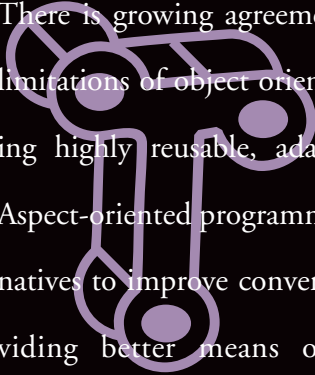




# ANALYZING THE ROLE OF ASPECTS IN SOFTWARE DESIGN

**How AOP technologies can promote better design practices is an essential issue concerning the potential benefits of the technology.**



There is growing agreement in the software community about the limitations of object orientation to cope with the problem of building highly reusable, adaptable, and extensible software systems. Aspect-oriented programming [5] is one of the most promising alternatives to improve conventional OO techniques. AOP aims at providing better means of addressing the well-known problem of separation of concerns [7] by using specialized mechanisms to encapsulate concerns whose behavior crosscuts essential application functionality. Several AOP approaches have been proposed in the literature, however, their application in practical cases is still uncertain. There are three basic approaches to addressing the process of separation of concerns: a linguistic view, a pure OO view, and an architecture-oriented view. Each of these involves several tradeoffs including aspect definition

**J. Andrés Díaz Pace and  
Marcelo R. Campo**

and weaving, performance, flexibility of aspect policies, and aspect evolution.

Besides the pros and cons of these alternatives, a central remaining question is what and how AOP technologies contribute to promoting good design practices. Realizing AOP's promises requires that such technologies also deal with the problems where objects have been traditionally successful. For this reason, we carried out a simulation case study to empirically compare both OO solutions against aspect-oriented ones, and aspect technologies against each other.

### Aspect-Related Technologies

The linguistic approach is based on the definition of a set of language constructs used to express aspects and their interactions. Relevant concerns identified at the problem domain are translated to aspectual constructs and later integrated with the functionality-decomposed program via well-defined interfaces. The final application is obtained by weaving the primary structure with the cross-cutting aspects. Figure 1 illustrates such a process. AspectJ (see [www.aspectj.org](http://www.aspectj.org)) and Hyper/J [8] are typical cases of this view. A strength of these languages is performance, but they tend to be limited regarding the facilities provided, because it is not always possible to know all the aspects that may come up in advance. Also, this model cannot easily deal with aspect evolution, because the mapped concerns are somewhat linked to fixed constructions. Furthermore, the integration of specific aspect languages with other tools or frameworks is not always possible or requires important integration efforts.

Within the OO view, two main alternatives for handling aspects are frameworks [4] and reflective architectures [6]. The framework view usually provides more flexible constructs than the language counterparts. A typical framework approach is shown in Figure 2. Concerns are now materialized as aspectual classes either at the framework level or at the user-application level, and developers can customize these aspects using the mechanisms supported by the framework. These types of frameworks are known as AO frameworks, a typical example of which is the Aspect-Moderator framework [2]. There is a subtle difference between AO



frameworks and application frameworks. An AO framework explicitly engineers concerns (aspects), whereas a traditional framework implements some abstractions for a given domain, and perhaps, additionally promotes good separation of concerns. An advantage of frameworks is that they can be combined with other frameworks or systems in order to produce larger systems.

Reflective architectures incorporate structures for self-representation, so that the behavior of the base-level objects can be enhanced and adapted by the metalevel objects according to specific requirements [6]. Regarding AOP, reflective techniques permit a clear separation of concerns as the aspects are dealt with at the metalevel while the basic functional components reside at the base level.

