Scripting Applications in the Public Administration Domain

Gerti Kappel[1], *Jan Vitek, Oscar Nierstrasz*
Betty Junod, Marc Stadelmann

Centre Universitaire d'Informatique
University of Geneva
12 Rue du Lac, CH-1207 Geneva, Switzerland
E-mail: oscar@cuisun.unige.ch

ABSTRACT

Scripting is an approach for constructing open applications from prepackaged software components. A scripting model characterizes and standardizes the interconnection interfaces of software components appropriate to an application domain. We present a scripting model for the domain of public administration applications, and we provide a scenario of scripting applications in this domain. This scripting model is being incorporated into a prototype visual scripting tool which provides a graphical editing facility for interactively scripting applications.

Keywords: open applications, application scripting, object-oriented programming, scripting model, public administration domain.

Incentive

One of the outstanding challenges to software engineering is the ability to adapt software systems rapidly to the evolving needs of the user community. In the past, a customer with unique needs and stable requirements might have been willing to invest a great deal of time and effort into a specially-tailored software system which, when it finally ran, would run well. With the realization that the needs of most customers are perhaps not so unique, ''software reusability'' became a means to reduce the cost of building systems by amortizing the cost of developing generic software components and high-level programming tools. When we additionally consider customers with *evolving* requirements, however, rapid development and software reuse are no longer merely steps to reducing software costs -- they become imperative as means to maintaining and adapting open software systems.

The two most successful present-day techniques to streamlining software construction are the use of toolkits, and the use of specialized, high-level programming tools (i.e., languages, environments, etc.). The former technique is a natural evolution of software libraries, honed to a fine point through the use of object-oriented features such as multiple inheritance and genericity. The latter technique, on the other hand, packages some functionality in such a way as to make it very easy to build a limited range of

[1] Author's present address: Institute of Statistics and Computer Science, University of Vienna, Liebiggasse 4/3-4, A-1010 Vienna, Austria. Email: a4423dac@awiuni11.bitnet

applications very quickly. Spreadsheets, application generators, fourth generation languages, hypertext systems, expert systems shells, and so on, fall into this category (dBase III[2] and HyperCard[3] are perhaps the best-known examples of such systems).

Toolkits have the advantage that they are inherently open-ended: new functionality can always be added. The main disadvantage is that they can be hard to use. The interface is intended primarily for programmers. One cannot use a toolkit without writing some code. With very high-level languages and systems, however, the advantage lies in the ease-of-use, and the disadvantage in the closed architecture. Although programming interfaces to existing applications and databases are typically provided, it is not normally possible to extend the functionality of *the system itself.*

We propose *scripting* as an approach to application development that gives us the best of both worlds. In this approach, libraries of software components are designed to be "plugged together" according to an interconnection model (called a *scripting model*) tailored to a particular application domain. A scripting environment would provide the high-level (i.e., direct manipulation) support for designing applications, finding software components, and scripting and debugging applications. Such an approach encourages investment of energy in the construction of reusable software and generic tools (rather than in the development of monolithic, hand-coded applications), and, in the long term, permits the rapid construction of flexible, open-ended applications. Finally, the approach points us towards a long sought-after design method for object-oriented software, in which a well-designed software component is always part of a community of components that can easily be scripted together.

What is Scripting?

We have adopted the term *scripting* to capture the idea that applications should be constructed by plugging together existing components rather than by individually programming every line of code. A script identifies a set of *software components* and how they are linked together to accomplish some task.

Any piece of reusable software is parameterized by a number of values that must be provided either at compile-time or at run-time. These include type parameters, initialization values, arguments to operations, instance variables and methods (i.e., in object-oriented languages, when inheriting from a superclass), temporal ordering (of actions), input and output streams, and so on. In the context of scripting, we refer to such parameters in general as the *ports* of a software component. Within a script, *links* may be established between the ports of a number of selected software components, or between a port and a fixed value. A script may itself be encapsulated as a component by indicating which ports are to be visible to the clients of the script component.

