THE PLATFORM INFLUENCE ON OBJECT-ORIENTED DEVELOPMENT
BASED ON THE MULTIAGENT MODEL.¹

Francis Losavio
Centro de Ingeniería y Sistemas ISYS,
Escuela de Computación, Universidad Central de Venezuela,
Apdo. 47567, Caracas, Venezuela
e-mail: flosavio@conicit.ve

Carlos Gálvez
Centro de Computación
Instituto Venezolano de Investigaciones Científicas, IVIC
Altos de Pape, Venezuela
e-mail: cgalvez@ivic.ivic.ve

Abstract

This work presents an ongoing research in designing interactive graphical applications following an object-oriented approach based on multiagent models, under different development platforms and graphical toolkits: Unix-X/Xt/Motif and MS-DOS-Windows/ObjectWindows. The architecture of the resulting system is obtained using the Presentation, Abstraction and Control (PAC) multiagent model, inspired on the Model, View, Controller (MVC) approach, or a combination of these models. The advantages of object-oriented design based on the multiagent model are pointed out. General schemas are presented for building the system architecture under both platforms, and their comparison is established. Finally, a C++ general pattern implementing the communications among multiagents using the ObjectWindows toolkit is provided.

Keywords: User-interface Design, Multiagent Model, PAC Model, Object-Oriented Methodology, Object-Oriented Programming.

1. INTRODUCTION

The choice of the platforms and graphical toolkits has major implications on the architecture of graphical interactive systems. The main goal of this work is to present two general schemas, based on the multiagent model, for the architecture of interactive systems, separating the application level from the user-interface level. The global view of the system's architecture presents two main layers, the application level, corresponding to the problem domain component, and the interface level, constituting the user-interface component. This approach favors reuse and communication among the agents through their respective controls, while maintaining the separation, whenever possible according to the selected platform, between the presentation and abstraction aspects of the application. One schema is proposed for a Unix², X/Xt³/Motif⁴ platform using the PAC (Presentation, Abstraction and Control) model [Cou 90], and the other, which combines different multiagent models, seems better adapted to the MS-DOS/ Windows⁵/ObjectWindows⁶ platform. The use of window toolkits helps to separate the application from the corresponding user-interface, but not much is said on design guidelines for achieving this goal. The Abstract Data Views and Abstract Data Objects is an interesting approach enhancing design for reuse [Cow&Al 93], [CL 95]. Another goal of this work is to introduce an object-oriented (OO) method based on

¹This research is supported by the New Technology Program of the BID-CONICIT
²Unix is a registered trademark of Unix System Laboratories.
³X-Window is a registered trademark of the Massachusetts Institute of Technology.
⁴Motif is a registered trademark of Open Software Foundation.
⁵MS-DOS, Windows are registered trademarks of Microsoft
⁶ObjectWindows is a registered trademark of Borland Int.
multiagent models, supporting complete development of interactive systems and combining known OO methods. The main aspects of the method are that any OO methodology could be used during the requirements and analysis phase, Jacobson's Use-Case approach [Jac&Ai 92] for example, in order to obtain the requirements for an interface prototype, and the interface, entity and control objects composing the whole system. During the design phase, Booch [Boo 94], or any other well known OO methodology could be used in order to construct the system object model.

Besides this introduction and the conclusion, this paper is structured in four sections: Section 2 presents some generalities on software development based on multiagent models, focusing on the PAC model. An OO multiagent based methodology is presented. Section 3 is devoted to the general schemas representing the system architecture under the two different platforms. An interactive graphical system built under MS-DOS/Windows/ObjectWindows is discussed as a case study. Section 4 gives a C++ pattern implementing the communications among the PAC agents under the MS-DOS/Windows/ObjectWindows platform and toolkit.

2. MULTIAGENT MODEL BASED OBJECT-ORIENTED DEVELOPMENT

In this section we will give some generalities on the multiagent model approach, emphasizing particularly the PAC model. Some differences with similar models are focused, pointing also out the advantages of this model for developing interactive systems with graphical user-interfaces. Finally, an object-oriented (OO) methodology based on the multiagent model approach is discussed.

