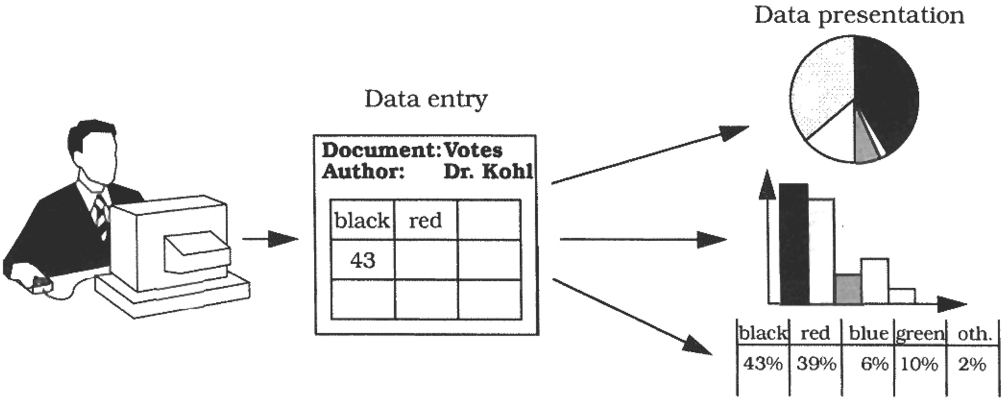
Presentation-Abstraction-Control

The *Presentation-Abstraction-Control* architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Example Consider a simple information system for political elections with proportional representation. This offers a spreadsheet for entering data and several kinds of tables and charts for presenting current standings. Users interact with the software through a graphical interface.



black	red	blue	green	oth.
43%	39%	6%	10%	2%

Different versions, however, adapt the user interface to specific needs. For example, one version supports additional views of the data, such as the assignment of parliament seats to political parties.

Context Development of an interactive application with the help of agents[19].

[19] In the context of this pattern an *agent* denotes an information-processing component that includes event receivers and transmitters, data structures to maintain state, and a processor that handles incoming events, updates its own state, and that may produce new events [BaCo91]. Agents can be as small as a single object, but also as complex as a complete software system. We use the terms *agent* and *PAC agent* as synonyms in this pattern description.

Problem Interactive systems can often be viewed as a set of cooperating agents. Agents specialized in human-computer interaction accept user input and display data. Other agents maintain the data model of the system and offer functionality that operates on this data. Additional agents are responsible for diverse tasks such as error handling or communication with other software systems. Besides this horizontal decomposition of system functionality, we often encounter a vertical decomposition. Production planning systems (PPS), for example, distinguish between production planning and the execution of a previously specified production plan. For each of these tasks separate agents can be defined.

In such an architecture of cooperating agents, each agent is specialized for a specific task, and all agents together provide the system functionality. This architecture also captures both a horizontal and vertical decomposition. The following *forces* affect the solution:

- Agents often maintain their own state and data. For example, in a PPS system, the production planning and the actual production control may work on different data models, one tuned for planning and simulation and one performance-optimized for efficient production. However, individual agents must effectively cooperate to provide the overall task of the application. To achieve this, they need a mechanism for exchanging, data, messages, and events.
- Interactive agents provide their own user interface, since their respective human-computer interactions often differ widely. For example, entering data into spreadsheets is done using keyboard input, while the manipulation of graphical objects uses a pointing device.
- Systems evolve over time. Their presentation aspect is particularly prone to change. The use of graphics, and more recently, multimedia features, are examples of pervasive changes to user interfaces. Changes to individual agents, or the extension of the system with new agents, should not affect the whole system.

Solution Structure the interactive application as a tree-like hierarchy of *PAC agents*. There should be one top-level agent, several intermediate-level agents, and even more bottom-level agents. Every agent is responsible for a specific aspect of the application's functionality, and consists of three components: presentation, abstraction, and control.

The whole hierarchy reflects transitive dependencies between agents. Each agent depends on all higher-level agents up the hierarchy to the top-level agent.

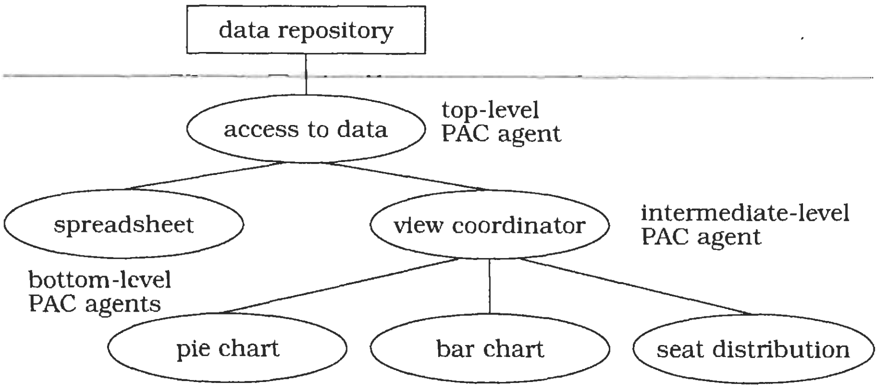
The agent's *presentation* component provides the visible behavior of the PAC agent. Its *abstraction* component maintains the data model that underlies the agent, and provides functionality that operates on this data. Its *control* component connects the presentation and abstraction components, and provides functionality that allows the agent to communicate with other PAC agents.

The *top-level PAC agent* provides the functional core of the system. Most other PAC agents depend or operate on this core. Furthermore, the top-level PAC agent includes those parts of the user interface that cannot be assigned to particular subtasks, such as menu bars or a dialog box displaying information about the application.

Bottom-level PAC agents represent self-contained semantic concepts on which users of the system can act, such as spreadsheets and charts. The bottom-level agents present these concepts to the user and support all operations that users can perform on these agents, such as zooming or moving a chart.

