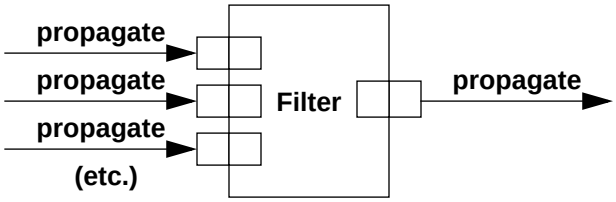
From these basic types of ports and links we can generate some of the others we will need. A *store* is a kind of component that initializes itself by a query link, can be updated, and can *propagate* its value to other components. An *initialize* link is a query link that will be used only once, when a component is instantiated. Note that it is the store that *requests* initialization, hence an output port is required. The send link allows the store to be updated. The *propagate* link is a combination of a send link *and* a query link, and so combines an input port and an output port. The propagate link is used by the component to update any connected components whenever it changes state, and it also allows the connected components to query the current state of the store at any time.



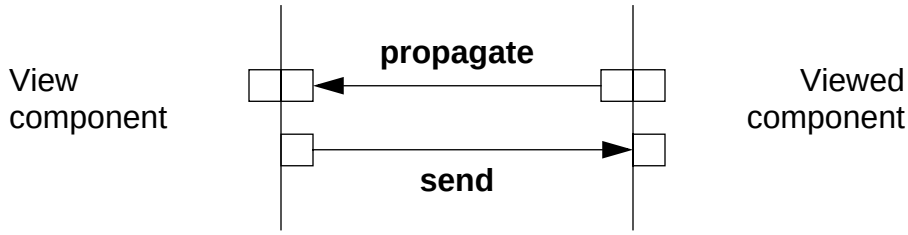
A *constant* is even simpler, since it cannot be updated:



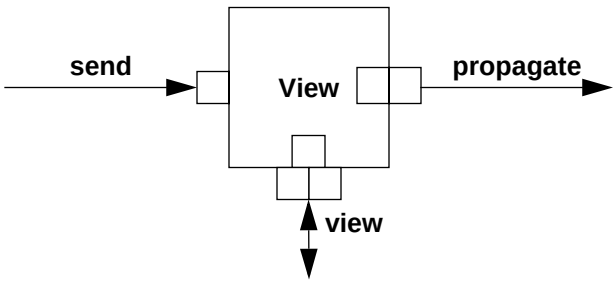
A *filter* is a component with no memory that computes a value from a number of parameters. It can be used either on a demand basis, or by triggering (hence it supports propagate links rather than just query links):



Finally, a *view* field makes use of another kind of composite link called a *view* link. The



view component can both ask for the current value of the viewed component and be notified of updates. (This combination is simply a propagate link towards the view.) In addition, The view component will pass on any updates it receives from other sources. A view component then, accepts updates (from the user or another component), synchronizes with another component by means of a view link, and propagates any changes onward.



With these generic component types, we can classify most of the fields we encountered in our active form example. The customer and address fields are stores and the identifier is a constant. The arithmetic functions and the fields that display them are filters. The fields of the view surfaces are view components (in the case of modifiable fields) and filters (the frozen ones).

The missing links belong to the approval field, which must be capable of communicating between surfaces, and to the order entry container, which must know the kinds of components it has to manage. Each of these links represents the binding of another set of services, which we will not describe here.

5. Vista

We have implemented a prototype of a visual scripting tool for object-oriented applications, called *Vista* [7][8][19]. Vista is an object-oriented successor to VST, an earlier prototype based on a Unix scripting model [23]. The current version of Vista supports the selection of components by means of a menu, the graphical linking of components, interactive execution of scripts, saving of scripts and the use of scripts as components within other scripts.

A small set of components has been implemented for demonstration purposes: text fields, labels, toggle and arrow buttons, slider, system date, "orchestra", simple arithmetic calculator, simple database of records and document. The scripting model presently supported is essentially a dataflow model: components make values available on their output ports and these values propagate to the input ports of connected components. In effect, the only kind of link presently available is the *send* link described earlier. In general, though, links may represent arbitrary sets of services associated with a particular scripting model. In these cases we plan to use the built-in dataflow links as a means to distribute access to services amongst components. The job of Vista will be to keep track of which connections are permissible and to actually establish the connections at the level of the underlying objects.

