Visual Scripting
Towards Interactive Construction of Object-Oriented Applications[1]

Oscar Nierstrasz
Laurent Dami, Vicki de Mey, Marc Stadelmann
Dennis Tsichritzis, Jan Vitek

Abstract

Object-oriented programming techniques are known to improve the flexibility and reusability of certain kinds of software. Libraries of object classes, however, continue to be difficult both to develop and to reuse. We present an approach to object-oriented application development in which applications are constructed by interactively "scripting" cooperating, reusable software objects. A visual scripting tool is being developed within ITHACA[2], an Esprit II project which seeks to produce an integrated environment for the rapid and flexible development of object-oriented applications for selected application domains.

1. Introduction

Traditional methods of application development are acknowledged to be slow, expensive and error-prone, and the results typically inflexible and difficult to maintain. With the observation that large numbers of modern applications are really just slight variations on one another, various techniques have emerged to exploit these similarities to great effect. Basically these techniques either (1) provide libraries of reusable software together with browsers, debuggers and software management tools or (2) provide very high level (VHL) tools or languages for constructing restricted classes of applications[3]. The former class of techniques is typified by Object-Oriented Programming (OOP) and the latter by application generators and Fourth Generation Languages.

Object-oriented programming is clearly the more open-ended approach since reusable class libraries can be potentially provided for any application domain. Unfortunately OOP is still programming, and programming, even with reusable software, can be a time-consuming, low-level activity. Although the VHL tools and languages can improve application development and maintenance for specific classes of applications, their advantages either break down or turn into disadvantages when exceptional requirements arise [14]. We would like to combine the open-endedness of OOP with the directness and power of VHL tools.

Visual scripting is a direct manipulation paradigm for constructing applications by coordinating the behaviour of collections of objects. An application developer would use a visual scripting tool by selecting visually presented application objects, such as electronic forms, mailboxes and office procedures, and tailor their behaviour to the application requirements by graph-

1. © Dennis Tsichritzis et auteurs, 1990. Tous droits réservés.
2. *ITHACA* is a Technology Integration Project (#2705) in the Office & Business section of the Esprit II Programme. The partners are Nixdorf (Berlin), Bull (Paris), Datamont (Milan), Tècnics en Automatitzaciò d'Oficines (Barcelona), the Foundation of Research and Technology, Hellas (Iraklion) and the Centre Universitaire d'Informatique of the University of Geneva.

ically editing and linking them. The objects are like character actors who are capable of playing certain kinds of roles in a variety of plays. The application developer is then a combination playwright and theatre director who assigns specific roles to objects and directs their interactions.

Visual scripting requires (1) that objects to be scripted have a visual presentation, and (2) that objects have a "scripting interface" which permits their behaviour to be graphically edited. Since these scripting interfaces are intended to be used from a direct manipulation editor rather than a programming language, they differ somewhat from the programming interfaces to objects that we see in object-oriented languages. A *scripting model* for an application software library defines the kinds of scripting interfaces that objects may have, and determines, therefore, the model for application construction available to the developer.

In section 2 we shall motivate scripting within the broader context of object-oriented development as we have defined it for the ITHACA project. We then elaborate the scripting paradigm through the use of an example, we give an overview of VST, a rapid prototype of a visual scripting tool, and we provide a framework for an object-oriented scripting model. In section 4 we outline an architecture for *VISTA*, a second-generation scripting tool being developed for the ITHACA application development environment. We conclude with some remarks on open topics for further exploration.

2. A Scenario for Object-Oriented Development

The traditional software engineering methods that have worked reasonably well in the past for stable, long-lived applications fail to yield good results today because of two central assumptions that no longer hold: (1) that the application requirements be stable and well-defined, and (2) that the application run in a closed and finite universe. Increasingly, modern-day applications are *open systems*, which can be "open" in each of the three following ways[21]:

1. *Platform:* the hardware and software platform continuously expands.
2. *Interoperability:* open applications must be prepared to exchange information and otherwise interact with systems that may not yet exist.
3. *Evolving requirements:* the application must be flexible enough to adapt to continuously changing requirements.

At this point, it is important to observe that there is another assumption that no longer holds, namely that every application is unique. It was this assumption, we believe, that justified the use of slow and laborious "waterfall" methods of development. In fact, large numbers of superficially different applications really perform very similar functions, which is the secret behind the success (however limited) of application generators and fourth generation languages. This suggests to us that object-oriented techniques, which are well-suited to factoring out common functionality into often highly reusable object classes, offer a more general way of allowing many applications in similar domains to share a large part of their code. Reuse of object classes may or may not streamline application development – reuse does entail overhead – but it should result in more flexible, robust systems. Unfortunately, it is not possible to just sit down and write libraries

of instantly reusable classes: the design of reusable classes is necessarily an iterative, evolutionary process [12].

For this reason we propose a very different model of software development in which *application engineers* are responsible for developing generic, reusable software for specific application domains, and *application developers* are their clients, reusing not just object classes but also pre-designed *generic application frames* (GAFs) that guide the developer towards standard ways of constructing applications. Initially, class libraries and GAFs may be developed by *re-engineering* existing applications in an object-oriented way (i.e., so as to factor out common functionality). As application developers encounter more demanding requirements, the software base must evolve to improve its generality and reusability. Application engineers and application developers thus cooperate in a producer/consumer relationship (*cf.* [4][19]).

To support such an object-oriented software life-cycle, it is clear that tools for storing and managing software information are essential [8] in addition to tools to support the development of classes and applications. ITHACA[1] is a 5-year, 100 person-year/year Esprit II project to build an environment supporting the development of object-oriented applications [16]. The ITHACA environment includes an object-oriented language (called CooL) closely integrated with an object-oriented database, in addition to tools to support application engineers and application developers. A central component in the environment is the *software information base* (SIB) which stores and manages structured "descriptions" of software (i.e., interface descriptions of object classes, GAFs, documentation, etc.). 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 the *visual scripting tool.*

Let us suppose that our application engineers have stocked the SIB with a library of classes and GAFs for an application domain such as "public administration" (one of several "demonstrator" domains for ITHACA). We would expect the application developer to proceed in the following way to produce a specific application:

1. Se*lect an application frame:* using only a rough sketch of the application requirements, the developer searches and browses to find a corresponding GAF.
2. *Select useful classes:* the GAF *drives* requirements collection and specification according to a pre-existing, generic design, thus guiding the developer in the selection of reusable classes.
3. *Tailor classes:* the selected classes are incrementally modified by supplying parameters or by refining their behaviour through inheritance (i.e., by programming, if necessary).
4. *Script application:* link the selected classes together by means of a "script" that specifies how the objects will cooperate to implement the required application.
5. *Monitor behaviour:* test and validate.

1. *ITHACA* stands for "Integrated Toolkit for Highly Advanced Computer Applications."