Intermediate-level PAC agents represent either combinations of, or relationships between, lower-level agents. For example, an intermediate-level agent may maintain several views of the same data, such as a floor plan and an external view of a house in a CAD system for architecture.

➥ Our information system for political elections defines a top-level PAC agent that provides access to the data repository underlying the system. The data repository itself is not part of the application. At the bottom level we specify four PAC agents: one spreadsheet agent for entering data, and three view agents for each type of diagram for representing the data. The application has one intermediate-level PAC agent. This coordinates the three bottom-level view agents and keeps them consistent. The spreadsheet agent is directly connected to the top-level PAC agent. Users of the system only interact with bottom-level agents.



❑

Structure The main responsibility of the *top-level PAC agent* is to provide the global data model of the software. This is maintained in the abstraction component of the top-level agent. The interface of the abstraction component offers functions to manipulate the data model and to retrieve information about it. The representation of data within the abstraction component is media-independent. For example, in a CAD system for architecture, walls, doors, and windows are represented in centimeters or inches that reflect their real size, not in pixels for display purposes. This media-independency supports adaptation of the PAC agent to different environments without major changes in its abstraction component.

The presentation component of the top-level agent often has few responsibilities. It may include user-interface elements common to the whole application. In some systems, such as the network traffic manager [TS93], there is no top-level presentation component at all.

The control component of the top-level PAC agent has three responsibilities:

- It allows lower-level agents to make use of the services of the top-level agents, mostly to access and manipulate the global data model. Incoming service requests from lower-level agents are forwarded either to the abstraction component or the presentation component.
- It coordinates the hierarchy of PAC agents. It maintains information about connections between the top-level agent and lower-level agents. The control component uses this information to

ensure correct collaboration and data exchange between the top-level agent and lower-level agents.

- It maintains information about the interaction of the user with the system. For example, it may check whether a particular operation can be performed on the data model when triggered by the user. It may also keep track of the functions called to provide history or undo/redo services for operations on the functional core.

➥ In our example information system for political elections, the abstraction component of the top-level PAC agent provides an application-specific interface to the underlying data repository. It implements functions for reading and writing election data. It also implements all functions that operate on the election data, such as algorithms for calculating projections and seat distributions. It further includes functions for maintaining data, such as those for updating and consistency checking. The control component organizes communication and cooperation with lower-level agents, namely the view coordinator and spreadsheet agents. This top-level PAC agent does not include a presentation component. ❑

Bottom-level PAC agents represent a specific semantic concept of the application domain, such as a mailbox in a network traffic management system [TS93] or a wall in a mobile robot system [Cro85]. This semantic concept may be as low-level as a simple graphical object such as a circle, or as complex as a bar chart that summarizes all the data in the system.

The presentation component of a bottom-level PAC agent presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it. Internally, the presentation component also maintains information about the view, such as its position on the screen.

The abstraction component of a bottom-level PAC agent has a similar responsibility as the abstraction component of the top-level PAC agent, maintaining agent-specific data. In contrast to the abstraction component of the top-level agent, however, no other PAC agents depend on this data.

The control component of a bottom-level PAC agent maintains consistency between the abstraction and presentation components, thereby avoiding direct dependencies between them. It serves as an adapter and performs both interface and data adaptation.

The control component of bottom-level PAC agents communicates with higher-level agents to exchange events and data. Incoming events—such as a 'close window' request—are forwarded to the presentation component of the bottom-level agent, while incoming data is forwarded to its abstraction component. Outgoing events and data, for example error messages, are sent to the associated higher-level agent.

Concepts represented by bottom-level PAC agents, such as the bar and pie charts in the example, are atomic in the sense that they are the smallest units a user can manipulate. For the election system this means that users can only operate on the bar chart as a whole, for instance by changing the scaling factor of the y-axis. They cannot, for example, resize an individual bar of a bar chart.

Bottom-level PAC agents are not restricted to providing semantic concepts of the application domain. You can also specify bottom-level agents that implement system services. For example, there may be a communication agent that allows the system to cooperate with other applications and to monitor this cooperation.

➠ Consider a bar-chart agent in our information system for political elections. Its abstraction component saves the election data presented in the chart, and maintains chart-specific information such as the order of presentation for the data. The presentation component is responsible for displaying the bar chart in a window, and for providing all the functions that can be applied to it, such as zooming, moving, and printing. The control component serves as a level of indirection between the presentation and abstraction components. The control component is also responsible for the bar-chart agent's communication with the view coordinator agent. ❑

Intermediate-level PAC agents can fulfill two different roles: *composition* and *coordination*. When, for example, each object in a complex graphic is represented by a separate PAC agent, an intermediate-level agent groups these objects to form a composite graphical object. The intermediate-level agent defines a new abstraction, whose behavior encompasses both the behavior of its components and the new characteristics that are added to the composite object. The second role of an intermediate-level agent is to maintain consistency between lower-level agents, for example when coordinating multiple views of the same data.

The abstraction component maintains the specific data of the intermediate-level PAC agent. The presentation component implements its user interface. The control component has the same responsibilities of the control components of bottom-level PAC agents and of the top-level PAC agent.

