Technical Note PR-TN 2006/00648

Issued: 08/2006

# Combining component-based and aspect-oriented software development in a resource constrained environment

P.J.L.J. van de Laar

Philips Research Europe

Authors' address P.J.L.J. van de Laar WDC21 pierre.van.de.laar@philips.com

**Title:** Combining component-based and aspect-oriented software development in a resource constrained environment

**Author(s):** P.J.L.J. van de Laar

**Reviewer(s):** Golsteijn, R., IPS Facilities, Ommering, R.C. van

**Technical Note:** PR-TN 2006/00648

**Additional Numbers:**

**Subcategory:**

**Project:** TRADER (2004-119)

**Customer:**

**Keywords:** aspect-oriented programming, resource constraints, component-based development, industrial experience

**Abstract:** In this article, we combine component-based and aspect-oriented software development using the component model's architecture description language. Our approach is applicable to any component model that provides the architectural descriptions, directly or indirectly, e.g., via a Code Document Object Model or via reflection. Unlike the approaches of Suvée et al. [2003], Lafferty and Cahill [2003], and Noguera [2003], our approach can be applied in environments where resources are limited. We illustrate this using an existing component-based software stack that uses Koala [van Ommering et al., 2000, van Ommering, 2004]: a component model designed for resource-constrained environments. We experienced that combining component-based and aspect-oriented software development was beneficial because we could increase the modularity of our software.

 

 

# Contents



 

 

# 1. Introduction

Philips is one of the largest television manufacturers in the world. Fierce competition in the television market is leading to smaller profit margins, price erosion, shorter time to market, and a battle for shelf space. To remain competitive, we must minimize the bill of material and the cost of system development. Minimizing the bill of material puts constraint on the resources of a television, such as memory, bandwidth, CPU cycles, and footprint. We minimize the cost of system development by modularization, which allows:

- Parallel development of the modules at multiple sites,
- Specialisation on individual modules,
- Buying modules from third parties, and
- Mixing and reusing modules in different products of the same product family.

When these benefits outweigh the cost to compose the modules, modularization results in lower costs, higher profits, shorter time to market, and a larger variety to fill shelf space. Yet, one question remains: How can a system be effectively modularised? According to Parnas [1972] the effectiveness of a "modularization" is dependent on the criteria used in dividing the system into modules.

In a component-based design, the system is modularised based on functionality: related functionality is localised in the same component. A component is a reusable module that provides and requires functionality. A component has interfaces that channel the interactions with its environment. Composition by a third party encompasses instantiating components and connecting their interfaces. Components and interfaces are described in the component model's architecture description language.

Philips applies component-based software development for televisions using the component model Koala [van Ommering et al., 2000, van Ommering, 2004]. In the television software stack, we made two observations with respect to the effectiveness of component-based modularization:

1. Many (non-functional) concerns are not localised in one component but are scattered over multiple components, and
2. Multiple concerns are tangled in one component.

Given these observed limitations of component-based modularization, we are looking for alternatives to more effectively modularize our software.

In an aspect-oriented design, the system is modularised based on concerns: each concern is localised in one aspect. An aspect is a module that specifies not only what behaviour the concern exhibits but also where the concern crosscuts other concerns. An aspect has predefined join points that control where aspects can crosscut. Composition by a third party encompasses selecting aspects and weaving them together at the join points. Many [Elrad et al., 2001, Laddad, 2003a, Filman et al., 2005] consider aspects as the next step in modularization, after procedures, data structures, objects, and components.

Theoretically, an aspect-oriented design has only one type of module: an aspect [Ossher and Tarr, 2001]. Yet, in practice, a large amount of software already exists which is

 

modularised based on functionality, and aspect-oriented design is treated as an extension added to this dominant modularization dimension. Suvée et al. [2003], Lafferty and Cahill [2003], and Noguera [2003] modularised the system by combining component-based and aspect-oriented software development based on both functionality and concerns. Unfortunately, their approaches are "unsuitable in environments where resources are limited" [Suvée et al., 2003]. We propose a combination of component-based and aspect-oriented software development that is applicable in a resource-constrained environment. We illustrate this using our existing component-based software stack and prove the increased effectiveness in modularization.

The article is organised as follows. First, we explain the Koala component model that we use in our component-based software development process and aspect-oriented software development in general. Second, we discuss options that are available to combine component-based and aspect-oriented software development. Third, we describe our approach to combine component-based and aspect-oriented software development and illustrate it with a simple example. We end with a brief summary.