Clearly, ports may not be arbitrarily linked. For example, ports representing input streams should only be linked to those representing output streams. Furthermore, it is unlikely that one will be able to script applications from components that have not been *designed* to work together. A *scripting model* identifies the kinds of ports, links and components that are appropriate for a particular range of applications.

A scripting model must be defined with great care, after acquiring a good understanding of an application domain. Similarly the implementation of the generic software components of the scripting model is an expensive, iterative process, since one must

[2] dBase III is a registered trademark of Ashton-Tate

[3] HyperCard is a registered trademark of Apple Inc.

target not just a single specific application, but a range of existing ''precursor'' applications as well as anticipated future applications. The payoff is in terms of extremely reduced development and maintenance costs for future applications. Not only is coding and debugging effort reduced, but application specification and design are simplified by being restricted to the context of an existing scripting model. Finally, the approach is open-ended, since one can not only easily incorporate new functionality by adding new components conforming to the existing scripting model (i.e., with the same kinds of ports), but one may extend the scripting models themselves, if necessary, by adding new kinds of ports and ways of linking components.

Up to this point we have used the term ''software component'' freely. What are these components exactly and how are they built?

Very generally, any piece of software or template for software could conceivably be viewed as a reusable component, but from the point of view of scripting, it is important to encapsulate functionality in such a way as to promote reuse through a well-defined, standard interface. As such, we take *objects* specified in an object-oriented programming language as our basic software components. We believe that objects are an ideal basis both from the point of view of natural modeling of applications, and from the point of view of mechanisms for code reuse. Templates for objects (i.e., classes, generic classes) and configurations of cooperating objects (i.e., scripts) may then be viewed as higher-level components.

Although scripting exploits object-oriented techniques, it is distinct from object-oriented programming per se in that we seek to separate the responsibility of programming the basic components from that of reusing components to build applications. It is by this means that we concentrate programming effort where it has the most benefit, and we raise the level of programming in such a way as to streamline development and adaptation to changing requirements.

Scripting and the Public Administration Domain

The development of a scripting environment for constructing applications is an integral part of the ITHACA project (No.2121), supported by the Commission of the European Communities under its ESPRIT program. The ITHACA project aims to produce an integrated application development and support environment based on the object-oriented approach [Pröf89].

Within ITHACA several ''demonstrator'' application domains are identified to validate the approach and the usefulness of the environment in constructing real applications. Out of these target application domains we have chosen the *Public Administration Domain* (PAD) to develop a scripting model. The main reason behind this decision is that we can reuse existing domain knowledge and expertise in building PAD applications [Kapp89a], which we have gained by studying an application generation system for this domain.

Applications in the public administration domain deal with provided automated support for work in public institutions. This automation regards highly repetitive tasks performed by public servants. These are typically related to filling-in, sending, and storing office documents.

PAD Terminology

We explain the terminology used for introducing the PAD scripting model with an example, which is the ''Organization of a Public Event'' [Kapp89b]. A glossary of terms is given in Appendix A.

The public administration taken as reference in this example is responsible for handling external requests concerning the organization of public events, such as demonstrations, parades, and public celebrations. The request has to be approved both by the police and by an internal administration department, otherwise it is rejected.

An office is generally structured as hierarchy of *office units*. The office itself, in our case a specific public administration, is the root of this hierarchy. The intermediate nodes represent departments, divisions, project groups, and other existing working units. A leaf node is the most specific office unit, to which one or more *office workers* are assigned. The edges between the nodes are directed and represent part-of relationships. For each office unit various responsibilities and privileges are defined, which are inherited down the hierarchy.

Each office worker performs certain *steps* to meet these responsibilities. All the steps an office worker might perform are referred to collectively as the *workspace* of the office worker. A *task* requested by a client is handled by the office workers following a predefined *office procedure*. The task we are interested in is getting the permission for organizing a public event.

Information to be processed in an office procedure is encapsulated in *office documents*, some of which may be highly structured *forms*. The office documents concerning a particular task are referred to collectively as a *case*.

An office procedure can be broken down into a number of individual *steps*, each of which is either performed by a single office worker or processed automatically. Office procedures, then, are implemented by coordinating both automatically executing steps and steps which are based on interaction with an office worker.