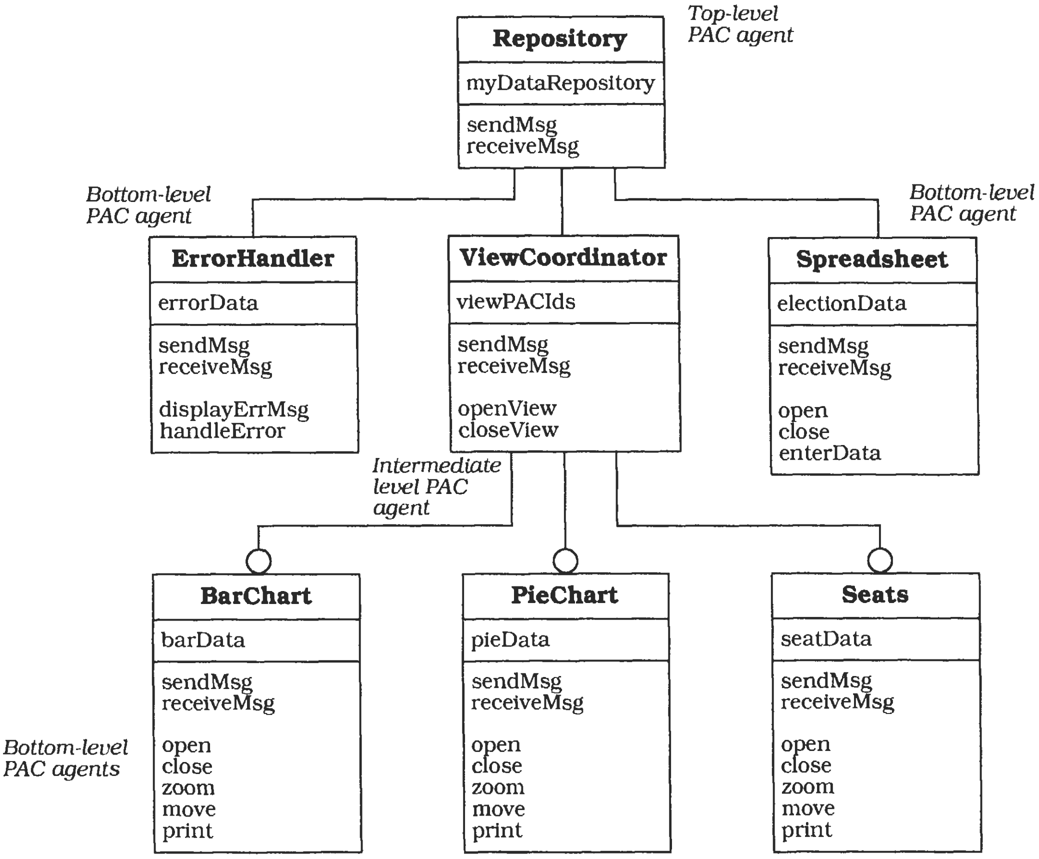
➥ Our example information system for political elections defines one intermediate-level PAC agent. Its presentation component provides a palette that allows users to create views of the election data, such as bar or pie charts. The abstraction component maintains data about all currently-active views, each of which is realized by its own bottom-level agent. The main responsibility of the control component is to coordinate all subordinate agents. It forwards incoming notifications about data model changes taking place in the top-level agent to the bottom-level agents, and organizes their update. It also includes functionality to create and delete bottom-level agents on user request. ❑

Class	***Collaborators***
Top-level Agent	• Intermediate-level Agent • Bottom-level Agent
Responsibility	
• Provides the functional core of the system. • Controls the PAC hierarchy.	

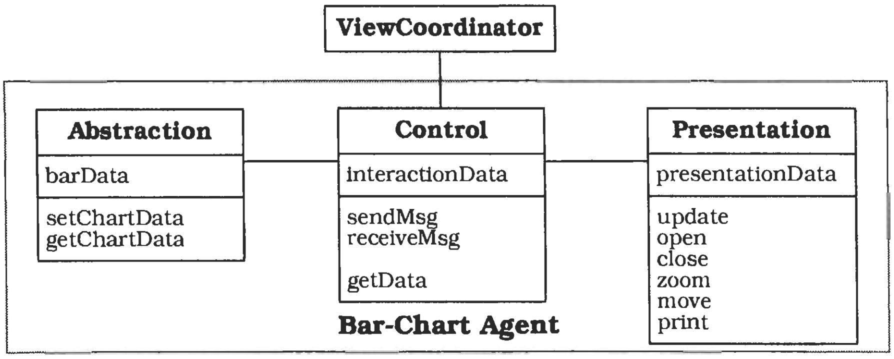
Class	***Collaborators***
Interm. -level Agent	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent
Responsibility	
• Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	

Class	***Collaborators***
Bottom-level Agent	• Top-level Agent • Intermediate-level Agent
Responsibility	
• Provides a specific view of the software or a system service, including its associated human-computer interaction.	

The following OMT diagram illustrates the PAC hierarchy of the information system for political elections. However, it only lists those functions that are necessary for controlling and coordinating the PAC hierarchy, or which are accessible to other PAC agents or to the user. We keep the interfaces of PAC agents small by applying the Composite Message pattern [SC95b]. All incoming service requests, events, and data are handled by a single function called `receiveMsg()`. This interprets messages and routes them to their intended recipient, which may be the abstraction or presentation components of the agent, or of another agent. Similarly, the function `sendMsg()` is used to pack and deliver service requests, events, and data to other agents. Another approach would be to provide an agent-specific interface that includes all the services the agent offers. The consequences of both these approaches are discussed in the Implementation section.



