Multidimensional Separation of Concerns[*]

Flavio De Paoli
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano - Bicocca
Via Bicocca degli Arcimboldi, 8 – 20126 – Milano – Italy
depaoli@disco.unimib.it

Abstract

Separation of concerns has been widely recognized to be a major issue in software design. Unfortunately, the definition of what concerns have to be considered of primary importance, and which is the best way to support software development based on those concerns are still missing. This paper presents three examples of separation of concerns and discusses possible solutions.
The taken approach is the definition of a conceptual model that underlies the definition of a reference platform to support the development of software. The approach is similar to the one taken be the Aspect-Oriented Programming, even if the projects described in the paper were not directly influenced by it. The current work is toward the definition of language constructs that unify the solutions and provide a start to develop a concern-oriented language.

1. Introduction

The traditional software construction does not address specific aspects by independent modules or constructs. Most programming languages supply the software engineers with a set of general-purpose constructs that can be used to implement each concern and can be embedded (almost) anywhere in the code. Moreover, some aspects are implemented as part of the abstract machine on which the language is based, thus preventing the programmer of modifying the standard behavior.
Some primary concerns have already been identified and accepted, and common programming languages supply clean constructs to address them. Examples are the separation between interface and implementation of modules and the exception handling mechanism. Actually, the exception handling mechanism is a halfway solution, it provides for a clean separation from regular code and exceptional code, but within each module, thus missing a global independence. In fact, the control of exceptional situations is spread over the whole software, thus making it obscure.
What would be useful is the possibility to develop independent modules, or components, to address specific concerns that can be composed to form the desired software systems. This would provide support for the construction of open software with well-known advantages. They deal with readability, maintenance, tailorability, reusability, reliability, portability and so forth.
In this paper, the experience with three projects is described. The first project deals with hard real-time embedded systems in which the concern is the predictable execution of tasks according to timing constraints. The second deals with synchronous cooperative systems in which the concerns are the sharing of applications and the controlled access to them. The last deals with knowledge management systems in which the concern is the contextualized presentation of information. These projects are examples of concerns that are not often addressed by general-purpose languages and framework; thus they are good case studies for further investigation and definitions of models to cleanly support separation of concerns.
The common goal is the definition of architectural models and frameworks to develop domain-specific applications. A common assumption is that concerns have to be addressed by independent components that can communicate each other someway, depending on the nature of the application and of the reference platform. Moreover, components have to be distinguishable even at run-time, thus enforcing the concern of producing open software by addressing tailorability and configurability.
The next sections sketch the three projects. Then a discussion is provided.

[*] The work described in this paper was partially supported by the European Community – Esprit Projects n. 20.592 OMI/MODES and n. 28.842 Klee&Co - and CNR - Progetto Coordinato n.99-02010-CT07.

2. Real-Time Systems

This section describes the experience with the TDE project developed within the Esprit Projects OMI/MODES. TDE is part of the HyperReal project that has the goal of developing a complete development environment and run-time support for hard real-time systems [4]. The goal of TDE was to develop a platform to support embedded real-time systems with a "time-driven" approach.

Real-time programming paradigms are often based on concurrency models that are intrinsically not suitable for real-time execution since they are based on non-deterministic execution of suspensive constructs [6]. In such systems, time remains external to the execution model; consequently, time constraints need to be translated to fit the execution model. TDE bases the execution of tasks directly on their time constraints. This approach allows system designers to concentrate on timing issues, and the run-time support to constantly control the system behavior according to timed plans.

The major concern to be considered is the control of the system execution. Traditional environments provide implicit mechanisms that are out of control of the programmers. It means that the programmer cannot have sharp control over the behavior of each element, nor over the behavior of the whole system. While this approach is reasonable for intrinsically concurrent systems, it becomes unacceptable for real-time systems in which activities cannot be performed eventually, but “at the right time” as specified by timing constraints. In time-driven systems, activities are scheduled in accordance with clock times as shown by an integrated independent clock or as determined from the readings of a system of clocks. As the logical clock reading reaches certain predefined values, an appropriate action is selected for execution from a lookup table [12].