5.1 A Vista example

We shall illustrate the functionality of Vista with an example of a simple office procedure. The example can be stated as follows:

> There exists a database of customer records of the format *name:fax number:value*, which supports queries and insertions. Given a customer name, the script must generate a fax to the customer containing the date, the customer's name, fax number and a balance calculated from the value in the customer's record and an input value.

Figure 6 shows an implementation of this office procedure as a Vista script. In this case Vista's built-in dataflow scripting model has been used to script the desired behaviour. Simple arrows represent links and component ports are only visible during the linking operation. All links are displayed attached to the upper left hand corners of their linked components. Each link is a dataflow link and the script as a whole can be read as a dataflow diagram.

Six kinds of standard components have been used in the example: *Label* components, like **query**, **entry** and **Sfr.->US$** which simply display a constant text label, *Text* components into which a user may type, a *Database* component that accepts queries and new entries, *RecordTo* components that extract fields from records, a *Calculator* component and a *Document* component that presents the assembled information.

The first two Label components serve to document two Text components which feed values to the Database component. The job of a Text component is to store a string. The current text stored in the component is displayed in a rectangle where it may be edited by the user. In addition, each Text component has an input port called **text in** where it may receive updates and an output port called **text out** which is used to propagate the current value to other components. In this script the **text out** ports are linked to the Database component and the **text in** ports are left unbound.

