Meta-programming composers in second-generation component systems

U. Assmann
Universität Karlsruhe
Institut für Programmstrukturen und Datenorganisation
Postfach 6980, 76128 Karlsruhe, Germany
E-mail: assmann@ipd.info.uni-karlsruhe.de

Abstract

Component systems require flexible composition of components. In contrast to current systems which only support a fixed set of composition mechanisms, future systems should provide a *composition language* in which users can define their own specific *composers*. For an object-oriented setting, we argue that this will be possible by meta-programming the class-graph.

Composers will be based on two major elements. First, they will express coupling with graph-based operators which transform parts of the class-graph (*coupling design patterns*). Second, during these transformations, elementary meta-operators will be used to transform data and code, rearranging slots and methods of their parameter-components. Thus during their reuse, components are queried by introspection and transformed by meta-programming.

Meta-programming composers generalize connectors in architectural languages. They encapsulate context-dependent aspects of a system, so that components become independent of context and can be reused more flexibly.

Keywords

Software reuse, component systems, composition, meta-programming

1 INTRODUCTION

For a long time, software engineers have had a dream: they want to build software from standard parts by composition (LEGO principle). Several authors have claimed that software composition will become a fundamental principle for the future software industry, because it supports flexible reuse [NM95] [Nie95]. However, it is not easy to build software like LEGO: it is not sufficient to reuse components *as is*, a software component has to be adapted extensively before it can be embedded into a larger system.

Systems Implementation 2000 R. Nigel Horspool (Ed.)

Usually components are coupled by hand-programming. To this end, component systems, such as JavaBeans [Jav96] or ActiveX [VN96], offer standardized coupling interfaces, by which the components can be plugged together. More elaborate *software architecture systems*, such as Darwin [MDK92], UniCon [SDK+95], or ACME [GAO95], offer a limited set of *connectors* by which components can be coupled in an abstract way [BF96]. Connectors link interfaces (*ports*) of components and arrange for embedding and control flow among them. Their major advantage is that they encapsulate all communication- and coupling-oriented aspects of a system. Then components can be programmed independently of their embedding context and reuse is improved. In general, two kinds of connectors can be distinguished. *Primitive connectors* are provided by the programming language or the operating system and comprise mechanisms such as method calls, pipelines, or event signaling. *Composite connectors* should be composed of these and should introduce complex interaction schemes among the components. However, there are only few systems which allow the specification of connectors [All97].

In this work, it is argued for an object-oriented setting, that complex connectors can be specified by *static meta-programming* on class-graphs. In an object-oriented setting, components are classes, and with meta-programming classes and methods can be introspected and adapted during composition. In essence, meta-programming can be used to define *composition operators* over components, so-called *composers*. These fall into two main categories: *composite connectors* link components and *encapsulators* encapsulate components to the outer world. Typically, composers are derived from *design patterns* that transform the structure of the class-graph. Hence this paper suggests that a *second-generation component system* should include a general *composition language* with meta-programming composers. This would allow to compose components very flexibly and improve reuse.

The next section presents some examples in which components are coupled by event-based design patterns (section 2). It turns out that these patterns can be seen as composite connectors, which may be specified separately from the components. This rises the need for meta-programming. In section 3 *composers* are defined formally and their tasks in future component-systems are discussed: complex composition (section 3.1), encapsulation, and configuration (section 3.3). It is also shown how composers can be represented by ordinary methods in an object-oriented language that supports meta-programming (section 3.2). Lastly, it is demonstrated that several well-known approaches from the literature are applications of meta-programming composers.

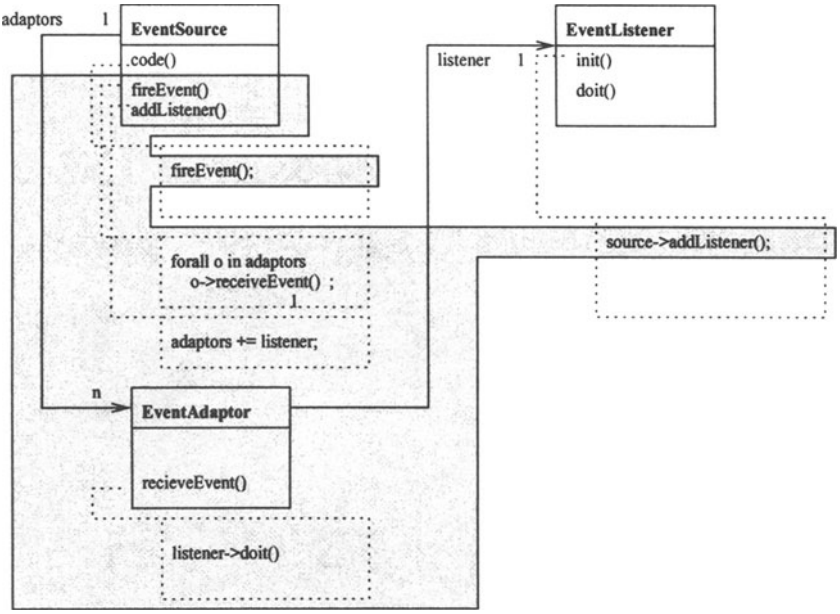

Figure 1 Coupling two components with JavaBeans EVENTADAPTOR design pattern. Coupling-specific parts are shaded. The event adaptor object receives the event and distributes to a listener.