2.1 The multiagent model

The multiagent model [Cic 84], [Gol 84]], [LVC 89] is used to describe the architecture of interactive systems. It is inspired from the stimuli-answers systems, which are organized as a set of agents, reacting at external events (stimuli) and generating events (answers). An event or stimulus is of a certain kind, it holds some information depending on its kind, it is produced by an emissary and received by a receptor. An agent may be seen as a processor, with receptors and emissaries for capturing and producing events. It is constituted by a two level memory, one to register detected events, the other to memorize a state, and it is characterized by a modular organization, parallel execution of processes and event-driven communications. This model adds another dimension, the parallelism, to the traditional models used in the construction of graphical user-interfaces for interactive systems (language based [F&D 84] and input/output based [Lan 86] models).

The basic OO notions [Mey 88] are present in the agent concept. Notice that a class may define a category of agents; the operations are the instructions of the processor, the attributes constitute the memory elements, modelling the agent's state. The constraints (e.g. preconditions reacting to the activation of an operator) specify the semantics of the processor's instructions. Moreover, agents may be connected by inheritance and/or association relations. An important issue of the multiagent model, as for the OO paradigm, is that an agent defines the granularity and modularity of the system. It is then possible to modify a behaviour without compromising the whole system.

2.2 The PAC model

The Presentation, Abstraction and Control (PAC) approach [Cou 90], based on the multiagent model, allows to structure recursively the architecture of an interactive system. The agents are organized according to three basic formalisms or perspectives: - The Presentation, defining the image of the system, reflecting its behaviour with respect to the user input/output. - The Abstraction, defining abstractly (independently from the graphical representation) the concepts and functionalities of the system. - The Control, maintaining the coherence and communications between the Presentation and the Abstraction perspectives.
since these are not allowed to communicate with each other. Moreover, the communications among the PAC agents are only performed by means of their respective controls. The PAC diagram of the application is recursively defined in a tree-like fashion from elementary PAC agents. Each tree level corresponds to an abstraction or semantical level of the application. On one hand, the highest level of the tree (root), called application level, contains a complex agent representing the semantical features of the application. On the other hand, the lowest level (leaves) contains elementary graphical features of the application. The other levels, besides the root, are grouped into a layer called interface level. Following the multiagent approach, each PAC agent is considered as a complete system. PAC agents work in parallel, and each agent is organized as three parallel processors, represented by the abstraction, control and presentation formalisms, with respectively three memories: an abstract state, a control state and a graphical state. The control instructions communicate with the other PAC agents, eventually performing translations, hence satisfying the requirements of these objects and the user's actions by calling the abstraction and presentation processors. As a consequence of its functions, the control may also work as an event recognizer. The instructions of the abstraction processor are functions working on the abstract state memory. The presentation processor contains instructions applied on the graphical state memory to display the objects on the screen and to receive the user's actions.

The main differences between the PAC and another well known similar approach, the MVC (Model, View, Controller) model [Gol 84] are: - The Model, View, and Controller perspectives constitute three Smalltalk objects. The Model perspective is like the PAC abstraction. Nevertheless, the View defines the user's perception of the system and the Controller only interprets the user's external events, so the PAC presentation perspective is represented in MVC by the view and the controller, which cannot be separated. Moreover, the central issue of the PAC model, the control perspective, which is considered as a translator from the abstract to the concrete world and allows communications among agents, is absent in the MVC model. - MVC agents may indistinctly communicate among them, while PAC agents can only communicate through their respective control perspective.

The main advantages of the PAC approach are presented below.

2.3 Advantages of the PAC model

We can resume the following advantages of this model: - It is a systematic approach recursively applicable to all the abstraction levels of an interactive system. - The control notion is introduced for communicating two different perspectives (abstraction and presentation), and for managing the interaction by means of cooperating local controls, facilitating the communications among agents. The control concept allows to maintain context depending information at different abstraction levels, facilitating general services such as help and historic. - It favors the distribution of semantical and syntactical actions. Particularly, the application may express itself in its own terms, independently from the presentation aspects, allowing the cohabitation of semantically distinct domains. In this way, functions which are usually mixed into the application domain, as for example the editing functions, are now kept only at interface level. These aspects facilitate the system maintenance (extensibility and modifications). - A PAC agent may hold different presentations with the same abstraction.