Finally, the pure architecture-oriented approach (see Figure 3) proposes an early identification of concerns using architectural organizational models, which can be later mapped to aspects through different implementation technologies. Note that object orientation is just a possible option for this materialization. An architectural viewpoint involves a higher level of abstraction than the previous approaches. It typically comprises two stages. First, developers should determine the problem architecture, that is, an architecture representing the underlying organization of the software to build and the tradeoffs imposed by nonfunctional requirements and architectural styles [1]. Unlike the previous approaches, concerns are initially mapped to architectural constructs, instead of coding them using framework or language constructs. Then, as a result of this general description of relevant concerns, the approach enables several kinds of aspect materializations through different frameworks, whereas these frameworks retain the properties inherited from the original architecture.

### A Case Study Comparing AOP Technologies

In order to evaluate the aforementioned AOP approaches, we undertook an empirical analysis based on a simulation case study of the temperature control of a building. Very broadly, this case study comprises a building with rooms requiring specific temperatures and a network consisting of radiators, pipes, and a boiler. The boiler is the source of heat, and it generates a heat flow that is distributed

Figure 1. A language-based approach.

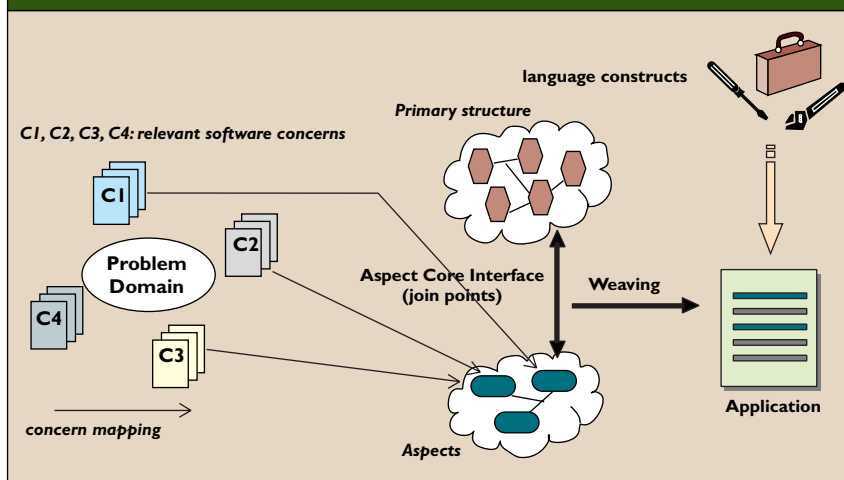


Figure 2. A framework-based approach.

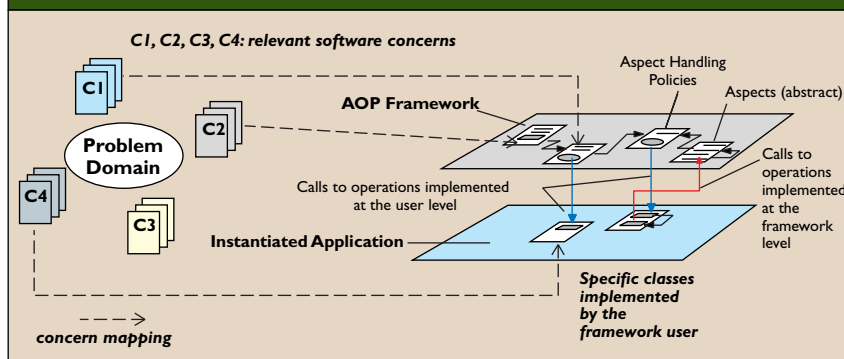
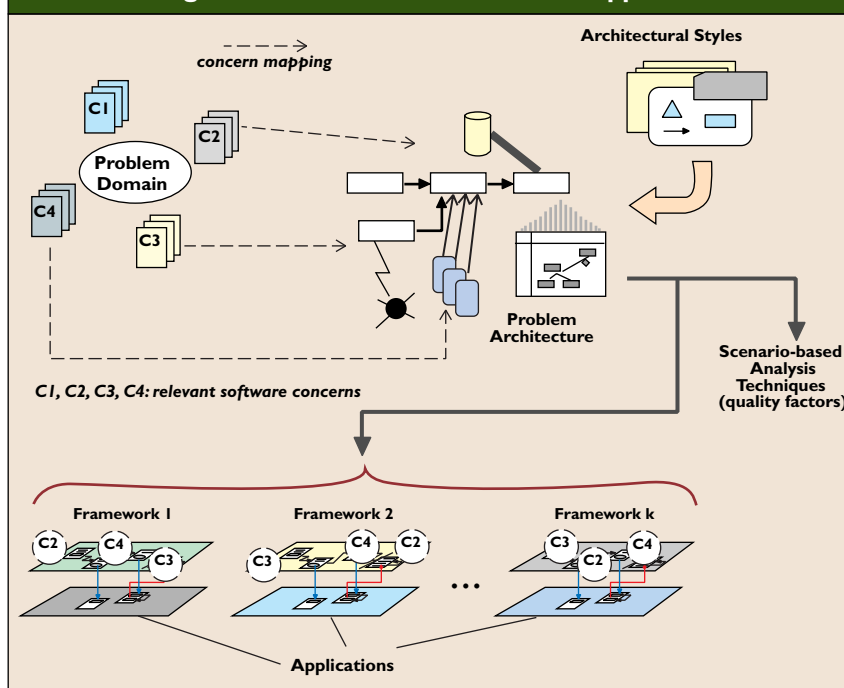


