INTERFACE AGENTS FRAMEWORKS FOR OBJECT-ORIENTED DESIGN*

F. Losavio, A. Matteo

Centro de Ingeniería de Software Y Sistemas ISYS,
Facultad de Ciencias, Universidad Central de Venezuela,
Apdo. 47567, Los Chaguaramos 1041-A, Caracas, Venezuela
flosavio @conicit.ve / @anubis.ciens.ucv.ve, amatteo@conicit.ve / @anubis.ciens.ucv.ve

Abstract

Frameworks, or semifinished generic architectures, have been successfully used in the development of graphical user-interfaces. Besides, the multiagent model describes the architecture of interactive systems. The main goal of this work is to discuss multiagent frameworks and their related design patterns for developing adaptable (reusable and extensible) user-interfaces. The well known MVC (Model-View-Controller) framework is presented, and a framework expressing the behaviour of the PAC (Presentation, Abstraction and Control) model is constructed. The instantiation of both frameworks is illustrated with an example, and both frameworks are compared, on the basis of the architecture of the resulting systems.

Keywords: framework, agent, multiagent model, human-machine interface, user-interface design, reusable software, design patterns, object-oriented design, object-oriented programming, groupware

1. INTRODUCTION

This paper aims to present two frameworks or semifinished generic architectures, for rapid development of adaptable (reusable and extensible) graphical interactive systems. This work is based on the functioning of stimulus-response systems, which are organized as a collection of agents that reacts to a given set of external phenomena, producing new phenomena. The graphical interactive system is focused as a collection of specialized agents that produce and react to events. An agent is a complete processing system, with a memory to maintain a state and a processor for cyclical treatment of inputs events, updating its own state, producing events or selecting input event classes. Agents that communicate directly with the operator are interactive objects, which should be seen as part of the operator: they listen to the operator, posses a convenient expertise and provide adequate feedback. Multiagent models are used to structure the architecture of interactive systems, favoring the separation between the application domain component and the interface component. Two known multiagent models MVC [Gol 84] and PAC [Cou 94] are described here. On the basis of the agent's behaviour in each model, a generic framework is constructed using design patterns [Gam&Al 95] as building blocks. An example illustrating the instantiation of both frameworks is presented. Finally, a comparison is established, considering the communications among the agents and the separation of the application and interface components in both frameworks.

Besides this introduction and the conclusion, this paper contains three sections: Section 2 introduces the design patterns and frameworks context. Section 3 presents the multiagent models MVC and PAC and the framework corresponding to their respective agents. Section 4 illustrates the instantiation of the presented framewrks with an example. Finally, Section 5 establishes a comparison of the communications among agents at application and interface levels.

2. DESIGN PATTERNS AND FRAMEWORKS

* This research is supported by the New Technology Program of the BID-CONICIT, the CONICIT MOODe Project and the Consejo de Desarrollo Científico y Humanístico (CDCH) OOMGRIN Project.

The approach to software design with patterns or templates that can be applied in many different situations, captures the experience involved in designing object-oriented (OO) software. Each design pattern systematically names, explains and evaluates an important and recurring design in OO systems. Recurring patterns of classes and communicating objects are found in many object-oriented systems. These patterns solve specific design problems and make OO designs more flexibles, elegant and reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer, familiar with such patterns, can apply them inmediatly to design problems without having to rediscover them. The following definition, given by the architect Christopher Alexander [Ale&Al 77], applies fairly well in the context of computational systems: "... Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice ...". Our solutions are expressed in terms of objects and their interfaces instead of architectonical elements such as walls and doors, but at the core of both kinds of patterns there is a solution to a problem in a given context.
Patterns, at a given level of abstraction, are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable OO design. The design pattern identifies the participating classes and instances, their roles and collaborations and the distribution of responsibilities. Each design pattern focuses on a particular OO design problem or issue. A design pattern, for implementation purposes, also provides sample code to illustrate an implementation. In [Gam&Al 95] C++ and/or Smalltalk codes are given, and these languages have been selected on the basis of experience in the language, increase of its popularity and language facility of expression. Actually, some patterns can be expressed more easily in one language than in the other.
A framework is defined as a reusable semifinished architecture for various application domains [Pre 95]. It is constituted by a set of classes, that may constitute several design patterns, conceived for working together and represents a generic subsystem that can be instantiated. The instantiation is acheived developing subclasses from the public abstract classes of the framework, called somtimes "hot spots" [Pre 95], which are seen as predefined places where application specific parts are composed. A developer using a framework must know the hot spots for a given problem and know how to adapt them to the application's needs. The MVC (Model-View-Controller) model [Gol 84] for building graphical user-interfaces (GUI) is considered one of the first known frameworks [KP 88].