2.4 An OO methodology for developing interactive systems

The PAC approach does not impose any particular design methodology, but PAC agents seem naturally adapted to be implemented using object-oriented programming techniques. In [Los & Al 94a], and [Los & Al 94b], a methodology has been proposed for developing interactive applications, whose architecture is based on the PAC model. In this work, the methodology is generalized to any multiagent model. Its steps are the following:

i. Formulate the requirements for the application.
ii. Study of the functionalities of the system with respect to user-interface, on the basis of the requirements formulated in step i.

iii. Identify the graphical objects and the multiagents based on the above results. A general multiagent diagram of the system is constructed. In parallel with the identification of the agents, the classes implementing the abstraction and presentation perspectives (when it is presented separately from its control) may also be identified.

iv. Introduce the user external actions on the multiagent diagram, for validating in a certain measure the architecture model. The attributes and operations identified in step iii may be used in this sense, allowing also the semantical validation of the operations already identified in the classes. This process points also out the control functionalities, whenever possible, in order to identify the attributes and operations of the corresponding classes.

v. Implementation of the classes corresponding to the multiagents which have been identified in steps iii and iv.

Considering an OO development, steps i, ii and iii constitute the requirements and analysis phase, which can be performed using an existing OO method. The use of Jacobson's Use-case approach [Jac&Al 92] as a front-end for our methodology and of Booch's design [Boo 94] as a back-end for modelling the application abstraction level, is under experimentation, and will be the object of future works. Notice that step iii refers to all the abstraction and presentation perspectives of the agents constituting the system. Particular care has to be taken for the abstraction corresponding to the application level agent, which represents the problem domain component in the Booch's or Coad&Yourdon's [CY 91] sense, and can be modelled accordingly. This paper aims to show that the structure of the PAC agents may be influenced, according to the choices of the platform and the graphical tools. Moreover, different multiagent models may be combined, still preserving some of the PAC principles. This issue will be discussed in the following section.

3. GENERAL SCHEMAS FOR THE SYSTEM ARCHITECTURE UNDER TWO DIFFERENT PLATFORMS. A COMPARISON

This section presents two different schemas for the system architecture under different platforms, depending on the toolkit used.

3.1 General schema representing the system architecture using the X/Xt/Motif toolkit

The schema presented in Figure 1 shows the PAC model based architecture of a system developed using the X/Xt/Motif toolkit. This diagram has been obtained following the methodology discussed in 2.3. The detailed steps will not be given here since the goal of this work is to show the platform and graphical toolkit influence on the internal organization of the involved agents. The APPLICATION LEVEL contains in general only one agent, corresponding to the problem domain component, with an Abstraction (shown as A in the oval) perspective holding information which is completely independent from the graphical representation of the objects manipulated by the application. The Presentation (shown as P in the oval) perspective may be absent, or it is reduced to an icon or button, for starting the whole application. In order to establish a general schema, at INTERFACE LEVEL, we have identified three layers of agents: the Interface_Manager level, with the Interface_Manager agent (shown as the Control name in the oval). This agent has in general a presentation containing the buttons or icons corresponding to the activation of the tools, at Tool level. Its abstraction is in charge of the information on the different tools opened or "clicked" by the user. The agents representing Tool_i, 1≤i≤n, with n integer, at Tool level, may be complex ones, often representing entire subsystems, which in turn may be developed applying the OO method described in section 2.4.
The last group of agents constitute what we have called the Tool_component level. Each agent, represented by the Tool_i_j_component, \(1 \leq i \leq n, 1 \leq j \leq k\), is constituted by the graphical elements conforming the windows of the application, such as window frame, drawing area, palette, menus. The Abstraction of these agents contains graphical information on the elements, such as coordinates, etc. The Presentation captures the user external events, such as the mouse click, etc. If Tool_i is a multiple document application, the Interface_Manager agent on the upper level, is charged to manage the lists of documents activated by the user. Each document is like an instance of the corresponding PAC agent. Graphically, this situation is represented in Figure 2 by three superposed PAC agents.

Figure 2. Multiple document application with three PAC agents representing three documents.

Notice also that Tool_i_j_component may also be a multiple document application, in which case the lists of documents are managed at the upper level by Tool_i. Moreover, Tool_i_j_component may be refined into other tool components (and the same argument applies) or be constituted by elementary graphical objects that are directly implemented using X/Xt/Motif facilities.

The communication among agents is bidirectional and it is hierarchically performed only through their respective controls. For example, an agent belonging to the Tool_i_j_component level cannot communicate directly with an agent at the Interface_Manager level. It must communicate first to a Tool level agent. In [Los\&Al 94a] a general C++ [Str 93] schema for implementing communication among agents, preserving...
the PAC principles, has been proposed using this toolkit. An application developed according to this schema is discussed in [Los&Al 94b].

