A Practical Framework For Interactive Applications in Guide, an Object-Oriented Distributed System.

Véronique Normand.

Unité Mixte Bull-Imag, Z.I. de Mayencin, 38610 Gières, FRANCE.

e-mail: normand@imag.imag.fr

Abstract. This paper describes our experience in constructing a tool to help and guide programmers in the development of interactive applications for Guide, a strongly typed object-oriented language and distributed environment. This tool consists of a set of reusable components designed as a practical evolution of prevailing theoretical conceptual models; we take into account requirements stemming from the Guide environment, such as the sharing of objects between applications, as well as the conformance to standard high-level graphical environments.

Introduction

Building interactive applications is a complex task which many different tools seek to simplify: user-interface toolkits, language-based User Interface Development Systems, graphical specification tools, automatic interface creation environments... (see [Myers 89] for a survey of the currently available tools). Each of these tools is designed for a precise development environment and target user, be he programmer, ergonomist, or graphics artist; it pursues two main goals: provide a design methodology, and limit the development effort of the target user.

Object-oriented environments are particularly well adapted to interactive application development tools since they promote software reuse and (for some of them) the separation of abstraction and implementation. They offer the notion of *framework* as a set of abstract and concrete classes embodying an abstract design but also providing a concrete reusable platform. Frameworks are tools designed for programmers; they are a means to enhance design reuse as well as code reuse in a programming environment.

This paper presents a framework designed as the keystone of a language-based User Interface Development System for the Guide object-oriented environment; it is a concrete realization of a conceptual model in the form of reusable components that will assist the programmer during the design as well as in the implementation of an interactive application.

This paper is organized as follows. Section 1 gives an overview of the Guide language, highlighting requirements specific to the target applications; it also describes our pre-existing working environment in terms of window management system and graphic toolbox. Section 2 defines our conceptual model for an interactive application as a practical object-oriented evolution of current prevailing models. Section 3 describes our implementation. Section 4 presents an evaluation as well as some experience in the use of our framework; elements of an evaluation of the Guide model and language in terms of reusability enhancement are also provided.

1. Context and requirements

Programming interactive applications in Guide raises general needs, mainly the need for a design methodology, and the need for a reusable implementation of services common to all applications. The Guide environment generates specific requirements however, stemming from its object model and the nature of its target applications. Providing access to a high-level graphical environment from within Guide is also part of the requirements.

1.1 Overview of the Guide system and programming language

The Guide programming language [Krakowiak 90] has been designed and

implemented for the programming of distributed applications. Its run-time support is provided by an object-oriented distributed operating system (also called Guide) designed and implemented at the University of Grenoble as a joint project of Laboratoire de Génie Informatique and Bull Research Center [Decouchant 88]. The Guide system and language implement the distributed object-oriented architecture defined by Comandos (Construction and Management of Distributed Open Systems), a cooperative project of the ESPRIT European program [Horn 87, Alves Marques 88]. The objective of Comandos is to design and implement a platform for the programming of large and complex distributed applications which manipulate permanent data.

The object model implemented in Guide is characterized by the following main features:

- Objects are typed. A type describes a behavior shared by all objects of that type in terms of the signatures of the operations applicable to the objects. Types are statically checked; in some cases, a dynamic check is necessary.

- A class defines a specific implementation of a type. Classes are used to generate instances of objects. Several classes implementing a type may coexist; an implementation of a type may dynamically be replaced by another one.

- Subtyping defines a hierarchy of types. The subtyping hierarchy is paralleled by a subclassing hierarchy, with single inheritance. Generic types and classes are provided.

- Objects are persistent. They are named by system references (system-wide uniquely generated internal names). Complex structures may be constructed by embedding references to objects within other objects.

In addition, the language supports concurrence and allows synchronization constraints for the sharing of an object by concurrent activities to be expressed.

Guide is currently implemented as an experimental version on top of Unix[1], on workstations connected by a local area network.

1.2 Guide applications

Expected pilot Guide applications are collaborative applications such as software development environments and structured document management systems. Several toy applications have already been programmed; real-size interactive applications are currently being developed, e.g. a browser for types and classes, graphical shell tools, a mailer, an agenda, etc.