6. *Continuously develop:* as requirements change, adapt the application; as the understanding of the application domain evolves, upgrade the contents of the SIB.

The role of scripting in this scenario is (1) to provide a very high-level means of rapidly and flexibly constructing applications from reusable classes and (2) to provide concrete guidelines for developing and *evaluating* object design. In our view, a well-designed object is one that can be scripted. To make this notion more precise, we shall explain the idea of scripting through several examples.

3. Application Construction by Scripting Objects

A script packages and links a number of *software components* into a running application. At the lowest level, components will correspond to objects (although, as we shall see, it is also possible to script other kinds of software components). A script itself is also a software component and, depending on how it has been packaged, may be used within higher-level scripts. Every component has a number of *ports* that parameterize its behaviour. Ports may either be assigned values statically or they may get their values dynamically by being linked to the port of another component. Ports that remain unbound may become ports of the script. Scripting, then, is the activity of selecting components, linking or assigning values to their ports, and packaging the result. A *scripting model* defines what *kind* of ports components may have and the rules for linking them.

In the discussion that follows, we shall see that, within this general framework for scripting, ports and links may be interpreted in a wide variety of useful contexts and that components may correspond to many different kinds of application objects.

3.1 Scripting Office Procedures

The domain of Office Information Systems contains many generic applications that can be modeled as electronic forms systems. There are three main aspects to most form applications. First, there are the forms that need to be designed. Second, there are simple, repetitive clerk procedures involving the forms, like selecting the proper forms to build a relevant case, and straightforward consistency checks, triggers and operations on the forms. Third, there are difficult decisions and parameter estimations that need to be performed directly by the users. These user actions are neither straightforward, nor can they be captured algorithmically. The scripting of forms deals with these three aspects in the following way: it provides the right environment to capture the first and second aspects, and it is then used to build a system in which users can interact graphically for the third aspect.

Let us consider the case of a mail-order retailer that maintains a catalog of the products it sells, their current price and stock at hand, as well as a record of customers' names, addresses and their current accounts. Various people are responsible for entering orders, maintaining the customer records, approving orders and so on. We would like to implement an electronic forms system in which order processing, billing and other activities are supported by automatically triggered office procedures. Electronic office procedures partially automate office work by handling routine cases and by helping the user to detect and manage exceptional cases[13][20]. Each

user will have a virtual desk with an in-tray, form-folders and various tools to support office work. Users may create, edit, file and retrieve forms and mail them to other users.

We shall illustrate the possibilities of visual scripting by showing how one might interactively script (1) forms and their behaviour, (2) triggered queries that assemble collections of related forms to support office work, and (3) office procedures that package actions to be performed on such collections.

Scripting Electronic Forms

Since electronic forms are end-user objects visible in running applications, we argue that it is natural to use this same presentation during their design and implementation. One should design forms in much the same way that documents are composed using direct manipulation, WYSIWIG document editors. In this way we can script both the *appearance* of a form (the form *template*) and its *behaviour*. Given a collection of basic interaction components, like fields, buttons, sliders or menus, designing an electronic form consists of selecting such components, customizing them and grouping them together. Components may have predefined behaviours that get coordinated through the form script.

Unlike their real-world counterparts, electronic forms may be *active*, that is, they may impose constraints on the way fields are filled and modified, they may cause certain fields to be computed automatically, they may present different views to users depending on the context or on their current state (or both), and they may cause actions to be triggered [7][15]. Specifying this behaviour is part of the scripting activity. Scripting forms, then, consists mainly of establishing *part-of* links between a form object and its contents, possibly establishing relationships between the various parts, and setting properties of the form and its parts.

Fields are elementary units of interaction. They have a collection of graphical attributes like size, location and colour, and they support operations for displaying and editing an associated value. Various types of fields can be grabbed from a palette (Figure 1) and modified to fit

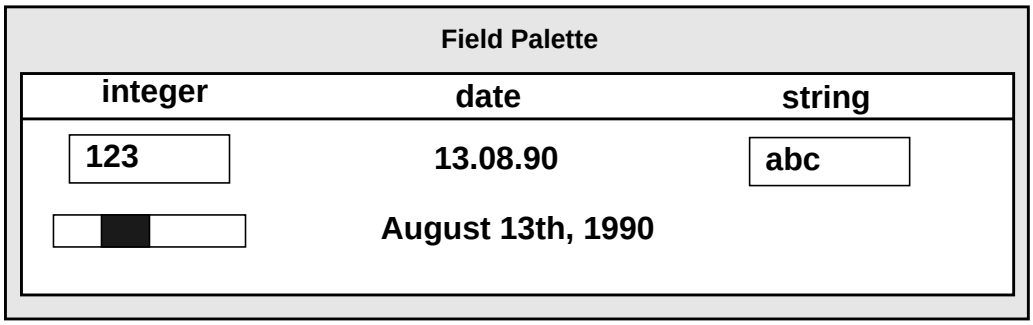

Figure 1: ***a (tiny) palette of available form fields***

particular graphical constraints. For a particular field type, it may be possible to choose among several views. For example, a field holding an integer value may be displayed either as a string of digits or as a slider. A simple example of a form is a customer record (Figure 2), which is easily scripted from existing fields.

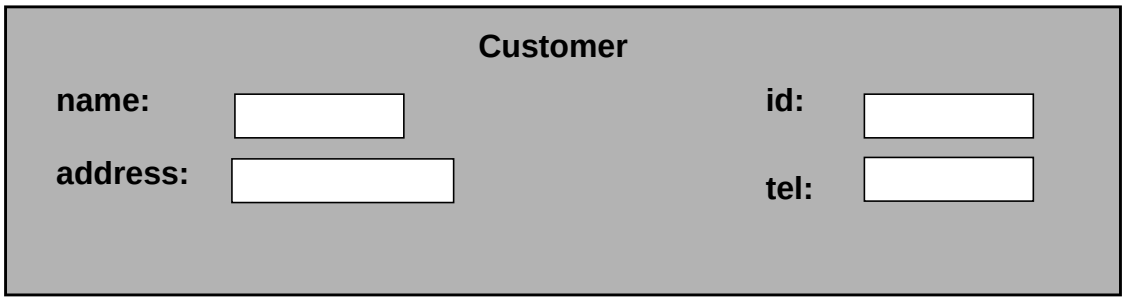

Figure 2: *a simplified customer form*

Our task gets a little more complicated if we want to add accounting information to the customer record. We would like to maintain a record of the amounts debited and credited to a customer's account. Since the account record grows during the lifetime of the customer record, it clearly cannot be modeled as a static form. Instead, we model each debit and credit as a simple account entry form and the account record as a form containing a *table* of account entries.

