design patterns, real-time software
software architecture

Janusz ZALEWSKI

REAL-TIME SOFTWARE DESIGN PATTERNS

The objective of this paper is to provide a road map to understanding and use of real-time design patterns. To meet this objective, we provide a brief overview of the origins of design patterns, then discuss a wide variety of specific real-time design patterns, and finally provide our own classification of patterns, which is derived from a systematic approach to real-time system design and real-time software architectures.

1. INTRODUCTION

Real-time systems are computer systems with bounded response time. They concern applications in a variety of areas, from microwave ovens, washing machines and cash registers, through automotive electronics, traffic lights and railways, through systems as complicated as missile guidance, air traffic control and airline reservations. Their primary characteristic is timing dependence, which usually translates into a necessity of completing urgent jobs by certain, well defined deadlines. If missing a deadline is critical and cannot be tolerated, then a system is called hard real-time system, otherwise it is called soft real-time. Many hard real-time systems are also safety critical, which means that their failure, such as inability to deliver the results of computations on time, will cause severe material losses.

With the extensive use of digital computers for that kind of applications, software is a mandatory technology to provide the required functionality and plays an increasingly important role in system operation. As far as the level of complexity of software-intensive systems continuously increases, methods to design software-based controllers do not keep pace with this progress. Engineers still use traditional software design methodologies, which are based on structured design paradigms developed for real-time systems in the past decades. These methodologies do no longer meet the needs of designers of modern real-time systems, due to the limitations of their notations, inadequacy of design techniques, and lack of appropriate automatic design tools to facilitate the development process. This is due to the fact that most methodologies do not address design at large or design with added complexity factor, where organizational and architectural decisions are critical but difficult to formalize.

As indicated in [28], even for such a small system as the cruise control system for a road vehicle to automatically maintain constant speed, there are a number of problems with the typical solution using the traditional design methodology:

- little consideration is given to the system architecture to be employed, and the consequences this will have for the rest of the design
- the control algorithm is being presented largely without comment, and does not use any of the classical techniques, such as PD or PID
- no hint is given how the effectiveness of the chosen approach should be assessed.

When complexity increases, the support provided by conventional development methods for designing such systems is not enough to be effective. Engineers face design challenges that usually go well beyond the experience and capabilities of single humans. One way to alleviate this type of problems, facing engineers designing software-intensive real-time systems, is to apply design patterns within the existing methodologies [19].

A pattern is usually considered a model of software or, better, a template of software to assist in the software development process. As such, it may refer to both the software product itself and the software process. Its description can be rather informal, since there are no good or widely accepted models of software of general applicability. That way of approaching the construction process has been widely used in engineering disciplines dealing with material objects, for example in civil engineering to construct building, bridges, etc. (patterns as building blocks), in chemical engineering to design chemical plants (patterns as unit operations), even in electrical and electronic engineering to design electric/electronic devices (patterns as hardware modules, such as a power generator or radio receiver).

Patterns have functioned as building blocks for complex systems for a long time. Such term as a handshake, for example, has been used for years at different hardware levels, such as the circuit level, bus level, network level, and is now being legitimately used at various software levels.

Patterns are normally not invented. They summarize previous engineering experience with solving typical problems and provide a way for applying reusability at the design level. Since on one hand, domain engineers are not fully trained in software development methodologies, and on the other hand, software engineers are not well familiar with the nature of physical processes to be controlled, we can use such patterns to record and reuse engineering knowledge in a form of software templates that serve as building blocks. It can help both professions to deal with and facilitate software development for real-time systems. But to be fully useful, these building blocks must reach far beyond the traditional program modularization techniques we are all familiar with, which rely on the use of subroutines, procedures and functions. Such patterns are needed even at the basic design level, before one can implement any subroutines or procedures in the code.

In this paper, we critically overview the state-of-the-art in the development and use of real-time design patterns, and present some of our own related experiences in real-time software design. The paper is structured as follows. Section 2 discusses previous work on software design patterns, in general, and presents some of the patterns developed specifically for real-time control systems. Section 3 presents our own view on the role of real-time design patterns in the development process and outlines our experiences in applying such patterns. Finally, Section 4 summarizes the paper and presents conclusions.

2. PREVIOUS WORK

2.1. DESIGN PATTERNS IN GENERAL

It is really hard to determine, who first came up with the idea of using the term *design patterns* in computing or software engineering. Work on software design patterns has been promoted mainly among software engineers and the classic book was published in 1995 [10]. The whole concept can be traced back to the late eighties and a series of workshops held at conferences on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).

In this and other similar works, a pattern is understood as a solution to a problem in a context. Patterns codify specific knowledge collected from experience in a domain. Types of software patterns vary from idioms at the programming language level, to frameworks for entire software systems. In this sense, at the intermediate level, procedures, functions, and classes can be considered pattern-like entities as well. In other words, patterns are repeatable units, building blocks.

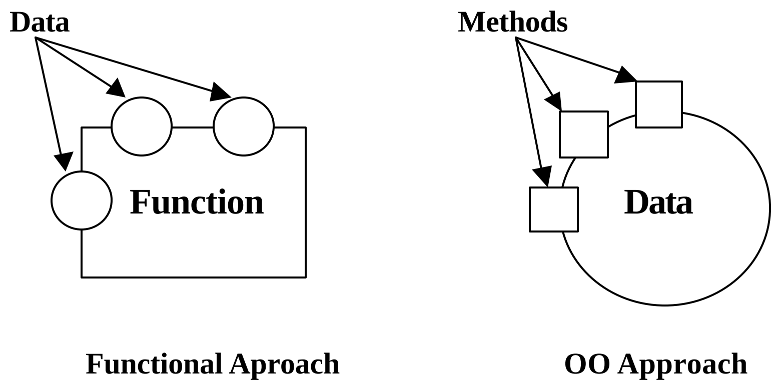

Figure 1. Relationship between Functional and Object-Oriented Approaches.

Gamma et al. [10] state that a design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The nature of object-oriented design and its relationship to traditional, structured design can be illustrated by considering the mutual roles of data and operations, as shown in Figure 1.