Guide has been designed to support the programming of applications manipulating permanent sharable data, which may also be distributed. Object persistence provides for the automatic storage of the data; system references as well as internal memory and execution management allow the sharing of any object by any number of concurrent activities on the network; synchronization constraints help programmers control concurrent accesses to an object. Transparency of object location is ensured by the object management system, so as to free the programmer from the need to be aware of distribution.

Guide also supports concurrence inside an application, so as to allow "parallel" execution of application tasks, possibly on different distributed processors.

Sharing of objects and concurrence are features of high interest to us for they mean specific requirements in the building of interactive applications. Sharing of objects implies that the same data may be visible through different views in the interface of a particular application but also in the interfaces of other applications executed at the same time throughout the network. This requires mechanisms which support the coherence of all views of the shared data.

Concurrence within an application affects choices concerning control distribution and communication schemes between the different components of the interactive application.

1.3 Graphical environment

[1] Unix is a registered trademark of AT&T.

[1]X Windows is now available as an established window system standard within Unix environments. Several user-interface toolkits are also provided along with X Windows; these toolkits are libraries implementing graphical interactive elements such as menus, pushbuttons, scroll bars, etc.

We chose to take advantage of these toolkit facilities, rather than develop our own dialog elements on top of low level X libraries. The advantage is that of all user-interface toolkits: relatively high level graphical services, homogeneity of external look of application interfaces, etc. The main drawback is the integration of foreign structured elements into the Guide object model.

2. Conceptual Model

2.1 General architecture models

Models for the architecture of interactive applications all agree on the necessity to separate the application from its interface; the main differences between these models are relative to three different aspects: abstraction level in the communication between application and interface, control distribution between the two components, and dialogue modeling. Two general theoretical models are considered as prevailing: the language model and the multi-agent model.

The language model considers human-machine interaction as consisting of three distinct levels: lexicon, syntax and semantic, just like human dialog. The application concentrates the semantics, while the interface acts as a lexical and syntactic server. This model is enticing for its simplicity and generality; however, its strict application leads to centralizing the handling of semantic, syntactic and lexical aspects, which does not fit easily with interactivity needs: input is processed as a sequential flow of data, forbidding multiple user activity threads. The language model must be considered more as a frame of reference than as an implementation model.

[1]X Windows is a registered trademark of MIT.

The multi-agent model considers an interactive application as a set of co-working specialized agents taking events as input and producing events as output. This model allows and promotes the handling of multiple threads of user activity, but makes no assumption on the manner in which an agent is organized.The principles of this model are closely related to those of object-oriented programming: an agent may be seen as an object the interface of which acts as the input protocol the agent responds to; the output protocol of the agent corresponds to the activation of the interface of other objects.

Our model is a combination of these two general models: a multi-agent model with the division of an application into general distinct knowledge domains.

2.2 Our model for Guide applications

The main principles of our model are the modularity resulting from a strict separation of concerns, and the distribution of control between interface and application.

An interactive application is divided into three general components corresponding to three precise knowledge domains within the application: the task domain, the interaction techniques domain, and the coordination domain.

- The task domain is the knowledge domain specific to the particular application, independent of user interface concerns. It is embodied in the *Task Component* (TC) of the application. The task domain may be seen as the semantic level in the language model.

- The interaction techniques domain is the knowledge domain of input and output. This is where the actual gathering of input from the user and the presentation of output is handled. This domain is embodied in the *Interaction Component* (IC) of the application. The interaction techniques domain may be seen as the syntactic and lexical level in the language model.

- The coordination domain is at the confluence of the two previously described domains. This is where the communication between TC and IC is realized. This domain

is embodied in the *Coordination Component* (CC) of the application.

The Task Component has no knowledge of the existence of the Interaction Component, and vice-versa. The division into three components is not only logical but also physical; this allows each component to be developed by a distinct programmer.

We now describe the internal organization of each component, highlighting the communication schemes between Task and Coordination Components, and between Coordination and Interaction Components. Each component is structured into specialized agents; an agent is characterized by the formalisms it understands and generates and the protocols it responds to. An agent is implemented as a Guide object.

Figure 1 illustrates a global view of the structure of interactive applications. Figure 2 summarizes the communication schemes between the agents of the three components.

Task Component :

The structuring of the Task Component is specific to the application. However some rules must be obeyed while designing this component in order to ensure correct communication with the Coordination Component.