3. AGENTS AND FRAMEWORKS FOR DESIGNING GRAPHICAL USER-INTERFACES

This section aims to present two well known multiagent models (MVC and PAC), for building object-oriented interactive systems, enhancing extensibility and reuse through the separation between the view (presentation or GUI) and the model (abstraction or application domain component) paradigm. The corresponding frameworks are shown, with an illustrative example.

3.1 Multiagent models

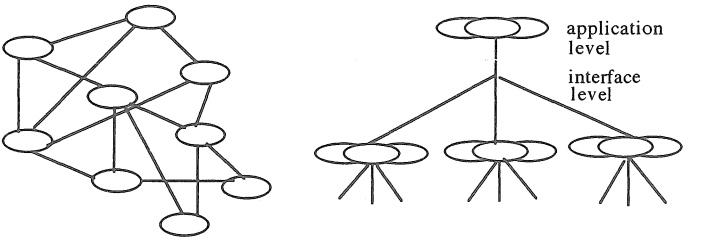
The multiagent model [Cic 84], [Gol 84]], [LVC 89] is used to describe the architecture of interactive systems. It is inspired from the stimuli-answers systems, which are organized as a set of *agents*, reacting at external events (stimuli) and generating events (answers). An *event* or *stimulus* is 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, such as the language [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 and its *instances* 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.

A well known multiagent model is the MVC paradigm [Gol 84], where the *Model, View, and Controller* perspectives, respectively, were originally presented as three distinct Smalltalk objects. The Model represents the abstraction or conceptual part of the application object, the View represents the screen presentation and the Controller defines the way the user interface reacts to user inputs. Each view has an associated controller that connects this particular view with an input device, such as mouse or keyboard. The traditional user-interface models tended to lump these objects together. The MVC agent or triad of classes (recent implementations consider only two classes, the Model and the View-Controller) has been originally conceived to build user-interfaces in Smalltalk-80 and the MVC model is now an accepted paradigm for building extensible GUI. The architecture of a system built according to the MVC paradigm is shown below in Figure 1.

Another interesting model inspired in MVC, the PAC (Presentation-Abstraction-Control) model [Cou 90], introduces the *Control* object notion for mediating between the *Presentation* and the *Abstraction*, maintaining the coherence between them, and assuring also the comunications among the agents. The Abstraction is similar to the MVC Model notion. The Presentation instead, corresponds to the MVC view-controller notion. Both approaches maintain the issue that the presentation or physical user interaction is kept separate from the semantics or conceptual part of the application, easing system development, extensibility and maintenance. Moreover, PAC considers agents communications only through their control perspectives, facilitating the development process and the work of programmer's teams [Los&Al 94]. The complete system architecture is expressed, considering a hierarchy of PAC agents: the top level agent constitute the whole system and its abstraction perspective corresponds to the application domain component (application level). The agents in the intermediate levels of the hierarchy constitute the user-interface of the system (interface level). Elementary agents, located at the lowest level, are constituted by predifined toolkit classes (widgets). Figure 1 shows the architecture of the system built following this approach.



Organization of MVC agents Organization of PAC agents

Figure 1. Different architecutres based on multiagent models

In what follows, the design patterns constituting the frameworks representing both models are discussed.

3.2 The MVC framework

Four basic design patterns characterize the MVC model: *Observer, Composite, Strategy, Factory Method.* Eventually, the *Decorator* pattern may also be used to add additional functionalities, such as scrolling to a view. The problem addressed, the complete description and reference number of each design pattern are found in [Gam&Al 95]. The notation followed to express graphically the framework is the OMT [Rum&Al 91] based notation, as used in [Gam&Al 95]. The four basic pattterns mentioned above are used for building the MVC framework presented in Figure 2 below, which captures the behaviour of an MVC agent.