The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. A set of cooperating classes that make up a reusable design for a specific class of software is called a framework. According to [10] pattern description should include:

- name (with aliases) and classification
- intent
- motivation
- applicability
- structure
- participants
- collaborations
- implementation
- sample code
- known uses
- related patterns.

Additionally, they categorize all software patterns into three groups:

- creational patterns (abstract factory, prototype), that concern the process of object creation
- structural patterns (adapter, bridge, proxy), that deal with the composition of classes and objects
- behavioral patterns (chain of responsibility, mediator, visitor), that characterize the ways in which classes or objects interact and distribute responsibility.

Since patterns describe software abstractions, they apply to both the product and process (that is, how to make a product). They are usually considered to be architectural building blocks, described by the following characteristics, in addition to Name:

- Problem the pattern is trying to solve
- Context for which a pattern is designed
- Forces (trade-offs) that make clear the intricacies of the problem
- Solution, that describes the structure and behavior of a pattern
- Force Resolution, that describes a resulting context, which forces are unresolved, what other patterns must be applied, and how context is changed by the pattern
- Design Rationale, which tells where the pattern came from, why it works, and why it is used.

Buschmann [6] characterizes software patterns in a similar way: “A pattern for software architecture defines a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution.”

All this is important to understand software design patterns, especially in object-oriented technologies, however, has very little to do with real-time systems or even with software-controlled systems. The main problem is that all this knowledge has been developed by software engineers, not real-time engineers, thus it contains very little, if any, domain specific information. Therefore in the following section, we present a review of known works on specific patterns related to real-time control technologies.

2.2. REAL-TIME DESIGN PATTERNS: AN OVERVIEW

Describing the use of real-time design patterns, we divide the available information into three categories, regarding its source:

- industrial applications, which show some of the prac tical uses of design patterns in real-time systems, mostly used for control
- tool vendors, who discuss software design patterns from the point of view of automatic support for development, and
- academic perspective, which describes some of the research issues in identifying real-time design patterns.

To include the most relevant patterns that may be related to real-time software design, we focus in the discussion on design patterns used in control systems. This is under the assumption that computer control systems are, in fact, real-time computing systems [49], as a consequence of adopting the definition that a real-time system is "a computer systems with bounded response time".

2.2.1. INDUSTRIAL VIEW

Lea [21] was probably one of the first authors to write on a subject concerning real-time design patterns. He discusses patterns related to the redesign of avionics control systems. An Avionics Control System (ACS) is the main navigation system of an aircraft, that continuously collects sensor information to estimate actual state of an aircraft, computes desired aircraft position with respect to guidance modes, and performs actions that advise pilots or directly manipulate the actuators to control the flight.

After presenting a general view of the ACS, which is primarily a black box interacting with sensors, actuators (which are called effectors) and displays, general system architecture is devised, composed of the following interacting models:

- navigation models, representing estimates of actual states
- objective models, representing desired states
- error models, representing those differences between actual and desired state that is useful for directing flight
- action models, representing desired actuator and display settings suitable for manipulating aircraft hardware
- intermediate models, used to provide lower-level representation of the four above mentioned models.

Since no specific reference to design patters is made in the text, it is assumed that models play the role of patterns in this paper. The description of models, however, does not identify any specific patterns in terms of three categories of patterns (creational, structural, or behavioral) or in terms of a particular descriptive features (as listed in

Section 2.1 above). However, design guidelines, in a form of design steps, are given to build these models, which may be viewed as process patterns.

Rubel [31] authored another early paper related to real-time design patterns. He used a model of mechanical control systems to present a series of, what he calls, patterns. They are inserted during the design process, between the engineer and the equipment, helping to automatize the control. He identifies and describes the following four patterns: pedestal, bridge, symmetrical reuse, and elevate reference to enhance reuse. They are, in fact, models of a hierarchical architecture of a control system, but have little to do with its software architecture.

Berczuk [1, 2] describes a pattern language to guide development of a ground based system that will process telemetry data from an earth-orbiting astronomical observatory. He considers patterns as forms/regularities for describing architectural constructs in a manner that emphasizes these constructs' potential for reuse. They provide a way to document and share design expertise in an application independent fashion.

His papers focus on the process of classification and interpretation of the telemetry data packets as received from the spacecraft, and dispatching the resulting data objects for further processing. The patterns he is referring to are mostly at the organizational, that is, process level. The following specific patterns have been discussed:

- Loose Interfaces, which is mostly a suggestion for decoupling interfaces because of physical distribution and loose connections among participating teams
- Parser/Builder, whose role is to decompose incoming packets into content-related units
- Composite Factory, which is responsible for producing the decomposed packet units
- Handlers, whose role is to deliver the assembled data units to the destinations.

A more contemporary approach to using design patterns in telemetry processing is presented by Hermann et al. [14]. They follow the recommendations of the Consultative Committee for Space Data Systems (CCSDS), which define the hierarchy of the data flow for transmissions (and corresponding hierarchy for data reception), as follows:

- create source packets from application date
- segmentation of source data packets
- insert telemetry packets into frames of appropriate virtual channels
- multiplex the frames of different virtual channels into a single sequence
- create pysical data channels.

Taking this hierarchy as a basis, telemetry packets and transfer frames are used as basic data elements, for which specific classes, TranferFrameHandler and TelemetryPacketHandler are designed. These classes, plus and ApplicationHandler class, then form a skeleton of two patterns:

- abstract factory, for choosing and creating telemetry objects responsible for remote sensing of particular satellite data, and
- facade, for encapsulating the telemetry standard functionality into a simple and unified interface.

Respective class diagrams for the facade pattern are shown in Fig. 2.