The TDE approach aims at setting a set of abstractions to support the definition of a system as a set of objects that behave as autonomous, reactive components (agents), and controllers, whose task is to drive the agents' execution [7]. Controllers include every issue related to synchronization and scheduling, thus making them first class concerns. In other terms, agents are designed without any embedded assumption on timing and synchronization. The controller drives the agents' execution to meet the system's requirements.

This separation of concerns has several advantages, ranging from readability and tailorability enhancement, to formal verifiability. In particular, this approach leads to modular design of systems. Modularity allows designers to build up tailored systems out of existing components, and supports the integration of these systems with existing environments.

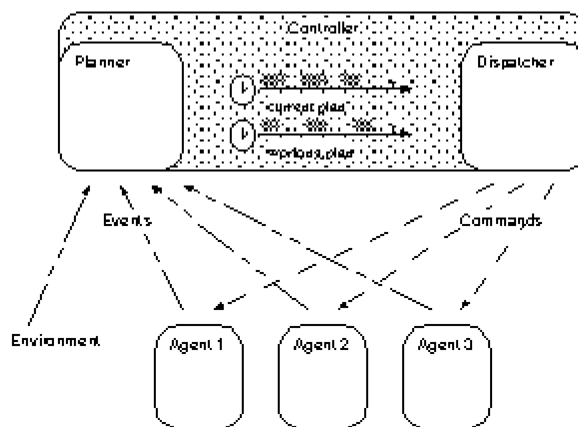

Figure 1: The TDE architecture.

Figure 1 illustrates the TDE architecture. A *planner* and a *dispatcher* form the *controller* of a system. In the simplest definition, the dispatcher receives a *plan* from a planner and lets agents run accordingly. A *plan* is a sequence of actions that represents the commands that can be scheduled and dispatched to agents. An *action* specifies an operation and the timing constraints associate with it. Constraints are expressed by the worst-case execution time (wcet), and by a validity interval ([after, before]), i.e., the lower bound after which the action can be executed and the upper bound before which the action must be completed. A plan is associated with a *reference clock* that defines the timeline.

This architecture leads to modular controllers that separate the machine dependent aspects from those dealing with the semantic of the application. A planner is in charge of setting up a timed plan that describes the activities to be performed by agents. The dispatcher is in charge of executing that plan by translating the actions of the plan in commands that drive the agents.

Agents are designed as objects with a private part and an interface. Agents are components that reacts to commands issued by the dispatcher. The implementation splits a command in two parts: the selection of the operation to be executed and the execution of that operation by the agent. To let the dispatcher select the operation, an agents exports a set of *control* variables that can be set to signal events to the agent. When an agent has a control variable set, it can execute. The effect of the execution is described by the behavior of the agent. The behavior is modeled as finite-state automaton that defines how the agent reacts to a certain event. This means that an agent behaves according to its internal status and external events.
The private part encapsulates data and operations. Operations are atomic, since they do not include any suspensive primitive, and they cannot be pre-empted to ensure deterministic execution. Agents are basic building blocks for systems that are driven by the *controller*. They are not aware of when operations are executed, nor are capable of taking any scheduling-related decision. An agent may notify the environment about the internal status by means of signals (events), and may exchange data with other components without explicit synchronization.