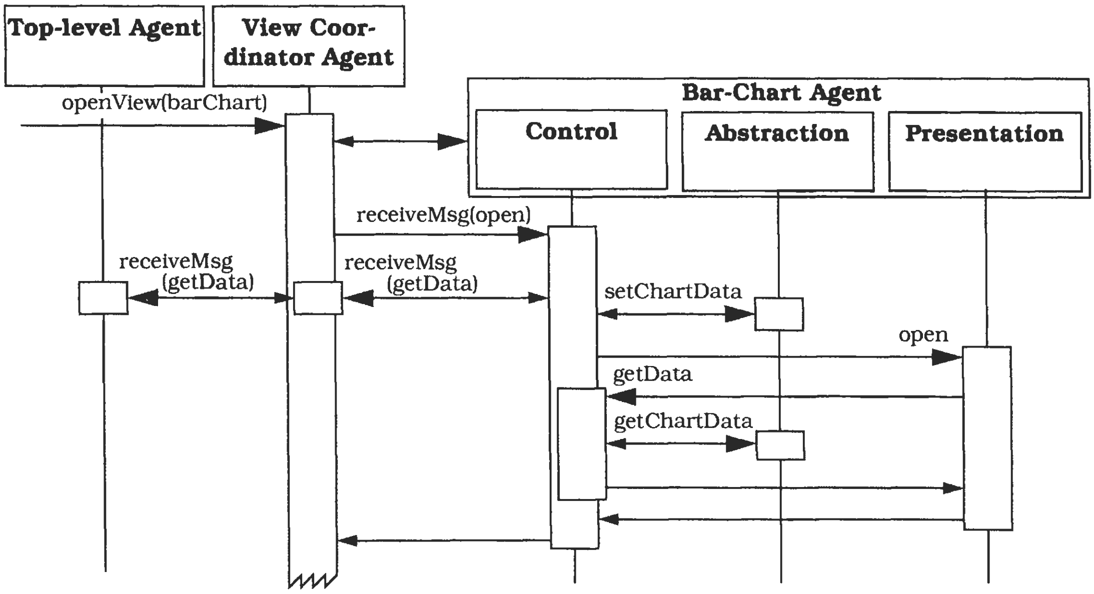
The internal structure of a PAC agent is shown below, using the bar-chart agent from our example:



Dynamics We will illustrate the behavior of a PAC architecture with two scenarios, both based on our election system example.

Scenario I describes the cooperation between different PAC agents when opening a new bar-chart view of the election data. The scenario also includes a more detailed description of the internal behavior of the bar-chart agent. It is divided into five phases:

- A user asks the presentation component of the view coordinator agent to open a new bar chart.
- The control of the view coordinator agent instantiates the desired bar-chart agent.
- The view coordinator agent sends an 'open' event to the control component of the new bar-chart agent.
- The control component of the bar-chart agent first retrieves data from the top-level PAC agent. The view coordinator agent mediates between bottom and top-level agents. The data returned to the bar-chart agent is saved in its abstraction component. Its control component then calls the presentation component to display the chart.
- The presentation component creates a new window on the screen, retrieves data from the abstraction component by requesting it from the control component, and finally displays it within the new window.

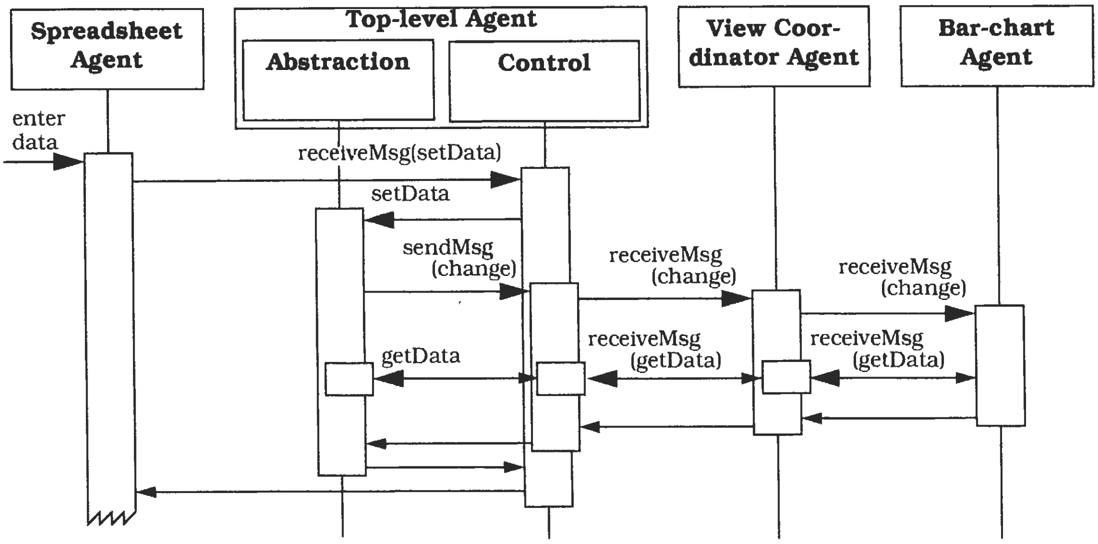


There are obvious optimizations possible here, such as caching top-level data in the view coordinator, or calling the bottom-level presentation component first and then storing the data. At this point, however, our emphasis is on explaining the basic ideas of the pattern.

Scenario II shows the behavior of the system after new election data is entered, providing a closer look at the internal behavior of the top-level PAC agent. It has five phases:

- The user enters new data into a spreadsheet. The control component of the spreadsheet agent forwards this data to the top-level PAC agent.
- The control component of the top-level PAC agent receives the data and tells the top-level abstraction to change the data repository accordingly. The abstraction component of the top-level agent asks its control component to update all agents that depend on the new data. The control component of the top-level PAC agent therefore notifies the view coordinator agent.
- The control component of the view coordinator agent forwards the change notification to all view PAC agents it is responsible for coordinating.

- As in the previous scenario, all view PAC agents then update their data and refresh the image they display.



Implementation To implement a PAC architecture, carry out the following ten steps, repeating any step or group of steps as necessary.

1 *Define a model of the application.* Analyze the problem domain and map it onto an appropriate software structure. Do not consider the distribution of components to PAC agents when performing this step. Concentrate on finding a proper decomposition and organization of the application domain. To this end, answer the following questions:

- Which services should the system provide?
- Which components can fulfill these services?
- What are the relationships between components?
- How do the components collaborate?
- What data do the components operate on?
- How will the user interact with the software?

Follow an appropriate analysis method when specifying the model.

2 *Define a general strategy for organizing the PAC hierarchy.* At this point we have not yet defined individual agents, but can specify general guidelines for organizing the hierarchy of cooperating agents.

One rule to follow is that of 'lowest common ancestor'. When a group of lower-level agents depends on the services or data provided by another agent, we try to specify this agent as the root of the subtree formed by the lower-level agents. As a consequence only agents that provide global services rise to the top of the hierarchy. For example, all agents in the election system depend on the central data repository. This is therefore provided by the top-level PAC agent. If only a fraction of all agents depend on the repository, we would try to group them into a subtree and define an agent holding the repository at the root of that subtree.