Figure 3. An architecture-oriented approach.



through the circuit. Thus, each radiator receives this heat and transfers a part to the rooms until they reach their predetermined temperatures. Figure 4 shows a diagram of a temperature control system (TCS). To regulate the heat flow between the radiator and the room, each radiator is associated with a special valve that can be adjusted by the control system. The user can set the desired temperature for a room, and when the current room temperature is lower than this value, a control action is activated asking for a given amount of heat to the boiler.

To simulate TCS, a simple mathematical model of temporal differential equations specifies the heat flow among the different components. Each simulation component performs its own computations collaborating in the evolution of the simulation running in independent threads. Although the real system is somewhat more complex, this simplified example is interesting enough to illustrate practical evidence of situations requiring aspectual treatment. In the simulation, aspects related to mathematical models, concurrency, scheduling, and optimizations appear scattered across the system. Table 1 summarizes the relevant aspects identified in TCS and their relationships. All these aspects appear combined and interrelated, which makes the TCS implementation tangled, difficult to understand, and more important, difficult to maintain and evolve.

The study involved four different groups of programmers implementing, in Java, the case study according to the AOP alternatives mentioned previously. The first group imple-

**Table 1. Relevant aspects in TCS.**

Aspect	Description	Relationships with other Aspects
Mathematical modeling	It models all the functionality with equations governing the simulation and with the numerical methods for the resolution of these equations. Generally, equations are expressed as relationships between some state variables belonging to the simulation entities.	It should be kept separated from the real-world mapping aspect in order to avoid tangling problems. In this way, a given mathematical model could be modified with minimal impact on the application. Synchronization and scheduling aspects may also affect this aspect.
Real-world mapping	It directly models TCS elements as independent entities. By doing this well, it should reduce the semantic gap between reality and simulated entities and promote reusability by composition of components.	There are some relationships with synchronization aspects, and it also affects the mathematical modeling.
Scheduling	It refers to how the simulated entities should run.	It works together with synchronization.
Synchronization	It basically involves access to shared variables and race conditions.	It works together with scheduling.