3.2 General schema representing the system architecture using the Windows/Object Windows toolkit.

The schema shown in Figure 3 presents the system architecture using the Windows/ObjectWindows toolkit [Bor 92a], [Bor 92b]. We found here three different kind of agents: the usual PAC agent, where the Presentation, Abstraction and Control perspectives are separated, which are user defined objects (e.g. Tool_1 and Tool_n_1_component); agents where the three perspectives cannot be separated, because they are predefined ObjectWindows objects (e.g. Interface_Manager and Tool_n); agents behaving like MVC agents (e.g. Tool_n_k_component), where only the Abstraction perspective is separated and correspond to a derived ObjectWindows object. So, a model with a combination of different kinds of multiagents seems much more adapted to represent the system architecture under the graphical Windows/ObjectWindows toolkit. A "pure" PAC or MVC model could not be used to represent this architecture, since this toolkit manipulates objects at a higher abstraction level. Moreover, the management of multiple documents is entirely performed by the toolkit objects, in consequence the three perspectives cannot be presented separately. Notice that, even if several multiagent models are used to represent this architecture, we preserve the PAC approach communication principle, in the sense that the agents cannot communicate directly among them. Moreover, the communication flows only hierarchically between the different levels established.

![Figure 3. System architecture using multiagents with Windows/ObjectWindows](image)

In the following section, an application developed with Windows/ObjectWindows toolkit, accordingly to the methodology presented in 2.4, will be briefly presented.
3.3 Case study: OO design of an application under MS-DOS/Windows/ObjectWindows platform

The application we want to build is a graphical interactive environment for drawing and manipulating graphs and digraphs, from the graphist point of view, e.g. drawing a picture of the graph, computing classical properties on it, like connectivity, shortest path, hamiltonian path, etc., write annotations on the drawing, have information on bibliographical references on some particular graph or digraph which is kept in an album of pictures. The system must offers facilities like sheets of paper for drawing and tools for the computations. For more detailed information, the reader is referred to [Los&AJ 91].

Following the methodology presented in Section 2.4, we could use any object-oriented methodology for steps i, ii and iii. We think that Jacobson's Use-Case approach [Jac&Al 92] is well suited for designing the requirements for this application, but they will not be detailed here, since they are outside the scope of this paper. A view of the user-interface may be seen in Figure 5. The multiagent diagram corresponding to the system architecture, is built in step iii and is presented below in Figure 4.

![Figure 4. Architecture of the AMDI_OO environment under Windows/ObjectWindows, shown with three digraph windows.](image)

According to the general schema in Figure 3, at APPLICATION LEVEL we find the problem domain component, the AMDI_OO (AMbiente para Dígrafos Orientado a Objetos) agent, which can be modelled using a known OO methodology (e.g. Booch [Boo 94] has been used in our case). The Abstraction perspective of this agent keeps all the graph abstract data structure, which is completely independent from the graphical details. The Presentation perspective is just an icon corresponding to the application. The INTERFACE LEVEL contains the Interface_Manager agent, which is in charge of the different windows of the application. The ObjectWindows toolkit manipulates transparently all the AMDI_OO windowing system, and this is the reason why the abstraction and presentation perspectives cannot be separated using this development platform. Notice that this is not the case for the Interface_Manager agent shown in Figure 1, using the X/Xt/Motif toolkit. The Palette agent, used for manipulating the graph, is a specific object of the AMDI_OO application, so it is designed as a PAC agent, with separate Presentation, Control and
Abstraction perspectives. The Digraph_WINDOW is the most important graphical agent of the application; it is an ObjectWindows multiple document object and so its perspectives cannot be separated. The last agent, the Canvas, is the drawing area for graphs and digraphs. Notice that it is designed like an MVC object, since all the graphical informations required for graph manipulations are kept in the Abstraction perspective. Like the MVC agent, it contains a unique presentation and control perspective, charged to capture the external events. Figure 5 below, shows an aspect of the AMDI_OO user-interface. Notice that each agent at INTERFACE LEVEL is completely identified.

Figure 5. Multiagents constituting the INTERFACE LEVEL of the AMDI_OO environment and representing a view of the user-interface.

Continuing with step iii of our methodology, the classes must be identified in order to implement the agents. In this step, any known OO methodology could be used for this purpose. For example, if Jacobson [Jac&Al 92] has been used as a front-end for requirement analysis, all the entity, control and interface objects will be identified. Booch methodology [Boo 94] can now be used as a back-end for constructing the corresponding object model. We have taken this approach for AMDI_OO and the Booch class diagram is shown in Figure 6. Notice that the class category A_amdi_OO represents the set of classes constituting the abstraction perspective of the AMDI_OO agent, containing the abstract information on graph and digraph objects, independently form its graphical representation.