 

# 2. Background

## 2.1. Component-based software development using Koala

Koala is a component model designed for component-based software development for resource-constrained environments such as televisions. Koala has an architecture description language to specify interfaces and components. An interface consists of a set of related functions. See Figure 1 for an example. A component has an external view: its provided and required interfaces; and an internal view[1]: the composition described by a list of components and the connections between their interfaces. See Figure 2 and 3 for a component description and its visualization, respectively. Visualizations like Figure 3 are key in discussions on the architecture of our systems. The component and interface descriptions are used to build products by generating make and header files; and to verify that the implementations adhere to the architecture.

```
interface IApe                          [ape]
{
        void Foo (int bar [in]);
        int Thud (char * grunt [out]);
}
```

*Figure 1. Example of an interface description in Koala.*

[1] Not all component models describe their internal view explicitly in the architecture description language.

 

```
component Composition {
        provides
                IApe    ape;
                INut    nut;

        requires
                IAbc    abc;
                IXyx    xyz;

        contains
                component     CFoo   foo;
                component     CBar   bar;

        connects
                foo.abc = abc;
                foo.xyz = xyz;

                bar.thud = foo.thud;
                bar.xyz = xyz;

                ape = foo.ape;
                nut = bar.nut;
}
```

*Figure 2. Example of a component description in Koala.*

 

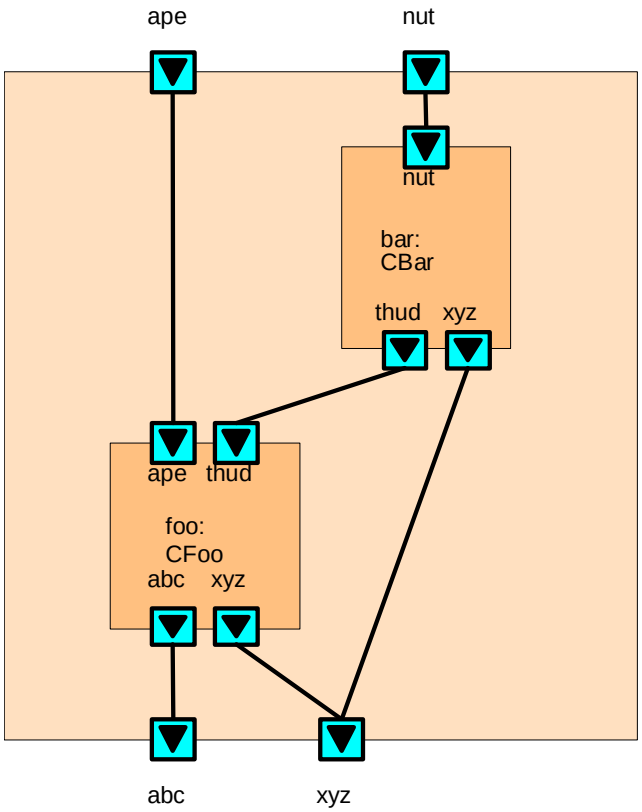


*Figure 3. Visualization of the component described in Figure 2.*

Koala has many similarities with other component models, such as OMG's CORBA component model [Siegel, 2000] and Microsoft's COM [Rogerson, 1997]. In Koala:

- Developers specify interfaces and components in an architecture description language.
- Components can only interact with each other via interfaces.
- A component provides functionality for which it may require functionality from its environment.
- A component refers to its interfaces by their instance names, and thus can provide and require multiple interfaces of the same type.
- A third party instantiates components and connects their interfaces.

Koala differs at one point fundamentally from the other component models: Koala fixes the composition at compile time, earlier than any other component model. Hence, Koala can instantiate all components, connect their interfaces, and even remove unused functionality without introducing any run-time overhead and increasing memory footprint. This behaviour is crucial for resource-constrained environments. Exactly for this reason, Philips developed Koala instead of using standard component models, as mentioned above.

 

## 2.2. Aspect-oriented software development

Aspect-oriented software development is a methodology to handle multiple concerns. For this article, we give only a black and white picture of aspect-oriented programming; for a more in depth description, see [Kiczales et al., 1997, Elrad et al., 2001, Laddad, 2003b, Filman et al., 2005].