**Table 2. TCS evaluation according to the different implementations.**

Analyzed Features	Standard OO Modeling	Architecture-Oriented (Bubble)	Aspects (TaxonomyAop and AspectJ)
Problem modeling	The system is decomposed on the basis of classes (or objects).	Description of the system in terms of autonomous entities (modules) and events.	The system is viewed as conventional objects plus aspects plugged into the primary structure.
Mathematical concerns	Numerical methods and mathematical computations are hard-coded and mixed with other components.	Mathematical matters are encapsulated in tasks assigned to entities, but they still remain somehow tangled.	Mathematical decisions reside at the aspect level, so they only interact with the primary structure via join points.
Scheduling and synchronization	Scheduling and synchronization are scattered across the components requiring these facilities (inheritance anomalies).	Synchronization and scheduling are provided as built-in features in the framework. It may be difficult to customize some of these policies.	There are specific aspects dealing with synchronization and scheduling. They are kept separated, but some problems about the way they interact (aspect composition) may arise.
Reusability	Reusability is limited to the usual object mechanisms such as inheritance, object composition, and some design patterns.	The framework allows for reusing design, using techniques such as design patterns, abstract methods, hook methods and template methods.	Reusability refers mainly to the separation between the simulation components themselves and the gluing mechanisms to make the simulation works.
Adaptability	Adaptability is quite low.	The event mechanisms supported by the framework provide a high degree of adaptability to the system.	Different aspects can be plugged into or unplugged from the primary components. AspectJ provides some constructors for dealing with unexpected changes (introduction clauses).
System organization	Conventional OO structure.	The application is built on top of an OO framework. Additionally, there is an architectural model underlying the framework.	One of the implementations uses a reflective AO framework. The other one is based on an aspect language.
Performance	Performance is very reasonable.	Some built-in features of the framework can produce performance overheads.	Performance is almost comparable with the standard implementation.

mented TCS under a typical OO view. Then, two other groups moved this design toward an aspect perspective, and modeled TCS using both an aspect language (AspectJ) and a reflective AO framework known as TaxonomyAop. Finally, the fourth group implemented TCS on top of an event-based architectural framework named Bubble. Table 2 summarizes the findings regarding the different TCS implementations.

For the analysis, we selected a number of quality factors such as performance, complexity, tangling reduction, reusability and adaptability, among others, that were applied to the TCS implementations. The metrics associated with these factors intend to give developers information about the OO structure of the design and code and provide guidelines for interpreting the values.<sup>1</sup> Here, we discuss a sample of the analyzed factors; a more complete description of this study can be found in [3].

We made several executions of the simulation programs, the performance of which is described in Table 3. The results were very similar, except in the case of the TaxonomyAop framework, where its figures were approximately three times the figures obtained with the rest of the implementations. The similarities found in the study seem to indicate that both the Bubble-based implementation and the implementation using AspectJ run almost like standard code. However, this consideration may depend on the particular problem.

<sup>1</sup>Indeed, it is important to agree on that any single metric has somewhat limited utility and developers should reason about all metrics in context. The metrics described are valid as preliminary indicators, but they should not be considered as fully accurate indicators because they can be biased by factors such as programming style, language features, or problem domain.

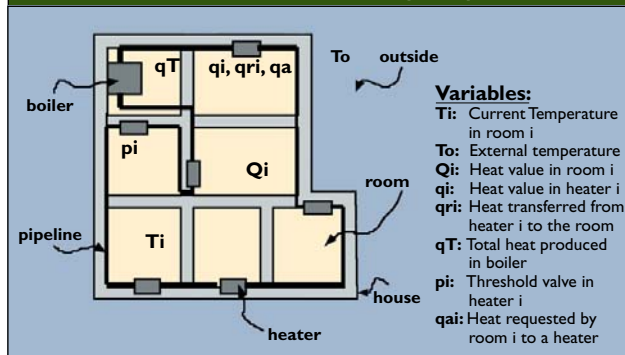
**Table 3. TCS performance with the different implementations.**

Performance	Standard (mseg) (*)	Bubble (%)	TaxonomyAop (%)	Aspectj (%)
100 Iterations	54762.00	100.93	330.91	101.15
1000 Iterations	515420.66	100.85	348.17	100.47
2000 Iterations	1024309.66	100.22	348.41	100.06
5000 Iterations	2539091.00	100.34	354.17	99.94

(\*) CPU time is expressed in milliseconds and represents average values

Simulation executions ranged from 100 to 5,000 thread iterations, on a single PC Pentium III. The configuration of the simulation consisted of 21 threads, corresponding to 4 rooms, 7 radiators assigned to those rooms, 8 pipes connecting the radiators, a boiler, and an environment.

