A MULTI-LEVEL DESIGN PATTERN FOR EMBEDDED SOFTWARE*

Ricardo J. Machado and João M. Fernandes
Dept. Sistemas de Informação & Dept. Informática, Universidade do Minho, Portugal

Abstract: It is a common practice amongst programmers to construct parts of software programs by imitating parts of programs constructed by more experienced professionals. This "learn by example" approach can be applied at the design level by using patterns as sets of rules and recommendations to solve well-defined tasks within the development of computer-based systems. This paper describes the multi-level ICIS pattern, to be used at various design levels of industrial control-based information systems, where embedded devices are networked to interact with the industrial processes and equipment. The proposed pattern is described using several UML diagrams.

1. INTRODUCTION

The research work of Edsger Dijkstra has demonstrated that it is advantageous to "waste time to think" in the organization, structure and internal partition of a system, instead of going directly to the implementation just after the requirements' modelling [1]. This methodological position, nowadays perfectly accepted by any professional system designer, has originated, at that time, several research lines that culminated in the emergence of a new sub-discipline called *systems architecture* [2], or *software architecture* in the particular situation of software intensive systems [3].

Architectural design involves the manipulation of general abstract models that can be applied to distinct systems, as long as these systems share a set of

* Research funded by FCT and FEDER under project *METHODES* (POSI/37334/CHS/2001).

common requirements. These general abstract models of systems' organization are called *design patterns* [4].

It is a common practice amongst programmers to construct parts of software programs by imitating parts of programs constructed by more experienced professionals. This current practice, at the implementation level, demands the search for a pattern within a third-party software code and the adaptation to the specific problem at hand. This "learn by example" approach can be applied at the design level by using patterns as sets of rules and recommendations to solve well-defined tasks within the development of computer-based systems [5].

2. DESIGN PATTERNS

One of the traditional problems of design patterns is the inexistence of a standard notation for its description, which allows different interpretations for each existing pattern [7]. A pattern can be characterized by using: (1) a pictorial diagram to describe the general context of the pattern; (2) a class/object diagram in a well-known notation (UML, for instance), to model the static relations amongst the pattern entities; (3) a sequence diagram to model the dynamic relations within the pattern; (4) any other semantic diagram, as long as the syntax is well-defined, to characterize a particular view of the pattern. Recently, Fontoura [8] has proposed the UML-F profile to describe framework architectures and to support framework modeling and annotation by using UML-compliant extensions.

Nowadays, design patterns have reached such a very mature state that they are organized in a catalogue fashion just like the old databooks of digital integrated circuits [9-13]. There are also some pattern catalogues for the analysis phase [14].

The idea of documenting the best practices in software development as patterns for building embedded and real-time systems is a recent research topic. The first important work on this topic was the "Recursive Control" pattern for real-time control systems [15]. Another major landmark is the collection of patterns proposed by Douglass to design object-oriented real-time systems [5]. Other work in defining patterns for embedded and real-time systems were also proposed in the last years [16-22].

3. THE MULTI-LEVEL ICIS PATTERN

A new design pattern, named *multi-level ICIS*, was defined as a result of the development of several industrial information systems [23, 28]. Within

these developments, embedded systems, web-services, and control applications had to work together to accomplish the easy interconnection between the lower (0, 1 and 2) and the upper (3 and 4) CIM (computer integrated manufacturing) levels [24]. These ICIS (industrial control-based information system) solutions are complementary, within the industrial organizations, to the well-known management information systems (MISs) [25]. Industrial information systems (IIS), which result from the integration of a MIS with an ICIS, are the answer to accomplish the definition of an applicational platform, based on ERP (enterprise resource planning) approaches, in order to integrate and unify the management and control of all organizational information.

The proposed pattern, based on the MVC pattern [6], was defined to support both the levels 2 and 3 of co-design [26]: (1) at level 2, the embedded software engineer must decide which functionalities will run directly on the processor and which ones will be synthesized for the reconfigurable devices; (2) at level 3, the information systems engineer integrates the previously designed components with the existing MIS. With this pattern we avoid the designer to follow a strict class-driven approach, where class diagrams are built before the object diagram [27]. The defined pattern provides a set of recommendations to support the architectural design of ICIS solutions.