A table is an object that provides a view on a homogeneous list of forms, presenting some (or all) of the contents of each form as a row of field values. At this point it is worth emphasizing that there may be many alternative ways of organizing and viewing lists of forms. We shall make use of *form folders* (an iconized, shared list of forms) and *in-trays* (a private list into which other users may deposit forms, and which may trigger other objects upon the arrival of a new form).

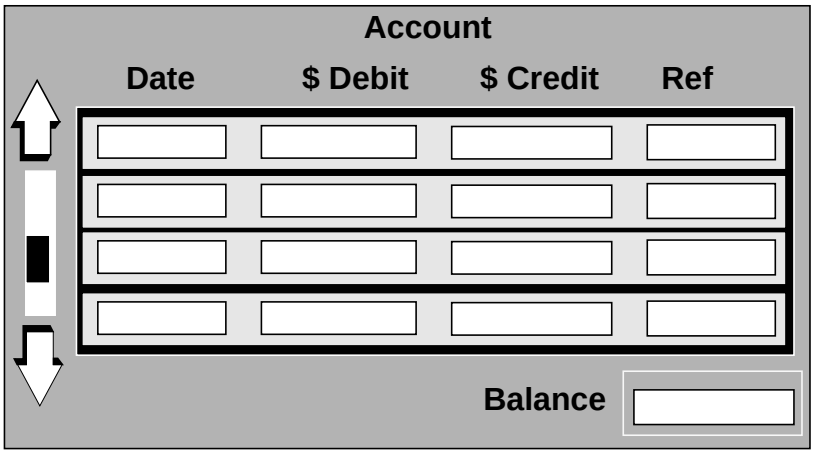

Figure 3: *a table of account entries*

In Figure 3 we see an account form that contains a table of account entries, where each entry has a date, an amount appearing in either the debit or credit field and a reference field for linking with bills or payments. Note the presence of a scrollbar for navigating through the table. The account form also contains a *computed field* displaying the balance of the account. Such a field could be scripted by associating a computed *subtotal* field with each account entry, which simply applies the debit or credit to the subtotal of the previous entry. The balance is then linked to the subtotal of the last entry in the table.

The account form, with its table of account entries, can now be included into the customer form (Figure 4). This demonstrates that complex data structures are supported by the form system: forms can recursively contain other forms.

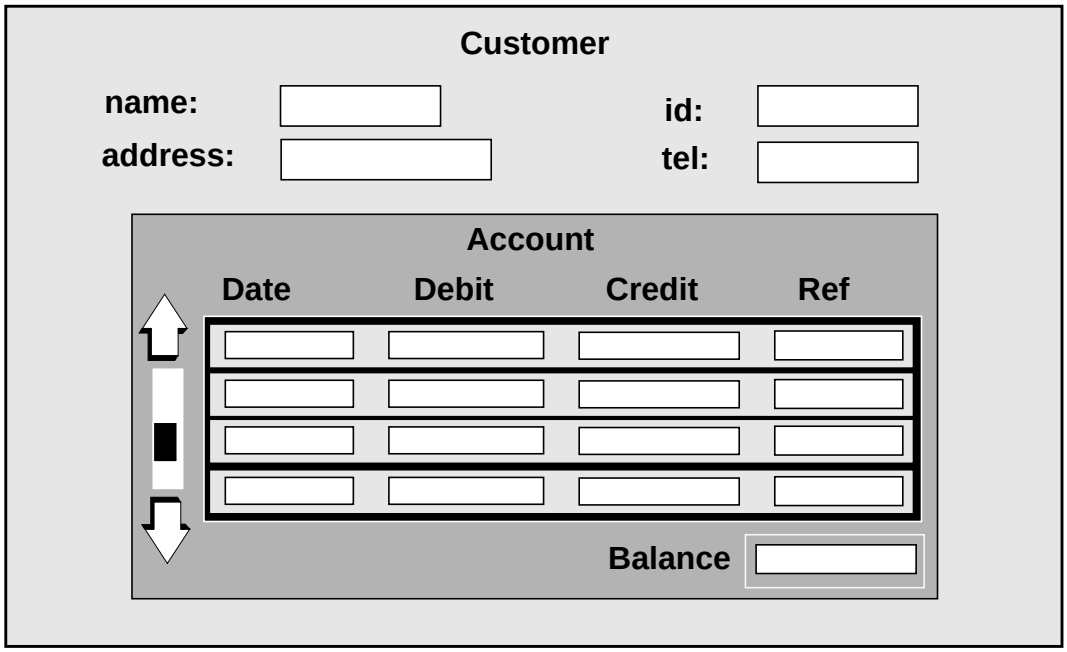

Figure 4: ***the complete customer form***

Scripting Queries and Triggers

Queries over collections of forms are naturally expressed by overloading form templates to specify retrieval criteria in a Query-by-Example [22] fashion. For example, consider a list of *product* forms (Figure 5) recording the name of a product, its price and the quantity in stock. The product form template can be used as an interface for selecting instances from a list by partially filling in the template. For example, to retrieve the record for a given product, the *name* field would be filled. To find out which items are running out, the condition "< 10" might be entered into the field representing the quantity in stock.

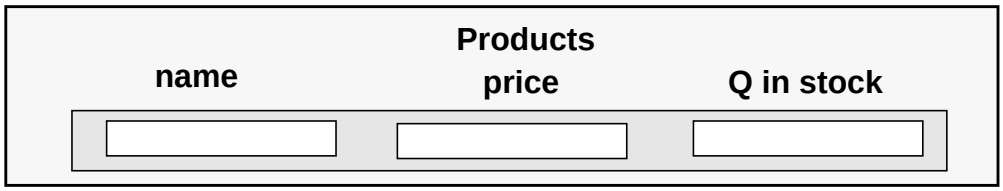

Figure 5: ***a product form***

Sets of matching forms of different types can be similarly retrieved by linking fields of different templates. Consider the order form in Figure 6. Each order form contains a table of individual orders for specific items. To retrieve the matching customer record, one would link the *from* field of an instance of the order form with the *name* field of the query template for the cus-

tomer forms. To retrieve *all* matching order forms and customer forms, one would link the *from* and *name* fields of query templates for both.