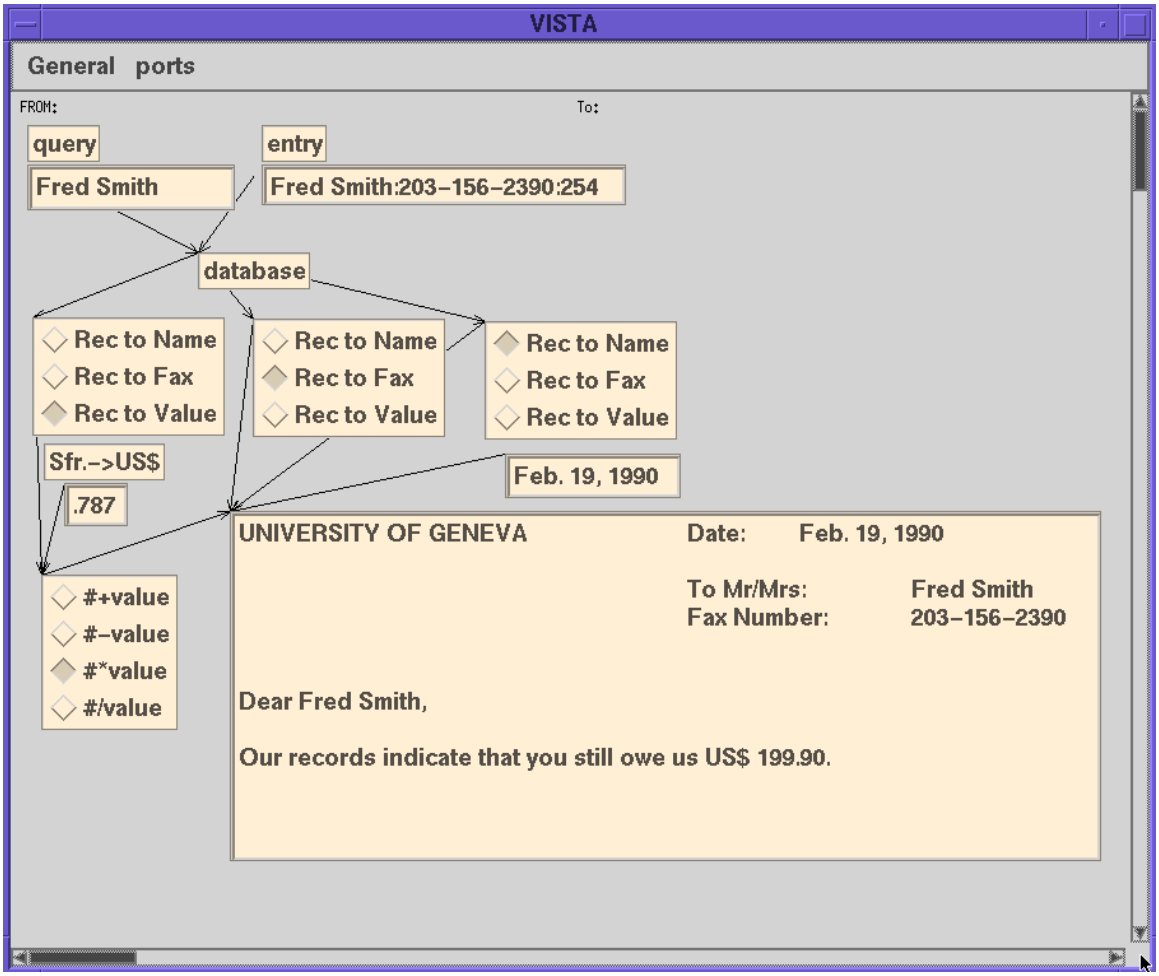


Figure 6 A Vista Script

The Database component is responsible for maintaining a set of records with the format *name:fax number:value*. It has two input ports, **query** and **entry**, where a string to match or a new record to enter may be received, and a **record out** output port where the last record stored or retrieved is made available to other components. In the script this output value is propagated to three RecordTo components each of which extracts and outputs a selected field of the record.

A Calculator component has two input ports for its arguments and a single **result** output port for the value computed. The Document component is used to group and format information produced by its connected components into a fax message that can be sent to the customer.

It should be noted that, in the current version of Vista, there is no difference between creating a script and executing a script. For small scripts this may be acceptable but for larger and more complex scripts, separate creation and execution modes could be useful. Furthermore, it should be possible to generate a stand-alone application from a script—presently scripts can only be executed from within Vista.

5.2 Vista Implementation

VISTA's internal representation of a script is a graph. The nodes of this graph represent components and the arcs represent links. This basic graph structure has no inherent syntax or semantics.

The above example can be represented as the simple directed graph of Figure 7. The example, by virtue of being essentially a dataflow diagram, maps nicely onto a directed, acyclic graph. In general, though, more complex applications will require more complex graphs for their representations.