3. Cooperative Systems

As cooperative systems, we define multi-user applications that let a set of user share a common working space. They are characterized by the need of controlling multicast communication in a selective way. Traditional approaches to software development do not separate concerns like implementation of the system's features, system's management, and policies to use the system [2] [3] [5].
As in the TDE project, open architecture, based on specialized components, has been proposed to address each concern separately, and to compose them in a single platform.
An example of the proposed architecture is shown in Figure 2. It is a multi-tier architecture with three kinds of components: Application components, Control components, and Connector components.
The example shows an application split in three components, each one dealing with a specific aspect. Assuming a spreadsheet application, the abstract data component maintains a database of cell values and manages interdependencies between them. The abstract user interface component defines the overall layout of the user interface (e.g., menus, buttons, scroll bars, as well as the graphical representation of data, say, in the conventional tabular format). Finally, the physical interaction component manages I/O devices. Components can communicate by exchanging requests (e.g., "set cell B3 with value 45"), and notifications (e.g., "cell B3 has value 45").

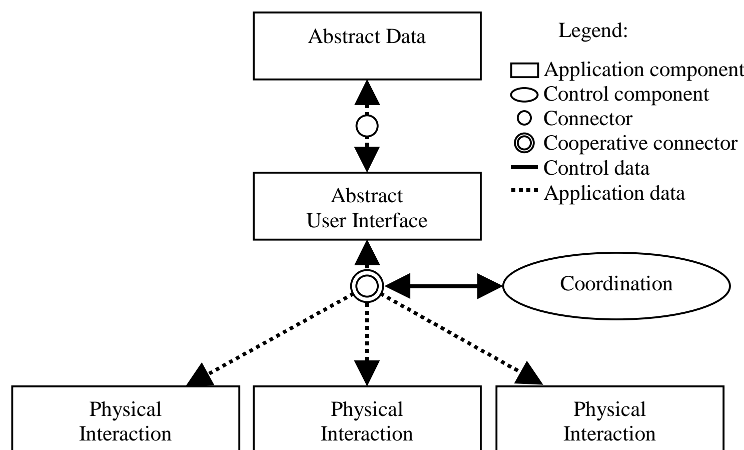

Figure 2: A cooperative system architecture.

Application components are connected by connector components that control the data flow between the connecting components. Connectors work as bridges between components to let them communicate. They are in charge of configuring the application and transfer data in a controlled way. This means that connectors filter (and possibly adapt) the exchanged data to implement an access control policy.
To let users share an application a *cooperative connector* is interposed between two application components. The cooperative connector functionality consists in *multiplexing* requests generated from multiple front-ends to the back-end component, and *demultiplexing* the notifications generated by the back-end component to the front-ends. In this case, the use of an access policy becomes manifest: users can be given access to the communication channels according to their rights, or roles. Note that the

choice of letting the application components exchange only requests/notification messages makes the introduction of cooperative connectors safe.
Control components are in charge of implementing the coordination policies by defining the user roles and associating them with access rights [3]. According to these rights, commands to control the behavior of the connectors are issued. Figure 3 illustrates the logical channels handled by the cooperative connector for applications like the spreadsheet. Assume there are two groups of users with the roles of writers and readers. A control component implements the two groups of people and issues commands (e.g., "open input to user Ui") to the cooperative connector. In such a way, the cooperative connector simply reacts to commands issued by the control component. The control components, and consequently coordination policy, can be replaced without effecting the rest of the system.

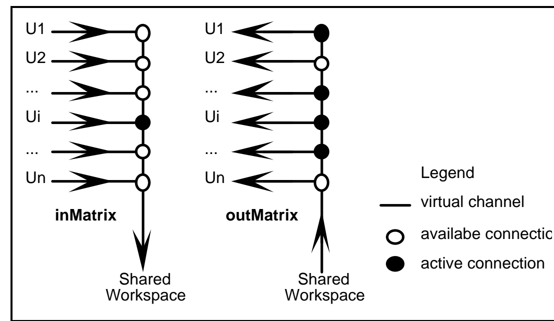

Figure 3: The logical channels of a cooperative connector.