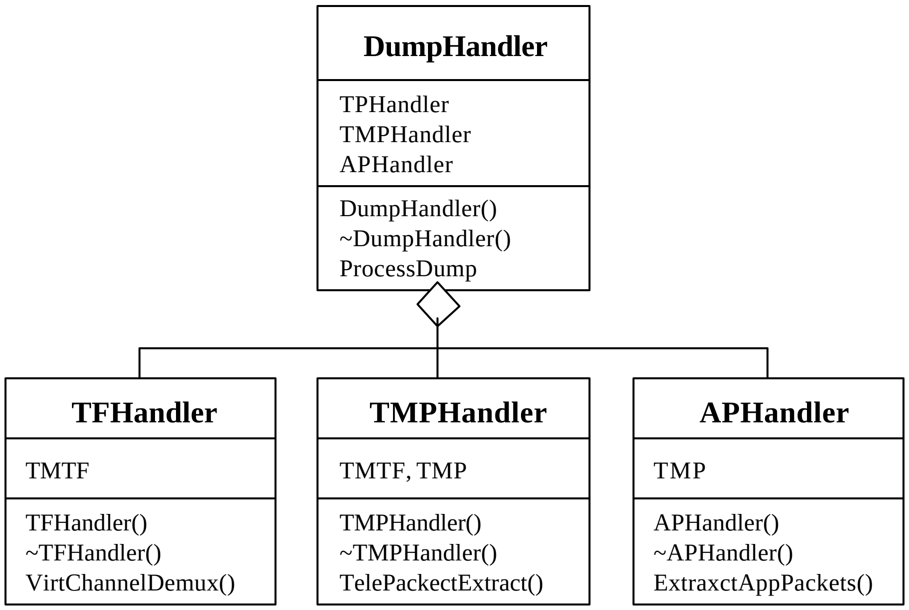

Figure 2. Facade Pattern for Telemetry.

Molin and Ohlsson [24] describe a series of design patterns related to the enhancement of microcontroller-based fire alarm systems. The key function of the system is to detect, via a number of sensors, that something out of the ordinary occurs, and generate an alarm. Generating an alarm involves activating alarm bells, posting messages to text displays, invoking extinguishers, calling firefighters. Respective design patterns are grouped into a hierarchy of the following: deviation, point and pool, periodic object, lazy state, and data pump; thus they form a language of patterns for fire alarm systems. The basic idea of a real-time system, based on the transformation of input signals into corresponding outputs after processing, is illustrated in Fig. 3.

Dagermo and Knutsson [8] describe an object-oriented framework for a vessel control system. In creating the framework they used the idea of design patterns from [10]. They describe the patters that have adopted and used in this particular problem: observer, singleton, proxy, and state patterns. For example, an observer pattern establishes a one-to-many relationship among objects, to serve the purpose of notifying all dependants when one object is changing state. A simple sequence diagram illustrating the behavior of the observer pattern is shown in Fig. 4.

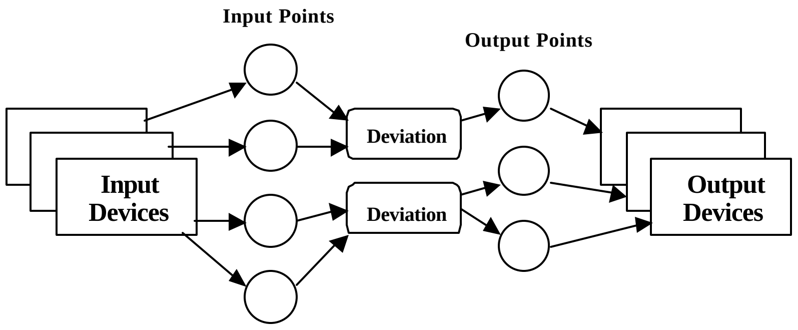

Figure 3. Transformation Scheme Leading to Deviation and Point Patterns.

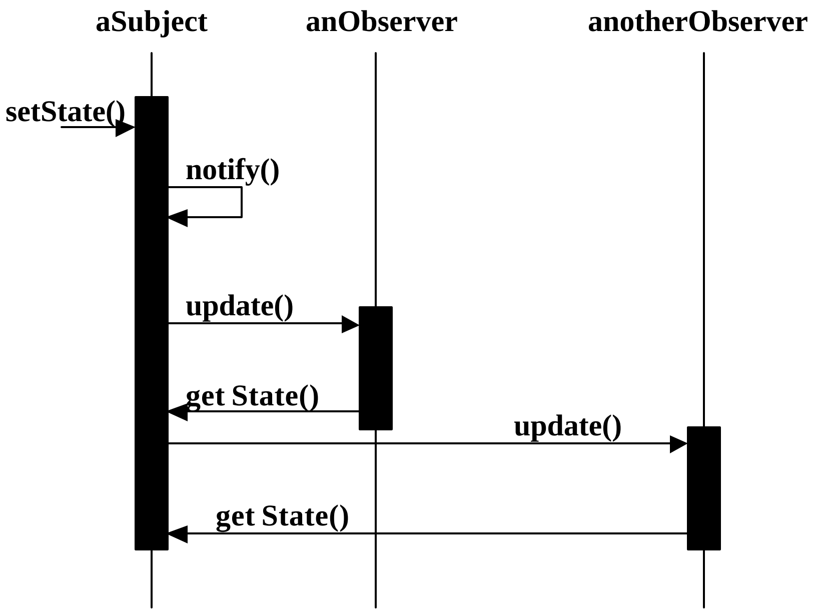

Figure 4. Sequence Diagram for the Observer Pattern.

There are many other published papers on applying design patterns to real-time systems. More recently, Heverhagen and Tracht [15,16] discussed their use of patterns to define a Function Block Adapter (FBA) as a stereotype in a Unified Modeling Language

(UML). Their objective is to integrate systems designed in UML Real-Time (UML-RT) with existing PLC environments designed according to IEC Std 61499. Buschmann et al. [6] present an outline, how the framework technology would help building software architectures for process automation systems. As a case study, they use a framework for hot rolling mills.

Several other papers are also worth noting, but cannot be summarized here due to space limitations. Woodward [43] described briefly his experiences in using design patterns to the simulation of an Airborne Early Warning (AEW) aircraft. Bottomley [3] presents an approach to pattern development for autonomous embedded systems with limited user interaction, variety of sensor inputs and control of a few outputs. Ihme [17] presented a related approach on developing patterns for measurement and data acquisition. Most recently, Tomura et al. [41, 42] described design patterns for use in large-scale systems for distributed simulation, and Sharp [40] applied design patterns to designing real-time software for avionics.