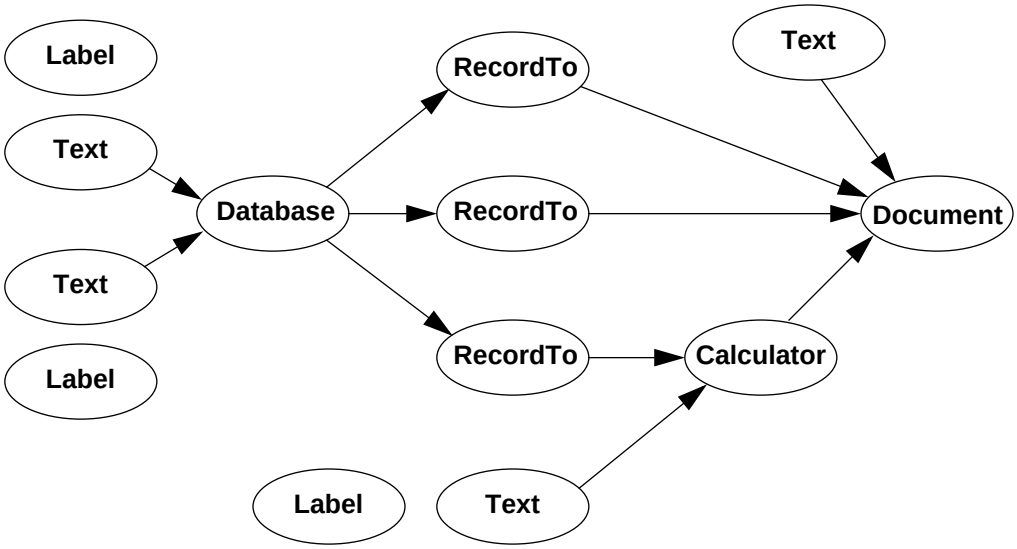


Figure 7 Example script as a graph

Vista is implemented in C++. The graph management of Vista is supplied by a set of class definitions and functions provided by the *Labyrinth* graphics editor [16]. Components and links displayed on the screen use the display capabilities of Labyrinth together with widgets created using the C++/Motif binding software from the University of Lowell. The currently supported hardware platforms are the Sun SPARCstation and 386 machines running SCO UNIX.

The user of Vista is presented with the scripting concepts of component, port and link. At the implementation level these concepts are mapped into groups of cooperating objects. Each component has an *activity*, that is, the internal representation of its state and services, and a *visual presentation*, which includes its appearance and its user interface. The activity of a component is either implemented as a C++ object or, in the case of encapsulated scripts, as a graph. Limited support for activities implemented as CooL[1] objects is also available. The visual presentation of a component is currently supported through Motif widgets. These two parts are kept separate to allow for the possibility of associating different presentations to the same component. The activity and presentation are connected and can communicate with each other via the graph structure representing the script. Each is wrapped inside a special *node* object that allows it to participate in the script.

Two components transfer information between each other when their ports are linked. Each port is represented in the graph by a special *port node* object and each link by a *link node* object. The port and link nodes are responsible for maintaining local and global constraints and to inform the user if invalid links are attempted. These constraints and linking rules are obtained from the scripting model.

1. CooL is the programming language developed within ITHACA — see §7.

Vista is intended to serve as an evolutionary prototype. The user interface of the tool is continuously changing as new functionality is added to Vista and as we get more insight into the possible ways of scripting applications. There is a wide variety of topics still left open in this first version of the tool which we plan to investigate. These topics include: a better way of adding new components, more support for components written in different programming languages, the general treatment of scripting models and generating code from a script.

6. Related work

There are presently a number of research prototypes and commercial products that illustrate the essence of scripting. Each of these tools, however, is based on a *single* scripting model for a particular component framework. We argue that, despite the success of some of these tools, the full potential for scripting applications is still largely untapped. Let us briefly consider three examples.

One such tool, Fabrik [15], is a visual programming environment that exploits bidirectional dataflow. The goal of Fabrik was to facilitate the *kit* approach to programming by taking advantage of emerging technologies such as iconic user interfaces. With kits, fixed rules govern the composition of kit components, thus restricting their possible reusability and the flexibility of application generation. Visual scripting, with scripting models, tries to factor out such restrictions to allow the creation of application templates that can be tailored to particular application domains.

HyperCard [14] is an authoring tool and information organizer based on the concept of *stacks* of information. HyperCard provides a fixed set of five components that can be configured interactively or through a high level programming/scripting language called Hypertalk. HyperCard is self-contained, no components can be added and there is no distinction between the tool environment and the application. In contrast, visual scripting supports the composition of components retrieved from continuously evolving component sets. The only restriction on new components is that they conform to the rules of a predefined scripting model.

Interface Builder [17] allows an application designer to define an application's interface graphically and to connect user interface objects to underlying application objects which have been programmed separately. Interface Builder was not intended to be an application generating tool, but it is gradually becoming more versatile in its functionality. Interface Builder, in contrast to visual scripting, does not support hierarchical design, that is, the encapsulation of scripts as components to be reused in future development.

Each of these tools has a very attractive direct manipulation user interface and attacks significant problems in the application development community but they are all limited to describing particular *instances* of applications whereas scripting in general can support the development of generic applications as scripts. Furthermore, by means of multiple scripting models, it is possible to support not only evolving component sets but to generalize scripting to different application domains each with its own standards for software composition.

7. A Scripting Environment

A visual scripting tool by itself is not sufficient to adequately support component-oriented software development. In addition, we need programming language tools to support the development of components, class management tools to support the organization, management, storage and retrieval of components, and tools to help in the task of matching requirements to available component frameworks.