In Figure 6, the C_amdi_OO class represents the Control perspective of the agent. Notice that classes TApplication, TMDIFrame and TWindow are ObjectWindows classes. C_amdi_OO inherits from TApplication class, in order to start the whole application. CP_interface_manager (corresponding to Interface_Manager agent) inherits from TMDIFrame, in order to manipulate multiple document objects. Classes CP_digraph_window (corresponding to the Digraph_Window agent), CP_canvas (corresponding to the Canvas agent) and P_palette (corresponding to the Presentation perspective of the Palette agent) inherits from TWindow class, in order to obtain window functionalities. Notice that all the other relations are use realations in the Booch sense, and they are bidirectional, representing the communications among the agents. The design continues with step iv of the method, introducing user external actions, for detailing the communication among objects. The attributes and methods are identified and/or completed during this step. The implementation pattern for communication among agents, part of step v of the methodology, will be shown in Section 4.
3.4 A comparison of the schemas

The global structure of the system is maintained in both schemas (Figure 1 and Figure 3). Notice that the abstraction level of the toolkit influences the underlying structure of the multiagents. When the toolkit has tools at a high abstraction level, the tools have more control on the agent than the user or implementor; this means that the presentation and the control are kept together and the two perspectives cannot be separated as for PAC agents. This is the case of MVC like agents, where the controller and the view are together in the presentation perspective, and the abstraction is separated from the rest of the agent. Notice also the case of agents with non separated perspectives. This combination of models seems well adapted to be used with the Windows/ObjectWindows toolkit. The PAC model, instead, is more adapted to toolkits with tools at lower abstraction level, like X/Windows/Motif. The implementor has more control on the agents and three separate perspectives can be always considered. An important issue is that at application level, an abstraction perspective is maintained in both schemas, and at interface level, the lower level the specific application component objects have also a separate abstraction. This fact guarantees that the PAC principles are maintained in the sense that this architecture favors the distribution of semantic and syntactical actions, enhancing the system flexibility to changes.

4. C++ PATTERN IMPLEMENTING COMMUNICATIONS AMONG MULTIAGENTS UNDER WINDOWS/OBJECTWINDOWS PLATFORM AND TOOLKIT

The communications among the agents constituting the AMDI_OO application shown in Section 3.3, have been implemented following the general C++ [Cor 92] pattern, derived form the schema for Windows/ObjectWindows platform, seen in Figure 3. For easing the reading of this Section, the reader is referred to this schema.

4.1 Definition of the classes

PAC Agent: Application

// Class implementing the Control perspective of the PAC agent Application
class Application: public TApplication //TApplication is an ObjectWindows class
{ private:
    Abs_Application *abstraction; // pointer to the Abstraction perspective of the Application PAC agent
    PWindowsObject MainWindow; // pointer to the Interface_Manager agent
public:
    Application(LPSTR name,HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmd, int nCmdShow);
    // constructor
    ~Application(); // destructor
    void InitMainWindow(); // main window creation
    ... f(...); ... // Function implementing one of the possible communications.
    // Methods for communicating the abstraction with the presentation
};
// Class implementing the Abstraction perspective of the PAC agent Application:
class Abs_Application
{ private:
    Application *control; // pointer to the Control perspective of the Application PAC agent
public:
    Abs_Application(); // constructor
    ~Abs_Application(); // destructor
    ... f(...); ... // Function implementing one of the possible communications
};
// Notice that the Presentation perspective of the Application PAC agent corresponds to the icon
// representing the application, which is an attribute of TApplication class

Agent: Interface_Manager

// Class implementing the Abstraction, Control and Presentation perspectives of the Interface_Manager agent
class Interface_Manager : public TWindow // inherits from the class TWindow of
// Objectwindows or inherits from the class
// TMDIFrame, in case of multiple document objects
{ private:
    Application *application; // pointer to the Control perspective of the Application PAC agent
    Tool_1 *tool1; // pointer to the Control perspective of the Tool_1 PAC agent
    ... 
public:
    Interface_Manager(PWIndowsObject Aparent, LPSTR Atitle,
    Application *p); // constructor
    ~Interface_Manager(); // destructor
    ... f(...); ... // Function implementing one of the possible communications
};