The office procedure for accomplishing the example public event handling task entails the following steps: An office worker enters all the information concerning a particular case by filling in the `EventRequestDocument`. Then a letter is printed and sent to the police to obtain approval for the event. If the police department accepts, the case is presented to another office worker who will decide for the administration. The request for organizing a public event can be either accepted or rejected. Finally the appropriate letter will be generated and sent to the client.

PAD Scripting Model

A scripting model identifies the kinds of ports, links, and components that are appropriate for a particular domain. This does not prevent us from reusing software components in several application domains, as long as they conform to the relevant scripting model.

In the PAD scripting model the interface of an object is re-packaged. The operations composing the object's interface are mapped to four kinds of ports: *attributes, acquaintances, actions and events*.

Attributes are the visible characteristics of an object. For example, the color and the size of a `Document` object when it is displayed.

Acquaintances are references to external objects with which the object will cooperate. For example, a `Document` object has to be acquainted to a `Printer` object in order to be printed.

Actions are the subset of the operations of an object that can be used for scripting. For example, the actions of a `Document` object are `edit`, `print`, and `display`.

Events occur whenever an object changes state. Upon occurrence of an event, a list of actions may be invoked. For example, each alteration to a text field of a `Document` may reset the date field of that `Document`. The action-list associated with an event may be extended for individual object instances as opposed to the entire class. In the preceding example, the event-action couple was defined for a particular document object and not for the whole class.

In Figure 1, the mapping from the set of operations of an object to its ports used in scripting is depicted. The former specifies the programming interface of the object, the latter specifies its *scripting interface*. Note that the kinds of attributes, acquaintances, actions, and events of an object are specified at the class level.

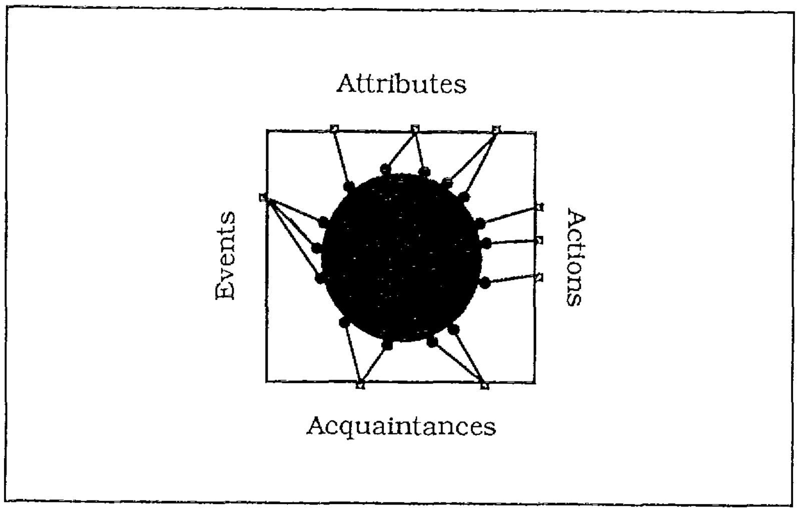


Figure 1: *The set of operations of an object is mapped to four kinds of ports (attributes, acquaintances, actions, events) used in scripting.*

Our study of the public administration domain has lead us to the design of a hierarchy of classes to implement scripting in this domain. In Figure 2, part of the class hierarchy is presented. The design distinguishes between application classes that represent the application domain knowledge and model classes that represent the application development knowledge. The application classes mainly model real world entities, like documents, letters, and clients. The model classes capture information about scripted applications, like steps, office procedures, and work spaces.

Coming back to the remark at the beginning of this section about the use of software components in several domains, it is worth pointing out that the PAD scripting model comprises both domain-independent (i.e. generic) scripting features, and PAD-specific scripting features. The former is the view of a software component being an object with a scripting interface, the latter are the kinds of software components and how they are linked together to build an application.