Vista is being developed as part of the *ITHACA*[1] project [1]. The goal of ITHACA, which stands for *"Integrated Toolkit for Highly Advanced Computer Applications,"* is to produce an advanced development environment for industrial applications based on object-oriented technology. This environment includes an object-oriented language called CooL, closely integrated with an object-oriented database, and tools to support the development of applications. A central component of the environment is the *software information base* (SIB) which stores and manages structured "descriptions" of software. The other tools interact primarily by exchanging and manipulating information stored in the SIB. They include: a *selection tool* for browsing and querying the SIB; MaX, a *monitoring debugger* for CooL classes; RECAST, a *requirements collection and specification tool*; and Vista. Standard tools and software components are provided by *application workbenches* for specific application domains supported by ITHACA, such as Public Administration and Office Systems.

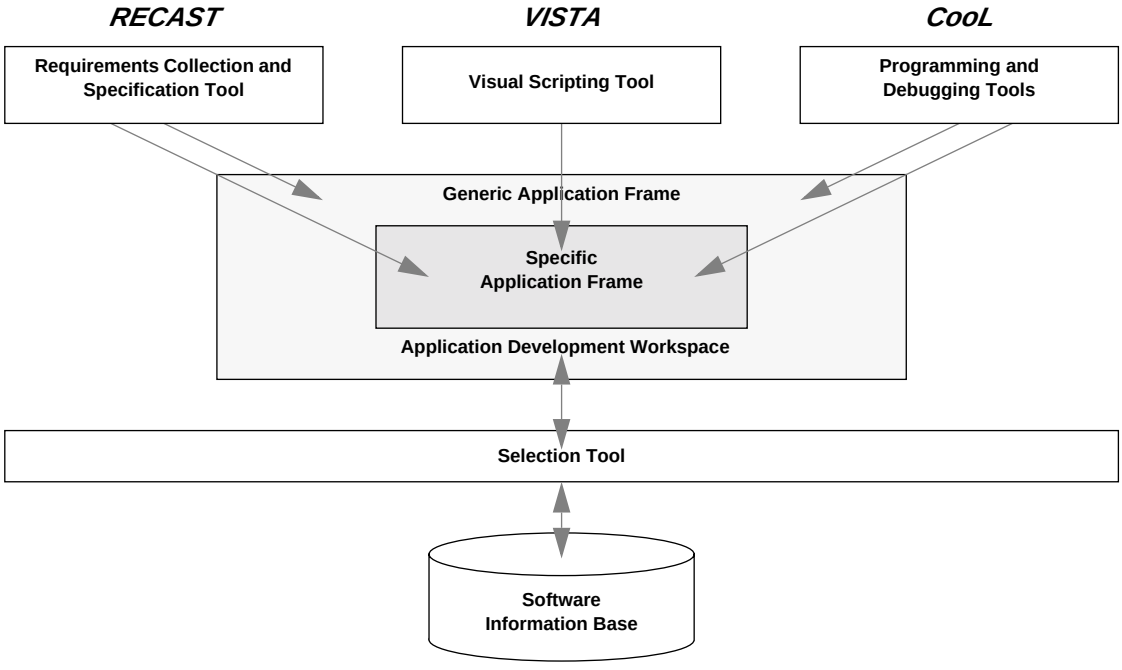


Figure 8 The ITHACA Application Development Environment

1. ITHACA is a 5-year, 100 person-year/year Technology Integration Project (#2705) in the Office & Business section of the European Community's Esprit II Programme. The partners are Siemens/Nixdorf (Berlin), Bull (Paris), Datamont (Milan), TAO—Tècnics en Automatitzaciò d'Oficines (Barcelona), FORTH—the Foundation of Research and Technology, Hellas (Iraklion) and CUI—the Centre Universitaire d'Informatique of the University of Geneva.