**Figure 4. Diagram of the temperature control simulation (TCS).**



**Table 4. TCS complexity with the different implementations.**

Complexity	Standard	Bubble (*)	TaxonomyAop (*)	Aspectj
Classes	7.00	21.00	15.00	14.00
Methods	143.00	263.00	232.00	213.00
NCSS	969.00	1605.00	1486.00	1354.00
Methods per class	20.43	12.52	15.21	15.21
NCSS per class	130.86	72.00	92.07	92.07
NCSS per method	5.85	5.15	5.63	5.63
Average CCN per method	2.64	2.32	2.51	2.51

(\*) It does not take into account the methods and classes of the frameworks, but just those new methods and classes involved in the TCS instantiations

Aspects were considered as normal classes (or objects) plus some additional mechanisms to define their associations with regular objects, so that the metrics applied to object code applied to aspect code as well.

To obtain a measure of the complexity of the implementations, we gathered code statistics (NCSS) about number of classes and methods, number of sentences per class and per method, methods per class, and cyclomatic complexity (CCN). These results are shown in Table 4. It must be noted that the apparent superiority of the framework-based approaches regarding the cyclomatic complexity values hides the fact that most of the implemented methods rely on the built-in features provided by the frameworks. These low values mostly correspond to decisions deferred by the applications to the frameworks. However, this means the applications have a relatively low complexity.

When it comes to reusability and adaptability, we used a scenario-based approach for software architecture analysis [1]. Reusability is defined as the

ability of software components to serve for the construction of many different applications. Adaptability, in turn, defines the ability of a given software system to cope smoothly with changes in the problem specification, producing a low impact on components previously implemented. Broadly speaking, scenarios are

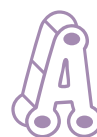
narratives that describe the use cases of the system, and can be used to capture either the system functionality or to detect possible flaws requiring potential enhancements. To give an idea of this process, Table 5 shows a possible scenario evaluation of TCS.

The scenario described in Table 5 appears overlapped with other scenarios. For example, the addition and deletion of new simulation components necessarily affects the global mathematical model of the simulation. Hence, those modifications will produce further changes in the target scenario.

### The Impact of AOP Technologies in Overall Design Complexity

From this partial study, we observed that the overall complexity of the applications was not markedly reduced or enhanced. The facts shown in Table 4, where the number of methods per class, the NCSS per method, and the CCN per method figures are low, seem to support this argument. This effect can be due to the balance

between the simplicity resulting of keeping concerns separated and the difficulty of reasoning about these concerns to integrate a subset of them into a coherent system. These two forces, and especially the concern integration force, cooperate, so that the overall system complexity remains almost unchanged. In particular, the best results were obtained with the Bubble framework. We argue that this payoff comes mainly from the autonomy of components and decoupling prescribed by the framework.



Another topic to analyze, not covered with the previous indicators, is the one of the interactions between aspects and primary components. There are some experimental results [9] highlighting the importance of the