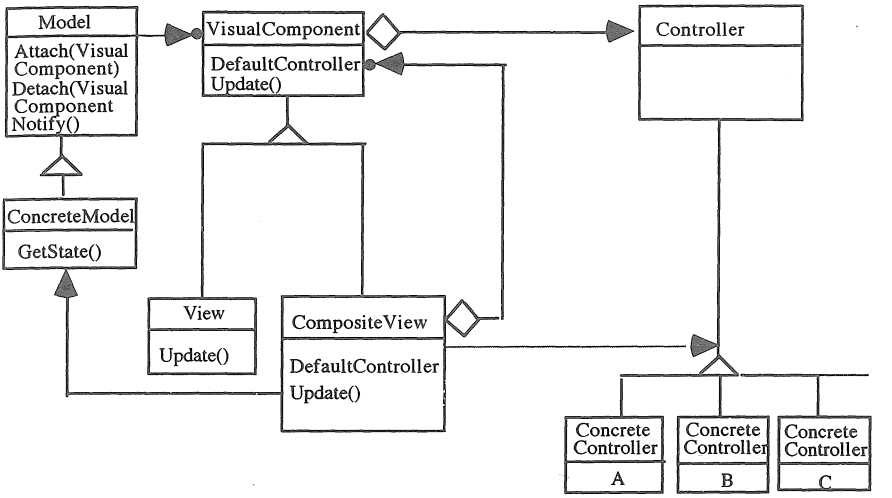


Figure 2. MVC framework

The *Observer* pattern causes the encapsulation of different aspects of an abstraction into separate objects, favoring their independent reuse and extension. In the MVC context, it is used to decouple views from models. In our framework, the Model creates the View-Controller pair, defining a one-to-many correspondence between the abstract classes Model (representing the MVC Model notion) and VisualComponent (representing the MVC View notion), so that, when the concrete class ConcreteModel changes its state, the corresponding concrete class CompositeView is automatically notified of the change. The *Composite* pattern groups objects and treats the group like an individual object. It is used to treat nested views. Objects are composed into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and composition of objects, uniformly. For example, a control panel of buttons can be implemented as a complex view containing nested button views. In the MVC framework, the reference relation between the classes VisualComponent and CompositeView establishes this recursive treatement. The *Strategy* pattern attaches a view to a controller allowing to change the way a view responds to user inputs. Notice that Strategy defines a family of algorithms, encapsulating each one and making them interchangeable. It lets the algorithm vary independently from clients that uses it. Finally, the *Factory Method* pattern is used to specify the default controller class for a view. So, for each instance of CompositeView, an instance of a particular algorithm for treating user inputs is selected. In Figure 1, this relation is denoted by a dashed arrow which points to the inheritance relation, instead of pointing to a

particular subclass (as in the OMT based notation), because, at framework level, we don't know which one of the algorithm has to be related to the particular CompositeView instance. Actually, this will be known only at the moment of instantiating the framework with a particular application. Notice also that we have added the dashed line, continuing the inheritance relation line, meaning that there may by any number of algorithms, depending on the application.

3.3 The PAC framework

Three main patterns characterize the PAC model: *Mediator, Strategy and Factory Method.* The framework corresponding to a PAC agent is shown in Figure 3 below.