Fig. 1 depicts a pictorial diagram of the multi-level ICIS pattern. This pattern is composed of four architectural blocks: (1) the *access interface* block, is responsible for the interface implementation with the MIS subsystem; (2) the *supervision interface* block is responsible for the interface implementation with the industrial processes and equipments (shop-floor); (3) the *operator interface* block is responsible for the interface implementation with the human operators that interact directly with the ICIS subsystem; (4) the *production, quality and management (pre-)processing* block is responsible for the data processing (stubbing and transformation) to support the interconnection of the three previous interfaces. Each architectural block can be implemented by adopting a reuse approach, based on the specialization or refinement of previously existing classes. This is the reason why, in fig. 1, there are one inheritance relation between each block and one class library. Additionally, each block can be developed within a CBD (component-based design) approach by using aggregation and composition of sub-objects as instances of specialized or refined classes from libraries.

To thoroughly understand the proposed pattern it is important to analyse the typical network topologies of final IIS solutions. In fig. 2, two distinct "zones" can be identified: (1) the first one corresponds to the CAN network supporting the ICIS implementation by using several embedded devices

(CANit, CANio, CAN-FPGA, CAN-RF and CAN-Server execute embedded software to support CIM level 2) and one (or more) PC-VAP (gateway executing LabVIEW software); (2) the second one corresponds to the Ethernet network supporting the MIS implementation by typical ERP and POS (plant operations system) software.

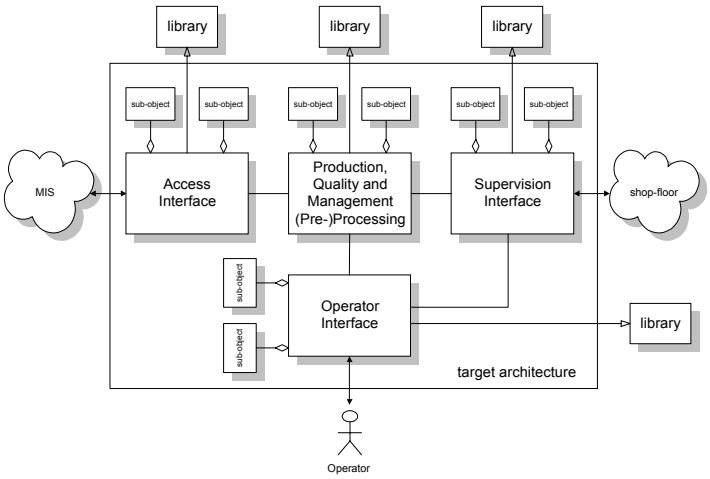

Figure 1. Pictorial diagram of the multi-level ICIS pattern.

UML deployment diagrams can be used to illustrate the three typical application scenarios for the multi-level ICIS pattern:

(1) *ICIS architecture*. The deployment diagram of the ICIS architecture is depicted in fig. 3, where the multi-level ICIS pattern is being used as follows: (i) the *PC-VAP* node supports the *access interface* component; (ii) the *CANio* node (embedded device topologically located near the industrial processes and equipments to acquire and send supervision information) supports the *supervision interface* component; (iii) the *CAN-RF* node (wireless embedded device used by human operators, along their walkthroughs the factory plant, to acquire and send supervision information) supports the *operator interface* component; (iv) the *CANit* node (embedded device that controls one or more groups of nodes, based on *CANio* and *CAN-RF* architectures, to coordinate the acquisition and sending of supervision information) supports the *(pre-)processing* component. Each *PC-VAP* node can concentrate, in a star topology, several sets of nodes based on *CANio* and *CAN-RF* architectures. Within this application context, the multi-level ICIS pattern is intended to clarify the kind of topology the information systems engineer should use to structure the existent computing nodes in a particular ICIS final solution.