4. Knowledge Management Systems

The experience described in this section deals with the ongoing project Klee&Co [11]. The projects goal is the design of a knowledge-management support system for design centers and other creative environments. Key feature of the system is the ability to provide information in a contextualized way.
The basic idea is to display an information (e.g., a document) surrounded by a context composed of related information like related documents, related persons, related knowledge areas and related e-mail messages [1]. To achieve the goal there is the need to collect, label and retrieve the knowledge of the designer in real-time with minor user involvement in collaborative design processes. The major concern is the capability of accessing the information in different ways according to the use context, and consequently the capability of retrieving the related information to be displayed. Unfortunately, languages or frameworks do not supply means to address the problem.

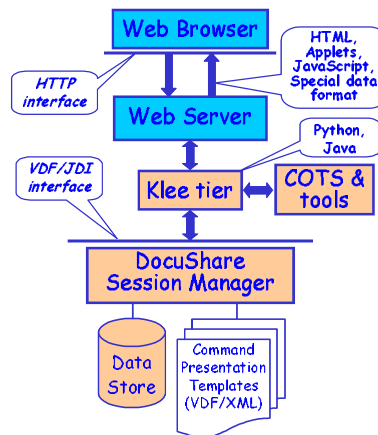

Figure 4: The architecture of the Klee&Co system.

The system development is based on the document management system DocuShare by Xerox. It comes with an XML-based API and a Java API that can be used to program the system to customize features. DocuShare is based on web technology, so it is natural to think of extending DocuShare by adding new components using existing tools, platforms and languages. Figure 4 illustrates the Klee&Co architecture. Unfortunately, the use different technologies to address new aspects of knowledge

management and presentation is not straightforward since they were not designed to accommodate different kinds of systems.
Moreover, some of the concerns are not easy to address because of conflicts. Examples of conflicting concerns are the capability of handling different kind of information like private information, community information and public information - which means to guarantee the right level of security but providing the right level of information access to make the system useful; the capability of tracking the user activity to store the design process and the related information without changing the existing practices - which means to include and extend the functionality of existing tools and data formats; the capability of providing data description to allow for personalized display - which means common data representation without assumptions on manipulating tools (in DocuShare the content representation is already based on XML to partially address this problem).
Solutions to effectively address these issues are still under investigation. Currently, we are developing prototypes based on open architecture that hosts specialized components. The development approach is to let the users evaluate the prototypes to collect information on both usability and functionality of the system. Once the set of requirements will be set, we plan to devise a framework to support the development of collaborative, knowledge-based applications in which information of different nature and format can be combined to provide the user for comprehensive presentation space and work space.

5. Discussion and Conclusions

As described in the previous sections, we have chosen to start from specific domains to address the issue of designing and implementing software according to the principle of separation of concerns. The lesson learned is that languages and frameworks should be based on basic constructs and assumptions with the slightest influence on specific concerns, or aspects. Constructs and libraries to address common concerns in typical applications should then augment such languages. Moreover, this approach makes designers free to extend languages and framework to address specific concerns in the cleanest way to fit special system requirements. The work to identify these basic assumptions and common concerns is still far to be completed.
In the described projects, a number of different languages and technologies have been adopted to achieve the goals. TDE platform was implemented by replacing the scheduler of a commercial real-time operating system [9] with a TDE dispatcher, while planners and agents were implemented in C. A set of macros was defined to supply the programmers with the basic constructs to address the concerns of setting up plans and designing agents [7] [8].
The cooperative system project was implemented by Unix technologies like C, X-11, Motif and TCP/IP to implement the application components and the communication part of the connectors. An important issue was to overcome limitations due to single-user assumptions of X-11 protocol and environment. To address the design and the implementation of the control components a specialized language named CSDL [3] was developed.
For the knowledge management development, the web technology is exploited, which means that several languages and platforms need to be used. Examples are proprietary APIs to program the document management system DocuShare; Pyton and Java to program CGI scripts and servlets; Java and JavaScript to program the user interface; general propose languages to implement tools. A problem we are facing is the need of writing tricky code to overcome some rigidity, and to address aspects not already addressed by languages/environments.
After these experiences, we plan to move forward looking for unifying models and platforms. Today, the design of a single language and framework to accommodate every possible concern is probably too ambitious, and probably unfeasible. What feasible is the classification of concerns and the identification of those that are compatible and influence each other. For example, consider the performance issue. When designing a system, it is fundamental to identify if it will be distributed or not. In fact, the way its components can communicate directly influences the system performance. As for the example discussed in [13], transferring a reference to an object or a copy of that object is meaningful in a distributed context, but it is of little, or at least different, impact in a centralized system. Therefore, a model for communication and data transfer is required, and a platform that implements that model is needed.
The HyperReal project taught us that even the execution model should not be part of the abstract machine of a language. It should be implemented as an independent feature that can be replaced as necessary. This is similar to the experience described in [13] where synchronization constructs were removed from Java to get a less intrusive kernel.
Moreover, the new language/framework should be designed to be able to host new features to address new concerns. For example, we learned from the cooperative system project that if messages exchanged by components do not depend each other, it possible to make a single-user application a