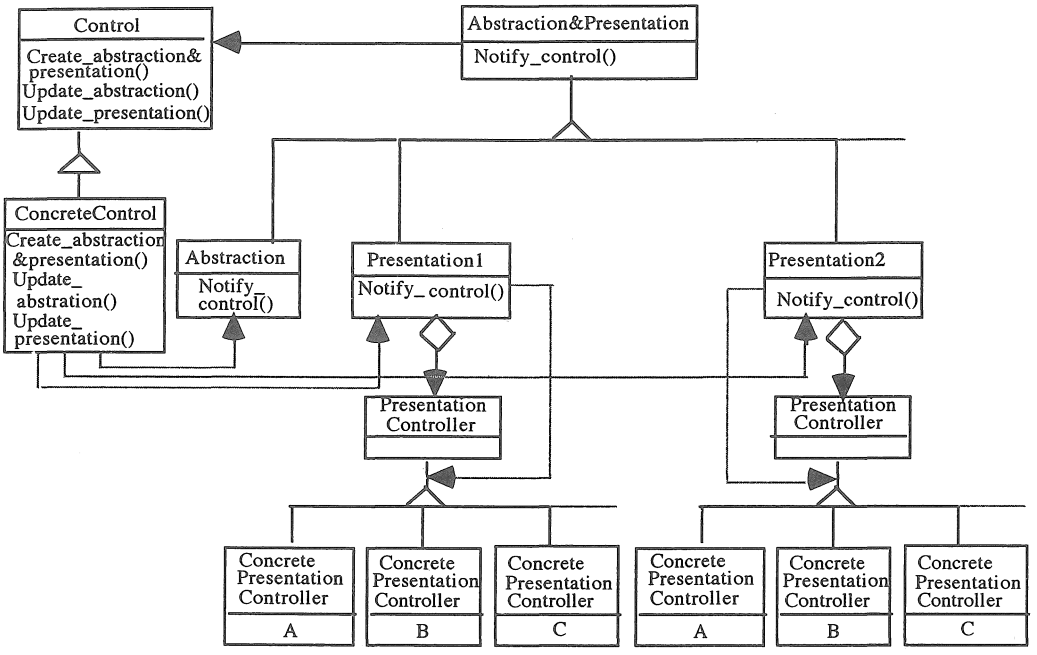


Figure 3. The PAC framework

The Control, which introduces the notion of communication between Abstraction and Presentation, mediating for the coherence between the two perspectives, is modelled by the *Mediator* pattern. An Abstraction and several optional Presentations are associated with each Control, denoted by Presentation1, Presentation2, etc. in Figure 3 (the PAC Presentation constitutes the View and Controller of MVC). Notice that, since a PAC agent may be decomposed in subagents, due to the PAC agents'hierarchy, each subagent has its own control, and the nested views are treated as subagents. The *Strategy* pattern is used in the same way as for MVC, and finally the *Factory Method* relates a view with the different incapsulated treatements of the user inputs, also like MVC.

4. AN EXAMPLE ILLUSTRATING THE INSTANTIATION OF THE MVC AND PAC FRAMEWORKS

Let us consider a simplified graphical editor, identified as the *Editor* agent in the main window, constituted by three additional agents (corresponding to three zones of the main windows): *Palette* for selecting the shapes to be drawn, *Menu* showing general facilities and *Edtion*, as the drawing area (see Figure 4). We will be considering only the Edition agent, in order to simplify the example, since the other agents are similarly treated.

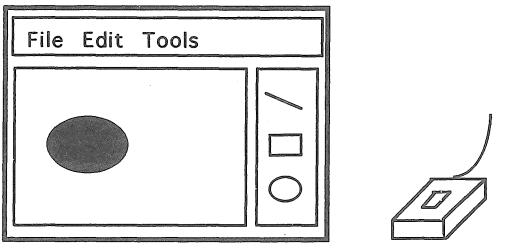
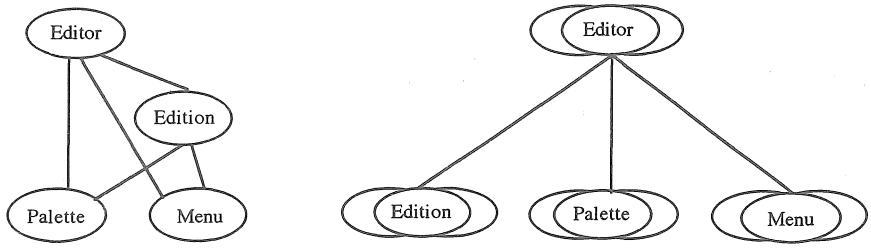


Figure 4. Agents for a simplified graphical editor, with one button mouse

Suppose now that we want to draw an oval in the drawing area. The figure will be displayed as a consequence of the following actions, supposing the oval sape selected on the palette (Palette agent) and the cursor moved to the drawing area (Edition agent): (a) the mouse botton is clicked twice and released and the figure is displayed on the drawing area. The display of the oval as a consequence of the double click is implemented by an algorithm that will be called A. Moreover, in the drawing area, the following situations may occur when the mouse button is clicked once: (b) If the cursor is inside the oval, the figure will be selected (filled with black color, for example); the actions corresponding to this situation are treated by algorithm B. (c) If we are in situation (a) and the cursor is outside the oval, the figure will be deselected; algorithm C treats this situation. (d) If the cursor is outside the oval, and situation (a) has not occurred, nothing happens. Notice then that we have four different situations in the drawing area, which is under control of the Edition agent: the figure is drawn, the figure is selected, the figure is deselected, there is no action taken, and three different algorithm handle the relevant situations.

Figure 5 below illustrates the different architectures obtained for the graphical editor, according to the to multiagent models presented:



Graphical Editor using the MVC model

Graphical Editor using the PAC model

Figure 5. Different architectures for a simplified graphical editor

Figure 6 below shows the instantiation of the MVC framework by the four agents of our editor, specifying the pattern classes only for the Edition agent. The classes corresponding to the Editor, Palette and Menu agents are not shown here in order to ease this presentation, but they can be developed in the same way as for Edition, according to the design patterns. The communications among the agentes are also shown in the figure.