The Task Component consists of a set of objects designed for the specific needs of the application. Part of these objects are *hidden*, i.e. their existence is transparent to the Communication Component. Among the remaining objects, some only offer operations corresponding to commands available to the user, while others hold data that may be visible in the user interface as well; the latter objects are called *displayable* objects. The data held in a displayable object is accessed by the Coordination Component through the type of the object (either as a variable declared as visible or through an explicitly declared access method); any modification of this data must be forwarded to the Coordination Component, so as to allow the interface to be updated. Any displayable object of the Task Component is potentially sharable between any number of applications, and therefore may be accessed by any number of Coordination Components; such an object (as well as other objects of the Task Component) is programmed without referencing any Coordination Component; the actual referencing is supported by our framework.

Task Component objects can be viewed as agents handling no other formalism than their own specific task level formalism; the protocol they respond to is defined in their types; the protocol they generate corresponds to the warning of modifications to the data of visible objects. Their only interlocutors are agents of the Coordination Component.

Interaction Component :

The interaction Component consists of two layers of objects: Interactive Objects (IO), and Views. IOs are objects resulting from the integration of high-level graphic toolkit elements into the Guide model, e.g. buttons, scroll bars, labels, etc. These objects are the actual gatherers of input, as well as the presenters of output. An Interaction Object acts as an agent between the X Windows graphical server and a View; it has its own graphic-toolkit-dependent formalism.The role of an Interaction Object is to translate user input into toolkit level formalisms, and toolkit level formalisms into effective graphical output.

A View object creates, displays, updates and manages a precise part of the image of the user interface through the management of a set of IOs. A View acts as an agent between Coordination Component objects and IOs. The role of the View agent is to translate Coordination formalisms into Interaction level formalisms, and vice-versa. A View acts as a filter of user input it perceives through IOs; local feedback may be implemented in the View. The View also manages part of the interaction state, at its level. The View responds to a double protocol: update and management from a Coordination Component agent, and handling of user input as transmitted by IOs. Hierarchical relationships are defined between the Views so as to support inclusion relationships between the image parts they manage.

Coordination Component :

The Coordination Component is structured as a set of Coordination Objects (CO). A CO controls a set of Task Component objects and a set of Interaction Component Views. The role of a CO is to translate Task formalisms into Views formalisms, and vice-versa; it reacts to modification warnings sent by displayable objects, and creates and modifies Views so as to update the image of the user interface; it also reacts to actions of the user as forwarded through the Views, and executes the corresponding operations in the Task Component. The CO also manages the state of the dialogue at its level. The CO responds to a double protocol: handling of Task Component modifications, and handling of user requests, as transmitted by Views. This protocol may be added application-dependent operations so as to support relationships between COs, such as synchronization relationships.

Control Distribution :

We propose a mixed control configuration: part of the control remains in the Interaction Component, so as to closely "listen" to user actions; part of the control remains in the Coordination Component, so as to asynchronously detect modifications to the Task Component, and appropriately update the display .

2.3 Related Models.

Several models have been proposed as a combination of the general language and multi-agent models; we present two of them: MVC (Model, View, Controller) of Smalltalk [Goldberg 84], and PAC (Presentation Abstraction Control) [Coutaz 87] [Coutaz 88], and compare their concepts with ours.

MVC is an application structuring and implementing paradigm that separates each application component into three elements: the part that represents the model of the underlying application domain (Model), the part that handles the output presentation of the model to the user (View), and the part that deals with the way the user interacts with this presentation (Controller). Each M, V and C element is implemented as a distinct Smalltalk object.

MVC offers a clear separation between input and output handling, which are mingled in our Interaction Component. Such a separation in distinct V and C objects is not always desirable, particularly when input and output are closely related as is the case in feedback implementation. In reality view and controller classes are often designed in consort; as shown in [Krasner 88], the definition of “pluggable elements”, or graphical elements reusable through instantiation rather than subclassing (e.g. push buttons and windows) tends to present View Controller pairs as single entities; the Controller is “hidden” behind the View; the parameters relative to the Controller are passed to the View.

Our Task Component objects can be compared to Model elements. The notion of Coordination Object however is diluted between the three MVC elements since the Smalltalk View directly accesses the Model formalisms, and the Controller sends Smalltalk messages to the Model as well as to the View, and optionally to other controllers when scheduling interactions with other V-C pairs.