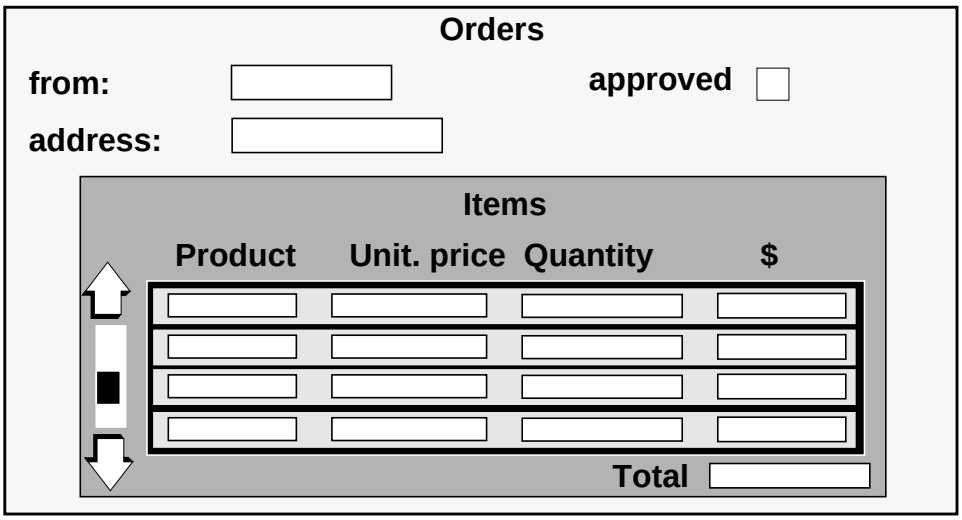

Figure 6: *an order form*

So far we have been concerned with *ad hoc* queries. A *triggered query* is a kind of persistent query that attempts to collect forms matching retrieval criteria whenever a triggering event occurs. For example, the query for products whose stock is low may be triggered upon modification of the quantity in stock field. The result of this triggered query is a list of products that may be consulted at any time. As we shall shortly see, triggers can be especially useful for initiating a set of actions. Failure of a triggered query can in turn trigger other actions or be simply used to warn the user that manual intervention may be required.

For our order processing application, we require a triggered query to build a *case* consisting of an order form, the matching customer record, and the records of all products ordered. The trigger condition will be the arrival of an order form in an in-tray. Once the query is triggered, if any condition fails (for example, if no previous record exists for that customer) the user will be able to examine the partially constructed case and take exceptional action.

In Figure 7 we see how this might be scripted: the thick, dashed boxes represent triggered queries. Black arrows represent triggers, grey arrows represent the source or result of matching forms, and the grey lines represent the fields to match. The outer query is triggered by the arrival of an order form. It then retrieves matching customer records from the customer folder. The inner query is triggered by the retrieval of the order form and selects a matching product record for each item ordered. The results of the query are maintained in a case folder.

Scripting Office Procedures

Office procedures in an electronic forms system can be conveniently viewed as a collection of associated triggered queries and actions. We can script office procedures in much the same way that we script triggers and queries by simply overloading the function of form templates. The procedures can be directly specified on the templates representing the forms retrieved by a triggered query or, as in this example, an output case of a triggered query can be used as input for the procedure which is defined in a Programming-by-Example fashion [9]. In Figure 8 we script

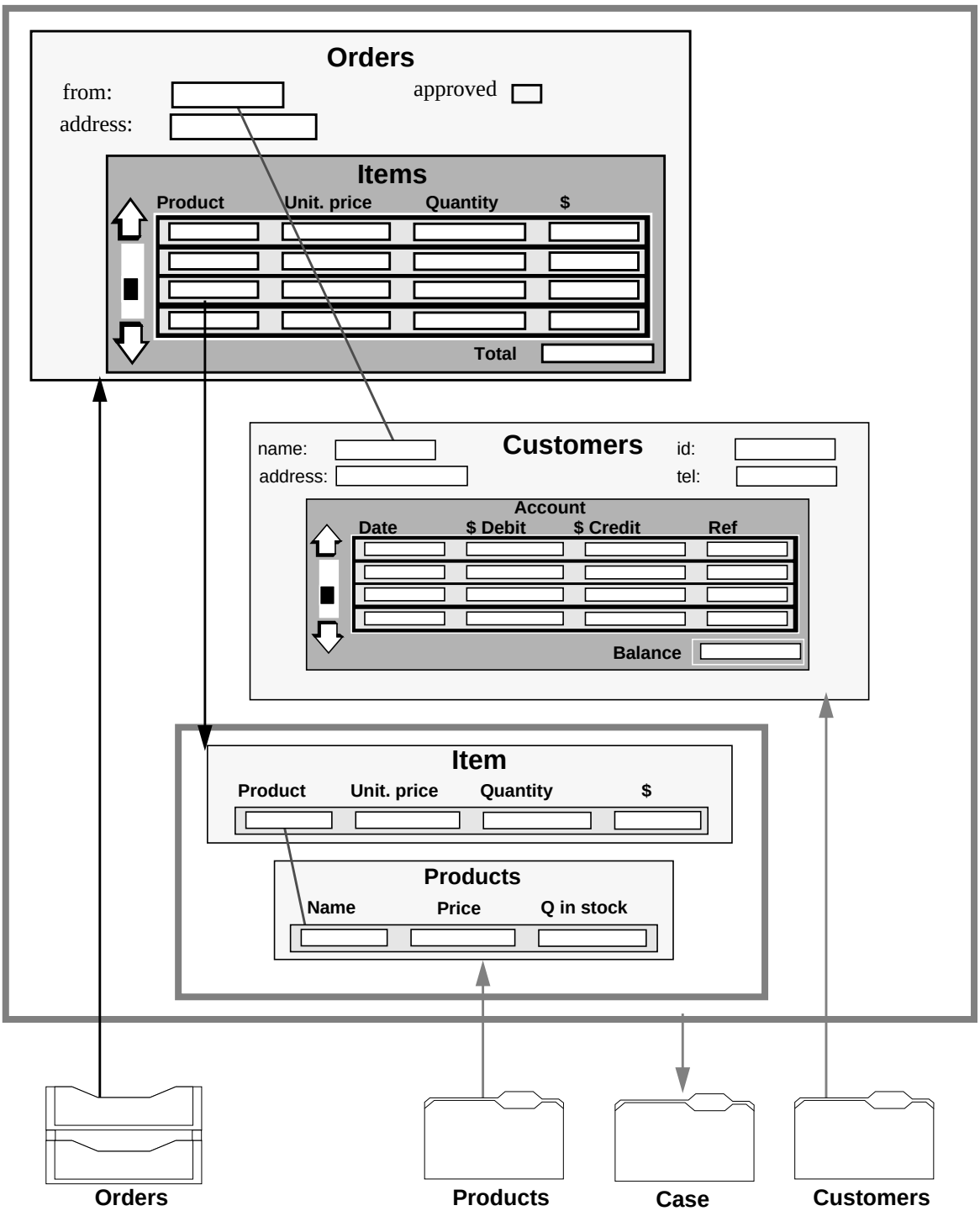

Figure 7: ***building a case with a triggered query***

the following actions: (1) create an account entry, (2) fill in the current date, amount to debit and the reference to the order form, (3) debit the customer's account (i.e., insert the account entry into the customer's account table), (4) compute the new quantity in stock and update the product form, (5) approve the order form and (6) forward the approved order to shipping. The thin black arrows copy values or references from one object to another, and the thick grey arrows insert forms into tables or in-trays. The order in which the actions occur depends on the availability of information from the case.