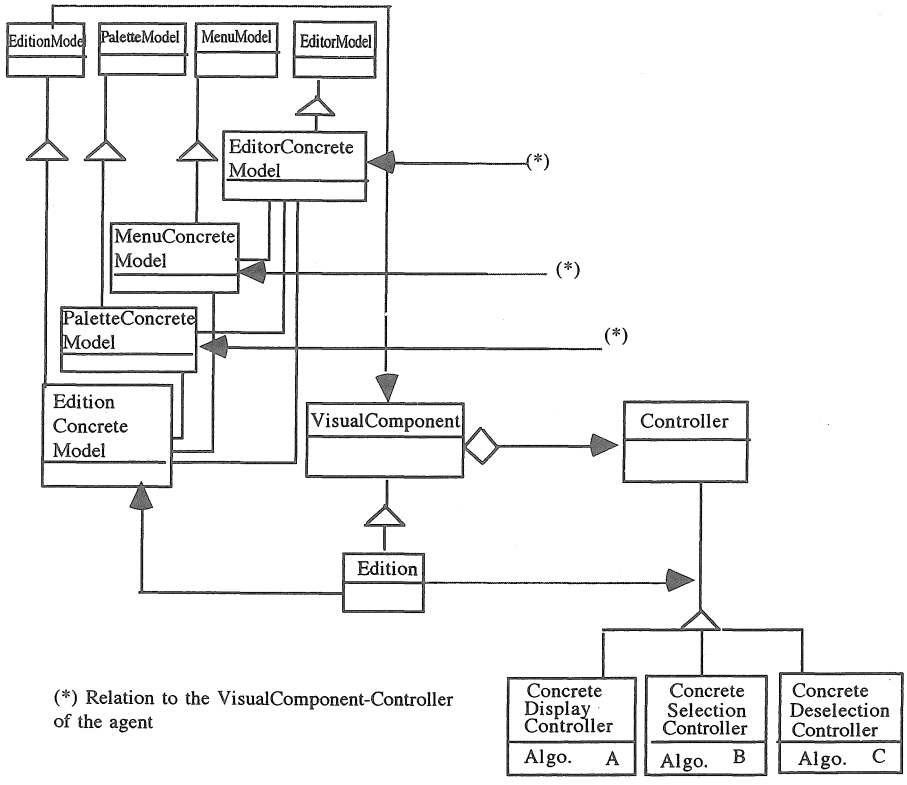


Figure 6. Instantiation of the MVC framework with the four agents of the graphical editor, showing also communications among the agents

Figure 7 below shows again the instantiation with the four agents of the editor, for the PAC framwork. Notice that the communication among the agents involved in the example, through their respective controls, are also shown: ConcreteEditorControl class is related with ConcretePaletteControl, ConcreteMenuControl and ConcreteEditionControl classes, respectively.