A major drawback in MVC is the close dependance between the View-Controller and the associated Model; this often leads Smalltalk programmers to define intermediary models between views and “actual” models, so as to isolate the viewing behavior from the modeling application as is enforced in our model. This is particularly necessary since the neat original MVC structuring model tends to get cluttered when programming applications using pluggable VC pairs: view and controller information may be introduced in the models.

PAC is an application structuring model that is not based on any implementation technique. This theoretical model defines the notion of an interactive object as composed of three elements: the Presentation defines its input and output behavior relative to the user; the Abstraction consists in its abstract concepts and functions; the Control manages the coherency between the two previous components. A composite interactive object is a structured

agent whose Presentation and Abstraction inherit from the Presentation and Abstraction of the components but include their own characteristics; the Control maintains the consistency between the presentation and abstraction sides managing the collaboration or dependency between the components. An interactive application can be viewed as a set of interactive objects hierarchically organized; the Abstraction of the toplevel interactive object represents the application task domain component and the Presentation is the whole image of the application.

PAC's three-way structuring is analogous to ours. The hierarchical decomposing of an interactive application is enticing; such a recursive design may be implemented with our model through the definition of hierarchical relationships between Coordination Objects. Abstraction elements that are not part of the task domain may be implemented as internal data of the Coordination Objects, or as independent objects managed by COs. Such hierarchical relationships are not systematic however, which is the reason why they are considered as application-dependent and are not built in our model.

Moreover, centralizing the task domain elements in the toplevel Abstraction can be very expensive at design and execution time since any access to application data is done through this element, and any data modification is only detected by the toplevel Control. We preferred a distributed solution where any Coordination object may declare an interest in any Task Component Object.

3. Implementation

We implemented our conceptual model as a set of types and classes of objects designed for reuse by the programmer. This set of objects implements the basic mechanisms inherent in the model we propose; it may be seen as a platform onto which the programmer can hook the objects of his application.

We describe the upper level interface of this platform, i.e. the basic types and classes explicitly used by the programmer, as socket elements to use to plug in his application.

In order to illustrate our presentation we describe a very simple application built on top of this platform. This application is an electronic note-book that allows the user to create, browse, modify and delete a list of notes. A note consists of an index, a title and an unstructured text. The interface of the application is shown below; it allows to browse through the note-book page after page (i.e. note after note), and modify or delete the current page; it also allows a note to be detached from the note-book and posted independently on the screen.

3.1 Task Component of the application

Type Displayable and its associated implementation class are provided as a basis for constructing objects of the task domain containing data potentially displayable in the interface. This type is defined as a subtype of type Traceable in order to allow the use of internal debugging facilities.

```
TYPE Displayable SUBTYPE OF
                        Traceable IS
  METHOD Init;
  METHOD TCModified(IN String);
  METHOD AddCOAccess(IN REF CO);
  METHOD SubCOAccess(IN REF CO);
END Displayable.
```

Method Init initializes internal structures of the object. Method AddCOAccess allows to register a coordination object as having an interest in the object, so as to add a new

receiver of warnings of modifications to this object. Method SubCOAccess allows a coordination object to be unregistered. Method TCModified provides for the propagation of warnings that some modification has occurred to the data of the displayable object; the execution of this method sends warnings to all coordination objects which have expressed an interest in the object.

Developing the Task Component of an application means (as to the part related to the interface) :

- Defining the types of displayable objects, as subtypes of type Displayable so as to inherit the definition of the previously described methods.

  Two types were defined for the note-book: a Note type that offers access to variables holding the index, title and text of a note, and a NoteBook type that offers operations allowing the browsing and modification of the note-book in terms of references to objects of type Note. The NoteBook type is partially shown in figure 3.

  Objects of type Note and Notebook are potentially sharable, that is the same note-book may be edited at the same time by several different users throughout the network.

- Programming the classes implementing the types of displayable objects, as subclasses of class Displayable, so as to inherit the internal mechanisms realizing the previously described methods. None of these methods needs to be redefined. The only rule in programming the class of a displayable object is to make sure the Init method gets called at initialization time, and the TCModified method gets called when a notable modification occurs to the data of the object: each time the title or text fields of a Note is modified in the Note class, and each time a Note is created or deleted in the NoteBook class.