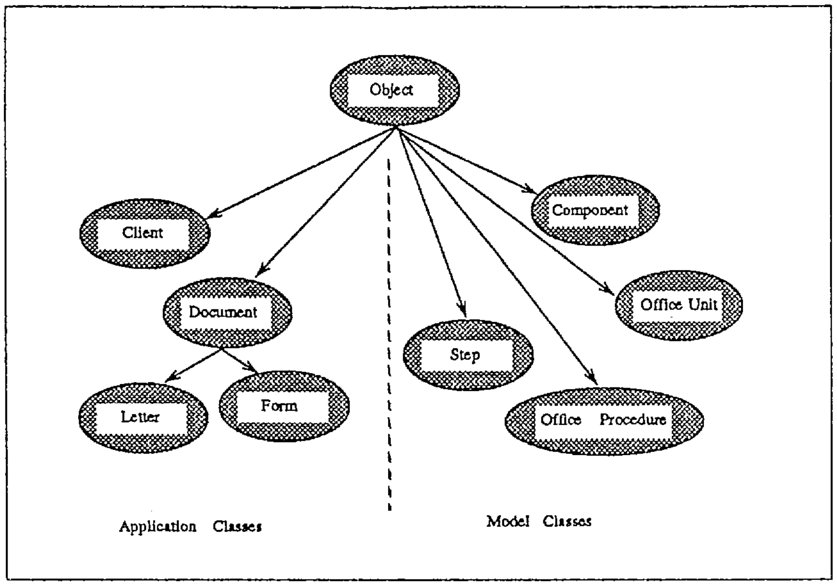


Figure 2: *The design of the PAD classes distinguishes between application classes that represent the domain knowledge, and model classes that represent the application development knowledge.*

PAD Scripts

Scripting PAD applications means constructing ready-to-run applications using a set of pre-packaged software components. These software components are application classes and model classes as outlined in Figure 2.

We have identified four levels of scripting which are composed out of different kinds of software components. These are:

- scripting a *step*: defining a unit of work within an office. At this level the components are application classes.
- scripting an *office procedure*: specifying a coordination of different steps to fulfill a task. The components are the steps scripted in the first level.
- scripting the *workspace* of an office unit: setting the responsibilities and privileges of office units. The possible components are office procedures, steps, and office units.
- scripting the *user interface*: defining the user interface of the application. The components are both application classes and model classes.

The output of each of these levels are scripts that have to be interconnected in order to build a PAD application. For the purpose of clarity and readability, however, we will investigate each level independently. To be able to compare and combine the various scripting levels we distinguish for each level between (1) kinds of possible components, (2) linking ports of the components, and (3) possible encapsulation of the script as a component for further reuse.

For all four scripting levels we assume an interactive scripting environment for building scripts, storing and retrieving components, and executing scripts. Note that when an object class is made available as component in the scripting environment, the mapping from its set of operations to its scripting interface has to be provided. A concise description of a prototype scripting environment is given in the section on ongoing work.

Scripting a Step

A step represents a unit of work either processed by a single office worker or triggered and executed automatically. The functionality of a step is defined by a set of object components and actions which are performed on these components in a predefined order. Scripting a step entails:

- *Selecting* application classes from the component library. The components of a step are application objects of the public administration domain, such as a `DocumentTemplate` for a certain kind of document, a `Date` field capable of representing dates in various formats, or an `Archive`, keeping track of all the documents coming into the office.
- *Linking* the ports of the components. For each component an object instance is either created or retrieved. The ports provided in the component's scripting interface (attributes, acquaintances, actions and events) may be linked to fixed values or to compatible ports of other components. The control flow of invoked actions is specified through linking control ports.
- *Encapsulating* the script as a parameterized step component for re-use when scripting office procedures.

The objects referred to in a script must correspond to actual object instances when the script itself is instantiated. These objects may either be pre-existing objects, or objects created by the script. To this end, we provide each application object component with *object identifier* (oid) input and output ports. The oid input port may either be linked to the oid output port of an existing object, or it may be left unbound, in which case the object will be created when the script is executed.

To increase the potential of reusability of a step, certain ports of the components which compose the step might be left unbound until the step is executed. For this purpose, it is possible to specify input ports of a step, which are linked to input ports of components. We can also export values from within the step by specifying output ports of the step, which are linked to output ports of components. These *user defined ports* might be of basic type (like integer, real, char, and boolean), or of type object.