In Figure 8 we can see the relationships between these tools. The purpose of RECAST [10] [11] is to provide a "guided" tour of the SIB [3]. It is this activity that aids the application developer in "negotiating" between the application requirements and the available software components. Without this negotiation, the chance of arriving at any reusable components as a result of an object-oriented analysis and design is significantly reduced. Since this is a critical activity, the SIB cannot merely be a repository of component frameworks. In addition, it must store and maintain application domain models, requirements models, implementation "hints" etc.—in short, software descriptions that encode *experience*.

The ITHACA term for one of these decorated component frameworks is a *Generic Application Frame* (GAF). The negotiation task is essentially one of retrieving and refining a GAF up to the point where a specific application can be largely scripted. If necessary, new components can be programmed at any point.

In the long term, we hope to make the job of the application developer as easy as possible. This requires, however, a substantial investment not only in developing the tools and environment, but in populating the SIB. The activity of developing scripting models, reusable components and their associated GAFs is called *application engineering* (which is analogous to the development of product lines, whereas application development is analogous to sales). The job of the application engineer is much harder than the traditional one of a software engineer, as he or she must develop reusable software for unknown application with unknown requirements. The only plausible way to attack such a problem is to adopt an evolutionary software life-cycle in which previous and future experiences feed back into the iterative development of component frameworks and GAFs (Figure 9).

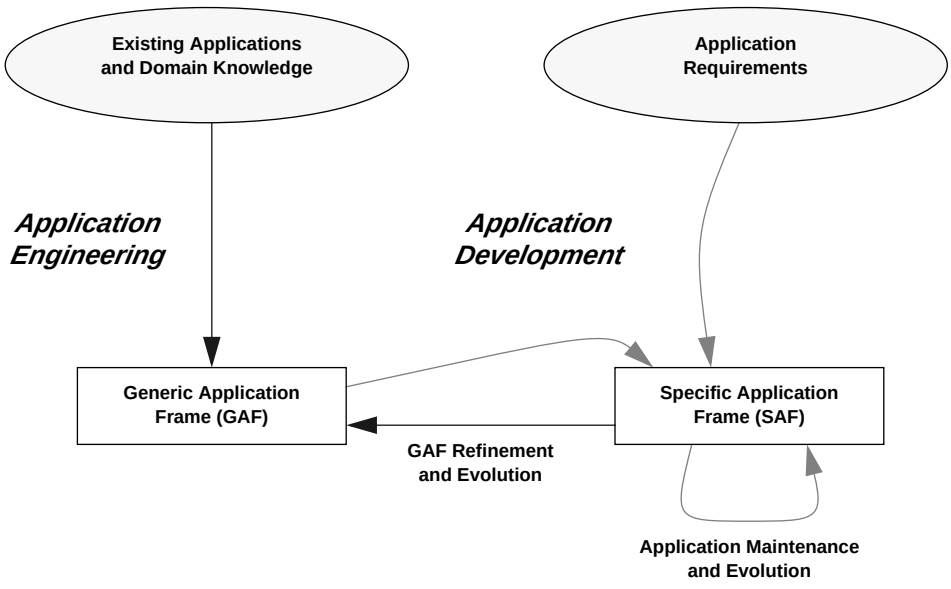


Figure 9 The Evolutionary Software Life-cycle of Application Engineering

8. Concluding Remarks

Object-oriented technology addresses very well issues of encapsulation of state and behaviour, instantiation of objects through classes, and incremental modification and reuse of code and interface through inheritance and related mechanisms. But object-oriented programming alone does not necessarily encourage *component-oriented* development of applications—one must still *program* new classes to make use of existing ones. We have argued that *scripts* are the previously missing summand in the equation: *objects + scripts = applications*. It should be noted that end users need applications, not programs or objects or components.

Scripts serve to introduce and bind objects that, like character actors, know how to play a well-defined role. A *scripting model* defines and standardizes the possible binding relationships between components, each of which stands for some set of abstract services available at the level of the object-oriented language. A scripting model determines the standard interfaces for the reusable components of a component framework. Components that conform to the scripting model will be *plug-compatible*, and so can be scripted to produce application. Familiar examples include the stream-based scripting model of Unix shell scripts and the event-based scripting model of user interface toolkits.