3.2 Interaction Component of the application

Graphical environment integration

A utility tool was developed to automatically generate the Guide programming interface to any set of X graphic toolkit libraries. This programming interface is a thin layer in Guide that integrates the data and functions of the libraries into the Guide object-oriented model as smoothly as possible. The [1]Motif set is currently being used. Our goal was to remain as close as possible to the original form of the primitives declaration, so as not to loose functionality, and to keep the original X and Motif documentation useful and valid in our programming environment.

Utility types and classes and Interactive Object types and classes were defined. Utility types and classes integrate into the object paradigm a large number of utility primitives with no predisposition to such an integration. Interactive Object types and classes match modules of primitives already apparent in the libraries; the classes feature one or two instance variables corresponding to one or two common parameters to these primitives; the signature of the declared methods matches the declaration of the primitives deprived of these one or two parameters.

Utility and Interactive Object classes are used by direct instantiation inside Views; they are not designed to be subclassed.

Views

Type and class View are offered as a basis for developing the Views of the application. Type View is defined as a subtype of type View_Internals so as to inherit definitions of a lower level.

```
TYPE View SUBTYPE OF
                    View_Internals IS
  METHOD Init(IN MrmHierarchy;
                           REF View);
  METHOD CreateImage;
  METHOD DisplayImage;
  METHOD UndisplayImage;
END View.
```

Method Init initializes internal structures of the object. Methods CreateImage, DisplayImage, and UndisplayImage respectively create and initialize the IOs

[1] Motif is a trademark of Open Software Foundation, Inc.

corresponding to the managed part of the user interface, display the image of the IOs, and undisplay it.

Developing the Interaction Component of an application means :

- Decomposing the image into parts managed by View objects. Modularity is required here, as well as a concern for reuse.

  Two View objects were designed for the note-book application: one for the presentation of the whole note-book (main view of the application) and one for the presentation of a detached note.

- Defining the types of View Objects, as subtypes of View. The declarations in a View type are two-fold: part of them define the protocol used by COs to update the View; the remaining part defines the protocol used by IOs to transmit user actions to the View. See figure 3 for a partial presentation of type NoteBookView.

- Programming the classes implementing the types of Views, as subclasses of class View, so as to inherit the implementation of the previously described methods. Method CreateImage needs to be redefined to create and manage the IOs of the application.

3.3 Coordination Component of the application

Type and class CO are offered as a basis for developing the Coordination Objects of the application. Type CO is defined as a subtype of type CO_Internals so as to inherit definitions of a lower abstraction level, concerning the implementation of mechanisms such as the reception of modification warnings from the Task Component.

```
TYPE CO SUBTYPE OF CO_Internals IS
 METHOD Init(IN REF GDisplayer);
 METHOD HandleTCModif(IN
                      TCModifMess);
END CO.
```

Method Init initializes internal structures of the object. Method HandleTCModif gets called when a warning message is received from a displayable object for which the control object has expressed an interest.

```
                              // object of the Task Component
TYPE NoteBook SUBTYPE OF Displayable IS
  METHOD Init;
  METHOD First: REF Note;
  METHOD Last: REF Note;
  METHOD Next(IN REF Note): REF Note;
  ...
END NoteBook.
                              // object of the Interaction Component
TYPE NoteBookView SUBTYPE OF View IS
  METHOD GetTitleValue: String[80];        // protocol used by
  METHOD SetTitleValue(IN String[80]);     // NoteBookCO
  ...
  METHOD DetachMenuItemRequest(...);             // protocol used by managed
  METHOD NextButtonRequest(...);           // Interaction Objects
  ...
END NoteBookView.
                              // object of the Coordination Component
TYPE NoteBookCO SUBTYPE OF CO IS
  METHOD NextNote;                                // protocol used by
  METHOD DetachNote;                              // NoteBookView
  ..
END NoteBookCO.
```

Figure 3.

Developing the Coordination Component of an application means :

- Decomposing the Coordination Component into COs. The simplest application may have a unique CO; a CO is designed to handle one high level application functionality, according to the modularity principle.

Three COs were defined for the note-book application: one for the coordination of each View object and the Note object they display, and another one that acts as the coordinator of the two previous COs and controls the access to the Notebook object.

- Defining the types of each Coordination Object, as subtypes of type CO. The declarations in such a type correspond to the protocol known to the Views managed by the CO; declarations may also be defined for inter-CO communication, depending on the needs of the application. A partial view of type NoteBookCO is shown in figure 3.