2.2.2 TOOL VENDORS PERSPECTIVE

Use of design patterns is the most effective, if they not only remain on paper, but are also supported by automatic development tools. Such tools need to adhere to a certain notational standard and allow developers to support software construction at various stages of a lifecycle.

For real-time software development, the tool has to demonstrate real-time related features, such as concurrency, responsiveness, etc. Universal Modeling Language (UML) is such a standard being more and more accepted nowadays, especially with its emerging Real-Time Profile (UML-RT). Several vendors, such as Rational, I-Logix, Artisan, and others publish extensively on their views of supporting real-time software design using UML, especially with design patterns [39].

Selic [37,38] presents an architectural pattern for real-time control software. It is called Recursive Control pattern, and relies on separating the control part from the functional part of the system (Fig. 5). It is designed to deal with what is traditionally considered ancillary software functions, such as startup and shutdown, failure detection and recovery, online maintenance, etc. Separation of control and service related functions allows each to be modified independently.

The pattern is described in terms of structure, collaborations, applicability, participants, consequences, and relationships to more basic patterns. Its behavior is described using a state machine diagram. It is interesting to note that this pattern applies the classical feedback control model, that is, separating the functional part from the controller. It is applied to design software for alternating bit protol and to client/server system.

Douglass [9] also defines a design pattern as a general solution to a commonly occurring problem. A pattern has three parts: a problem context, a generalized approach to a solution, and a set of consequences. Patterns are usually constructed by abstraction, that is, extracting out the things from large set of specific instances. To him, patterns are described involving classes interconnected via relations. He groups design patterns into two major super-categories: architectural patterns, which are heavy-weight pieces related to architectural decisions, and mechanistic design patterns, which are much smaller in scope.

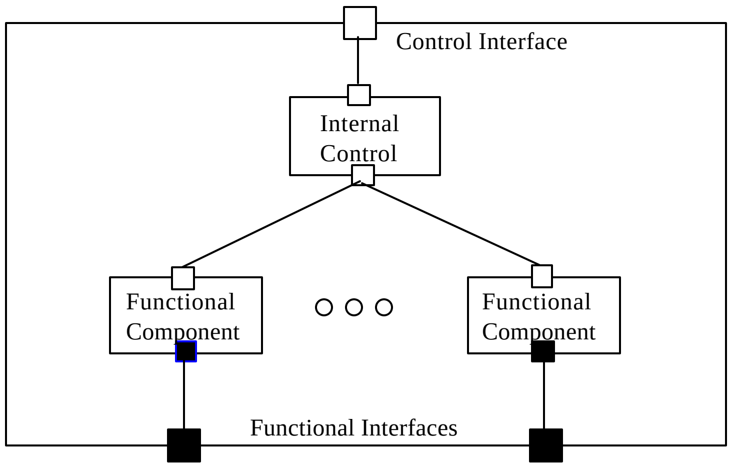

Figure 5. Recursive Pattern Structure.

In the architectural patterns group, he lists a large number of design patterns divided into several categories:

- execution control (preemptive multitasking, cyclic executive, time slicing, cooperative multitasking)
- communications (master-slave, time-division multiplexing, bus-mastered)
- distributed systems (proxy, broker, asymmetric multiprocessing, symmetric multiprocessing, semi-symmetric multiprocessing)
- resource (static allocation, fixed size allocation, priority ceiling)
- safety and reliability (homogeneous redundancy, heterogeneous redundancy, sanity check, monitor-actuator, watchdog, safety executive).

In the mechanistic patterns, he distinguishes among the following categories:

- simple pattern, such as observer, transaction, smart pointer
- reuse patterns, including container, interface, policy, and rendezvous,
- state behavior, inclusing state and state table.

Douglass uses a slightly extended UML notation to describe patterns. For example, Fig. 6 presents the structure of a rendezvous pattern. Strictly speaking, even though called real-time design patterns, very few of the patterns from Douglass report [9] are related to real-time operation.

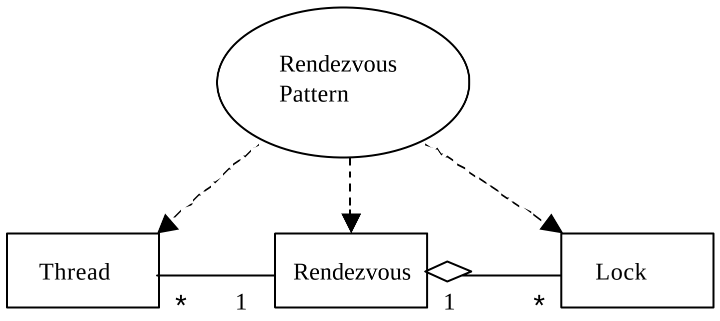

Figure 6. Structure of a Rendezvous Pattern.

2.2.3 ACADEMIC VIEWS

There is only a handful of authors from academia, who write seriously on design patterns for real-time systems.

The most systematic view of software design patterns for real-time systems has been presented by Sanden, with the use of his entity-life modeling approach. He defines a design pattern as a known solution for a problem, that is, an existing model of software. In a series of papers [32, 33, 34], he discusses several design patterns suitable for use with his entity-life modeling approach, for concurrent and real-time systems:

- shared-resource pattern
- assembly line pattern
- state-machine pattern.

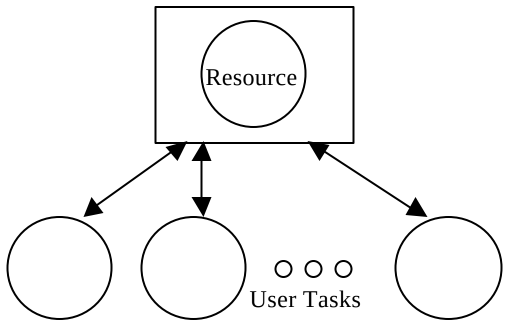

Figure 7. Shared Resource Pattern.