**Table 5. A typical scenario-based analysis for reusability and adaptability.**

Scenario Description				
Changes to the mathematical model. Developers would need to upgrade the mathematical model, either to include new equations or new bindings among their variables. Other issues that can vary are the numerical methods used to solve the equation set. A more radical change may involve changing the underlying model to a discrete-event model.				
<b>Architecture-Oriented (Bubble)</b>	The architecture-oriented implementation locates the simulation equations in tasks, but the framework itself provides little support for new equations or variable bindings. Therefore, all these modifications are up to the developer's criteria. For instance, some of these decisions include deciding which tasks should implement the equations, how to assign them, or which variables to compute. Furthermore, the numerical methods to solve the equations have to be hard coded along with the equations, which complicates the modifiability of the concern. An alternative design consists of adding a blackboard component responsible for these computations, in order to move mathematical matters from tasks to a centralized repository. Although the framework can incorporate this blackboard with minimal effort, this option may introduce some bottlenecks into the system. Other techniques, typically discrete-event-based simulation, are feasible to model with the framework. For example, in this case, the event mechanisms of the framework can be extended to implement scheduling components and the tasks can be seen as actions bounded to events.			
<b>Aspect-Oriented (Taxonomy-Aop and Aspectj)</b>	When we move to the aspect-oriented implementations, we have other point of view for the scenario. Aspects permit encapsulating the equations and the variable bindings, keeping them separated (in aspect modules) from the functionally decomposed program. Developers simply need to define how these aspects are associated with the simulation components (using associations between base and metaobjects in TaxonomyAop, or point-cuts plus advice clauses in Aspectj). In this way, when a change arises (new equations, bindings, or numerical methods), it is somehow confined to the mathematical aspects and it loosely affects the rest of the components. These aspects work like small blackboards. The only disadvantage with this design is that the interaction between aspects and functional components can sometimes be quite complex, producing an additional complexity to understanding and debugging of the system. On the other hand, regarding other simulation techniques, it is not easy to clearly determine how discrete-event simulation should be implemented with aspects.			
Evaluation Summary				
Quality Factors	Standard	Bubble	TaxonomyAop	Aspectj
<b>Reusability</b>	Low	Low	Medium	Medium
<b>Adaptability</b>	Low	Medium-to-High	Medium	Medium

aspect-core interface in achieving development benefits with AOP. These results show, among other conclusions, that the separation provided by aspect orientation is most helpful when the interface is narrow, that is, in a well-localized scope. These comments were applicable to our implementations as well. We observed during the development of the aspectual implementations that synchronization and scheduling were more understandable (and therefore required less debugging) than the mathematical aspects. In the latter case, the interaction between aspects and simulation components presented a wide interface that increased in some degree the complexity of the application itself and also affected the impact of changes. The analyzed perspectives showed very brittle results about mathematical issues.

Certain differences concerning reuse and adaptability facilities can also be noticed from the scenario evaluation. The Bubble-based implementation had limited reuse advantages compared with the OO implementation, whereas the aspectual implementations presented a moderate score. If

designing for reusability means that the system should be structured so its components can be chosen from previously built products and integrated in larger systems, we note that AOP technologies do not focus specifically on this matter. Rather they help maintenance by controlling change. Regarding adaptability, the Bubble-based implementation showed very good results, strongly related to the underlying architecture of the framework. Conversely, the aspectual implementations provided adaptability in a more restricted sense, one that just refers to the incremental inclusion of aspects into the system plus some mechanisms to partially enable or disable their capabilities.

The experiments show the application of AOP technologies has no significant disadvantages in performance. In fact, two of the implementations ran almost identical to the standard version of TCS. The only exception was, obviously, the reflective implementation. This strength in performance is mainly attributed to language-based approaches, but framework-based approaches can also provide a quite reasonable performance.

Development efforts were not magnified at the implementation level. Instead, the major amount of work was concentrated at design levels. Most of the time was devoted to engineering the interactions between relevant concerns and the primary structure of the application. We observed the development of the aspectual examples consumed approximately half the time to understand and model the problem itself and the other half to adapt the system to the aspectual characteristics. Conversely with the Bubble-based implementation, the group spent more time modeling the problem and fitting it into the framework architecture. After that, the adaptability was mostly straightforward.