- Programming the classes implementing each Coordination Object type, as subclasses of class CO, so as to inherit the implementation of the previously described methods. Method HandleTCModif needs to be redefined. Programming a CO class implies holding references to managed Task Component objects, declaring an interest in these objects, creating and displaying Views, stocking the mapping between TC objects and Views, calling TC operations and updating Views.

3.4 Control Distribution

Guide is a passive object model: execution structures must be defined separately from the objects. We propose several possible execution configurations so as to match application needs and specificities in terms of dialog threads, length of operation execution, etc. As a minimum execution configuration we propose two concurrent controllers: the GDisplayer and ListenToTC controllers.

The GDisplayer controller is dedicated to the watching for user actions; upon reception of an X-level event indicating a user action the controller executes the handling method defined in the View of the Interaction Object corresponding to the event.

The ListenToTC controller is dedicated to the watching of Task Component modification warnings; upon reception of such a message it executes the HandleTCModif method of the Coordination Object the message is sent to.

4. Evaluation and Experience

Our framework is currently being used in the development of a mailer system, a browser of types and classes, and graphic shell tools. A library of View objects is being elaborated. Additional services are also being implemented, such as support for an error and help system, and support for inter-application "cut-copy-paste" facilities.

The main lessons learned from the design and implementation of this framework, as well as the preliminary experience with the building of applications on top of it may be summarized as follows.

4.1 The framework

• *Existing frameworks*. The main reference application framework currently is MacApp [Schmucker 86], which was realized in Object Pascal. MacApp was designed mainly as a utility tool to help programmers to use the Macintosh toolbox, at the expense of the design model. MacApp offers a set of classes implementing the three aspects of an interactive application: control, application concepts and presentation views. In MacApp however no formalism transfer is offered, which implies that the task domain elements directly reference the presentation elements. Close dependence between components is a major drawback for the application design-implementation-test cycle which our model manages to avoid, as is explained thereafter.

• *Design-implementation-test cycle.* We provide an architecture model that allows an iterative development of interactive applications. Task and Interaction Components may be developed completely separately. The Coordination Component only requires to know the types of Task objects and Views. Thanks to the dynamic linking facilities in Guide each component may be modified without recompiling the other components, provided the interface of this component (i.e. the type of its elements) remains unchanged.

• *Help system*. Reusing the framework implies understanding the inherent methodology for building interactive applications, and the precise functions of each type and class provided in the framework, as well as their rules of use. We provide a programmer's manual describing the practical details of the process of reusing the framework; each type is presented according to a help "template" explicating its role within the application, and for each of its declarations the function, calling rules, parameters meaning and redefinition rules are also given. Such a type by type description is not enough to understand the organization of an application and the relationships between object within the application. Learning by example is a way to reach that goal; another way is to use a set of skeletons of subtypes and subclasses of our types and classes, these skeletons being provided as programming guides.

• *Shortcoming and future objectives* . The main shortcomings of the framework are related to its ease of use. We are now considering building a new category of tools, interactive tools to assist the programmer in the development of its application on top of the framework.

4.2 Object-oriented design and language

• *Helping the programmer to work for reuse*. Reusability is said to be the major contribution of object-oriented languages; however, programming reusable types and classes is not automatic, as stressed in [Johnson 88]. First of all, programming reusable elements is a way of programming that must be acquired; we tend to build types and classes to answer our specific needs; an extra effort is necessary to take a new perspective on our work and isolate in a new type and class functionalities that may answer other programmers' needs. We hope the methodology inherent in our framework will help to initiate this process, particularly to extend the library of View objects. Once this creation of reusable elements is realized however, we are only half way: the use of the new class and type has still to be leaned by other programmers. We propose basic help tools in the preceding section; another way of making the learning process easier is to define

type protocols according to standard rules. For example, the initialization method of an object is always named "Init"; so the programmer knows that method Init in a new type should be called only once, at instantiation time. We believe such simple rules can be of great help in the learning process; a framework such as ours can help define these standard easily-recognized protocols; for example we designed the View type as containing the method CreateImage even though its implementation in the View class is empty (it must be entirely redefined in the programmer's subclass) so as to help enforce the rule that the creation of the image of a View object is realized by the execution of method CreateImage.