A *visual scripting tool* provides a graphical, direct manipulation interface for the interactive construction and execution of scripts. We have implemented a prototype, called *Vista*, which supports scripting of both the behaviour and the user interface of applications. Vista supports the visualization and graphical linking of components selected from an extensible component set. Scripts can be interactively executed, saved, and encapsulated as components for use in other scripts. At present Vista provides a single, hard-wired dataflow-based scripting model. Vista has been designed, however, with the goal of eventually supporting multiple scripting models for a range of component frameworks. This implies the need for a standard notation for defining these models. More experience with various kinds of scripting models is required before such a notation can be formally defined. At present we are developing a scripting model for active forms and its associated component set.

Vista is being developed as part of *ITHACA*, an Esprit Technology Integration Project which seeks to produce a complete environment for application development based on object-oriented technology. Such an environment includes interactive programming and debugging tools, a Software Information Base for storing descriptions of software components and application domain knowledge, a selection tool for querying and navigating through the SIB, a requirements collection and specification tool to aid in the task of negotiating between application requirements and available component frameworks, and Vista, the visual scripting tool. ITHACA also provides, by means of a set of *application workbenches*, the means to test the tools and environment in a selected set of well-understood industrial application domains.

In addition to continuing work on Vista, we are pursuing a number of other directions closely related to visual scripting. Our first prototype of a scripting tool was *TEMPO*, a language for scripting temporal objects, in particular, animations [9]. Recent work on an object-oriented multimedia framework supports scripting [25]. We have also developed scripting interfaces for musical composition [6].

Another closely related topic is the specification of object-oriented programming languages that are better suited to the development of reusable software components. If one observes that object-oriented mechanisms supporting reuse, such as classes, inheritance, mixins, generics etc., all work in essentially the same way, that is by functional composition of various kinds of software components, then it is very natural to propose a *pattern* mechanism that generalizes this principle. In fact, scripts and components are both examples of "software patterns". A pattern language for active objects would be a natural foundation both for the development of reusable components and for scripting [20][21].

We shall close with a philosophical remark on the nature of scripting. Scripting is far from being a new idea—humans have used scripting to convey a general plan of action between loosely cooperating actors. Football coaches make up scripts to be executed by their players. Film directors outline scripts for their actors. Generals script battle plans for their troops. Programmers used to combine language statements to write programs. We suggest instead that they should script their objects to build applications.

Acknowledgements

Vista has been implemented by Vicki de Mey, Serge Renfer, Betty Junod and Marc Stadelmann. The precursor to Vista, VST, was implemented by Marc Stadelmann and Jan Vitek. We thank our ITHACA partners and associates for stimulating discussions on the nature of application engineering, GAFs, and scripting.

References

[1] M. Ader, O.M. Nierstrasz, S. McMahon, G. Müller and A-K. Pröfrock, "The ITHACA Technology: A Landscape for Object-Oriented Application Development," Proceedings, Esprit 1990 Conference, pp. 31-51, Kluwer Academic Publishers, Dordrecht, NL, 1990.

[2] G. Anderson and P. Anderson, *The UNIX C-shell Field Guide*, Prentice-Hall, 1986.

[3] P. Constantopoulos, M. Dörr, E. Pataki, E. Petra, G. Spanoudakis and Y. Vassiliou, "The Software Information Base – Selection Tool Integrated Prototype," ITHACA report FORTH.91.E2.#3, Foundation of Research and Technology – Hellas, Iraklion, Crete, Jan 12, 1991.

[4] B.J. Cox, *Object Oriented Programming — An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.

[5] B.J. Cox, "Planning the Software Industrial Revolution," IEEE Software, vol. 7, no. 6, pp. 25-33, Nov. 1990.

[6] L. Dami, "Scripting Musical Components in Application Interfaces," in *Object Management*, ed. D.C. Tsichritzis, pp. 357-366, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[7] V. de Mey, "Vista Implementation," ITHACA Report CUI.90.E.4.#1, Centre Universitaire d'Informatique, University of Geneva, Dec, 1990.