(2) *Software architecture of the PC-VAP node*. The software architecture of the *PC-VAP* node is depicted in fig. 4. The *access interface* component of fig. 3 is being decomposed into a set of sub-components organized by the architecture defined by the multi-level ICIS pattern. The *access interface*, *supervision interface*, *operator interface* and *(pre-)processing* components appear again, but now within the *PC-VAP* software. For instance (and only just an example, since the graphical LabVIEW language has not been introduced here), it is possible to identify, within a portion of LabVIEW code depicted in fig. 5, the code blocks that correspond to those components.

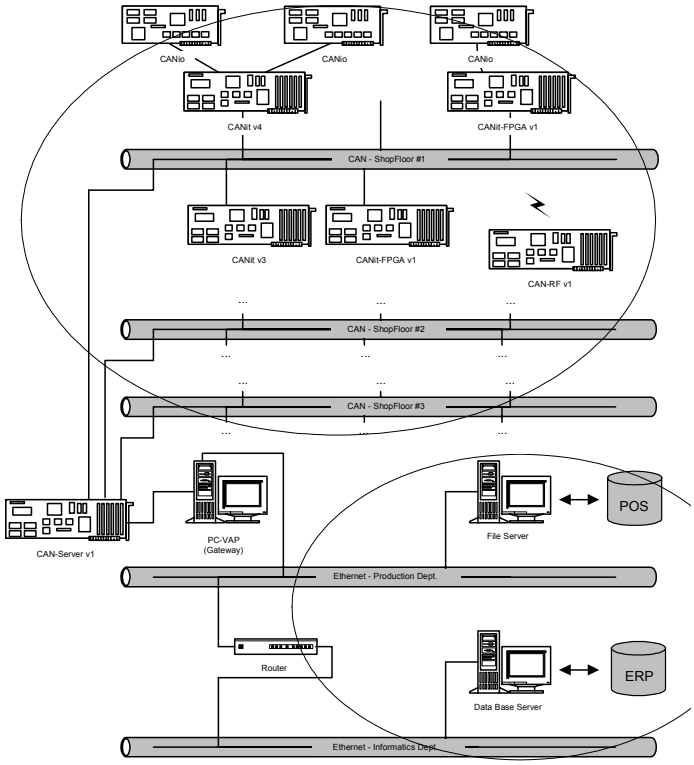

Figure 2. Network topology for typical IIS solutions.

(3) *Software architecture of the embedded components*. The *(pre-)processing* component of fig. 3 is decomposed into a set of sub-components organized by the architecture defined by the multi-level ICIS pattern. Again, *access interface*, *supervision interface*, *operator interface* and *(pre-)processing* components appear, but this time within the *CANit* embedded device, by using aggregates of objects in the Oblog language, just as referred in [29], following the methodology described in [30], with some well-known limitations [32].

A class diagram that describes, generically, the relations between the entities involved in the components suggested by the pattern is depicted in fig. 6. This diagram follows a class-driven approach (since it defines an instantiation template) and identifies six distinct classes: *Access*, *Supervision*, *Controller*, *Sub-Controller*, *DataRepository* and *Operator*.

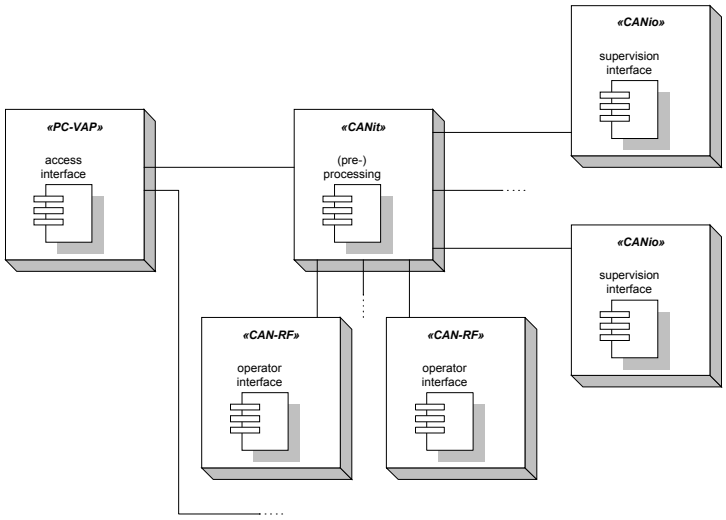

Figure 3. UML deployment diagram of the ICIS architecture.

