Acquisition-Computing-Execution-Expression (ACEE)

A Software Architecture Pattern for Computer-supported Automation and Control Systems

Yongmei Wu

Darmstadt University of Technology
Department of Computer Science
Programming Languages and Compiler
wu@pu.informatik.tu-darmstadt.de

Context

In the domain of Computer-supported Automation and Control Systems (CsACS), an application is used to help human to finish some control tasks. Peripheral devices like sensors gather data from the controlled objects, this data is processed according to the specific control rules. The control decisions (output) have to be produced in form of control signals for the execution mechanisms of the controlled objects fulfilling the tasks. Generally, the control process is real time and automatic. In some situations letting the operators take part in the control process is still required.

To build an application in CsACS, the whole contents described above must be taken into account. We also need strategies for flexible support of development and maintenance of an application. The **ACEE** architecture pattern aims to aid in constructing the application to meet the requirements.

Example

Suppose we develop an application for controlling a robot. The robot has three motors for driving its arm and the attached gripper, as shown in figure 1. The robot is used to lift an object to a target position.

In order to control the robot fulfilling the task, the application should include the code for driving the motors to move the robot arm and to open and close the gripper. It should integrate the control rules for deciding where to and how the object will be moved. The control decisions should be regulated timely according to the position of the robot arm and the width of the robot gripper. To allow the operator to interact with the application to change the system dynamic behaviors (e.g., to change the target position when the application is running), the application should express the robot's current status (the arm position, the width of the gripper), the manipulation possibilities (e.g., the representation of the gripper) and the system control decisions in some expression devices (e.g., monitor with a graphical user interface). An input command from the operator is rather abstract and has to be translated into concrete actions (e.g. the command which is used to move the robot arm to a concrete position has to be translated to a sequence of switching motors on and off). But first the application also has to decide if the command is valid.

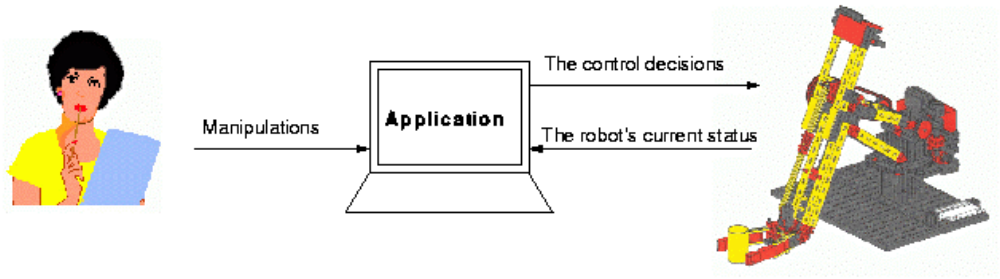

Figure 1. A Computer-supported Robot Control System

Problem

What should a CsACS application look like to be highly maintainable, reusable and efficient to develop?

Forces

The development of an application for CsACS crosscuts several technologies. It does not only deal with computer science but also other engineering techniques. It requires that the application designers are familiar with software development and the necessary hardware technology such as knowing the characteristics of peripheral devices. Even more, in order to develop a user-friendly software, they should also have knowledge about psychology and ergonomics. This means that building an application for CsACS requires several application designers with different skills to work together. For instance, in order to acquire the signals from the robot and to drive the robot fulfilling the control tasks, knowledge about robot technology is required, which is generally mastered better by electronic engineers. For arranging the low level interactions, e.g., the interactions between the robot and the computer, software engineers need to be proficient in system programming. Support for user-friendly Human Computer Interactions requires specific user interface designers who master psychology and ergonomics in the field of CsACS. For making the control decisions, the domain specific control algorithms are required to be summarized and integrated into the application, which is the special skill of mathematicians, and so on. If the software architecture is not well constructed, it is hard to assign the development tasks to the application designers adequately. It could require that the co-operating application designers must know the code details written by the others during the development. It obviously adds to their burdens and could result in low work efficiency.