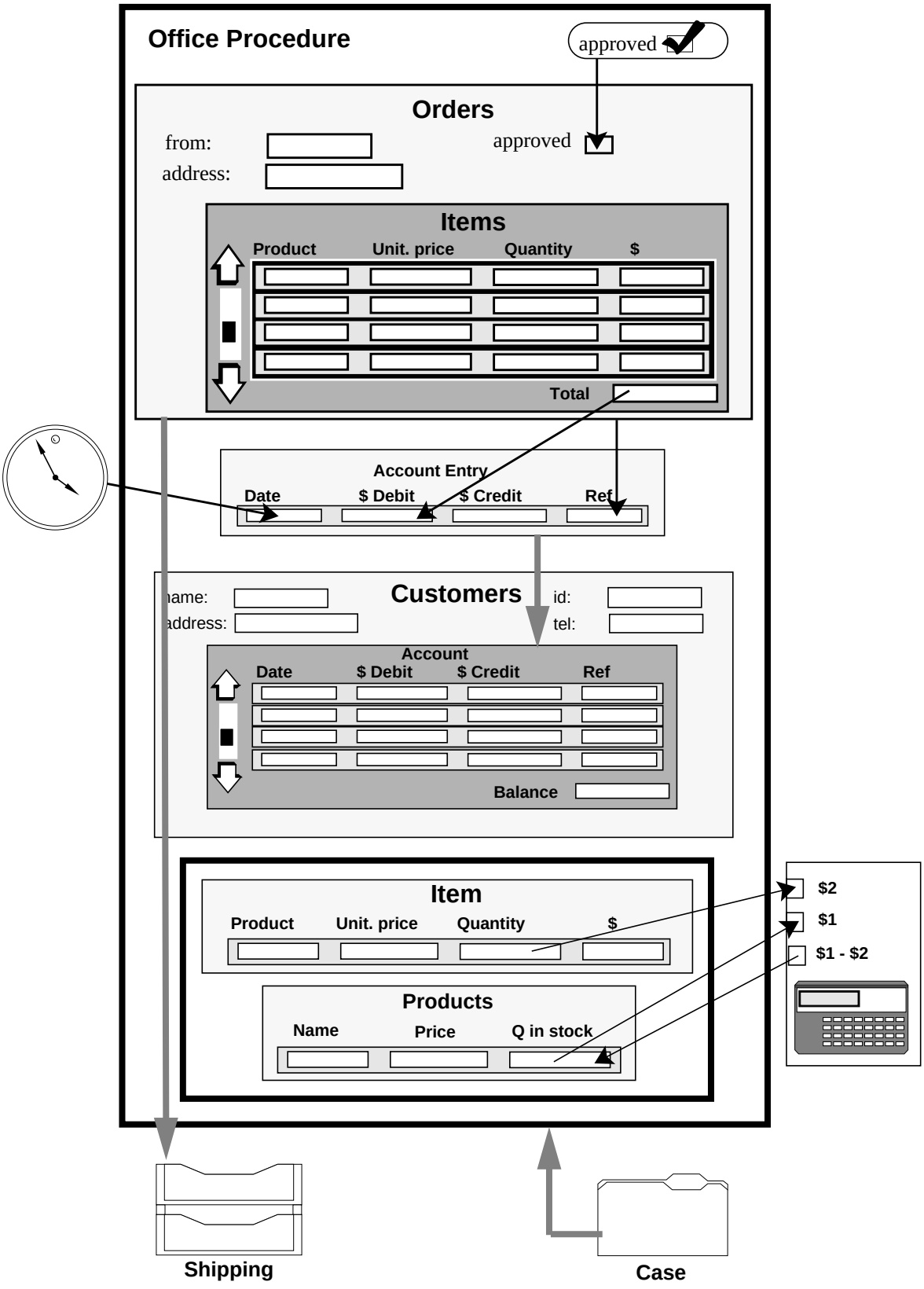

Figure 8: ***scripting the actions of an office procedure***

3.2 VST – A Unix-Based Visual Scripting Tool

We have implemented a proof-of-concept prototype of a Visual Scripting Tool (VST) based on a Unix scripting model [18]. Unix was chosen as a basis mainly because of the ready availability of a large collection of existing software components (i.e., Unix commands) and because Unix provides powerful mechanisms for connecting existing commands by means of pipes, file redirection and environment variables.

For the purposes of demonstrating the potential of visual scripting, we have selected those mechanisms of the Unix shell that lend themselves well to visualization and expressed them as a scripting model. In this model, components are (1) Unix commands or shell scripts, (2) Unix files, (3) strings or (4) visual scripts. Any given port of a component is directed, depending on whether values are input or output at that port. The types of ports provided in the prototype are (1) *text* ports (used mainly to provide arguments to Unix commands), (2) *stream* ports and (3) *sequencing* ports to control the execution order of commands (if necessary). Links may be made between input and output ports of the same type (as links are made, they are checked for consistency). Links are also used to define the ports of a script being encapsulated as a new component by connecting internal ports to the border of the script.