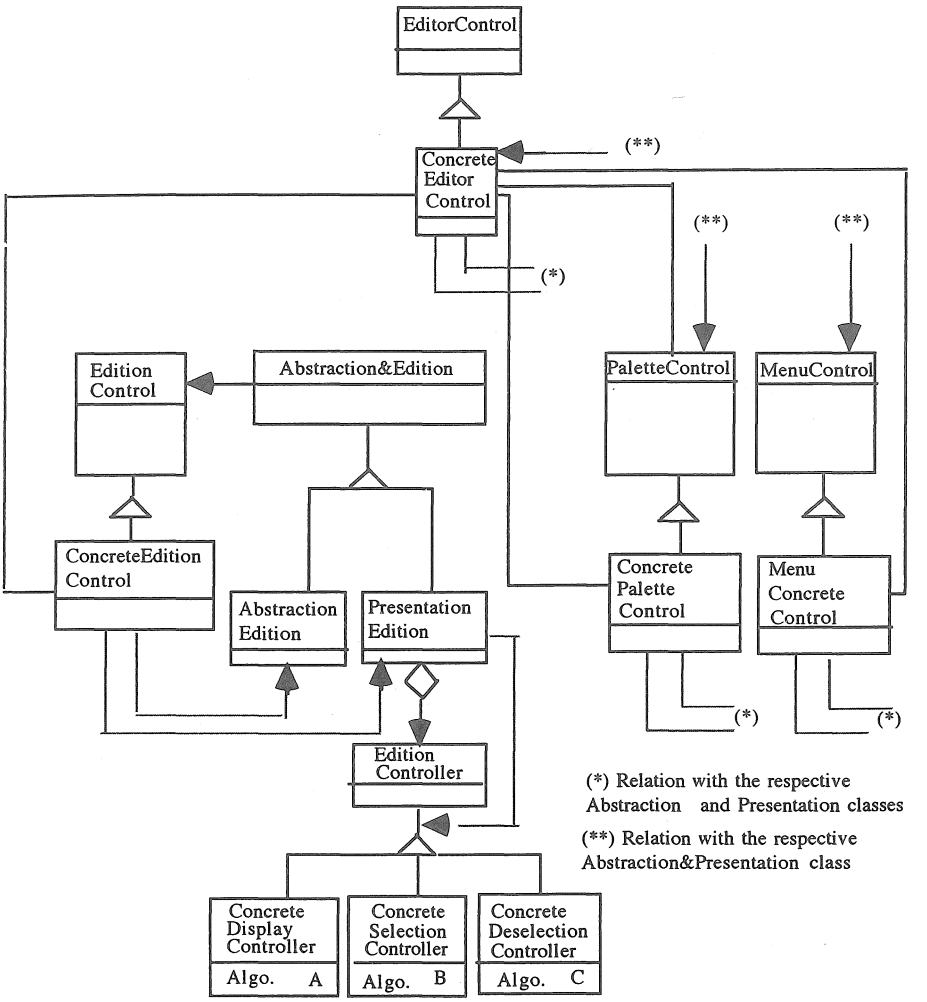


Figure 7. Instantiation of the PAC framework with the four agents of the graphical editor, showing also communications among the agents

Notice that the communication among the agents in both approaches establishes the different architectures of the resulting systems. Using the PAC model, a hierarchy of agents constituting the whole system is established, and the two main levels (application and interface) are quickly separated. Moreover, during a detailed design step, the communications between the agents through their respective ConcreteControl class, is directly obtained from the PAC model. We have to add that the PAC principles of communicating the agents only through their control perspective, has easily been established as an additional communication

framework [Los&Al 94]. The reusable frameworks and clear identification of communications' responsibilities among the agents facilitate groupware, in the sense that the different perspectives constituing a PAC agent can be coded by different programmers' teams. Moreover, the integration of the agent and the implementation of the control communications can also be the responsibility of another team, easing the whole software project control. With the MVC model, instead, we don't have an ordering on the relations and/or communications among the agents constituting the GUI of the system. These are identified and placed on a unique level and only some of the possible relations among them can be established. In our editor example (see Figures 1 and 5), it is easy to notice that the Edition agent has to communicate with the Menu, Palette and Edition agents and that Palette communicates with Edition, but in other applications this schema may be completely different. At detailed design level, the instantiation of the MVC framework only establishes that the agents may be related through their respective ConcreteModel class; nothing can be said about the communication of each individual agent with its neighbours, because no hierarchical relations are established among them.

5. A COMPARISON OF THE MVC AND PAC ARCHITECTURES AND THEIR FRAMEWORKS

We have seen that the MVC and PAC frameworks express the behaviour of an MVC and PAC agent respectively. Both agents consider multiple views, that is to say only one abstraction or model with different presentations or views. Moreover, an MVC agent takes account of nested views. A PAC agent, instead, considers a hierarchy of subagents. As we have mentioned in 3.1, the abstraction of the top level agent may be considered as a subsystem or set of classes constituting the conceptual part of the whole application, and cannot be considered like the other agents, which are part of the GUI. So, the PAC agent framework presented here cannot be applied for the application level agent. Notice also that, in general, this special agent lacks the presentation perspective, which is sometimes reduced to an icon. The agents constituting the GUI component of an interactive system built according to the MVC paradigm may only communicate through the ConcreteModel class (hot spot) of the framework (see Figures 2 and 6).Observe that VisualComponent and Controller classes capture and treat external events, encapsulating the corresponding algorithms. So, the communication among the agents is acheived only through the respective ConcreteModel classes Instead, if the PAC model is followed, the agents constituting the GUI of the interactive system will communicate through their respective ConcreteControl class (hot spot) of the framework (see Figure 3). The *Facade* pattern is used in both cases to model the communications between the objects of the GUI component and the objects of the application domain component. Figure 8 and 9 illustrate both architectures. Notice that in Figure 8, the communication established between Facade and the agents' ConcreteModel class, may involve any agent of the GUI component, enhancing high coupled systems. Nevertheless, the rigidity of the system structure imposed by the PAC model may be seen as a disadvantage, with respect to the efficiency established by the MVC model.