In addition, the peripheral devices are manifold. Moreover, the development of hardware is so fast that newer and more powerful hardware is continuously appearing in the market which results in a strong demand for the system to flexibly support new devices. On the other hand different operators may need different ways for presenting the system's status in order to interact with the system (sometimes text expression is more sufficient, some users only can deal with graphical user interface, and sometimes voice information is required). E.g., a small monitor in a factory hall may only offer text output or there may be some standard visualization tool that has to be connected to the system. This means that the parts deal with peripheral devices and human interaction in the application are prone to change, while the control rules of a specific application domain are relatively stable. If the software architecture is monolithic, it is hard to change parts of the system. It might even be necessary to rewrite the whole application.

Solution

According to the functions needed by CsACS, **ACEE** divides the application into four components:

> **A**cquisition, acquires the signals from the external objects, i.e., the status of the controlled objects (e.g., robot arm position) and the control events from manipulation devices (e.g., mouse movements which are used to initiate the control decisions for driving the robot arm or a hand held teachIn device).
>
> **C**omputing, makes the control decisions.
>
> **E**xecution, drives the execution mechanism to fulfil the control decisions.
>
> **E**xpression, is the interface to the operator, whatever this could be. By use of visualization, audio and other possible technology, it expresses the status of the external objects, the control decisions and provides the manipulation possibilities to the operator as required.

The principle of **ACEE** is: once Acquisition obtains new data, Computing will make the control decisions and Expression will express the data to the operator as required. After the control decisions are made, Execution will drive the execution mechanism to fulfil the control decisions and Expression will express the control decisions to the operator in the demanded ways.

Additionally, we can see that the communication between the four components is unidirectional. That is, it is necessary for Computing and Execution to be informed about the change (acquired data) in Acquisition, and for Execution and Expression to know the change (control decisions) in Computing, but not vice versa. **ACEE** uses the Broadcaster/Listeners mechanism (also known as Observer design pattern [2]) to realize the change notifications.

For one Acquisition component, application designers can build several Computing and Expression components for attaching to it. And several Execution and Expression components can also be built for a Computing component. In this respect **ACEE** is similar to the MVC pattern [1], a well known Architecture Pattern for User Interface Design. MVC can support flexibly adding or changing the View and Controller components to a Model. With **ACEE**, application designers can also flexibly integrate or change different acquisition interfaces, control models, expression forms and execution mechanisms in one application.

Structure

The structure of **ACEE** is shown in figure 2.

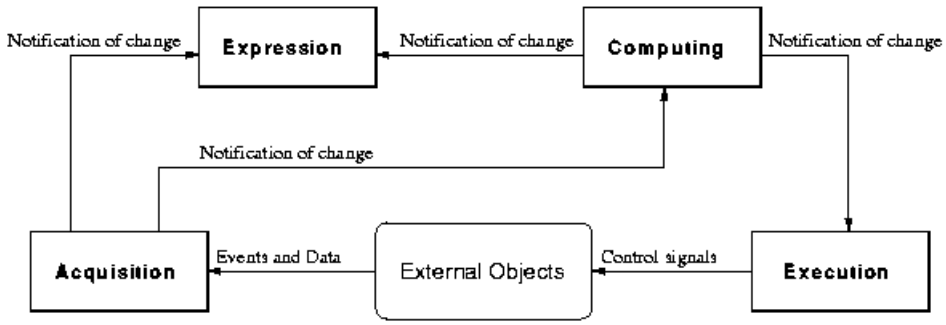

Figure 2. The Structure of ACEE

Participants:

Acquisition component

- Monitors the interface connected to the external objects for occurring events.
- Acquires data from the external objects.
- Translates the acquired signals to a computer processable format if necessary.
- Informs all registered Computing and Expression components about its change after new signals are acquired and translated.

Computing component

- Keeps its state consistent with that of its Acquisition component.
- Makes the control decisions after a change in its Acquisition component.
- Informs its Expression and Execution components about the control decisions.