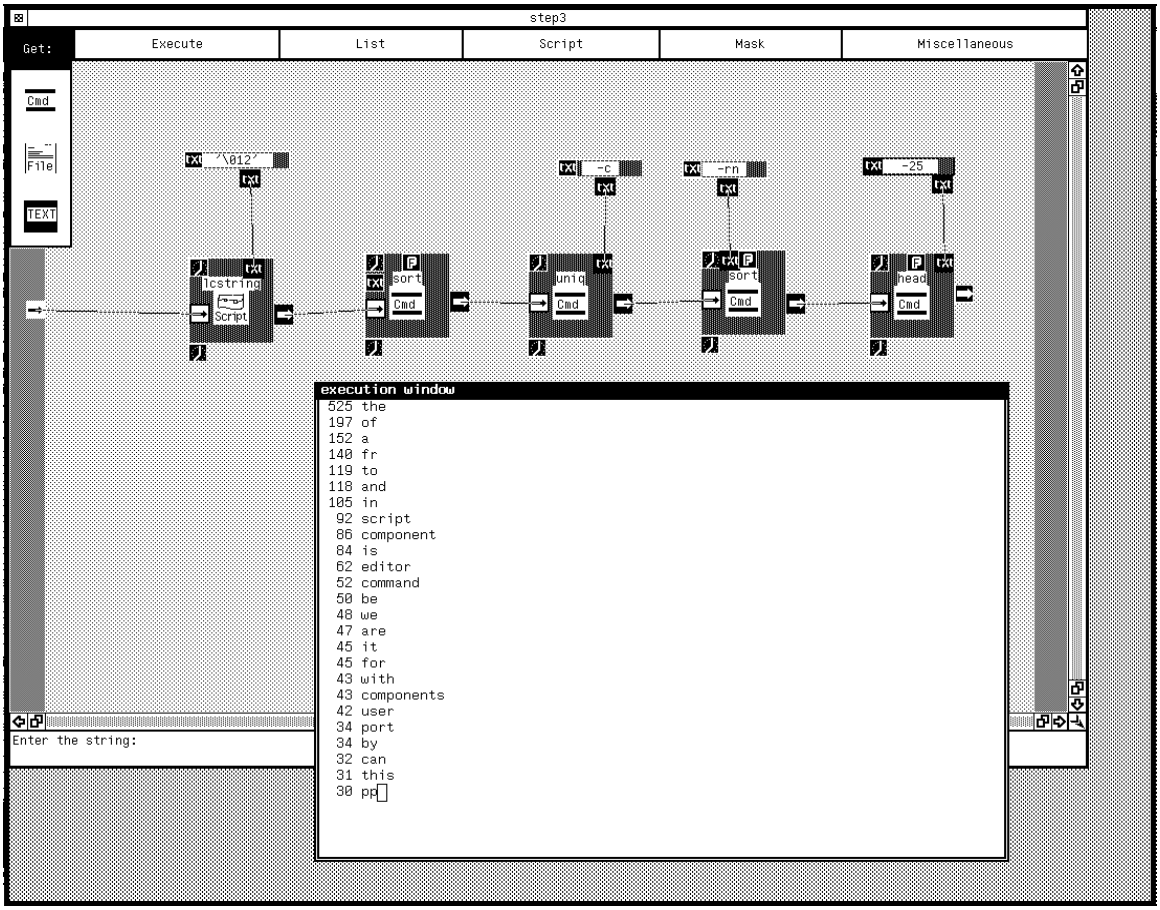

Figure 9: *scripting a word frequency counter*

In Figure 9 we see a visual script of the *word frequency counter* problem[2]. This script extracts the *n* most frequently occurring words in a file and reports how many times they occur. The *lcstring* component is a previously packaged visual script that converts all alphabetic strings in its input to lower case, separating them by a character given as a parameter. In this case we separate all words by newline characters (\012). We sort the resulting list of words, count each occurrence with the command uniq -c, re-sort the output in reverse numerical order, and finally extract the 25 most common words. In the figure we see the output of a previous test of the script still visible in a separate execution window. Note that the stream input port to *lcstring* has been connected to the border of the script; if this script is now packaged as a component, it will generate word frequency counts for files connected to that port.

Although the prototype falls short of demonstrating that a visual scripting interface is more "natural" or "user-friendly" than, say, a shell script (the equivalent shell scripts are typically much more compact), it should be clear that the logic and organization of applications are often better expressed visually than textually. In the case of this example, visualization is straightforward because the Unix scripting model is heavily dataflow-oriented. Many other visual programming tools[17], such as Fabrik[11], similarly exploit dataflow. Scripting, however, can be used for the construction of a much larger variety of applications.

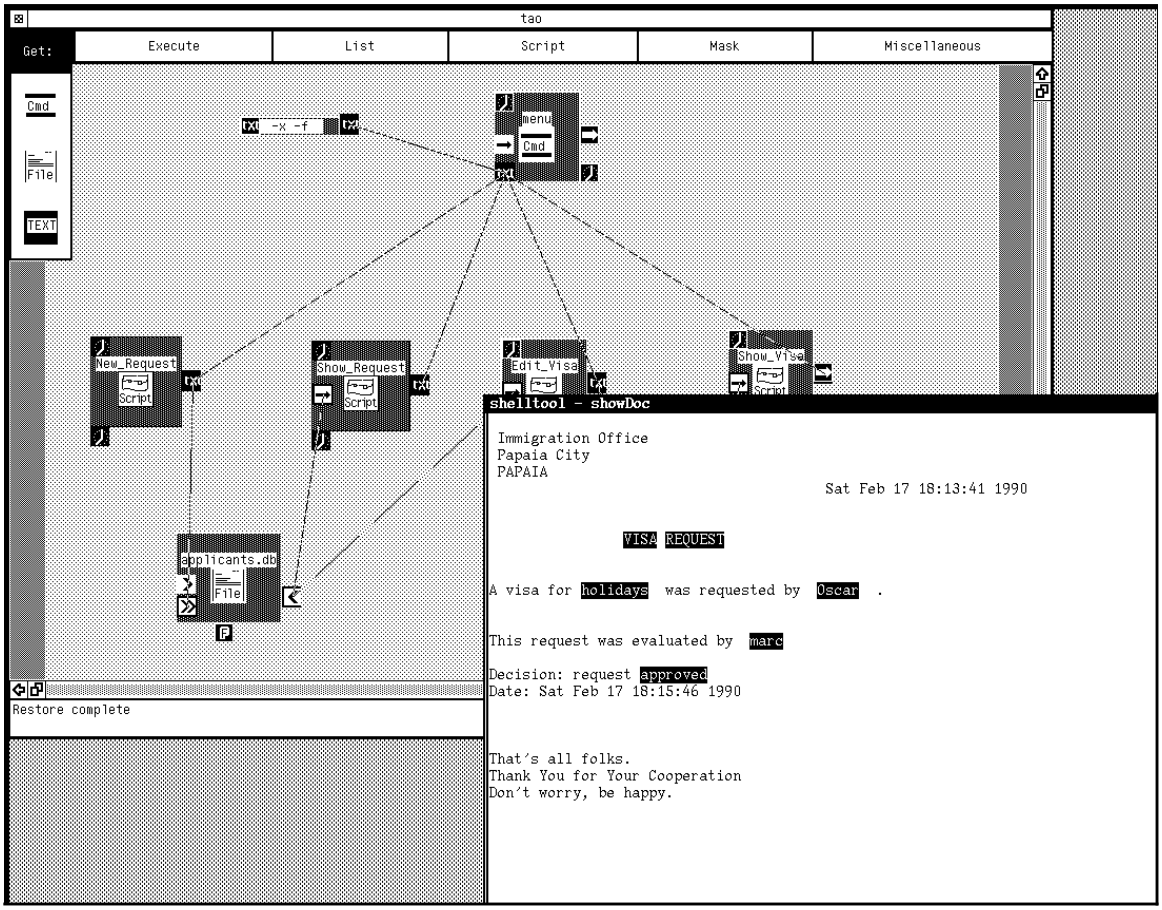

Figure 10: *scripting a visa approval application*

In Figure 10 we see the script of a simple *visa approval* application implemented as a hierarchy of menus. The top-level menu provides access to sub-menus for (1) generating new visa

requests and filing them in applicants.db, (2) retrieving pending requests, (3) approving or rejecting visa requests and (4) browsing the file of visas.

