# Aspect Orientation Enables Differentiation with Software

Piërre van de Laar (pierre.van.de.laar@esi.nl)
TU/e Campus, Laplace-building 0.10
Den Dolech 2
P.O.Box 513, 5600 MB Eindhoven
The Netherlands

Companies develop large numbers of embedded systems that contain a significant amount of software. The amount of software in these embedded systems is exponentially growing according to Moore's law. With the increase of software also the complexity increases. Separation of concerns, i.e., the ability to deal with the difficulties, the obligations, the desires, and the constraints one by one [Dijkstra, 1976], is needed to cope with this growing complexity and to ensure that companies can continue to differentiate with software.

Currently, embedded software is modularized based on functionality. Unfortunately, this kind of modularization cannot separate all concerns as can be observed in the current software:

- Many (non-functional) concerns are not localised in one software module but are scattered throughout the software.
- Multiple concerns are tangled in one software module.

Since concerns are not separated, complexity increases; independence decreases; and decision points must be preponed. Throughout the whole software development process, the impact of a limited separation of concerns is noticeable:

- Traceability and localisation of requirements is reduced.
- Independent (multi-site) development and the associated integration and validation is prevented.
- During implementation, code is duplicated which not only is the root cause of errors due to inconsistencies, but also prevents specialisation and wastes scarce and limited [Kaashoek, 2005] developer resources.
- Maintenance becomes more difficult.
- Evolution and reuse within a product family is hampered.

To solve these problems, we ask ourselves the question: how can we improve the effectiveness of our modularization?

Aspect orientation is a technology that improves the effectiveness of modularization. Aspect orientation modularizes the system based on concerns. In 1978 [Sandewall, 1978] the principles of aspect orientation were described for the first time. Yet, it became hot due to Xerox [Kiczales *et al.*, 1997], who applied aspect orientation amongst others for image improvements. Currently, on top of every popular programming language an aspect oriented programming language exists. For example,

- AspectC++,
- AspectC,

- AspectSharp, and
- AspectJ.

The interest for aspect orientation is not limited to these Open Source projects, also Microsoft is investigating it for their Developer Studio[2]. We have investigated aspect orientation in the scope of hybrid (analog and digital) television.

In the remainder of this extended abstract, we will first give a black and white picture of aspect orientation. For a more in depth description, we recommend [Kiczales et al., 1997, Elrad et al., 2001, Laddad, 2003, Filman et al., 2005] to the interested reader. We will then describe how we added aspects to the component-based software of television, and share our experiences with applying aspect orientation in this context. We will end with a summary.

## *What is aspect orientation?*

Aspect orientation introduces join points around the execution of instructions to handle concerns related to these instructions. Join points are also the only points where aspects can interact with other pieces of software. To give an example, join points around instructions that change items in a database for a user, enable that:

- Before the instructions are executed, the access to the database is logged;
- The instructions are only executed when the user has the rights to modify the items in the database; and
- After execution of the instructions, all observers of the database are notified to ensure accurate visualizations.

An aspect contains pointcuts and advices. A pointcut specifies, by selecting join points, where an aspect crosscuts other aspects. Join points can be selected based on, amongst others, the type and name of functions and its parameters. An advice specifies in a function-like construct what behaviour to exhibit around the selected join points. For the implementation of an advice, an aspect may require functionality of other aspects, use meta-data about the selected join point, and introduce variables.

Making a product from aspects is called weaving: joining the aspects at the selected join points. Weaving can occur at different points in time. To give a few examples: Before compile time by code weaving, at load-time by the class loader, or at run-time by the virtual machine.

## *Adding aspect orientation to component-based software*

Component-based software has besides source and binary/byte code also an architectural description. We decided to weave based on these architectural descriptions to leverage the following advantages:

1. The architectural description contains information, some of which is lost in the source code. For example, since the C programming language has no interface concept, the information of which functions constitute an interface is lost. Similarly, the direction of parameters of functions is lost in C.
2. The source code of a component is often not available, while the architectural description is always available. But even when source code is available,

[2] For more info, see http://research.microsoft.com/workshops/aop/.

weaving at source code level typically invalidates the warranty and support of components.
3. The architectural description language is implementation-language agnostic, which makes the weaving implementation-independent.
4. The sensitivity of the system for modifications at architecture level is by design less than at the source code level. Computations that cross component boundaries must be able to handle the allowed variations in the implementation of interfaces and are thus less sensitive for modification compared to computations within a component that typically exploit implementation details to optimize throughput and response time.
5. The architectural description has a higher abstraction level and is more stable than the implementation; this positively influences the independent evolution of aspects and components.