2 EVENT COUPLING WITH META-PROGRAMMING COMPOSERS

This section presents two examples how meta-programming composers may look like, in particular composite connectors. The examples are based on a certain class of design patterns, namely those that couple components (*coupling design patterns*) [Tic97] [GHJV94]. These design patterns can be meta-programmed, i.e. described with programs on the meta-model.

Event coupling is a flexible method to link components [SN92] [GHJV94]. In an event-based coupling, events are fired by an *event source* component and delivered to a *mediator* context* which redistributes them to *event listeners* components. All event source and listener components have to register with the mediator, so that the mediator can distribute events correctly. When new listeners register, or the mediator changes, the behavior of the system changes also. However, such a change is transparent to the sources and listeners since they do not know to whom an event is delivered and from whom an event originates. This is the reason why event-based coupling is so flexible.

*Also called *event adaptor*, *event handler*, or *event manager*.

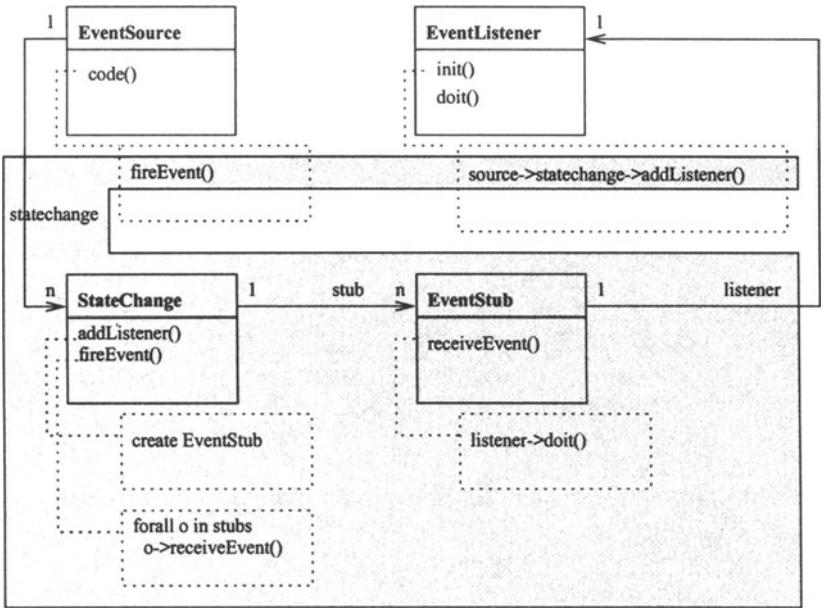

Figure 2 Coupling two components with design pattern EVENTNOTIFICATION. Coupling-specific parts are shaded. The listener registration and event signaling is dispersed into *StateChange* and *EventStub* components.

JavaBeans event adaptor Figure 1 shows the design pattern EVENTADAPTOR from JavaBeans 1.0 [Jav96] (also called *mediator* in [SN92]). In this coupling scheme, the event source maintains a list of listeners to which events are to be distributed. When the source fires an event with method `fireEvent`, an *event adaptor* component is called. This adaptor may modify events and has to distribute them to a listener. To be ready for an EVENTADAPTOR-coupling, a listener only has to register for an event at the event source (calling method `addListener`) and to provide an ordinary method that is called when an event occurs (method `doit`). The source has to do more for the coupling: it has to maintain the list of listeners. Hence, when components should be reused that were not prepared for event-based coupling, the components have to be extended by event-handling code manually (*white-box reuse*).

Event notification design pattern Event management can be decoupled from event sources. Figure 2 shows another event coupling design pattern, the EVENTNOTIFICATION [Rie96]. Here the management of the listener list is dispersed into a new component, the *StateChange*. When the coupling is initialized, the source creates a *StateChange* object for each event that it may fire.

The listener that registers with an event creates an event-specific *EventStub* object. This combination of *StateChange* and *EventStub* objects substitutes the event-specific listener queue of the event source in EVENTADAPTOR. When an event occurs, the source signals the corresponding *StateChange* object. This object distributes the event to all registered *EventStub* objects, which in turn call the `doit`-routine in the appropriate listener.

From design patterns to meta-programming composers In EVENTNOTIFICATION, almost all elements of the coupling are separated from the event source and the event listener, except the registration and the event-firing. This introduces a new view on the coupling of source and listener: the context-related parts of the scheme can be regarded as *glue code* which is introduced by a *composer* while connecting the components. In essence, such a composer has to do two things: it has to allocate new components and their interactions (the glue code), and it has to modify the components so that they can be coupled. In EVENTNOTIFICATION the composer is a *composite connector* which creates the glue classes *StateChange* and *EventStub* and mixes the calls of `fireEvent()` and `addListener()` into the code of the source and the listener. Hence the composer performs *grey-box reuse* on its *parameter-components*: although the user is not forced to edit the parameter-components manually (white-box) they are not reused as-is (black-box), but adapted by mixing-in connector code.