Aspect-oriented programming introduces join points around the execution of instructions to handle concerns related to these instructions. For example, join points around instructions that change items in a database for a user, enable that:

- Before the instructions are executed it is verified that the user has the rights to modify the items in the database;
- The instructions are executed as a transaction to ensure that the database remains in a valid state; and
- After execution of the instructions, all observers of the database are notified to ensure accurate visualizations.

An aspect exposes its join points explicitly or implicitly based on the usage of particular language constructs. Yet, in both cases an aspect should expose only implementation-independent join points for understandability and stability. For this reason, an aspect in AspectJ [Laddad, 2003b] exposes a join point for every method call, object instantiation, field access, and exception handler, but it exposes no join point for a “for loop”.

An aspect specifies in its pointcuts and advices where the concern crosscuts other concerns and what behaviour the concern exhibits, respectively. 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.

Aspects can be weaved together using the source or binary/byte code at different points in time. To give a few examples: before compile time by code weaving, at load-time by the component loader, or at run-time by the virtual machine. Based on the requirements of overhead and flexibility, one can select the optimal point in time to weave aspects. Of course, a single program can contain many aspects weaved together at different points in time.

 

# 3. Combing component-based and aspect-oriented software development

Component-based developed software has besides source and binary/byte code also an architectural description. Hence, when combining component-based and aspect-oriented software development, one can also weave based on the architectural description of the software. Weaving based on the architectural description has 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, 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. Computations within a component usually exploit implementation details to optimize throughput or 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.

And the following disadvantage:

1. The granularity of weaving is equal to the granularity of the architecture description language. This granularity is too coarse for use cases that require detailed information not available in the architectural description.

Summarizing, by combining component-based and aspect-oriented software development, one gets an additional option to weave that may better fulfil the requirements of the system under development. In the remainder of this article, we will focus on this new option to weave. Furthermore, we will focus on code weaving, since in our resource-constrained environment we do not need the flexibility and cannot afford the overhead of weaving at a later point in time. But see [Suvée et al., 2003] for an example of run-time weaving. Furthermore, unlike [Lieberherr et al., 1999], we aim at weaving without modifications of (binary) components and without explicit aspect deployment in the source code by the user. Finally, Lafferty and Cahill [2003] also propose a language-independent aspect weaver, but they exploit the multi-language support of the Common Language Infrastructure instead of the architecture description language.

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.

Based on the analogy with AspectJ [Laddad, 2003b] that exposes join points for object instantiation and initialisation in the object-oriented programming language Java, one would expect join points for component instantiation, initialisation, and destruction. But in Koala join points for component instantiation and destruction are irrelevant since all Koala components are instantiated at compile time and never destroyed. In addition, in Koala no special join points for component initialisation are needed since by convention Koala components are initialised by calling the *Init* function on their provided *IComponentInit* interface (see also Figure 4), and join points for functions in interfaces are available.

```
interface IComponentInit            [ini]
{
        void Init (void);
}
```

*Figure 4. Description of the component initialisation interface IComponentInit.*

 

# 4. Adding Aspect-Orientation

We experienced a chicken and egg situation when combining component-based with aspect-oriented software development. Designing a component-based aspect-oriented language and developing a weaver takes effort that is only justifiable when aspect-orientation adds value. Furthermore, the specific features of the language needed to add aspects to components would only become clear by using that language. We will now describe the approach we took to prove the benefits of adding aspect-orientation and explorer the requirements for the language with minimal effort. After describing our approach, we will present a simple example for illustration. Based on our experiences with the current tooling, we will describe the work that we see ahead.

## 4.1. Our Approach

For simplicity, modularity, and to keep research out of the production roadmap, we decided not to mix the weaver with the Koala compiler, but to make the weaver a pre-processor, initially. Consequently, we must accept some limitations including that the exact product configuration is unknown to the weaver. In addition, we decided that connecting to the provided and required functionality of aspects will be handled outside the component model, for now.

 

The weaver exploits the fact that one can easily introduce an indirection using wrapping as follows. See also Figure 5.