Besides these user-defined ports a step component has *predefined ports*, defined in the PAD scripting model, for linking attributes, acquaintances, events, and actions. A step component has a default `execute` action which implements the functionality specified in scripting the step.

As an example, consider the `RequestInitialisation` step of our office procedure described earlier. It is the first step of the office procedure which should create a document containing all the information for the public event request. This document will be filled in and added to the archive where all requests are stored. The step will have as only output port the created document.

To script this step we start by choosing two components out of the component library, the `EventRequestDocument` class and the `Archive` class. We leave the oid input port of the former unbound, i.e. a new document instance is created whenever this step is executed. We bind the name of an existing archive to the latter, i.e. each created document instance is stored in this specific archive. Then we link the ports of these components. The attribute ports of the `EventRequestDocument` are bound to fixed values. They specify the size of the document, and the fonts used for printing. The only acquaintance port of the document will be connected to a `Printer` object. The action ports are linked to fixed values which represent the actions invoked on the object component. Choosing actions for a component is equivalent, i.e. translates to sending a message to the object bound to the component, when the step is executed. The `edit` action is selected for the `EventRequestDocument` that enables the filling in of the document. For the `Archive` component the `add_new_document` action is selected. The `onEditing` event port of `EventRequestDocument` will be linked to the `update_document` action of the `Archive`. In general, we can specify a set of actions of several object instances to be invoked (i.e., equivalent messages are sent to the object instances) as response to an event.

Scripting an Office Procedure

An office procedure represents a description of how to accomplish a task. This description is a configuration of steps which together provide the functionality of the office procedure. Steps which need user interaction can be executed in parallel, but they have to be executed in a predefined order concerning a specific case instance. Scripting an office procedure entails:

- *Selecting* step components from the component library. There are no other kinds of components than step components in an office procedure. However, a step component might be used several times within the same office procedure.
- *Linking* the ports of the components. For each step component, we may link its event port to the `execute` action of the successor steps which are automatically triggered on completion of execution of the current step. These successor steps must not depend on user interaction. In addition, we may bind data input and output ports which represent case specific information.
- *Encapsulating* the office procedure script as office procedure component which may be reused for scripting a workspace.

The links between steps represent either control flow – when the execution of successor steps is triggered – or data flow. For supporting conditioned data flow, we encapsulate the evaluation of conditions within steps and model an output port for each possible result. Thereby OR branching of data flow is supported by feeding selected output ports of a step.

Concerning multiple occurrences of the same step in an office procedure, we encapsulate this information with the step. For each occurrence of a step in an office procedure we keep its particular bindings (successor steps, values bound to data ports, etc.). When a step is executed, in fact, a particular occurrence of the step in an office procedure is executed.

We outline the script of the example office procedure for handling public event requests. We distinguish six different step components: `RequestInitialisation`, `PoliceLetterPrinting`, `PoliceAnswerProcessing`, `PublicAdmin-DecisionTaking`, `RejectionLetterPrinting`, and `ApprovalLetter-Printing`. They are linked in the following way: the initialisation step triggers the printing of a letter which is sent to the police station. In addition, the output port of the initialisation step referencing the created document is linked to an input port of the letter printing step which is bound to a document component. Whenever the answer of the police arrives it is processed by the `PoliceAnswerProcessing` step. This step has two output ports, both of the same type which is the type of the document manipulated in the answer processing step. The step evaluates the answer letter sent by the police, updates the `EventRequestDocument` with the received answer (approved or rejected), and transmits the document to either of its output ports for approved requests and rejected requests. The ''rejection'' output port is linked to an input port of

`RejectionLetterPrinting`. The "approval" output port is linked to an input port of `PublicAdminDecisionTaking`, a step for deciding on the request based on internal information of the public administration. This step has again two output ports of the same type which are linked to different steps. The approval output port is linked to the step for printing the approval letter, and the rejection output port is linked to the step for printing the rejection letter. Note, the `RejectionLetterPrinting` step is used two times within our example office procedure.