A second aspect to consider is the depth of the hierarchy. Most PAC architectures comprise several intermediate levels of PAC agents. In the Mobile Robot system [Cro85], for example, bottom-level agents are composed to environments which again are composed to workspaces —this is covered in more detail in the description of the Mobile Robot system in the Known Uses section. The deeper the hierarchy, the better it often reflects the decomposition of an application into self-contained concepts. On the other hand, deep hierarchies tend to be inefficient at run-time, and also hard to maintain. Finding the appropriate decomposition of a system into PAC agents is important to be able to gain the benefits of this architecture.

3 *Specify the top-level PAC agent.* Identify those parts of the analysis model that represent the functional core of the system. These are mostly components that maintain the global data model of the system, and components directly operating on this data. Identify also all user interface elements that are common to the whole application, such as menu bars or dialogs with information about the system. All components identified in this step will be part of the top-level agent.

4 *Specify the bottom-level PAC agents.* Identify those components of the analysis model that represent the smallest self-contained units of the system on which the user can perform operations or view presentations. In our example system, these units are the various diagrams and charts presenting election data, and the spreadsheet for entering this data.

For each of these units, identify those components that provide the human-computer interaction associated with them. The bar chart in our example requires a window in which the diagram is displayed, and functionality to manipulate the diagram, such as zooming and printing. Each semantic concept such as a bar chart and its user interface components together form a separate bottom-level agent.

5 *Specify bottom-level PAC agents for system services.* Often an application includes additional services that are not directly related to its primary subject. In our example system we define an error handler. Other systems may provide services for communicating with other systems or for configuration purposes. Each of these services, including their human-computer interaction, can be implemented as a separate bottom-level agent [BaCo91].,

6 *Specify intermediate-level PAC agents to compose lower-level PAC agents.* Often, several lower-level agents together form a higher-level semantic concept on which users can operate.

In the mobile robot system described in [Cro85], several wall, place, and route PAC agents form an environment. Users of the system can specify new environments, and missions for robots within environments. Environments are displayed on the screen, and users perform actions such as scrolling and zooming on these presentations. An environment is therefore a higher-level concept with its own functionality and human-computer interaction. Such concepts are implemented as separate agents. They provide their own human-computer interaction, and operate on their constituent lower-level agents.

➥ Our election example does not provide semantic concepts above individual charts, diagrams, and spreadsheets. Therefore we do not define PAC agents for composing other PAC agents. ❑

7 *Specify intermediate-level PAC agents to coordinate lower-level PAC agents.* Many systems offer multiple views of the same semantic concept. For example, in text editors you find 'layout' and 'edit' views of a text document. When the data in one view changes, all other views must be updated. Such coordination components, which you may have identified when modeling the analysis model, provide their own human-computer interaction; for example, menu entries and associated callback functions. The view coordinator agent of our example system is such an intermediate-level agent. To implement agents that coordinate multiple views you may apply thc View Handler pattern (291).

Note that views are not the only aspect of an application that must be coordinated. The network traffic management system described in the

Known Uses section [TS93], for example, implements an agent that coordinates the different concurrent jobs the system performs in a telecommunication network.

8 *Separate core functionality from human-computer interaction.* For every PAC agent, introduce presentation and abstraction components. All components that provide the user interface of the agent, such as graphical images presented to the user, presentation-specific data like screen coordinates, or menus, windows, and dialogs form the presentation part. All components that maintain core data or operate on them form the abstraction.

You can provide a unified interface to the abstraction and presentation components of a PAC agent by applying the Facade pattern [GHJV95]. The control component exports those parts of the abstraction and presentation interfaces that other components can use.

For some PAC agents it may be hard to specify presentation or abstraction parts. For example, top-level PAC agents often do not provide a presentation component. [BaCo91] suggest the implementation of the top-level presentation as a general geometry manager that maintains spatial relationships between the presentation components of lower-level PAC agents. You can apply the Command Processor pattern (277) to further organize the presentation component. This allows you to schedule user requests for deferred or prioritized execution, and to provide agent-specific undo/redo services.

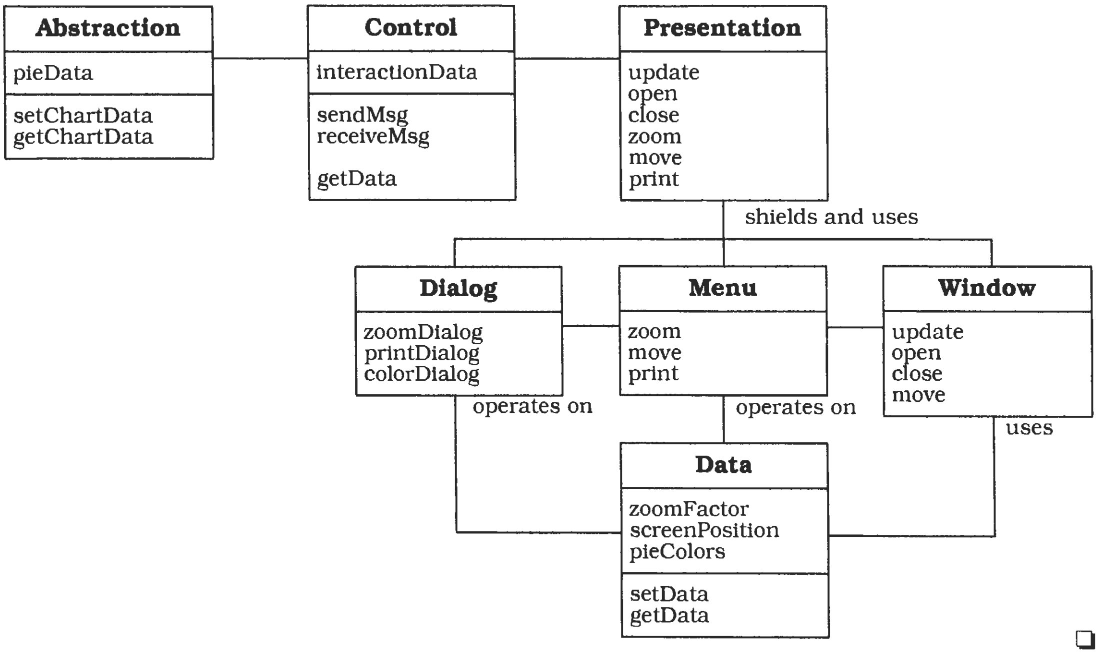
Some abstraction components, especially those in lower-level agents, often operate on data provided by other PAC agents. In this case, you may either not specify an abstraction component, or design the application such that the abstraction component just serves as a data cache. In the first case, you save all the effort of implementing components to keep replica data, and the functionality to keep these replica consistent. In the latter case, you save additional communication effort between PAC agents, for example when refreshing a view after a window is moved.