• *Abstract types and type conformity relationships in designing for reuse*.

Guide separates abstract types and implementation classes; a relation of syntactic conformity is defined between types; this relation provides for type checking at compile time as well as execution time for special cases, in order to control the assignment of a variable to objects of different types.

Separating abstract type from implementation class is a highly valuable feature to us: it allows the independence of our application components. Combined with subtyping and subclassing, it allows to simplify the framework interface offered to the programmer: all he needs to know is contained in the types defined in the upper level of the platform; internal definitions are hidden in super-types of these types. Abstract types and conformity relationship offer an additional facility in terms of reuse enhancement: they allow the definition of libraries of View objects. Indeed the only application-dependent feature interfering when defining a View is relative to the Coordination Object dedicated to the management of this View; more precisely, the application dependance may be represented by the set of operations called by the View on the Coordination object in order to transmit user action. This set of operations may be gathered in a CO type definition; this allows the View to be reused and managed by any CO whose type conforms to the CO type initially defined.

Conclusion

We have presented an application framework which offers an architecture model and a programming guide for the development of interactive applications in Guide, but also provides the implementation of services common to all applications, such as the access to standard graphic libraries and a support for the coherency of views on shared task domain objects. This application framework is implemented as a set of abstract types and classes to be reused by the programmer through subtyping and subclassing or through instantiation.

The benefits to be reaped from this tool are a programming methodology that not only ensures the evolution capability of the interactive system to be built, but also promotes reuse as well as the design of reusable application components.

Acknowledgements. The tools to automatically integrate the graphic toolkit libraries were developed by Malcolm Bennett. Guide is partially supported by the Commission of European Communities through Esprit project 834 (Comandos).

References

[Alves Marques 88]
Alves Marques J., Balter R., Cahill V., Guedes P., Harris N., Horn C., Krakowiak S., Kramer A., Slattery J., Vandôme G., Implementing the Comandos architecture, *Proc. 5th Annual ESPRIT Conference*, Bruxelles, North-Holland (Nov. 1988), pp. 1140-1157.

[Coutaz 87]
Coutaz J., PAC, an Implementation Model for Dialog Design, Interact'87, Stuttgart, September, 1987, pp. 431-436.

[Coutaz 88]
Coutaz J., Interface Homme-Ordinateur : Conception et Réalisation, Thèse de Doctorat d'Etat, Université Joseph Fourier, Décembre 1988.

[Decouchant 88]
Decouchant D., Duda A., Freyssinet A., Paire E., Riveill M., Rousset de Pina X., Vandôme G., Guide: an implementation of the Comandos object-oriented architecture on Unix, *Proc. EUUG Aut. Conf.*, Lisbon (Oct. 1988)

[Goldberg 84]
Goldberg A., Smalltalk-80 : The Interactive Programming Environment; Addison-Wesley Publ., 1984.

[Horn 87]
Horn C.J., Krakowiak S., An object-oriented architecture for distributed office systems, *Proc. 4th Annual ESPRIT Conf.*, Brussels (sept. 1987)

[Johnson 88]
Johnson R. E., Foote B., Designing Reusable Classes, *Journal of Object-Oriented Programming*, June-July 1988, pp. 22-35.

[Krakowiak 90]
Krakowiak S., Meysembourg M., Nguyen Van H., Riveill M., Roisin C., Design and implementation of an object-oriented, strongly typed language for distributed applications, *to appear in the Journal of Object-Oriented Programming.*

[Krasner 88]
Krasner G. E., Pope S. T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, August-September 1988, pp. 26-49.

[Myers 89]
Myers B. A., User-Interface Tools : Introduction and Survey, IEEE Software, January 1989, pp. 15-23.

[Schmucker 86]
Schmucker K., MacApp : An Application Framework, Byte 11(8), 1986, pp. 189-193.