The framework-based implementations showed that this approach usually favors some concerns within the framework organization and neglects others. Thus, it is sometimes difficult for application developers to select the right concerns of the framework without dealing at the same time with

other undesired concerns. Moreover, these unfavored concerns often produce poor flexibility. We observed this situation in the Bubble framework, where the relevant concerns were incorporated implicitly into its organization instead of defining them as aspects.

## Conclusion

The claims about advantages and disadvantages of aspect technologies are quite broad. We see that the central problem of aspect technologies, whatever the approach we consider, is not just about cross-cutting or separation of concerns, but it involves deeper research about how to understand a number of software parts as separated artifacts and then integrate some of them into a coherent system.

This situation also bears the issue of locality of changes, because the more interactions with other components (or aspects) the developer has to know in order to understand the system, the more complex the maintenance of this software results. Particularly, the best results were obtained with the Bubble framework.

Here, we have performed a comparative study exploring the possibilities of AOP technologies. From the results, two observations can be extracted:

- AOP technologies make it easier to write and change certain kinds of concerns. However, other concerns have to be matched with the aspect constructs. We observed that the two aspectual implementations assumed from the beginning a separation between simulation components and aspects, but sometimes this separation was quite premature in the sense there was little guidance about how to decide the design of the applications.
- The underlying design of aspectual applications is more important than the mechanisms provided by the AOP technologies, no matter how sophisticated. Note that we are not discussing the value of existent technologies. Rather, we stress that good separation of concerns is ultimately enforced by architectural means. Technologies are useful in this context, as they promote good design practices.

Given these results, the architecture-oriented approach proposes, at its essence, reasoning about

relevant aspects at the very conception of the system architecture. By expressing aspects in an architectural model we can ensure these quality factors will also affect the chosen aspects. Using this architectural guidance, we can later map a given architecture (and its relevant concerns) to different types of framework organizations addressing different needs, while retaining the characteristics inherited from the architecture into this materialization. At this point, a linguistic approach could be useful to facilitate the implementation of good design practices. Therefore, this discussion connects with the old, but never out-of-date, arguments by Parnas [7] concerning the value of early design decisions in any software development project. **C**

## REFERENCES

1. Bass, L., Clement, P., and Kazman, R. *Software Architecture in Practice*. Addison-Wesley, 1998.
2. Constantinides, C., Bader, A., Elrad, T., and Fayad, M. Designing an aspect-oriented framework. *Computing Surveys* 32, 41 (2000).
3. Díaz Pace, A. *An Empirical Study about Separation of Concerns Approaches*. Tech. Rep. RR-001-2001, ISISTAN Research Institute, UNICEN University, 2001.
4. Fayad, M., Schmidt, D., and Johnson, R., Eds. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley, 1999.
5. Kiczales, G., Lamping, J., Mendhekar, J., Maeda, C., Videira Lopes, C., Loingtier, J., and Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag, 1997.
6. Maes, P. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22, 12 1987.
7. Parnas, D. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972).
8. Tarr, P. and Ossher, H. *HyperJ User and Installation Manual*. HyperJ homepage; [www.research.ibm.com/hyperspace/HyperJ](http://www.research.ibm.com/hyperspace/HyperJ).
9. Walker, R., Baniassad, E., and Murphy, G. *An Initial Assessment of Aspect-Oriented Programming*. Tech. Rep. TR-98-12, University of British Columbia, 1998.

---

**MARCELO R. CAMPO** ([mcampo@exa.unicen.edu.ar](mailto:mcampo@exa.unicen.edu.ar)) is a professor in the Computer Science Department and head of the ISISTAN Research Institute of the UNICEN University at Tandil, Buenos Aires, Argentina.

**J. ANDRÉS DÍAZ PACE** ([adiaz@exa.unicen.edu.ar](mailto:adiaz@exa.unicen.edu.ar)) is an assistant professor in the Computer Science Department and the ISISTAN Research Institute of the UNICEN University at Tandil, Buenos Aires, Argentina.

---

This work was partially supported by the CONICET research agency.

---