In a shared resource pattern (Fig. 7), a thread represents each resource user. A resource itself may be a non-executable entity's data item or another executable entity, such as a thread. In an assembly line pattern (Fig. 8), each thread represents a resource and the data items, being operated upon, travel along the line of threads. It is a variation of a pipe pattern but possibly with a resource to wait for, which is not the case for pipes.

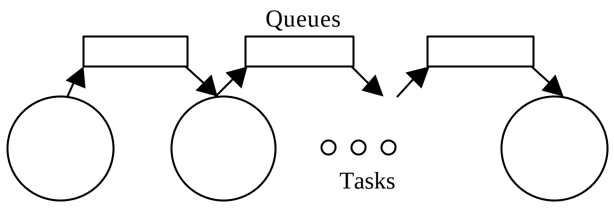

Figure 8. Assembly Line Pattern.

A state machine pattern, as presented by Sanden [32, 34], is a template for handling state machines in real-time software. It is not just another representation of a state machine (finite automaton), such as a state transition diagram, but involves such representation as one of the structural elements of the pattern. Other elements form an access layer to the automaton and include (Fig. 9): event captor, E, activities, C, and passive interfaces, P. Sanden discusses two examples: an odometer and a weather buoy.

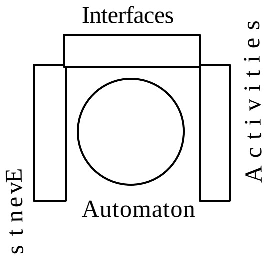

Figure 9. Structure of a State-Machine Pattern.

Carvalho et al. [7], introduce a sensor-reactor design pattern to separate sensing capabilities from reacting capabilities needed to return the system to a stable state. They introduce an AbstractSensor and AbstractReactor classes that are superclasses of concrete sensor and reactor models. Connecting sensor and reactor models is done via a Logic class that incorporates all required computations.

Parikh et al. [26] discuss the use of design patterns for data fusion. They present several versions of the classifier pattern for use in condition monitoring and fault diagnosis of a diesel engine cooling system. They experimented with the following three patterns:

- multi-layer perceptron classifier
- radial-basis-function classifier, and
- neuro-blackbord approach.

Nelson [25] presents a design pattern for control of autonomous or robotic vehicles. It is a tri-level approach to vehicle control, calles STESCA (Strategic-Tactical-Execution Software Control Architecture). At the highest, strategic, level mission specification is expressed. This is converted at the tactical level into actual vehicle component control commands. The lowest, execution, level interfaces the commands to the individual vehicle components.

Graves and Czarnecki [13] discuss behavior-based patterns for robotics. In a structural type of pattern description they focus on robot control architecture and identify three kinds of patterns to distribute control functions between man and machine:

- traded control, in which responsibilities for producing behavior are traded between man and machine,
- shared control, in which the functions are integrated, such that an operator is guiding the robot to target, while collision avoidance is autonomous, and
- supervisory control, where the controller is preprogrammed to perform the entire task, while an operator is overseeing the problems.

Gomaa and Farrukh [11] describe the way to map features of a distributed application to a number of architectural patterns. An architectural pattern contains the following information:

- declaration of component types
- declaration of other architectural patterns required by this pattern
- instantiation of components
- definition of interconnections between internal components
- definition of interconnections between this pattern and other included architectural (prerequisite) patterns, including the kernel architectural pattern.

A case study of a flexible manufacturing system, composed of three patterns, the kernel pattern, factory production pattern, and flexible manufacturing pattern, is analysed.

More patterns addressing some fundamental problems of concurrent computing applied to real-time control systems can be found, for example, on distributed rendezvous [18], deadlock avoidance [44] and client/server systems [12]. McKegney [22] reviewed some issues of pattern application to real-time object-oriented software design and postulated development of specialized tools [23]. Spectacular work has been done by Schmidt and coworkers [36], in relation to distributed object-oriented systems, in many aspects related to real-time systems. Pont [27] collected a large number of design patterns for embedded systems and used them in a variety of applications [28, 29, 30]. Sanz et al. [35] presented the use of design patterns for intelligent control systems.

Even though Buschmann represents an industrial company, patterns he has developed, a master-slave pattern [4] and real-time constraints as strategies [5], are so fundamental that both qualify to be included in the academic view. Especially interesting is the latter design pattern that decouples real-time specific constraints and behavior from the application service to which they apply. The application service is provided by a service class and real-time related aspects are delegated to strategies which implement these in a specific manner.

3. AUTHOR'S APPROACH

This author has advocated a top-down approach to designing real-time software, which naturally leads to the formulation and application of design patterns. The fundamental idea comes from the requirements view of the system under development and relies on the fact that all real-time systems interact with their environment in four basic ways, via [45]:

- process (plant) interface, involving measurement and control actions
- user interface, handling interaction with an operator
- communication link, taking care of controller's interaction with other parts of a control system via a communication network
- database interface, responsible for storing and retrieving data.

This is because all functional requirements on real-time software are normally expressed in terms of input/output functions. If we assume that, in addition to that, every real-time system includes a processing component and a timing component, a generic architectural pattern is generated as shown in Fig. 10.

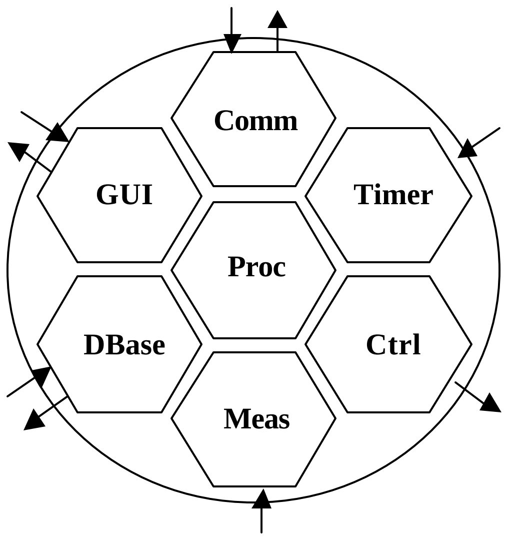

Figure 10. Architectural Pattern for Real-Time Software.