Scripting the Workspace of an Office Unit

The workspace of an office unit comprises the set of steps an office unit is responsible for. When scripting the workspace the information about what an office unit is expected to do is specified, but not how it will do it. Scripting a workspace entails:

- *Selecting* components representing office procedures, steps and office units from the component library.
- *Linking* the ports of the components. Both office procedure and step components have an acquaintance port which is linked to office unit components. Hence, workspaces of several office units can be scripted at a time.
- *Encapsulating* the workspace script as work space component which is reused when scripting its interface.

Office units are organized hierarchically. The workspace of an office unit is inherited by all office units which are part of this particular unit. Office workers are assigned to the leaf nodes, thus specifying what the persons working in the office are supposed to do.

Concerning the assignment of a step to an office unit, the user can choose between assigning a step independent of its use in several office procedures, assigning a step as it is used in a particular office procedure, and assigning an occurrence of a step in a particular office procedure.

The concept of an office unit resembles the "user object" in [Nier85]. For each user object a set of allowed operations may be defined. In our approach, this functionality is accomplished when specifying the workspace of an office unit. The workspace can be both created and updated via scripting.

Scripting the User Interface

We assume a graphical user interface to our scripting environment, both for the development of scripts and for their execution. Therefore, for all visible software components, a graphical representation has to be specified. How this can be accomplished depends on the user interface construction paradigm, hence, on the user interface toolkit we use. But the minimal requirements will be that software components be built in a way which makes it possible to link them with a graphical representation, and that generic user interface classes, like `Button`, `Menu`, and `Scrollbar`, are available. For example, scripting the user interface of a particular work space entails:

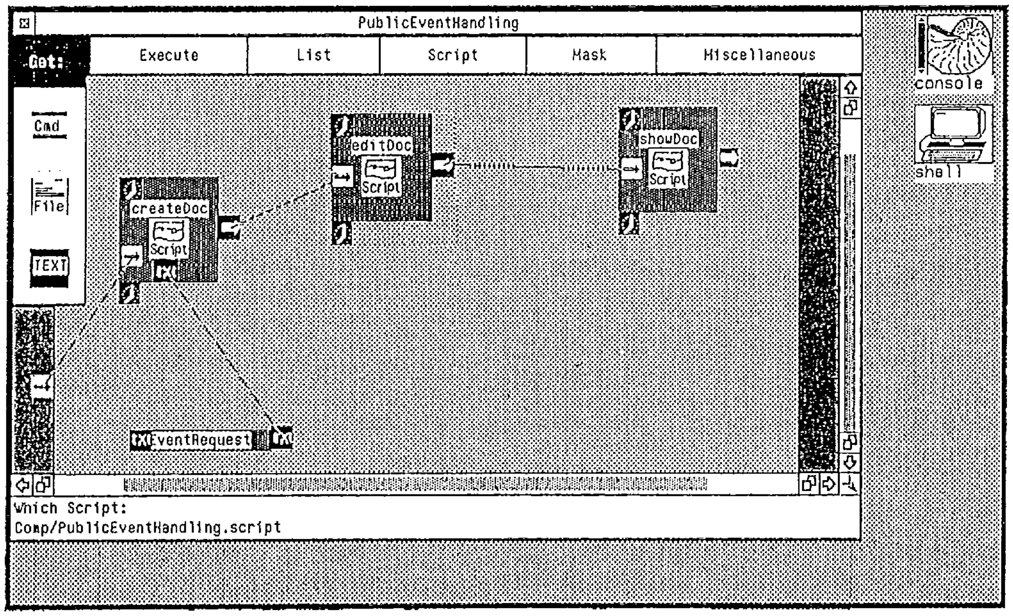
- *Selecting* user interface classes and the workspace component from the component library.
- *Linking* the ports of the components. The component will be equipped with a special-purpose port for linking its graphical representation.

Note that there is no encapsulation of a user interface script supported, since we do not use such a component in scripting other scripts. Two existing user interface construction systems, InterfaceBuilder of NextStep [Thom89] and Fabrik [Inga88], support scripting-like features for building user interface objects and linking them to application objects.