On the basis of the two examples, we can make the following observations:

- Visual software components have an enormous potential for self-documentation since the ports of a component are immediately visible. On-line help should be closely integrated to obtain quick explanations of the behaviour of a component or any of its ports.
- Direct manipulation should be exploited wherever possible. Forms and menus, as in the visa example, are better represented directly, rather than as boxes with ports.
- Careful integration of a browser is essential for locating software components quickly. It must be possible to generate queries of the sort: “tell me what I can connect here,” or “tell me what has been connected here in other scripts” automatically.
- Examples of existing scripts can be extremely useful as documentation and as a step towards designing a new script. In this sense, a script itself can function as a GAF (§2).
- The ability to toggle between editing and executing a script can be very useful when debugging.

The prototype was implemented in *Objective-C* [4] using the *ICpak 201*[1] user interface class library and runs on Sun workstations (3/50s, 3/60s and SPARCs). Visual scripts are translated to Unix C-shell scripts [1] and passed to the C-shell interpreter.

3.3 Object-Oriented Scripting Models

The examples of visual scripting presented thus far have emphasized either direct manipulation of office objects or visualization of dataflow using Unix streams. We have seen a variety of software components and different kinds of ports and links, but we have not yet demonstrated the relevance of object-oriented programming. We shall now argue that scripting models map naturally to a setting in which the software components are objects.

First, we note that all ports tend to be either input or output ports and have an associated type. Next, we note that links always fall into one of the following categories:

- *Value links:* provide a fixed value to an input port (e.g., options to Unix commands)
- *Reference links:* establish a client/server relationship between two components (e.g., between a Unix command and a stream, or between an office procedure and the input tray that triggers it)
- *Export links:* specify which ports internal to a script are accessible to external clients (e.g., when packaging a Unix script or when defining which fields of a form may be entered and modified by the user)

In the case of value links, the port type may be either a primitive data type or, more generally, that of a complex object. Reference links differ from value links primarily in that the objects passed by a value link become private to the receiving object, whereas a reference link passes a

1. O*bjective-C* and *ICpak 201* are registered trademarks of Stepstone.

handle to a shareable object. (Ultimately, of course, a reference is also a value.) The type of a reference link is simply "reference to object of some type," for example, a Unix command expects to be connected to something of the type *stream*, supporting the operations *read* and *write*. Office procedures expect to be connected to trays and form folders containing forms of a given type. Finally, an export link simply copies the type of an internal port to the interface of a script.

Note that at the level of scripting, we do not "pass messages to objects." Instead, we instantiate and *introduce* objects that communicate by "message-passing." Value links permit us to vary the pre-packaged behaviour of objects according to instantiation parameters. Reference links permit us to vary the way in which objects cooperate by specifying who is acquainted with whom. An object-oriented scripting model then, is a specification of the parameters available for creating objects and the types of the relationships that can be established between objects.

The essence of scripting is gluing elements into a whole, i.e. composing software components into executable applications. Object-oriented languages traditionally support two major composition principles, namely inheritance and message passing, both of which imply directed forms of communication between related entities. For scripting we need more flexible composition principles based on events and triggers. Objects can raise events, and other objects (*"sensors"*) wait on events and react by triggering a sequences of actions or further events. This differs from message passing in object-oriented languages since the client-server relationship is reversed: the object communicating an event is providing a service, not initiating a request. Sensors can be created, removed or even modified at run-time. They therefore add a degree of flexibility to composition, and can be used to model dynamic changes in the rules governing an executing system.

4. VISTA — An Object-Oriented Visual Scripting Tool

We are presently implementing *VISTA*, a second generation scripting tool to be part of an integrated suite of application development tools for the ITHACA environment. We shall briefly summarize some of the design requirements and architectural considerations. First, VISTA itself is intended to be an open system. In particular:

- VISTA will support scripting models for multiple application domains.
- Scripting should be (largely) target-language independent. Initially C++ and CooL (§2) will be supported.
- In the interest of portability, storage of scripts, scripting models and software component descriptions will be independent of VISTA.
- The environment must be open to new tools; context-switching and tool cooperation (i.e., through a programmers' interface) must therefore be provided.

A nucleus of common functionality is needed to serve as a foundation for tools to share resources and communicate. This functionality includes persistent storage, a *workspace* for objects shared by the tools, and a platform for a common user interface.

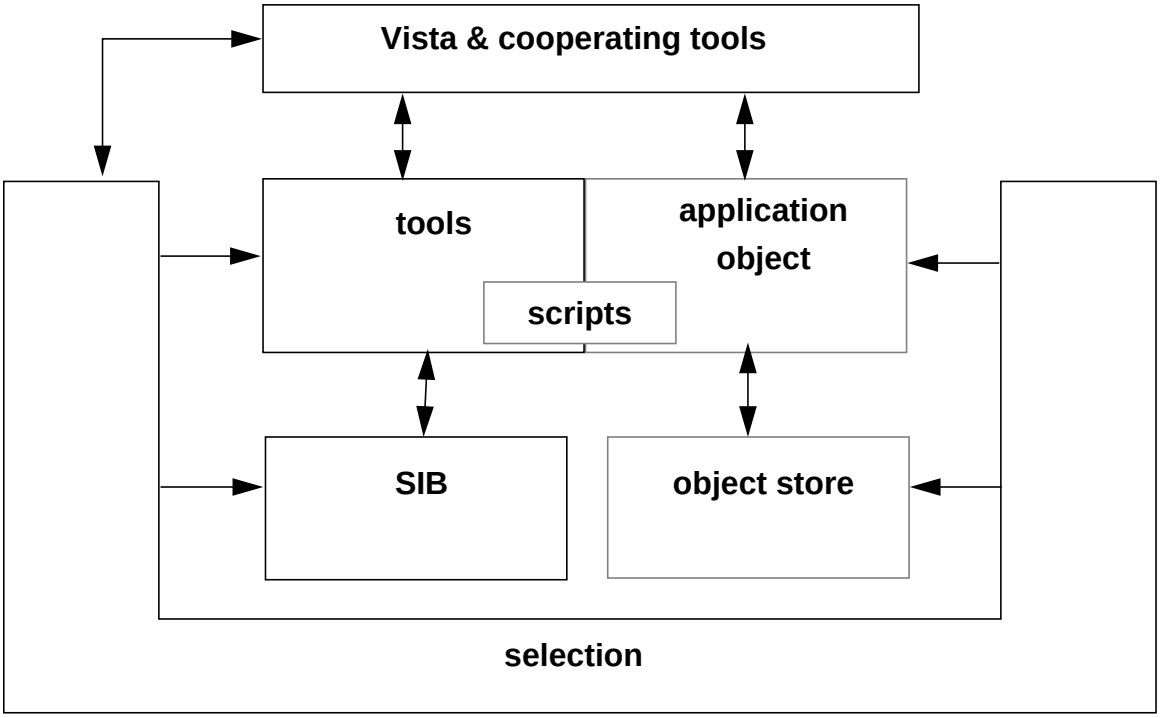

Figure 11: *architectural overview*

Persistent storage is divided into the *object store*, which contains the object instances of running applications, and the *Software Information Base* (SIB), which contains *descriptions* of software for use by the application development tools (see Figure 11). During a tools session, information will be retrieved from the SIB into the workspace either directly by the user browsing and querying via the selection tool, or indirectly, by the tools generating queries.