Refining the architectural pattern from Fig. 10 leads us to different design patterns depending on a specific real-time application. Several sophisticated real-time applications have been developed for this architectural pattern, including those for controlling elementary particle collider, satellites, and combat vehicles [20], as well as for simpler applications, such as instrumentation software [46] or cruise control [50]. Other applications are being developed, such as air-traffic control software and satellite ground control station.

It is important to note that the architecture from Fig. 10 does not imply the use of any particular design methodology for further development, nor does it preclude the use of any specific methodology for designing real-time software. In particular, traditional structured design approaches for real-time software design can be successfully used, as well as newer object-oriented methodologies can be applied. For example, software components from Fig. 10 can be mapped onto traditional modules in a sequential structured design, or onto concurrent tasks in a multitasking structured design. They can be also mapped onto objects in an object-oriented design, or onto a mixture of modules, tasks and objects in a hybrid design.

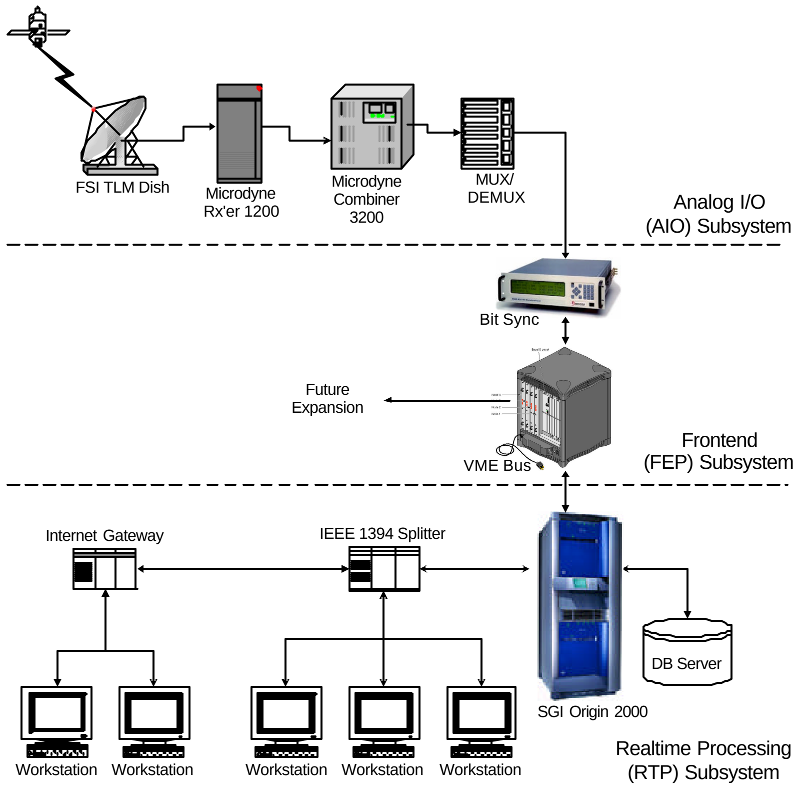

Figure 11. Satellite Ground Control Station Physical Diagram.

This architectural pattern can be also mapped onto a number of different hardware architectures. In particular, the combination of modules can be run on a single processor, but also, a family of concurrent tasks can be executed on a single processor or on a multiprocessor system. Ultimately, a combination of objects can be distributed over a network using a middleware technology, such as CORBA [20, 35]. It is important to note that CORBA offers a good platform for pattern-based design for real-time software, since patterns do naturally lead to component-based development.

An example of using this approach to develop software for a complicated distributed real-time system involves satellite ground control station, whose physical diagram is shown in Fig. 11. Following the architectural principle outlined in Fig. 10, one can group all necessary components into those responsible for major functionalities: GUI, Telemetry, Database, and GPS Timing. Respective real-time design patterns are described both at the structural and behavioral level, using UML notation. Due to space limitations we only present here a simple sequence diagram showing the behavior of the GUI pattern (Fig. 12).

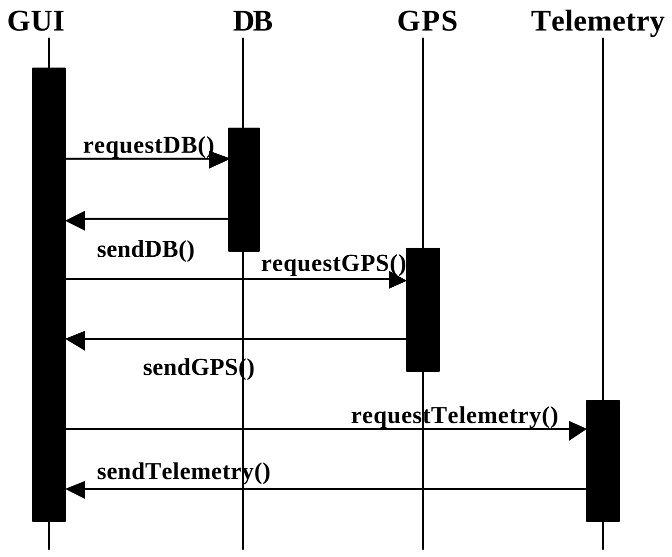

Figure 12. Sequence Diagram for a Simple GUI Pattern.

4. CONCLUSION

As shown in the review of the most important works, the use of design patterns for real-time applications is more and more common nowadays. Design patterns increase software reusability and facilitate development by arming software developers with knowledge encapsulated in patterns by domain experts. In this view, the author advocates a specific approach to designing real-time software, which is based on treating real-time systems as control systems. This approach naturally leads to the formulation of design patterns in terms of handling particular interfaces with the external environment. Applying patterns to real-time applications has several disadvantages, however, to name only indeterminism introduced at the design level [47] and inadequate support by the development tools [48].

REFERENCES

[1] BERCZUK S.P., A Pattern for Separating Assembly and Processing, Pattern Languages of Program Design, J.O. Coplien and D. Schmidt (Eds.), Addison Wesley, Reading, Mass., 1995, pp. 521-528.