In figure 9, instead, Facade communicates the application domain component (Abstraction perspective of the top level PAC agent) with the GUI component, only through the ConcreteControl class of the agents located at the first level of the PAC hierarchy, playing the role of the control perspective of the top level PAC agent and enhancing low coupled systems, favoring adaptability.

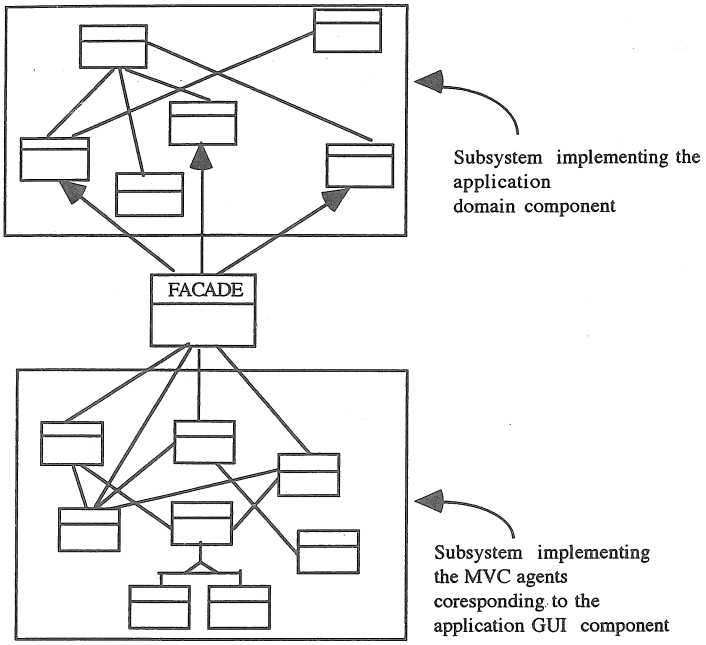


Figure 8. System architecture using the MVC model.

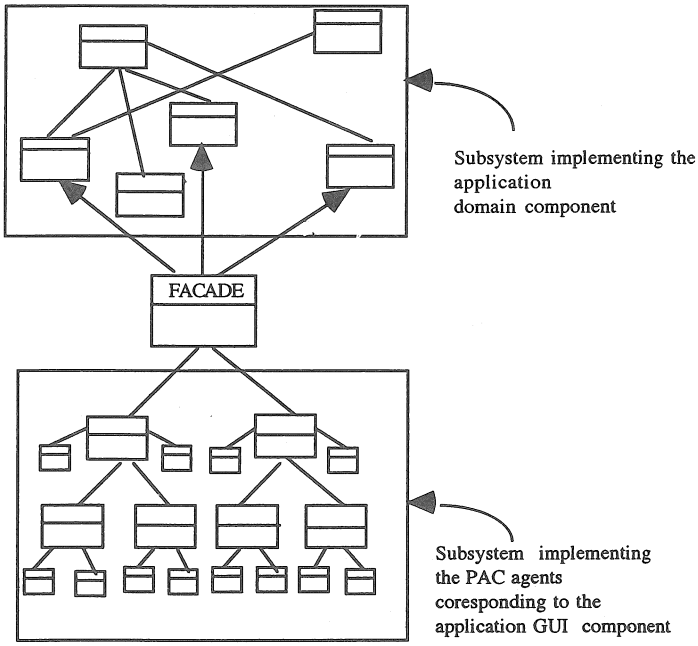


Figure 9. System architecture usign the PAC framework.

6. CONCLUSION

This work has presented two frameworks for agents used in the development of GUI. The MVC paradigm is now widley applied for building extensible GUI. Our experience in using the PAC approach is that it greatly favors groupware, facilitating the work of programmers' teams, at detailed design level: the agent framework is given to programmers, separating the teams developing abstraction and presentation classes (a programmer can be responsible of several agents); the team responsible for the control perspective integrates abstraction and presentation of the agents and establishes communications among the GUI agents, according to an established communication pattern, preserving the PAC principles [Los&Al 94]. Prototyping is also facilitated because a graphical interactive system is quickly constructed: the agents at interface level can be implemented without caring much about the application domain component (top level agent), that can be developed independently, since it has nothing to do with the graphical presentation aspects of the system. A method [Los 95] is being developed, based on a schema defined in [LM 95], for using the OOSE (Object Oriented Software Engineering) analisys model [Jac&Al 92] to obtain entity, interface and control objects as products of the requirements step and study their traceability through the design step, into PAC (or MVC) agents. A detailed design follows, applying the frameworks described here. A rapid and systematic object-oriented development is acheived, using C++ [Str 93] as the implementation language, on Unix[1]/X,Xt[2]Motif[3] and MS-DOS/Windows[4], ObjectWindows[5] platforms.