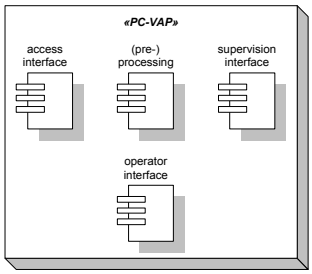

Figure 4. UML deployment diagram of the PC-VAP node.

Access, *Supervision* and *Operator* are *«interface»* classes (according to [32], an *«interface»* class models behaviour and information dependent on the system's interface) and allow *Controller* e *Sub-Controller* classes to be considerably independent from the particular mechanisms adopted to implement the relation with the outside world (the system's environment). Both the access (border between the embedded architecture and the upper MIS) and the supervision interfaces (border between the embedded

architecture and the industrial process and equipment) deal with external entities topologically located very far from the embedded device, thus adopting an asynchronous flow mechanism in relation with the embedded device's main thread responsible for the system's general control.

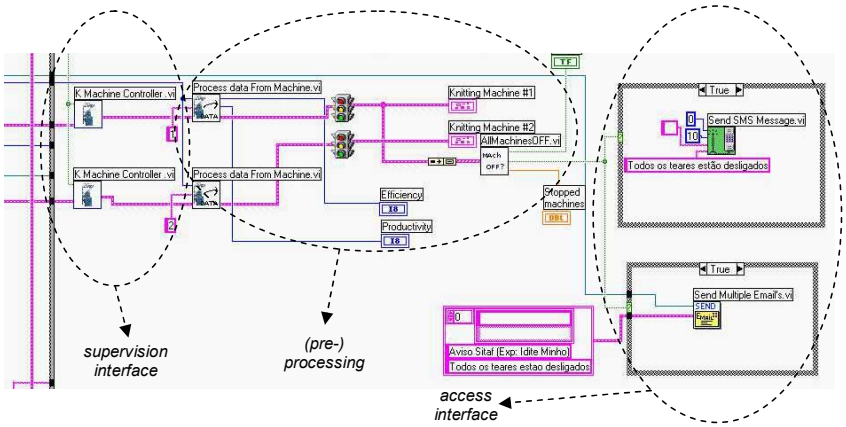

Figure 5. Multi-level ICIS pattern implemented in LabVIEW code.

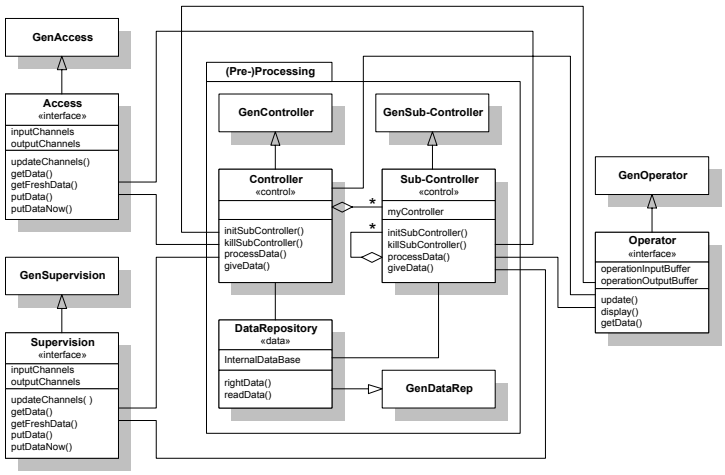

Figure 6. UML class diagram for the multi-level ICIS pattern.