Finally, introduce the control component to mediate between the abstraction and presentation components, and to avoid direct dependencies between them. The control component is implemented as an Adapter [GHJV95]. It links the presentation and abstraction components together by performing interface and data adaptation between them. In this step, do not consider the parts of the control component that deal with the communication between the agent and other PAC agents. That is a different role of the control component, and should therefore be separated from the mediation between the agent internal abstraction and presentation components.

➥ To illustrate this step in our example, we refine the bar-chart agent from the example, as described in the Structure section. The abstraction component keeps a copy of the election data displayed in the bar chart.

The presentation component is structured into components that provide the functionality of windowing, menus, dialogs, and of maintaining presentation-specific data. To shield clients from this structure we provide a Facade [GHJV95].

The control component of the pie chart PAC agent is simple. It just forwards data read requests from the presentation component to the abstraction component. Communication with higher-level agents is handled in the next step.



❑

9 *Provide the external interface.* To cooperate with other agents, every PAC agent sends and receives events and data. Implement this functionality as part of the control component.

Within an agent, incoming events or data are forwarded to their intended recipient. The recipient may be the abstraction or the presentation component of the agent, but may also be lower or higher-level agents. For example, the view coordinator agent of our information system regularly receives change notifications from the top-level PAC agent and forwards them to the view agents. It also receives requests from lower-level agents that are forwarded to the top-level agent. In other words, the control component is a mediator—you may use the Mediator pattern [GHJV95] to implement this role.

One way of implementing communication with other agents is to apply the Composite Message pattern [SC95b]. This keeps the interface of an agent small. It also allows agents to be independent of the specific interfaces of other agents, and also of particular data formats, marshaling, unmarshaling, fragmentation and re-assembling methods. Applying the Composite Message pattern requires, however, that the control component interprets incoming messages. It must decide what to do with them—calling the abstraction or presentation components, or forwarding the message to another agent. This functionality is usually very complex and hard to implement.

A second option is to provide a public interface that offers every service of an agent as a separate function. These functions 'know' how to handle data and events when called. Compared to the Composite Message solution, this reduces the inner complexity of the control component, but introduces additional dependencies between agents—they depend on the specific interfaces of other agents. In addition, in this approach the interface of an agent can 'explode'. For example, an intermediate-level agent must offer all the functions of the top-level agent that are called by its associated lower-level agents. Vice versa, the intermediate-level agent must offer all the services of its associated lower-level agents that are called by the top-level agent. The interface of an agent may become complex and hard to maintain as a result.

A PAC agent can be connected to other PAC agents in a flexible and dynamic way by using *registration functionality*, as introduced by the Publisher-Subscriber pattern (339). For example, if a new instance of the bar-chart agent in our election system is created, it is dynamically registered with the view coordinator agent.

If a PAC agent depends on data or information maintained by other PAC agents, you should provide a change-propagation mechanism. Such a mechanism should involve all agents and all levels of the hierarchy and work in both directions. When changes to data occur within an agent, its abstraction component starts the change propagation. The control component forwards change notifications to all dependent PAC agents, but often also to the presentation component. Incoming change notifications from other agents cause the abstraction and presentation components to update their internal states. One way to implement such a change-propagation mechanism is to use the Publisher-Subscriber pattern (339). Another way is to integrate change propagation with the general functionality for sending and receiving events, messages, and data; see the example code below.

The interface for these communication and cooperation functions should be the same for all PAC agents. This supports re-configuration and reuse of PAC agents, and the extension of the application with new PAC agents.

➥ The control component of the view coordinator PAC agent in our election example provides the following interface:

```
enum ViewKind { barChart, pieChart, seats };
     // type of available views of election data
class DataSetInterface { /* ... */ };
     // Common interface for datasets, messages, and
     // events, according to the specifications of the
     // Composite Message pattern [SC95b]
class PACId { /* ... */ };
     // Provides a handle to a PAC agent
class VCControl {
     // Data member specifications
     PACId         parent;  // higher-level agent
     List<PACId>   children;// lower-level agents
     // More data member specifications ...
private:
     void attach(PACId agent, parentAgent = 0);
     void detach(PACId agent);
          // Registration functionality for connecting
          // dependent view agents and the top-level agent
          // with the view coordinator agent.
```

```
        DataSetInterface sendMsg(DataSetInterface data);
            // Sending events, messages, or data to other PAC
            // agents including change notifications
        void openView(ViewKind kind);
        void closeView(PACId agent);
            // Opening and closing views including
            // creation, registration,and deletion
            // of bottom-level agents displaying charts
    public:
        DataSetInterface receiveMsg(DataSetInterface data);
            // Receiving events, messages, or data from other
            // PAC agents including change notifications
    };
```

`sendMsg()` and `receiveMsg()` return objects for holding answers to the messages sent and received. ❑

10 *Link the hierarchy together.* After implementing the individual PAC agents you can build the final PAC hierarchy. Connect every PAC agent with those lower-level PAC agents with which it directly cooperates.