7. ACNOWLEDGMENT

The authors are indebted to Francisco Marchena and wish also to thanks the creative programming efforts of the ISYS programmers' teams.

8. REFERENCES

[Ale&Al 77] Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., Angel S. "A pattern Language", Oxford University Press, NY, 1977.

[Cic 84] CICCARELLI E. C., "Presentation Based User-Interfaces", Technical Report 794, Artificial Intelligence Laboratory, Massachussets Intelligence Laboratory, August 1984.

[Col 95] COLLINS D., "Designing Object-Oriented User Interfaces", Benjamin/Cummings Publishing Company, Inc. 1995.

[Cou 90] COUTAZ J., "Interfaces homme-ordinateur", Dunod 1990.

[F&D 84] FOLEY D.J., VAN DAM, "Fundamentals of Interactive Computer Graphics", Addison Wesley 1984.

[Gam&Al 95] Gamma E., Helm R., Johnson R.. Vlissides J. "Design Patterns. Elements of Reusable Object-Oriented Software" Addison Wesley Publishing Co., 1994.

[Gol 84] GOLDBERG A., "Smalltalk-80 The Interactive Programming Environment", Addison-Wesley 1984.

[Jac&Al 92] JACOBSON I., CHRISTERSON M, JONSSON P., ÖVERGAARD G., "Object-Oriented Software Engineering, a Use Case Driven Approach", Addison Wesley, 1992.

---

[1]Unix is a registered trademark of Unix System Laboratories.

[2]X-Window is a registered trademark of the Massachusetts Institute of Technology.

[3]Motif is a registered trademark of Open Software Foundation.

[4]MS-DOS, Windows are registered trademarks of Microsoft

[5]ObjectWindows is a registered trademark of Borland Int.

[KP 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 1(3), 1988.

[LM 95] LOSAVIO F., MATTEO A. "Use-Case and Multiagent Models for Object-Oriented Design of User-Interfaces" LRI, Orsay, France, Research Report No. 992, September 1995. To appear in Journal of Object Oriented Programming

[Los&Al 94] LOSAVIO F., MATTEO A., ORDAZ O., MEZA O., GONTIER W. "An implementation of the PAC architecture using object-oriented techniques", IFIP'94, 13th World Computer Congress 94, Volume 2, 149-155. Hamburg-Germany, August 1994, K. Brunnstein and E. Raubold (Editors) Elsevier Science B.V. (North-Holland)

[Los 95] LOSAVIO F. "An Object-Oriented Methodology for User-Interface Design Based on the Multiagent Model", Proceedings of the Information Systems Analysis Synthesis, ISAS'95, 89-92, Baden-Baden, Germany, August 1995.

[LVC 89] LINTON M.A., VLISSIDES J.M., CLADER V., "Composing User-interfaces with Interviews", IEEE, Computer, 8-22, Feb. 1989.

[Pre 95] PREE W. "Design Patterns for Object-Oriented Software Development", Addison Wesley 1994.

[Mey 88] MEYER B., "Object-Oriented Software Construction", Prentice Hall 1988.

[Rum&Al 91] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F., LORENSEN W. "Object-Oriented Modeling and Design" Prentice Hall International, Inc (1991)

[Str 93] STROUSTRUP B., "The C++ Programming Language", Second Edition, Addison-Wesley Publishing Company, 1993.

Francis LOSAVIO received the Doctor degree in Computer Science in 1991 and a 3ème. Cycle Doctor degree in Computer Science in 1985 from the Univ. Paris-Sud, Orsay, France. She also received a MSC degree in Computer Science from the Univ. Simón Bolivar , Venezuela in 1983. At present, she is Titular Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and coordinates the ISYS Research Center. Her research includes software engineering environments, methodologies for software development, automatic program construction and formal specifications.

Alfredo MATTEO received the Doctor degree in Computer Science from the Univ. Paul Sabatier, Toulouse, France in 1984. At present, he is an Aggregate Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and is member of the ISYS Research Center. His research includes software engineering environments, methodologies for software development, formal specifications and algorithmic complexity.