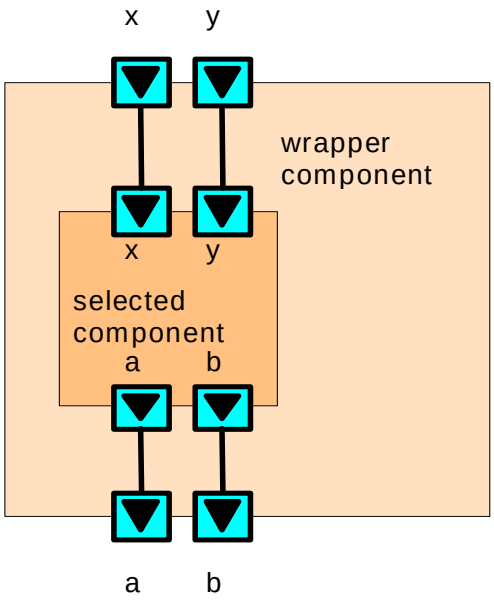


*Figure 5 Wrapper component with indirection.*

1. Select a component.
2. Create a wrapper component with the same signature, i.e. the same provided and required interfaces.
    - Instantiate the selected component in the erapper component.
    - Connect all interfaces of the wrapper component to the selected component. The wrapper component surrounds not only the provided functionality, like [Edwards et al., 2004], but also the required functionality.
3. Whenever the selected component is instantiated, use the wrapper component instead.

The weaver uses the indirection in the wrapper to weave in the aspects. This will be described in more details in Section 4.2.

For simplicity, flexibility, and language requirements exploration, we decided not to design and implement a component-based aspect-oriented language, but to use a plug-in framework instead. The framework handles, using reflection on the component and interface repository, i.e., the Code Document Object Model, the generic functionality of an aspect weaver:

- It finds the join-points of a pointcut.
- It generates wrapper components and exploits the indirections to insert additional code, i.e. the advices.

Each aspect is implemented as a plug-in. A plug-in is a compiled class (in C#.Net [Hejlsberg et al., 2004]) that implements a specific interface:

- It defines the pointcut.
- It implements the advices, by returning function bodies for the C-language, and introduces variables.

 

- It handles advices' specific dependencies.

Furthermore, since aspects are just classes, all object-orientation abstractions equally apply to aspects, e.g., abstract aspects are just abstract classes. Summarizing, we do not interpret an aspect but compile it into a plug-in instead.

We will now elaborately describe the framework together with an example aspect that verifies the usage of resources.

## 4.2. Example: Resource Usage Check

```
interface IResources            [res]
{
        int     NrOfSemaphores(void);
        int     NrOfMailboxes(void);
        int     NrOfThreads(void);
        …
}
```

*Figure 6. Description of the reflection interface IResources.*

```
interface IRealTimeKernel               [rtk]
{
        Semaphore    CreateSemaphore(…);
        Mailbox      CreateMailbox(…);
        Thread       CreateThread(…);
        …
}
```

*Figure 7. Description of the real time kernel interface IRealTimeKernel.*

Many components need resources, such as semaphores, mailboxes, and threads. Since televisions are resource constrained, Philips manages resource usage in a television explicitly. For this,

- All components must provide a reflection interface specifying the needed resources. See Figure 6 for the description of this interface. Note that the needed resources can depend on the context, specified by the diversity settings, of the component. After summing the needed resources as specified by all components, the system allocates exactly this amount of resources.
- Each component can only create resources during component initialisation via its real time kernel interface. See Figures 4 and 7 for the description of the initialisation and real time kernel interface, respectively.

To check resource usage, we have written an aspect that counts for each component the creation of resources and checks after initialisation whether the amount of resources specified and created match. Figure 8 shows a wrapper component generated by the framework with our aspect. The framework uses some of the indirections in the wrapper component to add additional components. The framework exploits that introducing com-

 