Ongoing Work

The notion of scripting as an aid for flexible application development has been introduced. A script specifies a set of software components and how they are linked together to build a specific application. Which kinds of software components, ports, and links are supported is defined in a scripting model. We have developed a scripting model suitable for building applications in the public administration domain. The main motivation behind this choice was that we could reuse existing application domain knowledge and existing application development knowledge which is key for constructing the "right" software components and linking mechanisms.

The construction of PAD scripts will be supported by an interactive integrated scripting environment. This makes it easy to link the several levels of scripting since everything is done in the same environment. So far, we have implemented a visual scripting tool, which allows the graphical construction of scripts, their encapsulation, and reuse as components in building more complex scripts [Stad89]. For first demonstration purposes, we used the UNIX shell scripting model as underlying scripting model. This choice was purely pragmatic: software components and script interpreter were readily available. An adaptation of the prototype for PAD scripting is underway. Figure 3 is a glance on the visual scripting tool in use. It shows part of the `PublicEventHandling` office procedure composed out of three step components.



Which Script:
Comp/PublicEventHandling.script

Figure 3: *A glance on scripting the office procedure for public event handling using the visual scripting tool.*

Furthermore, we are investigating the support of several interfaces for one software component and domain specific standard protocols for communication between components.

References

[Inga88]

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

[Kapp89a]

G. Kappel, ''Reusable Software Components for Applications of the Public Administration Domain,'' ITHACA.CUI.89.E.#12, Sept. 25, 1989.

[Kapp89b]

G. Kappel, ''Proposed Reference Example for the TWG in ITHACA,'' ITHACA.CUI.89.E.#7, April 30, 1989.

[Nier85]

O.M. Nierstrasz, ''An Object-Oriented System,'' in *Office Automation*, ed. D. Tsichritzis, pp. 167-189, Springer.

[Pröf89]

A.-K. Pröfrock, D.C. Tsichritzis, G. Müller, 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.

[Stad89]

M. Stadelmann, G. Kappel, J. Vitek, ''Ithaca Visual Scripting Tool: A First Implementation based on the Unix Shell Scripting Model,'' ITHACA.CUI.89.E.4.#5, Dec. 8, 1989.

[Thom89]

T. Thompson, ''The NextStep,'' Byte, vol.14, no.3, pp. 265-271, March 1989.

APPENDIX A -- Glossary

Task:

A task is an identifiable job to be done in an office. Examples of tasks can be as simple as the filling-in of a form, or as complex as the organization of a public event.

Office Procedure:

An office procedure is a (procedural) description of a way to accomplish a task. Such a procedure will typically break down into a partially ordered collection of steps in which various office documents are created, modified and destroyed.

Step:

A step is a logical unit of work within an office procedure, entailing the transformation of a set of office documents by a single office worker. That is, within a step, an office worker starting with some set of office documents relating to a particular task, modifies those documents, possibly creating or destroying some. Examples are ''sending a memo'' or ''taking a decision.'' Steps may, in some cases, be partially or fully automated, e.g., filling in today's date. (The precise of a step is left undefined, but in practice should correspond to ''the smallest unit of work worth talking about.'')

Office Documents:

An office document is a multimedia representation of structured and unstructured information. Office documents are typically grouped into classes which share a common organization, such as standard fields for creation date, author, etc., and a common format for presentation of text etc.

Form:

A form is a very regularly structured office document, for example, containing only a fixed number of fields, some repeating fields, and only a limited capability for attaching unstructured text. Forms may be ''intelligent,'' that is, they may require fields to be filled in a particular order, restrict visibility of information to certain office workers, automatically fill in certain fields or forward themselves to another office worker upon completion of a step. Forms may also be implemented as views on a database, in which case they have no autonomous identity.

Case:

A case comprises the collection of office documents concerning a particular task. The documents of a case may, at any time, be distributed amongst several ''locations'' in the office (hence the use of the term ''case'' rather than ''folder'' or ''dossier'').

Office Unit:

An office unit is an organisational unit within an office for which various responsibilities and privileges are defined. The office itself is organized as hierarchy of office units, the root being the whole office and the leaves corresponding to the most specific unit. Each leave office unit is assigned to a set of office workers.

Office Worker:

An office worker is a person responsible for certain work to be done within an office.