Agent: Tool_n

// Class implementing the Abstraction, Control and Presentation perspectives of the Tool_n agent
class Tool_n : public TWindow // inherits from the class TWindow of
// Objectwindows
{ private:
    Interface_Manager *interface_manager; // pointer to the Interface_Manager agent
    Tool_n_1_Component *tooln_1_component; // pointer to the Control perspective of the
    // Tool_n_1_Component agent
    ... 
public:
    Tool_n(PWIndowsObject Aparent, LPSTR Atitle, Interface_Manager * p); // constructor
    ~Tool_n(); // destructor
    ... f(...); ... // Function implementing one of the possible communications
};

Agent: Tool_n_k_Component
// Class implementing the Control and Presentation perspectives of the Tool_n_k_Component agent
class Tool_n_k_Component:
public:
    TWindows // inherits from the class TWindow of the
{ // private:
    Abs_Tool_n_k_Component * abstraction; // pointer to the Abstraction perspective of
    // the Tool_n_k_Component agent
    Tool_n * tool_n;
    public:
    Tool_n_k_Component( Tool_n * p);
    ~Tool_n_k_Component();
    f (...); ...
};

// Class implementing the Abstraction perspective of the Tool_n_k_Component agent
class Abs_Tool_n_k_Component
{ // private:
    Tool_n_k_Component* control; // pointer to the Control and Presentation perspective
    // of the Tool_n_k_Component agent
    public:
    Abs_Tool_n_k_Component(Tool_n_k_Component* p); // constructor
    ~Abs_Tool_n_k_Component(); // destructor
    f (...); ...
};

4.2 Definition of the methods

PAC agent: Application

Application::Application(LPSTR name, HINSTANCE hInstance, HINSTANCE hPrevInstance,
{ abstraction = new Abs_Application(this); }; // creates an instance of the Abstraction of the PAC
// agent Application
Application::Application() { }; Application::InitMainWindow() { MainWindow = new Interface_Manager(NULL,"caption",this); };
Abs_Application::Abs_Application(Application *p) { control = p; }
// saves the link to the instance of the Control perspective for the PAC agent Application
Abs_Application::~Abs_Application() { }; Agent: Interface_Manager
Interface_Manager::Interface_Manager(PTWindowsObject Aparent, LPSTR Atitle, Application *p)
{ application = p;
    tooln = new Tool_n(HWND,"caption",this); }; // creates an instance of the Tool_n agent
Interface_Manager::~Interface_Manager() { delete(tooln); }; // destruction of the class instance of the
// Tool_n agent
Agent: Tool_n

Tool_n::Tool_n(PTWindowsObject Aparent, LPSTR Atitle, Interface_Manager * p)
{ interface_manager = p; // saves the link to the instance of the Control perspective for the
// Interface_Manager agent
tool_n_k_component = new Tool_n_k_Component(HWND,"caption",this); }; // creates an instance of the Tool_n_k_Component agent
Tool_n::~Tool_n() { delete(tool_n_k_component); }; // destruction of the class instance of the
// Tool_n_k_Component agent
Agent: Tool_n_k_Component

Tool_n_k_Component::Tool_n_k_Component(Tool_n * p)
{ tooln= p; // saves the link to the instance of the Control perspective for the Tool_n agent
    abstraction = new Abs_Tool_n_k_Component_i(this); }; // creates an instance of the Abstraction perspective of the Tool_n_k_Component agent
Tool_n_k_Component::~Tool_n_k_Component() { delete(abstraction); };
// destruction of the Abstraction class instance of the PAC agent Tool_Component_i
Abs_Tool_n_k_Component:: Abs_Tool_n_k_Component(Tool_n_]Component*p)
{control = p; } // saves the link to the instance of the Control perspective for the
// Tool_n_k_Component agent
Abs_Tool_n_k_Component::~Abs_Tool_n_k_Component() {};

5. CONCLUSION

Two schemas using multiagent models for different platforms and toolkits have been presented. We found that the abstraction level of the objects manipulated by the toolkit determines the choice of the particular multiagent model. Nevertheless, the advantages of the PAC model approach are preserved in both schemas presented: the abstraction perspective is separately kept in the implementor-defined agents which are specific to the application, like the application agent and the tool component agents, ensuring the independence from the graphical representation of the application. This fact guarantees that the PAC principles are maintained also for the MS-DOS/Windows/ObjectWindows platform, favoring the distribution of semantic and syntactical actions and enhancing the system flexibility to changes. The C++ pattern, expressing communications among agents, facilitates the reuse of code during the implementation step and the multiagent architecture also favors the work of programmers' teams.

6. REFERENCES