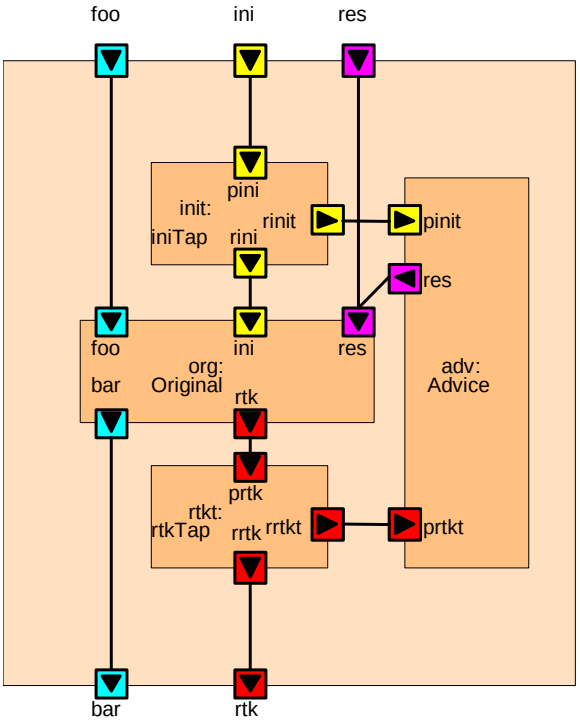


*Figure 8 The generated wrapper component.*

ponentns in Koala does not use resources, and adds many small components instead of a single component for three reasons:

1. To save resources, by enabling that only the relevant parts of aspects are included,
2. To prevent interference with Koala's ability to remove unused functionality, and
3. To visually reflect where an aspect crosscuts and what behaviour is added.

For each interface with at least one function in a pointcut, a subcomponent ending with *Tap* is added. This *Tap* component inserts the necessary before, around, and after function call notifications. In Figure 8, two of these *Tap* components are added, one for the initialisation interface and one for the real-time kernel interface. The subcomponent *Advice* contains all advices of pointcuts with join-points in the original component. Also the usage interfaces [Lieberherr et al., 1999] of expected operations by the aspects from the original component are visually reflected as required interfaces of the *Advice* subcomponent. In Figure 8, we see that the resources interface is in this case a usage interface.

```
…
static int CountSemaphores = 0;
…
void prtkt_BeforeCreateSemaphore (void) { CountSemaphores ++; }
…
void pinit_AfterInit (void) {
   assert (res_NrOfSemaphores() == CountSemaphores);
…
}
```



*Figure 9 Generated code for the resource usage aspect.*

Figure 9 shows the code generated by the framework based on the resource usage aspect for the *Advice* component. The framework generates components in Koala and aspect code in the C programming language, yet it is independent of the programming language used for the original component.

The resource usage aspect was weaved in during testing of multiple products, and was simply left out during release. With this single module, we were able to check that the implementation was according to its specification of a large number of components in multiple products. In fact, we found a few violations. One violation was caused by an erroneous component that claimed more resources than specified. All other violations were components wasting resources, since they claimed fewer resources than specified.

## 4.3. Experiences

We have used our framework, without any change to the original code, for

- Logging and tracing;
- Checking interface contracts, i.e., invariants, pre- and post conditions;
- Checking component states, ranging from asserting that a component is only used after initialisation to checking states in protocols; and
- Time measurements, e.g. the duration of semaphore locks.

In all of these cases, we observed that the combination with aspect-orientation increased the modularity of the software since it localizes crosscutting test concerns for component-based software in a single module. Thanks to aspect-orientation, the component-based software could be tested more effectively. The addition and removal of test code not only took less time due to the automation of the weaving process but also had a lower probability of introducing errors.

We observed that the properties (including context) of definitions, i.e., functions, interfaces, and components, become more important. Although we have exploited our naming convention to extract some of the properties relevant for our aspects, we experienced that a naming convention is not the solution. First, names become unreadable if they must capture all properties. Second, the discovery of a new property results in a non-localised change: not only the name of a definition with this new property but also all references to this definition are affected. Third, retrieving properties from a given name might be impossible: Is the function *setup(int)* a setter of up or a setup function? And what about *setupperbound(int)*? A good solution is to add attributes to definitions to expose their properties. We know two alternatives to add attributes:

1. Add the attributes to the definition, as is done in for example .Net [Hejlsberg et al., 2004].
2. In a separate module add the attributes by reference [Lieberherr et al., 1999].

The requirements for locality, readability, and flexibility determine which alternative to use. The attributes added to the definitions help not only to find the right join points but also to write compact, robust, and reusable aspects.

 

## 4.4. Future Work

Based on our experience, we decided to start developing a component-based aspect-oriented language. This language will support the selection of functions in interfaces of components based on the names, types, and attributes of functions, interfaces, and components. To give a few examples, select functions based on the attributes of their parameters and return types; select interfaces based on inheritance; and select components based on their unit of deploy, a.k.a. assembly in .Net [Hejlsberg et al., 2004].