[8] V. de Mey, "Vista User's Guide," ITHACA Report CUI.90.E.4.#2, Centre Universitaire d'Informatique, University of Geneva, Dec, 1990.

[9] E. Fiume, D.C. Tsichritzis and L. Dami, "A Temporal Scripting Language for Object-Oriented Animation," Proceedings of Eurographics 1987 (North-Holland), Elsevier Science Publishers, Amsterdam, 1987.

[10] M.G. Fugini and B. Pernici, "RECAST: a tool for reusing requirements," in *Proceedings CAiSE'90*, LNCS 436, Springer Verlag, 1990.

[11] M. G. Fugini, M. Guggino and B. Pernici, *Reusing Requirements Through a Modeling and Composition Support Tool*, Trondheim, May 1991, Accepted to CAiSE'91.

[12] N. Gehani, “The Potential of Forms in Office Automation,” IEEE Transactions on Communications, vol. Com-30, no. 1, pp. 120-125, Jan 1982.

[13] S.J. Gibbs, D.C. Tsichritzis, E. Casais, O.M. Nierstrasz and X. Pintado, “Class Management for Software Communities,” Communications of the ACM, vol. 33, no. 9, pp. 90-103, Sept. 1990.

[14] D. Goodman, *The Complete Hypercard Handbook*, Bantam Books, 1988.

[15] D. Ingalls, “Fabrik: A Visual Programming Environment,” Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 23, no. 11, pp. 176-190, Nov. 1988.

[16] M. Katevenis, T. Sorilos, C. Georgis and P. Kalogerakis, “Laby User’s Manual (version 2.10),” ITHACA report FORTH.90.E3.3.#7, Foundation of Research and Technology – Hellas, Iraklion, Crete, Dec 31, 1990.

[17] *NeXT Preliminary 1.0 System Reference Manual: Concepts*, NeXT Inc., 1989.

[18] O.M. Nierstrasz and D.C. Tsichritzis, “Integrated Office Systems,” in *Object-Oriented Concepts, Databases and Applications*, ed. W. Kim and F. Lochovsky, pp. 199-215, ACM Press and Addison-Wesley, 1989.

[19] O.M. Nierstrasz, L. Dami, V. de Mey, M. Stadelmann, D.C. Tsichritzis and J. Vitek, “Visual Scripting — Towards Interactive Construction of Object-Oriented Applications,” in *Object Management*, ed. D.C. Tsichritzis, pp. 315-331, Centre Universitaire d’Informatique, University of Geneva, July 1990.

[20] O.M. Nierstrasz and M. Papathomas, “Viewing Objects as Patterns of Communicating Agents,” ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP’90, vol. 25, no. 10, pp. 38-43, Oct 1990.

[21] O.M. Nierstrasz, “The Next 700 Concurrent Object-Oriented Programming Languages,” submitted for publication, Centre Universitaire d’Informatique, University of Geneva, Feb 1991.

[22] M. Shaw, “Prospects for an Engineering Discipline of Software,” IEEE Software, vol. 7, no. 6, pp. 15-24, Nov. 1990.

[23] M. Stadelmann, G. Kappel and J. Vitek, “VST: A Scripting Tool Based on the UNIX Shell,” in *Object Management*, ed. D.C. Tsichritzis, pp. 333-344, Centre Universitaire d’Informatique, University of Geneva, July 1990.

[24] D.C. Tsichritzis, “Form Management,” CACM, vol. 25, no. 7, pp. 453-478, July 1982.

[25] D.C. Tsichritzis, S.J. Gibbs and L. Dami, “Active Media,” submitted for publication, Centre Universitaire d’Informatique, University of Geneva, Jan 1991.

[26] P. Wegner, “Capital-Intensive Software Technology,” IEEE Software, vol. 1, no. 3, July 1984.

[27] R. Wirfs-Brock and R.E. Johnson, “Surveying Current Research in Object-Oriented Design,” Communications of the ACM, vol. 33, no. 9, pp. 104-124, Sept. 1990.