Execution component

- Keeps its state consistent with that of its Computing component.
- Translates the control decisions from its Computing component to the data format recognizable by the execution mechanism.
- Drives the execution mechanism to carry out the control decisions.

Expression component

- Keeps its state consistent with that of its Acquisition or its Computing component.
- Translates the expression contents to the data format needed by the expression devices (e.g., large display, voice cards, etc.) if necessary.
- Expresses the status of external objects, the control decisions from its Computing and provides the manipulation possibilities to the operator in the demanded ways.
- Drives the expression devices finishing the expressions.

Dynamic

Let's explore the dynamic behaviors of **ACEE**.

Acquisition acquires the signals form the external objects.

Acquisition includes the mechanism for monitoring the events that occur in the external objects. When an event is captured, Acquisition processes it and acquires data form the relevant object if applicable. Acquisition then transforms the acquired data to the required format if it is not processable by the computer. After interpreting, Acquisition notifies its Computings and Expressions about its change.

Computing makes the control decisions.

When a Computing is notified that new data has been acquired by its Acquisition, it will make the control decisions based on the integrated control rules and the new acquired data. After Computing has made the control decisions, it will notify its Expression and Execution components.

Expression expresses the information to the operator.

When an Expression is informed about new data from its Acquisition or new control decisions in its Computing, it will translate the data to the format required by its expression devices if needed. Expression then drives its expression devices, if those are not the standard devices supported by the computer system (e.g., large display, voice cards, etc.), refreshing the expression contents.

Execution drives the execution mechanisms to fulfil the control decisions.

When an Execution is notified of new control decisions from its Computing, it will transform the control decisions to the signals recognizable by its execution mechanism (e.g., motor, switches, actuator, etc.). It then drives the execution mechanism fulfilling the control decisions.

Scenario: In our example, the operator uses the mouse to manipulate the representation of the gripper on a screen to move the robot arm.

The location of the robot gripper is acquired by Acquisition. Acquisition translates it to the format processable by the system and then informs its Computing and Expression. In Computing, the data is processed for future decision. In Expression, the representation of the current location of the gripper is refreshed on the monitor.

The operator uses the mouse to manipulate the representation of the gripper on the monitor.

After the mouse driver acquires the data from the mouse, Acquisition translates the data to a position recognizable by the system. Acquisition then informs its Computing and Expression.

Computing makes the control decisions and its Expression and Execution react to them afterwards.

According to the integrated control rules, the current gripper location and the data that comes from the mouse, Computing decides if the manipulation by the operator is valid. If it is confirmed by Computing, the representation of the target position of the gripper on the monitor is refreshed and Execution will drive the motors to perform the control decisions. Otherwise, only a textual warning message will be displayed.

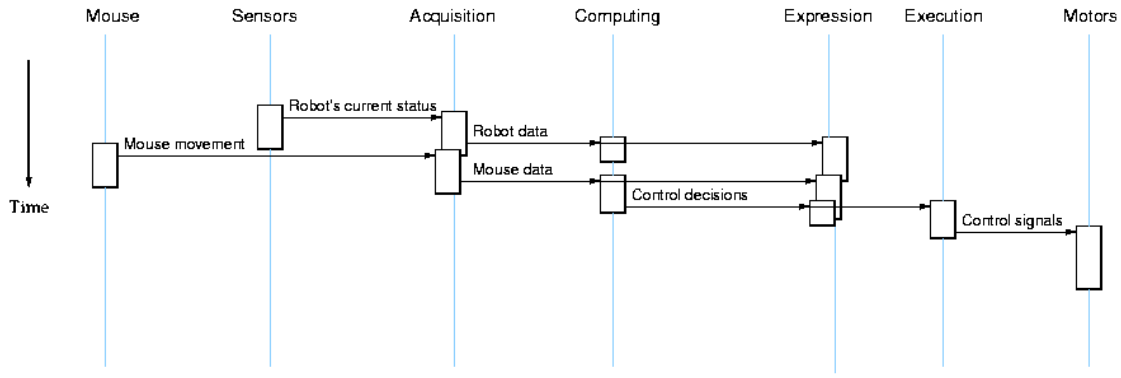

Figure 3. The Interaction Diagram of the Scenario

The interaction diagram is shown in figure 3.