[2] BERCZUK S.P., Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams, Pattern Languages of Program Design 2, J.M. Vlissides, J.O. Coplien, N.L. Kert (Eds.), Addison-Wesley, Reading, Mass., 1996, pp. 194-206.

[3] BOTTOMLEY M., A Pattern Language for Simple Embedded Systems. Proc. PLoP'99, 6th Annual Pattern Languages of Programming Conference, Monticello, Ill., 15-18 August 1999.

[4] BUSCHMANN F., The Master-Slave Pattern, Pattern Languages of Program Design, J.O. Coplien and D. Schmidt (Eds.), Addison Wesley, Reading, Mass., 1995, pp. 133-142.

[5] BUSCHMANN F., Real-Time Constraints as Strategies, Proc. EUROPLOP'98, Third European Conf. on Pattern Languages of Programming and Computing, Bad Irsee, Germany, 9 - 11 July 1998.

[6] BUSCHMANN F. et al., Framework-Based Software Architectures for Process Automation Systems, Annual Reviews in Control, 2000, Vol. 24, pp. 163-175.

[7] CARVALHO S., G. ROSSI, F. BALAGUER, Using Design Patterns in Real Time Applications, Proc. WRTP'96, IFAC/IFIP Workshop on Real Time Programming, Gramado, Brazil, 4-6 Nov. 1996, Elsevier, Oxford, pp. 93-96.

[8] DAGERMO P., J. KNUTSSON, Development of an Object-Oriented Framework for Vessel Control System, Technical Report ESPRIT III/ESSI/DOVER #10496, Dover Consortium, Karlskrona, Sweden, 1996.

[9] DOUGLASS B.P., Real-Time Design Patterns, iLogix Inc., Burlington, Mass., 2001, http://www.ilogix.com/

[10] GAMMA E. et al., Design Patterns: Elements of Reusable Software Design, Addison-Wesley, Reading, Mass., 1995.

[11] GOMAA H., G.A. FARRUKH, Methods and Tools for the Automated Configuration of Distributed Applications from Reusable Software Architectures and Components, IEE Proc. – Software, 1999, Vol. 146, No. 6, pp. 277-285.

[12] GOMAA H., D. MENASCE, M. SHIN, Reusable Component Interconnection Pattern for Distributed Software Architectures, SIGSOFT Software Engineering Notes, 2001, Vol. 26, No. 3, pp. 69-77.

[13] GRAVES A.R., C. CZARNECKI, Design Patterns for Behavior-Based Robotics, IEEE Trans. on Systems, Man & Cybernetics, Part A (Systems & Humans), 2000, Vol. 30, No. 1, pp. 36-41.

[14] HERRMANN A., T. SCHÖNINGS, Standard Telemetry Processing – An Object Oriented Approach Using Software Design Patterns, Aerospace Science and Technology, 2000, Vol. 4, pp. 289-297.

[15] HEVERHAGEN T., R. TRACHT, Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters, Proc. ISORC'2001, IEEE Int'l Symposium on Object-Oriented Distributed Computing, IEEE Computer Society Press, Los Alamitos, Calif., pp. 395-402.

[16] HEVERHAGEN T., R. TRACHT, Using Stereotypes of the Unified Modeling Language in Mechatronic Systems, Proc. ITM'01, 1st International Conference on Information Technology in Mechatronics, Istanbul, Turkey, 1-3 October 2001, pp. 333-338.

[17] IHME T., Design Patterns and Frameworks for Real-Time Embedded Control Software. VTT Electronics, Oulu, Finland, http://www.vtt.fi/ele/research/soh/projects/finesse/slides/Frameworks-Product-Line.pdf

[18] JIMENEZ-PERIS R., M. PATINO-MARTINEZ, S. AREVALO, Multithreaded Rendezvous: A Design Pattern for Distributed Rendezvous, Proc. ACM Symp. On Applied Computing, San Antonio, Texas, 28 Feb. – 2 March 1999, pp. 571-579.

[19] JOBLING C.P. et al., Object-Oriented Programming in Control System Design: A Survey, Automatica, 1994, Vol. 30, No. 18, pp. 1221-1261.

[20] van KATWIJK J., J.-J. SCHWARZ, J. ZALEWSKI, Practice of Real-Time Software Architectures: Collider, Satellites and Tanks Combined, Proc. IFAC Conference on New Technologies for Computer Control, Hong Kong, P.R. of China, 19-21 November 2001.

[21] LEA D., Design Patterns for Avionics Control Systems, DSSA Adage Project ADAGE-OSW-94-01, State University of New York, Oswego, New York, November 1994.

[22] MCKEGNEY R., Application of Patterns to Real-Time Object-Oriented Software Design, Master Thesis, Queen's University, Kingston, Ontario, Canada, 2000, http://www.cs.queensu.ca/home/mckegney/Research.html

[23] MCKEGNEY R., T. SHEPARD, Design Patterns and Real-Time Object-Oriented Modeling, Proc. OOPSLA 2000, Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum), ACM, pp. 55-56.

[24] MOLIN P., L. OHLSSON, Points & Deviations - A Pattern Language for Fire Alarm Systems. Proc. PloP'96, Third Annual Pattern Languages of Programming Conference, Monticello, Ill., 4-6 Sept. 1996, http://www.cs.wustl.edu/~schmidt/PLoP-96/

[25] NELSON M.L., A Design Pattern for Autonomous Vehicle Software Control Architectures, Proc. 23rd COMPSAC International Computer Software and Applications Conference, 1999, pp. 172-177.

[26] PARIKH C.R. et al., Towards a Flexible Application Framework for Data Fusion Using Real-Time Design Patterns, Proc. EUFIT '98, 6th European Congress on Intelligent Techniques and Soft Computing, Aachen, Germany, 7-10 Sept. 1998, Vol. 2, pp. 1131-1135.

[27] PONT M.J., Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers. Addison-Wesley, Harlow, England, 2001.