In the future, we will combine the Koala compiler with the weaver, and we will extend the language to explicitly describe and connect the required and provided functionality of aspects. In this language, one can express that an aspect that, for example, counts function calls in a multi-threaded environment requires real-time kernel primitives to make the counter increments atomic and provides a query interface that returns for a given function name the number of times that function is called. This will also remove an undesired side effect that we observed: Adding aspects to a component can change its properties and behaviour. For example, an aspect that creates a semaphore on a usage interface changes the component's resource usage. Components, interacting via reflection interfaces with components where aspects are added, might retrieve unexpected values leading to errors due to our current approach.

The combined Koala compiler and weaver will also support a technique we call selected join points enumeration. The weaver enumerates all join points selected in (at least) a pointcut. The weaver presents a list of the selected join points with for each join point its unique enumeration value, its meta-data, and the aspects (in order) applied to it. This list is an essential aid to determine that the actual weaving matches the expectations. Furthermore, the enumeration values minimize the memory, bandwidth, and processor-time overhead of reflection in the embedded system, while the list enables off-line post-processing to still end up with readable results.

 

# 5. Summary

In this article, we combined component-based and aspect-oriented software development using the component model's architecture description language. Our approach is applicable to any component model that provides the architectural descriptions, directly or indirectly, e.g., via a Code Document Object Model or via reflection. Unlike the approaches of Suvée et al. [2003], Lafferty and Cahill [2003], and Noguera [2003], our approach can be applied in environments where resources are limited. We illustrated this using an existing component-based software stack that uses Koala [van Ommering et al., 2000, van Ommering, 2004]: a component model designed for resource-constrained environments. We experienced that incorporating aspect-orientation in our component-based software development process was beneficial because we could increase the modularity of our software using this additional modularization technique.

## Acknowledgements

I would like to thank Rob Golsteijn, Lennart de Graaf, Mehmet Aksit, David Watts, Frank Pijpers, Paul Janson, Rico Kind, Aleksandra Tešanović, Robert Deckers, Merijn de Jonge, and Rob van Ommering for their valuable remarks on my research in general and this article in particular.

## References

Parnas, D.L., December 1972, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15 No 12, pp. 1053-1058.

van Ommering, R., van der Linden, F., Kramer, J., Magee, J., March 2000. *The Koala Component Model for Consumer Electronics Software*, IEEE Computer, Vol. 33 No. 3, pp. 78-85.

van Ommering, R., 2004, *Building product populations with software components*, PhDThesis Rijksuniversiteit Groningen. ISBN 90-74445-64-0 Available at http://irs.ub.rug.nl/ppn/27516956

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, R., November/December 2003a. *Aspect-Oriented Programming Will Improve Quality*, IEEE Software, Vol. 20 No. 6, pp. 90-92.

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

Ossher, H., and Tarr, P., October 2001, *Using multidimensional separation of concerns to (re)shape evolving software*. Communications of the ACM, Vol. 44 No. 10, pp. 43-50.

 

Suvée, D., Vanderperren, W., Jonckers, V., March 2003. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development*, Proceeding of AOSD 2003, Boston USA, pp. 21-29.

Lafferty, D., Cahill, V., November 2003, *Language-Independent Aspect-Oriented Programming*, ACM SIGPLAN Notices, Vol. 38 No. 11, pp. 1-12.

Noguera, C., 2003, *Compose* - A Runtime for the .Net Platform*, Thesis Vrije Universiteit Brussel. Available at http://janus.cs.utwente.nl:8000/twiki/pub/Composer/ComposeStarDocumentation/thesisCarlosNoguera.pdf

Siegel, J., 2000. *CORBA 3: Fundamentals and Programming*, OMG Press, ISBN 0-471-29518-3.

Rogerson, D., 1997. *Inside COM: Microsoft's Component Object Model*, Microsoft Press, ISBN 1-572-31349-8.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin,J., June 1997, *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

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

Lieberherr, K., Lorenz, D., Mezini, M., March 1999. *Programming with Aspectual Components*, Technical Report, NU-CSS-99-01. Available at http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html

Edwards, S.H., Sitaraman, M., Weide, B.W., Hollingsworth, J., November 2004. *Contract-Checking Wrappers for C++ Classes*, IEEE Transactions on Software Engineering, Vol. 30 No. 11, pp. 794-810.

Hejlsberg, A., Wiltamuth, S., Golde, P., 2004. *The C# Programming Language*, Addison-Wesley, ISBN 0-321-15491-6.

 