Implementation

The implementation of a CsACS application consists of six steps. Steps 2~5 maybe repeated several times to define several Acquisition, Computing, Execution, Expression components in order to meet the demands of an application.

1. Construct the Broadcaster/Listeners mechanisms.

   The communications in **ACEE** depend on two Broadcaster/Listeners mechanisms. One is between Acquisition, Computing and Expression components, where Acquisition is the Broadcaster and Computing and Expression are the Listeners. The other is between Computing, Execution and Expression components, where Computing is the Broadcaster and Execution and Expression are the Listeners. According to the work principles of the Broadcaster/Listeners mechanism, if there is any change in the Broadcaster, by sending a **`changed`** message to itself, the Broadcaster will trigger the Broadcaster/Listeners mechanism to activate **`update`** messages in the corresponding Listeners automatically.

   For the sake of meeting the demands of CsACS, the Broadcaster/Listener mechanism in **ACEE** should be able to allow the Broadcaster to notify its changes to its Listeners, triggering their updating. It should supply the interface for flexible attachment and detachment of the Listeners to the Broadcaster. It should also be independent of the application domain. Therefore, **`changed`**, **`update`**, **`attach`** and **`detach`** messages should be implemented. The **`changed`** message is used in the Broadcaster for informing its change, while the **`update`** message is used in the Listener for updating itself. The **`attach`** and **`detach`** messages are used in the Broadcaster for attaching and detaching the Listeners. For the implementation details, please refer to Observer pattern in [2].

2. Implement the Acquisition component.

   - Implement Acquisition as the Broadcaster component in the Broadcaster/Listeners mechanism.
   - Implement **`eventDispatch`**, **`eventHandler`**s and necessary acquisition interfaces. In CsACS the external objects are not only the traditional interaction devices like keyboard and mouse but also other facilities like sensors and wireless signals. In order to save the system consumption, Acquisition gathers data from the external objects only when an event happens. To deal with specific event, Acquisition must use specific scheme. If all the schemes are heaped up in one message, say **`eventHandler`**, it will be difficult to implement and maintain, because the interwoven code brings lots of pitfalls. E.g., if there is any execution error in one scheme, the message cannot work even through the other schemes are correct. And if we need to change or add any external object, the whole message may be needed to be rewritten. To reduce the possibility of error and improve the reusability and maintainability, diverse strategies for processing diverse events should be encapsulated in different **`eventHandler`**s. For recognizing an event and assigning a concrete **`eventHandler`** to it, we need also a mechanism, **`eventDispatch`**. And if there is any external object which is not supported by the system low level platform (e.g., operating system and system I/O), an acquisition interface must be built for it.

     Implement the acquisition interfaces if necessary. The acquisition interfaces are responsible for monitoring the events occurring in the external objects. Usually an operating system provides the mechanisms to support standard interactive devices, e.g., mouse driver, which are transparent to the application designers. Interfaces for the external objects which are not supported by the low level platform should be built in Acquisition. An acquisition interface depends on the characteristics of the external object and the low level platform. In general, Operating Systems like Windows NT provide corresponding functions such as reading and writing files and sockets for communication [7]. For the external object which can not produce events autonomously, a polling mechanism must be built in the interface in order to acquire data correspondingly.

     > If we use Windows NT as the operating system in our example and because the interface of the robot does not belong to standard devices supported by Windows NT, we can use a parallel port to connect the interface of the robot to a PC. All the data read from or written to the robot must go through the parallel port. Standard functions of Windows NT like File I/O cannot be used to achieve reading from and sending data to the parallel port. Thus the interface for

reading and writing the data from the parallel port must be wrapped in Acquisition. Besides, the robot interface does not have the ability to produce events, the acquisition interface must include polling mechanism, too.