[28] PONT M.J., Control System Design Using Real-Time Design Patterns, Proc. UKACC Int'l Conf. on Control, III. Conf. Publication No. 455, IEE, London, 1998, Vol. 2, pp. 1078-1083.

[29] PONT M.J., Designing and Implementing Reliable Embedded Systems Using Patterns, Proc. EuroPLoP '99. 4th European Conference on Pattern Languages of Programming and Computing, Dyson, P. and Devos, M. (Eds.), Bad Irsee, Germany, 8-10 July 1999.

[30] PONT M.J. et al., The Design of Embedded Systems Using Software Patterns, Proc. Workshop on Condition Monitoring, Swansea, UK, 12-15 April 1999, pp. 221-236.

[31] RUBEL B., Patterns for Generating a Layered Architecture, Pattern Languages of Program Design, J.O. Coplien and D. Schmidt (Eds.), Addison Wesley, Reading, Mass., 1995, pp. 119-128.

[32] SANDEN B., The State-Machine Pattern, Proc. TRI-Ada Conference, 1996, pp. 135-142.

[33] SANDEN B., Concurrent Design Patterns for Resource Sharing, Proc. TRI-Ada Conference, 1997, pp. 173-183.

[34] SANDEN B., A Design Pattern for State Machines and Concurrent Activities, Proc. Ada-Europe 2001 Conference, Springer-Verlag, Berlin, pp. 203-214.

[35] SANZ R. et al., Design Patterns for Intelligent Control Systems, Proc. 1999 IFAC Congress, Beijing, P.R. China, 1999.

[36] SCHMIDT D. et al., Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects. Vol. 2, John Wiley and Sons, New York, 2000.

[37] SELIC B., An Architectural Pattern for Real-Time Control Software, PLoP'96, Proc. Third Annual Pattern Languages of Programming Conference, Monticello, Ill., Sept. 4-6, 1996, http://www.cs.wustl.edu/~schmidt/PLoP-96/

[38] SELIC B., Recursive Control, Patteren Languages of Program Design 3, R.C. Martin et al. (Eds.), Addison-Wesley, Reading, Mass., 1998 pp. 147-161.

[39] SELIC B., J. RUMBAUGH, Using UML for Modeling Complex Real-Time Systems, Rational Corporation, Redwood City, Calif, 1998, http://www.rational.com/

[40] SHARP D., W. ROLL, Pattern Usage in an Avionics Mission Processing Product Line, Proc. OOPSLA 2001 Workshop Towards Patterns and Pattern Languages for OO Distributed Real-Time and Embedded Systems, Tampa, Fla., 14 October 2001, http://www.cs.wustl.edu/~mk1/RealTimePatterns/OOPSLA2001/submissions/DavidSharp.pdf

[41] TOMURA T. et al., Object-Oriented Design Pattern Approach for Modeling and Simulating Open Distributed Control System, Proc. 2001 ICRA, IEEE International Conference on Robotics and Automation, Seoul, South Korea, 21-26 May 2001, Vol. 1, pp. 211-216.

[42] TOMURA T. et al., Developing Simulation Models of Open Distributed Control Systems by Using Object-Oriented Structural and Behavioral Patterns, Proc. ISORC 2001, Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 428-437.

[43] WOODWARD K.G., Heading off Tragedy: Using Design Patterns Against a Moving Target, Proc. 2nd World Conf. on Integrated Design and Process Technology, Austin, Texas, 1996, pp. 280-285.

[44] WYNS J., H. van BRUSSEL, P. VALCKENAERS, Design Pattern for Deadlock Handling in Holonic Manufacturing Systems, Production Planning and Control. 1999, Vol. 10, No. 7, pp. 616-626.

[45] ZALEWSKI J., Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences, Annual Reviews in Control, 2001, Vol. 25, No. 1, pp. 133-146.

[46] ZALEWSKI J., Developing Component Based Software for Real-Time Systems, EUROMICRO Workshop on Component-Based Software Engineering, Warsaw, Poland, 4-6 September 2001, IEEE Computer Society, Los Alamitos, Calif., pp. 80-87.

[47] ZALEWSKI J., Object-Orientation vs. Real-Time Systems, Real-Time Systems, 2000, Vol. 18, pp. 75-77.

[48] ZALEWSKI J., Distributed Real-Time Software Architectures and Effective Use of Automatic Tools, Proc. KKIO 2000, 2nd National Conference on Software Engineering, Zakopane, Poland, 18-20 October 2000, pp. 127-136.

[49] ZALEWSKI J., Real-Time Systems: Design and Practical Aspects, Proc. 14th Fall Conf. of the Polish Information Processing Society, Mragowo, Poland, 16-20 November 1998, pp. 17-28.

[50] ZALEWSKI J., R. SANZ, M. PONT, Supporting the Design and Implementation of Control Systems Using Design Patterns, IEEE Control Systems, July 2003 (scheduled for publication).

WZORCE PROJEKTOWE OPROGRAMOWANIA CZASU RZECZYWISTEGO

Celem tego artyku³u jest przedstawienie wzorców projektowych do programowania systemów czasu rzeczywistego. Poszczególne jego czêœci omawiaj¹ kolejno: powstanie koncepcji wzorców projektowych w programowaniu i wzorce projektowe dla systemów czasu rzeczywistego z punktu widzenia: 1) u¿ytkowników takich systemów; 2) wytwórców narzêdzi programistycznych; 3) badañ naukowych. Omówiono tak¿e klasyfikacje takich wzorców wynikaj¹c¹ z systematycznego podejœcia do projektowania systemów czasu rzeczywistego.

Software Design Patterns - Explain the Analysis phase and Design phase, What are design patterns?, What are the four major categories in which design patters can be categorized? Name the methods that lie in these categories.Â 1. It belongs to the high level design stage 2. It helps in discovery of classes and specification of data and processes associated with those classes. If you remember, software engineers speak a common language called UML. And if we use this analogy of language, then design patterns are the common stories our culture shares, like for instance fairy tales. They are stories about commonly occurring problems in software design and their solutions. And as young children learn about good and evil from fairy tales, beginning software engineers learn about good design (design patterns) and bad design (anti-patterns).