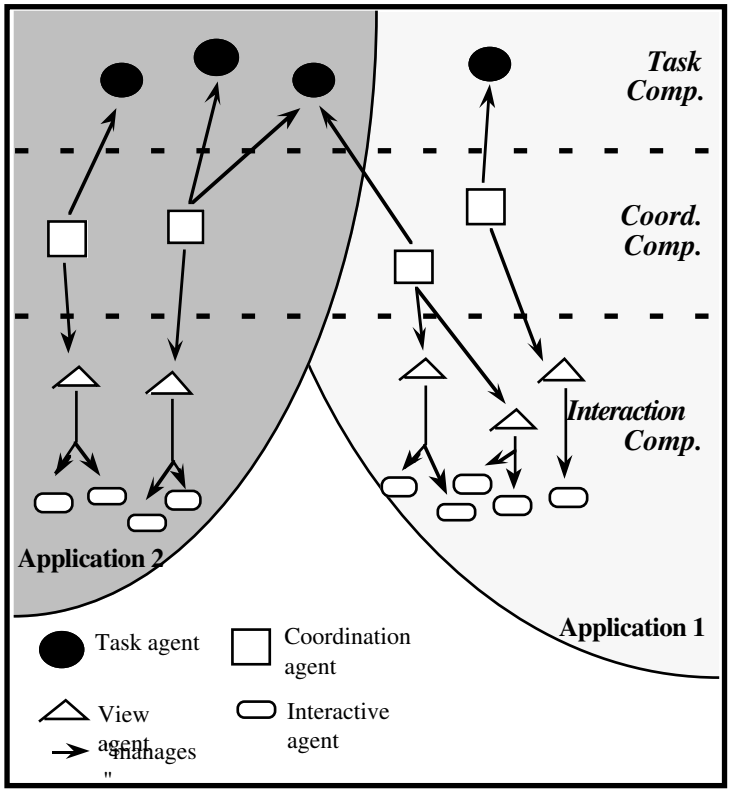

Figure 1

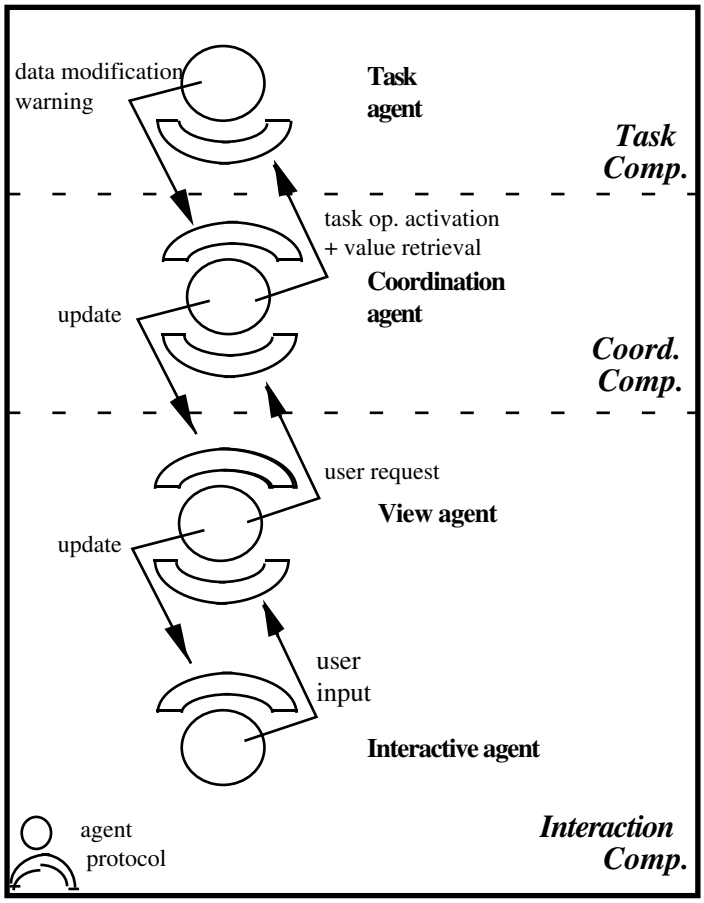

Figure 2

A Practical Framework For Interactive Applications in Guide, an Object-Oriented Distributed System.

Introduction……1

1. Context and requirements……1
   1.1 Overview of the Guide system and programming language……1
   1.2 Guide applications……2
   1.3 Graphical environment……2

2. Conceptual Model……3
   2.1 General architecture models……3
   2.2 Our model for Guide applications……3
   2.3 Related Models……5

3. Implementation……6
   3.1 Task Component of the application……7
   3.2 Interaction Component of the application……7
   3.3 Coordination Component of the application……8

4. Evaluation and Experience……10
   4.1 The framework……10
   4.2 Object-oriented design and language……10

Conclusion……11