Implement an **`eventDispatch`** message. The **`eventDispatch`** message deals with how to dispatch **`eventHandler`**s for processing the events. Reactor pattern [8] and Proactor pattern [6] describe two corresponding solutions and detailed implementation steps for building **`eventDispatch`**.

Implement the **`eventHandler`** messages. According to the characteristics of the external object, specific schemes for processing events should be encapsulated in a specific **`eventHandler`**.

In our example, data comes from the robot interface, keyboard and mouse. According to the captured event, `eventDispatch` assigns the specific `eventHandler` to process it. For instance, if an event, which is used by the polling mechanism of the robot acquisition interface, is triggered by the system clock, `eventDispatch` will assign the robot `eventHandler` to it. The robot `eventHandler` message is used for checking if the data read from the robot interface is complete, for example.

- Implement the **`translate`** messages if required. When the acquired signals are not processable by the computer, **`translate`** messages are needed. A **`translate`** message includes the code for translating one kind of the acquired signals to the data processable by the system.

  In our example, the robot arm's horizontal movement is driven by a motor. A pulse switch is used to monitor the motor's movements. In the application, the robot arm's horizontal position acquired by its `eventHandler` is a number recording the times that the pulse switch has been switched on and off. The `translate` message includes the algorithm to interpret the pulse count to a value, telling how many degrees the arm was moved.

- Implement the necessary messages for other components, i.e. Computing and Expression, to access its core data.

3. Implement the Computing component.
   - Implement Computing as the Listener of an Acquisition and the Broadcaster of Expressions and Executions according to the Broadcaster/Listeners mechanism.
   - Implement a **`compute`** message. The message **`compute`** should integrate the control rules related to the domain specific application for making the control decisions. The control decisions are determined by the integrated control rules, the current status of the controlled objects, and the control events from the manipulation devices (e.g., mouse movements activated by the operator) if they exist.
   - Implement the necessary messages for other components, i.e., the Execution and Expression components, to access its core data.
4. Implement the Execution component.
   - Implement Execution as the Listener of a Computing according to the Broadcaster/Listeners mechanism.
   - Implement a **`translate`** message. The Execution component is responsible for driving the execution mechanism to fulfil the control decisions. The **`translate`** message includes the strategies for transforming the control decisions to the data format recognizable by the execution mechanism.

     In our example, the `translate` message in Execution transforms the target horizontal position of the robot arm calculated by Computing, which is a number denoting how many degrees the arm has to be moved, to a number that tells how many times the pulse switch has to be switched on and off.

   - Implement a **`drive`** message. The **`drive`** message must include the code for driving the execution mechanism to carry out the control decisions.

     In our example, the `drive` message encapsulates the code for monitoring the pulse switch count.

5. Implement the Expression component.

   - Implement Expression as the Listener of an Acquisition or a Computing according to the Broadcaster/Listeners mechanism.
   - Implement a **`translate`** message. Expression is responsible for expressing the content, which involves the status of the external objects, the manipulation possibilities and the control decisions in a graphical user interface or other multimedia ways. Depending on the requirements, the **`translate`** message integrates the code for interpreting the contents to the interface of the expression device.

     In our example, the gripper width in the Computing is a number while its representation to the operator is a circle. `translate` thus transforms the number to a representation of a circle.
   - Implement a message **`express`**. The **`express`** message integrates the code for expressing the contents in expression devices. For the standard expression devices like the computer screen, the low level system platform provides standard functions which are transparent to the application designer. But for other specific expression devices such as some arbitrary audio devices, **`express`** must include the code to drive them.

     In our example, `express` includes the code for displaying the control decisions on a large screen.

   According to the demands and characteristics of the expression devices, by implementing the **`translate`** and **`express`** messages, the expression component makes it easy to plug other new expression devices or expression forms to the system without influencing other components.

6. Initialize the communication relationships between Acquisition, Computing, Execution and Expression.

   When the implementations of Acquisitions, Computings, Executions and Expressions are finished, implement their **`changed`** and **`update`** messages accordingly to establish the adequate communication relationships within the system.