Provide the PAC agents that dynamically create and delete lower-level PAC agents with functionality to dynamically extend or reduce the PAC hierarchy. For example, the view coordinator agent in our information system creates a new view PAC agent if the user wants to open a particular view, and deletes this agent when the user closes the window in which the view is displayed.

Variants Many large applications—especially interactive ones—are multi-user systems. Multi-tasking is thus a major concern when designing such software systems. The following two variants of PAC address this force.

PAC agents as active objects. Many applications, especially interactive ones, benefit from multi-threading. The mobile robot system [Cro85] is an example of a multi-threaded PAC architecture. Every PAC agent can be implemented as an active object that lives in its own thread of control. Design patterns like Active Object and Half-Sync/Half-Async [Sch95] can help you implement such an architecture.

PAC agents as processes. To support PAC agents located in different processes or on remote machines, use proxies (263) to locally represent these PAC agents and to avoid direct dependencies on their physical location. Use the Forwarder-Receiver pattern (307) or the

Client-Dispatcher-Server pattern (323) to implement the inter-process communication (IPC) between PAC agents.

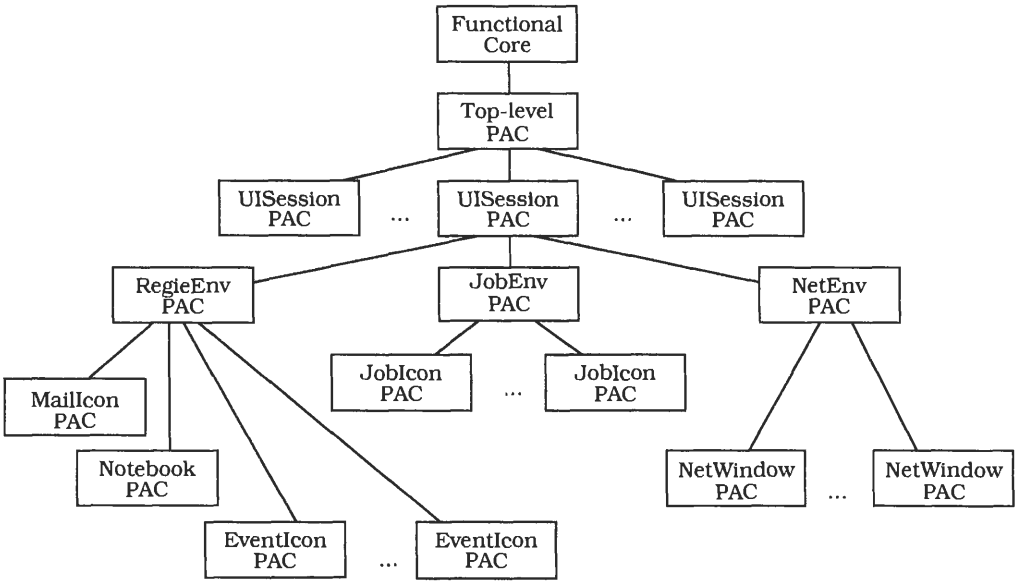
Since IPC is inefficient, you can also consider organizing coherent subtrees of the PAC hierarchy within different processes. Agents that cooperate closely in carrying out a particular task are then located within the same process. IPC between PAC agents is minimized, and is only necessary for coordinating different subtrees, as well as for accessing the services of the top-level PAC agent.

Known Uses

Network Traffic Management. This system is described in [TS93]. It displays the traffic in telecommunication networks. Every fifteen minutes all monitored switching units report their current traffic situation to a control point where the data is stored, analyzed and displayed. This helps with identification of potential bottlenecks and in preventing traffic overload. The system includes functions for:

- Gathering traffic data from switching units.
- Threshold checking and generation of overflow exceptions.
- Logging and routing of network exceptions.
- Visualization of traffic flow and network exceptions.
- Displaying various user-configurable views of the whole network.
- Statistical evaluations of traffic data.
- Access to historic traffic data.
- System administration and configuration.

The design and implementation of the system follows the Presentation-Abstraction-Control pattern. Every function of the system is represented by its own bottom-level PAC agent. There are dedicated agents for each view of the network, for the jobs the system can perform, and for the additional services the system offers, such as mail or help. Three intermediate-level PAC agents coordinate these bottom-level PAC agents, one for each of the three categories of application functionality: view, jobs, and additional services. In the diagram below, they are denoted by the agents NetEnv, JobEnv, and RegieEnv. An additional intermediate PAC agent organizes user sessions. The top-level PAC agent coordinates individual user sessions, and communicates with the functional core of the system. The core is implemented separately from the PAC hierarchy, probably because it incorporates legacy software. The PAC agent hierarchy of the system is dynamic. If, for example, a user starts a new session, a corresponding UISession agent is created and registered with the top-level agent. At the end of the session this agent is deleted.

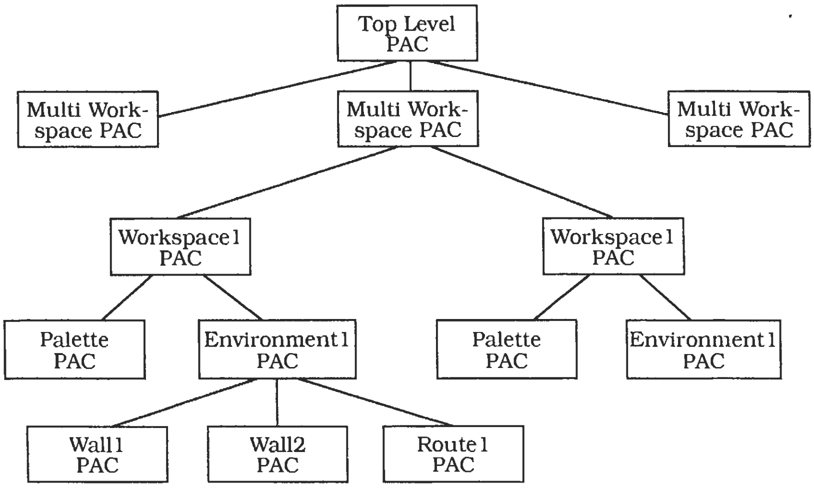


Mobile Robot. This system [Cro85] allows an operator to interact with a mobile robot that navigates within a closed and hazardous environment consisting of walls, equipment and people, either intruders or accident victims. The robot navigates using its own sensors and information from the system operator. The software allows the operator to:

- Provide the robot with a description of the environment it will work in, places in this environment, and routes between places.
- Subsequently modify the environment.
- Specify missions for the robot.
- Control the execution of missions.
- Observe the progress of missions.