shared application, otherwise the message ordering affect the correctness of the system [5]. The drawback is that independent messages may affect the performance of the system. As often happen, there is not a unique, best solution. That is why an open framework to host tailored solutions is needed.
Currently we are working on experiments based on Java to deliver a framework that enables for programming component-based systems in which each component can address specific aspects as described along the paper.

References

[1] De Michelis G., De Paoli F., Pluchinotta C., and Susani M., "Weakly Augmented Reality: observing and designing the work-place of creative designers ", in *Proceedings of DARE 2000,* Denmark, April 14-16, 2000.

[2] DePaoli, F. and Tisato, F., “A Model for Real-Time Co-operation,” in *Proceedings of the Second European Conference on Computer-Supported Cooperative Work,* Amsterdam, September 25-27 1991, pp. 203-217.

[3] DePaoli, F. and Tisato, F., “CSDL: a Cooperative Systems Design Language.” *IEEE Transactions on Software Engineering*, vol. 20, no. 8, 1994.

[4] De Paoli F., Tisato F., Bellettini C., "HyperReal: A Modular Control Architecture for HRT Systems", *Euromicro Journal of System Architecture*, 1996.

[5] DePaoli F. and Sosio A., “Requirements for a layered software architecture supporting cooperative multi-user interaction”, in *Proceedings of the 18th International Conference on Software Engineering* (ICSE-18), IEEE, Berlin (Germany), March 25-29, 1996, 1996, pp. 408-417.

[6] DePaoli F., Tisato F., “On the complementary nature of event-driven and time-driven models”, *Control Engineering Practice - An IFAC Journal*, Elsevier Science, June, 1996.

[7] De Paoli F., “Specification of Basic Architectural Abstractions of TDE”, Esprit Project 20592 OMI/MODES TR 5.9.1 version 2, October 1997

[8] De Paoli F., Tisato F., Bellettini C., “TDE: A Time Driven Engine for predictable execution of realtime systems”, in Proceedings of the Workshop on Object-Oriented Technology and Real Time System - European Conference on Object-Oriented Programming, Brussels (Belgium), July 20-24, LNCS, Springer, 1998.

[9] EOS web site, http://www.etnoteam.com/EOS.

[10] Kiczales Gregor, Lamping John, Mendhekar Anurag, Maeda Chris, Videira Lopes Cristina, Loingtier Jean-Marc, Irwin John, "Aspect-Oriented Programming", in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[11] Klee&Co web site, http://www.kleeandco.org.

[12] Nissanke N., “Realtime systems”, Prentice Hall Series in Computer Science, ISBN 0-13-651274-7, Prentice Hall, 1997.

[13] Videira Lopes Cristina, Kiczales Gregor, "D: A Language Framework for Distributed Programming", Technical report SPL97-010, P9710047 Xerox Palo Alto Research Center. February 1997.