By use of the variant of OMT notation [2], the class diagram of **ACEE** can be depicted as in figure 4.

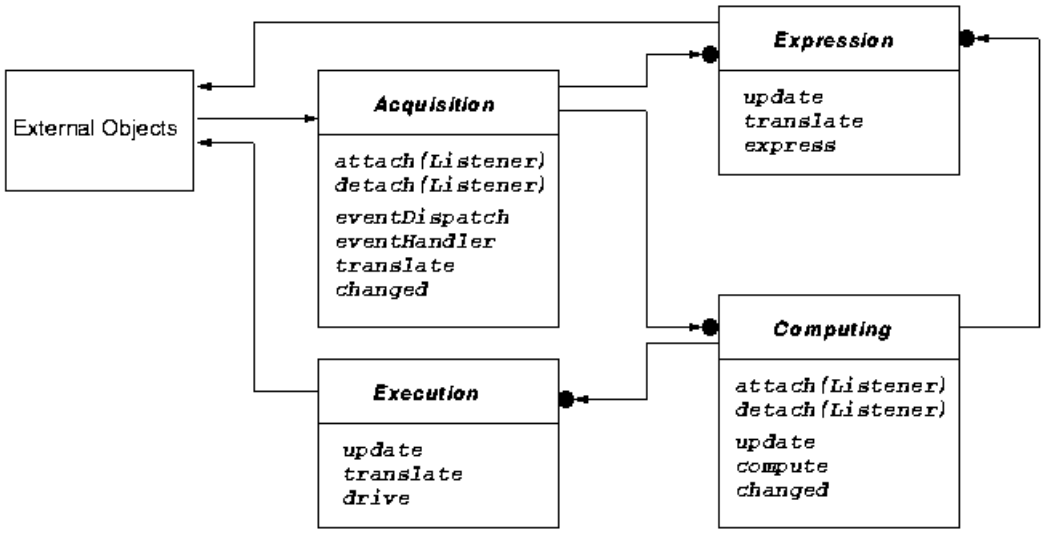

Figure 4. The Class Diagram of ACEE

Variant

The MVC Architecture Pattern [1] is a paradigm for interactive systems design. It divides an interactive software into three parts: Model, View and Controller. Model is the core structure and data for describing the domain specific application. View deals with displaying application states to the user. Controller is used to handle user input. The Broadcaster/Listeners mechanism is also used to accommodate the communications between them. In

MVC, for one Model there can be several View and Controller pairs. Once the Model has changed (often triggered by the Controller monitoring the user action, e.g., mouse movement), it will inform its Listeners in Views. The relevant Views will refresh the displays.

In application domains where no Execution is needed, Expression is limited to the graphical user interface, and there is only one Broadcaster/Listeners mechanism in Acquisition, Computing and Expression, **ACEE** can be simplified to MVC, where Computing, Expression, Acquisition correspond to Model, View, Controller respectively.

Known Uses

An application for process automation in Hot Rolling Mills developed by Siemens AG covers the whole range between data acquisition via sensors, computing data in complicated mathematical models, visualizing this data and controlling the mill with actuators. For data acquisition (Acquisition), visualization (Expression) and low level control tasks (Execution) standard products are used that have to communicate with a complex computing component (Computing).

SCUT Voice [10], a public telephone voice service system, is built according to the principle of **ACEE**. It allows the users to leave, inquire and delete voice messages through the public telephone. It provides one "Voice Mailbox" for one user and each "Voice Mailbox" can hold limited messages. In SCUT Voice, the software is divided into four parts: Acquisition, Computing, Execution and Expression. Acquisition is responsible for monitoring and acquiring the data from the telephone interface. Computing is used to check if the user's operations are illegible and to fulfil the user's requirements. Execution is driving the telephone interface for transferring the results, e.g., the messages, to the user. Expression expresses the system's working status, such as which interface channel is occupied, to the system administrator.