The embedded device should have another tread that acknowledges the *access* and *supervision* objects (instances of *Access* and *Supervision* classes) about: (1) the arrival of information from the environment to be stored in the interface objects by using the *putData()* method; (2) the need to update all

the values stored in the interface objects by using the *updateChannels()* method. Each *access* and *supervision* objects must implement, as attributes, two distinct data structures, one to store the interface input values (*inputChannels*) and another to store the interface output values (*outputChannels*), before they are disseminated throughout the other system components. In the main thread, the methods *getData()* and *putData()* should be used to have access to the *inputChannels* and *outputChannels* data structures in an asynchronous way. To assure a synchronous execution of the reading and writing operations the methods *getFreshData()* and *putDataNow()* should be used. These methods are blocking within the main thread, since *access* and *supervision* objects force an effective hardware refreshment of the data structures.

Controller and *Sub-Controller* are *«control»* classes (according to [32], a *«control»* class models behaviour that can not be naturally associated to any other kind of object, i.e., *«interface» or «data»*) and allow the instantiation of aggregations of state-machines. These state-machine aggregations have access to a small data base, internal to the embedded device, to store temporarily information about the industrial processes and equipment. This set of objects constitutes the *(pre-)processing* architectural block to be executed within the main thread. Fig. 7 presents an UML sequence diagram for the multi-level ICIS pattern.

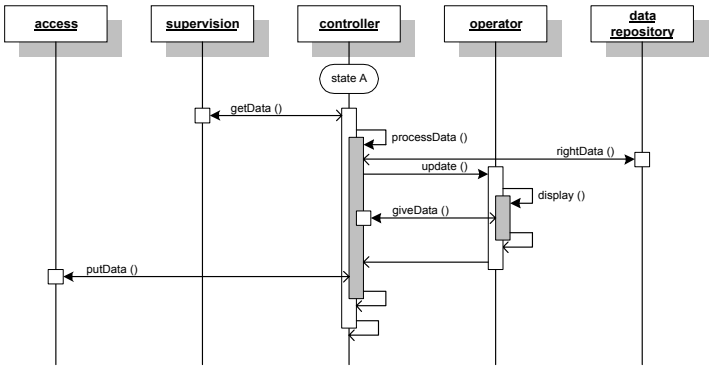

Figure 7. UML sequence diagram for the multi-level ICIS pattern.

4. CONCLUSIONS

The implementation of a pattern with the characteristics of the one presented here must be carefully thought for real-time applications, since it is very easy to adopt technological solutions that will introduce an enormous

temporal and space inefficiency. This additional implementation difficulty must be explicitly assumed as being the price to pay to benefit from the usage of patterns. The multi-level ICIS pattern should be faced as a semantic guideline for conducting the design of networked solutions for the supervision of industrial processes and equipment.

In practice, it is common to implement a simplified version of the architecture in which the three kinds of interfaces (*access*, *supervision* and *operator*) and the data repository are melted into a unique entity, named *interface controller*. This solution is more efficient than the one presented in fig. 7, since it is only necessary to maintain a unique set of data structures, which implies that the memory requirements can be significantly reduced and, consequently, the data processing operations can be speedup.

REFERENCES

1. E. Dijkstra, *The Structure of the 'T.H.E.' Multiprogramming System*, Communications of the ACM, vol. 18, no. 8, pp. 453-457, 1968.
2. E. Rechtin, M. Maier, *The Art of Systems Architecting*, Systems Engineering Series, CRC Press LLC, 1997.
3. P. C. Clements, *From Subroutines to Subsystems: Component-Based Software Development*, in A. W. Brown, *Component-Based Software Engineering*, Selected Papers from the Software Engineering Institute, IEEE CS Press, 1996.
4. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, ACM Press, 1995.
5. B. P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison-Wesley, 1999.
6. G. E. Krasner, S. T. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, vol. 1, no. 3, pp. 27-49, ACM Press, 1988.
7. A. Lauder, S. Kent, *Precise Visual Specification of Design Patterns*, 12th European Conference on Object-Oriented Programming, LNCS 1445, pp. 114-134, Springer Verlag, 1998.
8. M. Fontoura, W. Pree, B. Rumpe, *The UML Profile for Framework Architectures*, Addison-Wesley, 2001.
9. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
10. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
11. J. Coplien, D. Schmidt, *Pattern Languages of Program Design*, Addison-Wesley, 1995
12. J. Vlissides, J. Coplien, N. Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.
13. R. Martin, D. Riehle, F. Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

14. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
15. B. Selic, *An Architectural Pattern for Real-Time Control Software*, Third Annual Pattern Languages of Programming Conference, pp. 4-6, 1996.
16. M. Bottomley, *A Pattern Language for Simple Embedded Systems*, 6th Annual Pattern Languages of Programming Conference, pp. 15-18, 1999.
17. R. Mckegney, T. Shepard, *Design Patterns and Real-Time Object-Oriented Modeling*, Conference on Object-Oriented Programming, Systems, Languages, and Applications , pp. 55-56, 2000.
18. R. Mckegney, *Small Memory Software: Patterns for Systems with Limited Memory*, Addison-Wesley, 2000.
19. M. Pont, *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*, Addison-Wesley, 2001.
20. J. Zalewski, *Patterns Real-Time Software Design Patterns*, 9th Conf. on Real-Time Systems, 2002.
21. B. P. Douglas, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*, Addison-Wesley, 2002.
22. S. Sauer, G. Engels, *MVC-Based Modeling Support for Embedded Real-Time Systems: Position Statement*, Workshop on Object-Oriented Modeling of Embedded Realtime Systems, pp. 11-14, 1999.
23. J. M. Fernandes, R. J. Machado, H. D. Santos, *Modeling Industrial Embedded Systems with UML*, 8th Int. Workshop on Hardware/Software Codesign (CODES 2000), pp. 18-22, ACM Press, 2000.
24. P. G. Ranky, *Computer Networks for World Class CIM Systems*, CIMware Limited, 1990.
25. B. Scholz-Reiter, *CIM Interfaces: Concepts, Standards and Problems of Interfaces in Computer Integrated Manufacturing*, Chapman & Hall, 1992.
26. R. J. Machado, J. M. Fernandes, *Heterogeneous Information Systems Integration: Organizations and Methodologies,* Product Focused Software Process Improvement, pp. 629-643, M. Oivo e S. Komi-Sirviö (editors), Lecture Notes in Computer Science, LNCS vol. 2559, Springer-Verlag, 2002.
27. J. M. Fernandes, R. J. Machado, *From Use Cases to Objects: An Industrial Information Systems Case Study Analysis*, Object-Oriented Information Systems, pp. 319-328, Y. Wang, S. Patel e R. Johnston (editors), Springer-Verlag, 2001.
28. J. M. Fernandes, R. J. Machado, *System-Level Object-Orientation in the Specification and Validation of Embedded Systems*, 14th Symp. on Integrated Circuits and System Design (SBCCI'01), IEEE CS Press, 2001.
29. R. J. Machado, J. M. Fernandes, *A Petri Net Meta-Model to Develop Software Components for Embedded Systems*, 2nd Int. Conf. on Application of Concurrency to System Design (ACSD'01), pp. 113-22, IEEE CS Press, 2001.
30. R. J. Machado, J. M. Fernandes, H. D. Santos, *A Methodology for Complex Embedded Systems Design: Petri Nets within a UML Approach*, Architecture and Design of Distributed Embedded Systems, B. Kleinjohann (editor), chapter 1, pp. 1-10, Kluwer A.P., 2001.
31. J. M. Fernandes, R. J. Machado, *Can UML be a System-Level Language for Embedded Software?* Design and Analysis of Distributed Embedded Systems, B. Kleinjohann, K. Kim, L. Kleinjohann e A. Rettberg (editors), chapter. 1, pp. 1-10, Kluwer A.P.2002.
32. I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.