Each wall, route and place within an environment is represented by its own bottom-level PAC agent. These agents together visualize the environment. Environments are represented by intermediate-level

PAC agents. They control the constituent wall, route and place PAC agents. The control users can exert on an environment is implemented in a 'palette' PAC agent, which is also at the bottom level of the hierarchy. The environment PAC agent and the palette PAC agent form a workspace for the robot. This workspace is represented by its own intermediate-level PAC agent. To support multiple views of the same environment, a multi-workspace PAC agent coordinates the different views of the same workspace. The PAC agent at the top level of the hierarchy encapsulates the functional core of the application, which is a rule-based intelligent supervisor for navigating and controlling the robot.



Consequences The Presentation-Abstraction-Control architectural pattern has several **benefits**:

Separation of concerns. Different semantic concepts in the application domain are represented by separate agents. Each agent maintains its own state and data, coordinated with, but independent of other PAC agents. Individual PAC agents also provide their own human-computer interaction. This allows the development of a dedicated data model and user interface for each semantic concept or task within the application, independently of other semantic concepts or tasks.

Support for change and extension. Changes within the presentation or abstraction components of a PAC agent do not affect other agents in the system. This allows you to individually modify or tune the data model underlying a PAC agent, or to change its user interface, for example from command shells to menus and dialogs.

New agents are easily integrated into an existing PAC architecture without major changes to existing PAC agents. All PAC agents communicate with each other through a pre-defined interface. In addition, existing agents can dynamically register new PAC agents to ensure communication and cooperation. To add, for example, a new view PAC agent to our information system for political elections, we only need to extend the presentation of the view coordinator PAC agent with an appropriate palette field that allows users to create this new view. The functionality for handling this new PAC agent, for registering it with the view coordinator PAC agent, and for propagating changes and events to it is already available.

Support for multi-tasking. PAC agents can be distributed easily to different threads, processes, or machines. Extending a PAC agent with appropriate IPC functionality only affects its control component.

Multi-tasking also facilitates multi-user applications. For example, in our information system a newscaster can present the latest projection while data entry personnel update the data base with new election data. All that is necessary is for the shared data repository, or its control component, to take care of serialization or synchronization.

The **liabilities** of this pattern are as follows:

Increased system complexity. The implementation of every semantic concept within an application as its own PAC agent may result in a complex system structure. For example, if every graphical object such as a circle or square within a graphics editor is implemented as its own PAC agent, the system would drown in a sea of agents. Agents must also be coordinated and controlled, which requires additional coordination agents. Think carefully about the level of granularity of your design, and where to stop refining agents into more and more bottom-level agents.

Complex control component. In a PAC system, the control components are the communication mediators between the abstraction and presentation parts of an agent, and between different PAC agents. The quality of the control component implementations is therefore crucial to an effective collaboration between agents, and therefore for the overall quality of the system architecture. The individual roles of control components should be strongly separated from each other. The implementation of these roles should not depend on specific details of other agents, such as their concrete names or physical locations in a distributed system. The interface of the control components should be independent of internal details, to ensure that an agent's collaborators do not depend on the specific interface of its presentation or abstraction components. It is the responsibility of the control component to perform any necessary interface and data adaptation.

Efficiency. The overhead in the communication between PAC agents may impact system efficiency. For example, if a bottom-level agent retrieves data from the top-level agent, all intermediate-level agents along the path from the bottom to the top of the PAC hierarchy are involved in this data exchange. If agents are distributed, data transfer also requires IPC, together with marshaling, unmarshaling, fragmentation and re-assembling of data.

These are serious potential pitfalls. We take them into account in the following discussion about when to use, and when not to use, the Presentation-Abstraction-Control pattern.

Applicability. The smaller the atomic semantic concepts of an application are, and the greater the similarity of their user interfaces, the less applicable this pattern is. For example, a graphical editor in which every individual object in a document is represented by its own PAC agent will probably result in a complex fine-grain structure which is hard to maintain. On the other hand, if the atomic semantic concepts are substantially larger, and require their own human-computer interaction, PAC provides a maintainable and extensible structure with clear separation of concerns between different system tasks.

See also The *Model-View-Controller* pattern (125) also separates the functional core of a software system from information display and user input handling. MVC, however, defines its controller as the entity responsible for accepting user input and translating it into internal semantics. This means that MVC effectively divides the user-accessible part—the presentation in PAC—into view and control. It lacks mediating control components. Furthermore, MVC does not separate self-reliant subtasks of a system into cooperating but loosely- coupled agents.

Credits PAC was originally described in [Cou87] by Joelle Coutaz. One of the first systems to be implemented based on PAC was the mobile robot application [Cro85]. Further valuable guidelines for implementing PAC can be found in [BaCo91] and [CNS95].

We thank Joelle Coutaz and Laurence Nigay for fruitful discussions and valuable input that helped us to shape the description of this pattern. Steve Berczuk, Brian Foote, Ralph Johnson, Tim Ottinger, David E. DeLano and Linda Rising carefully reviewed an earlier version of PAC and provided us with detailed feedback for improvement.