Which components can be affected by an aspect? Even though the composition of a component is implementation dependent, we decided that an aspect could affect all components in a product. This choice enables more powerful aspects, which are needed, amongst others, for logging all components, and asserting that all components are only used after initialisation.

What are the join points in an architectural description? We consider the functions in the interface of a component as join points, since:

- A component only communicates via these functions.
- Developers explicitly describe both the functions in an interface and the interfaces of a component.
- Only these functions are implementation-independent.

## *Experiences with aspect orientation*

We first gained experience with and confidence in aspect orientation in the validation and verification phase. This path ensured that we reduced the risks associated with our ultimate goal: The introduction of aspect orientation into our products.

How to handle access before initialisation is a concern that affects all components. Although one can easily describe how to handle access before initialisation in general, it is currently handled per component. Even worse, this handling differs between components in the same software stack. With aspect orientation, we were able to write three different strategies to handle access before initialisation. The first strategy asserts that a component is not accessed before initialisation; the second strategy ignores accesses when the component is not yet initialised; and the third strategy calls the initialisation code when the component is accessed but not initialised before. With these strategies:

1. We could ensure that all components handle access before initialisation identically.
2. We could separate the initialisation implementation from the functional implementation. This not only reduces the lines of code by 2%, but also makes reuse more likely. Reuse becomes more likely since the reuse environment has only to match either the initialisation requirement (to reuse one of the three initialisation aspects) or the functional requirement (to reuse one of the components), but not both.
3. We could postpone the decision for an initialisation strategy from implementation to integration.

Resource usage is an important concern for resource limited systems. The functionality to check that a component does not use more resources than specified can be localised in one component. Yet, for each component under test one still has to do a lot of plumbing:

- Instantiate a test component, and
- Change the connections to the component under test to pass through this test component.

With aspect orientation, we were able to localise not only the functionality but also the plumbing in one aspect. This made the test process both easier and less error-prone.

Many pieces of software cannot be accessed multithreaded, but accidentally are accessed on multiple threads. Integration and testing would benefit from automatic detection of illegal multithreaded accesses. We have written an aspect that lists multithreaded accesses throughout the complete software stack. This list can help architects to pinpoint illegal multithreaded accesses. Of course, by adding attributes to the current code base and exploiting this information in a comparable aspect, also this last step can be automated [Hoogendijk *et al.*, 2005].

During integration and testing, understanding the dynamic behaviour is crucial. Tracing provides insight in this behaviour. We have written and applied an aspect to trace the interface function calls in an already finished television set. While manually adding trace statements requires programming effort linear with the number of interface function calls, this aspect required only a small programming effort that is independent of the number of interface function calls. Currently, we are using this aspect at NXP, the former Philips Semiconductors, to determine how the platform is accessed by the applications running on top of it.

## Summary

Aspect orientation improves our effectiveness to modularize software. As a consequence, separation of concerns is better supported. This reduces the complexity of the software, minimizes dependencies, prevents duplication, ensures consistency, makes reuse more likely, and enables to postpone design decisions. Our experience indicates that aspect orientation scales to industrial applications. Furthermore, with aspect orientation, companies will be able to continue the differentiation with software.

## References

[Dijkstra, 1976] Edsger W. Dijkstra, 1976, *A Discipline of Programming*, ISBN 0613924118.

[Sandewall, 1978] Erik Sandewall, 1978, *Programming in an Interactive Environment: the "LISP" Experience*, Computing Surveys, Vol. 10, No. 1, pp. 35-71.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, *Aspect-Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming, pp. 220-242. Available at http://www.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/for-web.pdf

[Elrad *et al.*, 2001] Elrad, T., Filman, R.E., Bader, A., October 2001. Special section on Aspect-Oriented Programming.Communications of the ACM, Vol. 44 No. 10, pp. 29-97.

[Laddad, 2003] Laddad, R., 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning, ISBN 1-930-11093-6.

[Filman *et al.*, 2005] Filman, R.E., Elrad, T., Clarke, S., Aksit, M., 2005. *Aspect-Oriented Software Development*, Addison-Wesley, ISBN 0-321-21976-7.

[Hoogendijk *et al.*, 2005] Paul Hoogendijk, Chritiene Aarts, Piërre van de Laar, Felix Ogg, Rob van Ommering, Jur Pauw, *Extending the Nexperia Home Component Model: Annotating and Checking Thread-safety Properties*, Philips Software Conference 2005.