LLDemo [9] applies the principle of **ACEE** to construct the software architecture. This results in the ability to flexibly change any component of the software without influencing the others. Different components can be developed by different designers concurrently.

Resulting Context

The advantages of the **ACEE** Architecture Pattern are:

- Clear separation of an application of CsACS into four components: Acquisition, Computing, Execution and Expression.

  The **ACEE** Architecture Pattern clearly divides an application of CsACS into four components according to the functionality. It makes the four components relatively independent of each other. This implies that each component can be developed individually without being much interwoven with others. Consequently, it enables an application designer to concentrate only on the component at hand.

- Easily attach/detach the Listeners to Acquisition or Computing.

  Because the Broadcaster/Listeners mechanism is used to accommodate the communications, the relationships between Acquisition, Computing, Execution and Expression are easily accomplished. To attach/detach Computings and Expressions to an Acquisition or to attach/detach Executions and Expressions to a Computing is very simple.

- Improve software reusability and maintainability.

  Since the four components are relatively isolated from each other with respect to functionality, they are easier to understand and maintain. The easily attached/detached Listeners improve the capability of reusing them.

- It can also support the design of those interactive systems that already go beyond the WIMP (Windows, Icons, Menus, Pointing devices) metaphor [3][5].

The drawback of the **ACEE** architecture is:

- Potentially unnecessary updates in the Listeners.

The Broadcaster/Listener mechanism implies a certain communications overhead which, if not properly taken care of, may slow the system down to an intolerable state. E.g. if a Broadcaster has too many Listeners or issues **`changed`** messages too often the consequent updates may take a while to compute. Therefore strategies, such as permitting a Listener to update itself only when any interesting change happens in its Broadcaster, should be applied to limit the potentially unprofitable updates.

Related Patterns

The Broadcaster/Listeners mechanism is used to construct the communication backbone of **ACEE**. For detailed information on it, please refer to the description of Observer pattern in [2].

Strategy design pattern [2] can be used to implement the **`update`** messages in Computing, Execution and Expression for flexible assignment of concrete schemes related to the application domains.

Proactor design pattern [6] and Reactor design pattern [8] can be selected to build **`eventDispatch`** of the Acquisition component in **ACEE**.

Acknowledgements

Many thanks to Christa Schwanninger who was the shepherd of this pattern and gave me lots of concrete advice for improvement. I would like to thank my supervisor Prof. Dr. Hans-Jürgen Hoffmann for the research proposal [4] which resulted in discovering the **ACEE** Architecture Pattern. Thanks to Jan Weerts, Patrick Closhen, Elke Siemon, Daniela Handl and Martin Friedmann who gave me some interesting suggestions when I documented this pattern.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: "Pattern-Oriented Software Architecture: A System Of Patterns", John Wiley & Sons, 1997.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[3] M. Green, R Jacob: "Software Architecture and Metaphors for Non-WIMP User Interfaces", SIGGRAPH'90 Workshop Report.

[4] H.-J. Hoffmann: "Research proposal: Design models and object-oriented framework for users interfaces in computer-supported automation and control technology", Darmstadt University of Technology, 1998.

[5] J. Nielsen: "Non-Command User Interfaces", Communications of ACM, April 1993, Vol.36., No.4.

[6] I. Pyaral, T. Harrison, D. Schmidt: "Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events", The 4th Annual Pattern Languages of Programming Conference, Allerton Park, Illinonis, September 2-5, 1997.

[7] R. Rajagopal, S. P. Monica: "Windows NT 4 Advanced Programming", Osborne/Mc Graw-Hill, 1998.

[8] D. Schmidt: "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", Pattern Languages of Program Design, Addison-Wesley Publishing Company, 1995.

[9] Y. Wu: "LLDemo -- A direct-manipulation user interface for the robot control", Technical Report, Darmstadt University of Technology, April 1999.

[10] Y. Wu, Y. Ni: "The Software Implementation Technique of Telephone Voice Mailbox", the Journal of South China University of Technology, Vol.25, No. 4, April 1997.