In the interest of developing a portable, common user interface, all tools will run as X clients, and the Motif toolkit will be available for user interface development. In addition, tools will be able to make use of *Labyrinth* (Laby), a general-purpose core for building graphical and CAD tools, being developed within ITHACA. Laby is a constraint-driven graphical editor that maintains graphs of connected “cells.”

VISTA will support user interaction modes similar to that of a powerful debugger that allows the application developer to toggle between editing operations and running the application. Furthermore, VISTA will generate script specifications (possibly in a compiled form) that will permit applications to run in a stand-alone mode (i.e., independently of the tools used for its creation and of the SIB).

5. Concluding Remarks

Asking how much can be done by scripting is like asking where automobiles can go. The answer is, everywhere there are roads. For scripting, the answer is, you can do practically everything, *provided* you have the right scripting model and adequate components. Scripting is not supposed to be complete in the sense of a general-purpose programming language. It is, however, open-

ended. In our environment, for example, we are using scripting ideas in several very disparate areas: animation, music, and “objectsheets.”

We have implemented a system for computer animation in which scripting controls the synchronization of animated actors [6]. The set of motions to be performed in a scene is specified hierarchically: local behaviours of actors, either programmed or scripted, can be encapsulated and then coordinated with other actors through the use of scripting operators that specify temporal relationships between participants. Although textual and not visual, this system uses scripting ideas heavily: it provides a generic framework for computer animation in which both the set of animated components and the set of temporal operators can be easily extended for meeting more particular needs.

Computer music is often confronted with problems of connections between components, either for controlling the flow of musical events or, at the sound generation level, for coordinating various primary devices like oscillators that are involved in a sound synthesis technique. A small prototype for structuring and modifying musical events, similar in several aspects to the animation system, is described in [5]. We are currently working on the integration of musical events in a more general user interface framework so that musical devices can be easily integrated into an application’s interface, either for sound feedback or as input devices (the rich variety of signals that can be generated on modern musical keyboards provides, in some situations, an interesting alternative to traditional input from the mouse).

Visual scripting resembles the way users work with spreadsheets. Some of the interesting features of spreadsheets are their intuitive presentation of data, their easy-to-use interface and their simple model of computation based on a global name space of cells. Another important feature is that they mix programming and execution: any change, be it to the value of cell or to a formula, triggers automatic global recalculation. We see potential to apply the spreadsheet metaphor to scripting, not only as a way of presenting object-oriented systems but also as a general philosophy behind a tool which would be easy to use and support a high degree of exploratory programming. We can apply the raster layout of spreadsheets to organize a workspace of collaborating components. Just as cells of a spreadsheet are related by formulas, cells of the *“objectsheet”* hold objects and are related through *scripting formulas* that link objects together to form an application. Objects may be directly manipulated via form-like representations. Modifications to an object can trigger computation in other cells. A Generic Application Frame can be seen as a “generic objectsheet” in which certain objects and formulas are already filled in. Finally, we feel that objectsheets can help to manage the complexity of large object-oriented systems since object addressability and relationships between components and subsystems are naturally addressed by varying the visual layout of objects on the screen.

References

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

[2] J. Bentley, “Programming Pearls,” Communications of the ACM, vol. 29, no. 6, pp. 471-483, June 1986.

[3] T.J. Biggerstaff and C. Richter, “Reusability Framework, Assessment and Directions,” IEEE Software, vol. 4, no. 2, pp. 41-49, March 1987.

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

[5] L. Dami, "Musical Scripts," in *Active Object Environments*, ed. D.C. Tsichritzis, pp. 162-171, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[6] 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.

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

[8] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz and X. Pintado, "Class Management for Software Communities," CACM, Fall 1990, To appear.

[9] D.C. Halbert, "Programming by Example," Ph.D. Thesis, Dept. of EE and CS, University of California, Berkeley CA, 1984, Also OSD-T8402, XEROX Office Systems Division.

[10] J. Hogg, O.M. Nierstrasz and D.C. Tsichritzis, "Office Procedures," in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, pp. 137-166, Springer Verlag, Heidelberg, 1985.

[11] 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.

[12] R.E. Johnson and B. Foote, "Designing Reusable Classes," Journal of Object-Oriented Programming, vol. 1, no. 2, pp. 22-35, 1988.

[13] G. Kappel, J. Vitek, O.M. Nierstrasz, B. Junod and M. Stadelmann, "Scripting Applications in the Public Administration Domain," SIGOIS Bulletin, vol. 10, no. 4, pp. 21-32, Dec 1989.

[14] S.K. Misra and P.J. Jalic, "Third-Generation versus Fourth-Generation Software Development," IEEE Software, vol. 5, no. 4, pp. 8-14, July 1988.

[15] 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.

[16] A-K. Pröfrock, D.C. Tsichritzis, G. Müller and M. Ader, "ITHACA: An Integrated Toolkit for Highly Advanced Computer Applications," in *Object Oriented Development*, ed. D.C. Tsichritzis, pp. 321-344, Centre Universitaire d'Informatique, University of Geneva, July 1989.

[17] Nan C. Shu, *Visual Programming*, Van Nostrand Reinhold company, 1988.

[18] M. Stadelmann, G. Kappel and J. Vitek, "ITHACA Visual Scripting Tool: A First Implementation Based on the UNIX Shell Scripting Model," ITHACA.CUI.89.E4.#5, Centre Universitaire d'Informatique, University of Geneva, December 8, 1989.

[19] D. Thomas and K. Johnson, "Orwell – A Configuration Management System for Team Programming," ACM SIGPLAN Notices, Proceedings OOPSLA '88, vol. 23, no. 11, pp. 135-141, Nov 1988.

[20] D.C. Tsichritzis, F. Rabitti, S.J. Gibbs, O.M. Nierstrasz and J. Hogg, "A System for Managing Structured Messages," IEEE Transactions on Communications, vol. Com-30, no. 1, pp. 66-73, Jan 1982.

[21] D. Tsichritzis, "Object-Oriented Development for Open Systems," Information Processing 89 (Proceedings IFIP '89), pp. 1033-1040, North-Holland, San Francisco, Aug 28-Sept 1, 1989.

[22] M.M. Zloof, "Query-by-Example: A Database Language," IBM System Journal, vol. 16, no. 4, pp. 324-343, 1977.