Also the EVENTADAPTOR pattern may be regarded as composer, the only difference is that it couples more tightly and changes more details in its parameter-components: it allocates a component *EventAdaptor*, adds methods and calls to the event source, and mixes the call to `addListener` into the listener.

Hence an event coupling scheme can be regarded as a meta-programming composition operator, which transforms the class-graph and mixes-in data and code into the parameter-components. Because nothing particular has been required, more design patterns should be realizable by meta-programming composers.

3 COMPOSERS: THE KEY TO SECOND-GENERATION COMPONENT SYSTEMS

This section defines formally how composers and second-generation component systems look like. These definitions can be realized directly in a programming language which supports reflection and intercession [KP97], and we outline how it would look like in Java.

3.1 Composition with composers

For the definition of composers, a simple object-oriented meta-model is assumed which may be easily extended in case of specific requirements. In this model components are classes. Additional meta-objects are methods and instructions. Component ports are method interfaces, and primitive connectors are method calls. The relations among the components are inheritance and association. A *class-graph* is an instance of the meta-model:

Definition 1 *A* meta-model $\mathcal{M} = (T, R)$ *is a scheme for a relational graph with a set of node labels* $T = \{\text{class}, \text{method}, \text{instruction}\}$ *and a set of relation labels* $R = \{\text{association}, \text{inheritance}\}$.

A class-graph $G = (V_i, E_j)$ *is a relational graph with a family of node sets* $V_i, i \in T$, *and a family of binary relations* $E_j \subset (\mathcal{I} \times \mathcal{I}), \mathcal{I} = \bigcup V_i, j \in R$. $\mathcal{I}$ *is called the set of* meta-objects.

T and R can be extended for other purposes with arbitrary other meta-objects and relations. A class-graph can be rewritten by a *composer*, an operator that uses graph rewriting and meta-programming:

Definition 2 *A* composer $C = (L \rightarrow R, P)$ *is a complex operator on the meta-model:* $L \rightarrow R$ *is a graph-rewrite rule with left-hand side class-graph* L *and right-hand side class-graph* R, *and* P *is a (meta-)program, performed on the items matched by* L.

A composer applies to a class-graph as follows:

Definition 3 *Let* G *be a class-graph. Then the composition* $G \rightarrow_C H$ *with composer* C *consists of the following steps:*

1. *The user specifies a subgraph* $G' \subset G$ *(the* application point*).* G' *is tested, whether* L *matches it, i.e. a non-induced injective graph homomorphism from* L *to* G' *is found.**
2. *The rewriting is performed, i.e.* G' *is rewritten to* G'' *which is injectively homomorph to* R.
3. *The meta-program* P *is performed on* G'.

The examples from section 2 can be regarded as graph rewrite rules on the class-graph: the left-hand side L consists of the white parts in the figures which are matched in the class-graph. The shaded part, i.e. the allocated glue methods and classes, is introduced by applying the right-hand side R to the

*An *induced* subgraph is matched, if together with a set of nodes in the graph *all* incident edges are matched also. Otherwise a *non-induced* subgraph is matched [BFG94].

matched subgraph. After that the meta-program P modifies the white parts appropriately, i.e. extends the matched meta-objects.

Based on these terms a component system can be formally defined:

Definition 4 *A* component system *is a tuple* $\mathcal{CS} = (\mathcal{M}, \mathcal{C})$ *where* $\mathcal{M}$ *is a meta-model and* $\mathcal{C}$ *is a finite set of composers. A* component-based software S *is the result of a sequence* Q *of compositions in the component system, starting from an initial class-graph* Z*:* $Q = Z \rightarrow_{C_1} G_1 \cdots \rightarrow_{C_{n-1}} G_{n-1} \rightarrow_{C_n} S, C_1, \ldots, C_n \in \mathcal{C}$.

Component-based software is the result of a sequence of composer applications in a component system. Unfortunately, $\mathcal{CS}$ cannot be an automatic graph rewrite system, since such a system would select arbitrary derivations, and not those the programmer wanted.

3.2 Composers in action

These formal definitions are concrete enough that they translate directly to a textual form in a Java-like style, and in the following the EVENTADAPTOR-connector from section 2 is demonstrated. All that is needed additionally is a meta-programming interface that provides reflection (querying meta-objects) and intercession (manipulating them). To this end, it is assumed that the reflection interface of Java is extended by intercession.

In such an extended reflection interface, the items of the meta-model, i.e. all meta-objects, are represented with ordinary classes. The following example uses the meta-objects `Class`, `Method`, and (implicitly) `Instruction` (Figure 3). We assume some basic reflective methods, such as `findMethod` which finds a method with a name, and `findClass` which finds a class with its name. Also several intercessory methods are required. The operator `new` may also allocate meta-objects, `addMethod` adds a method to a class, `prefix` prefixes the instruction list of a method with some instructions, and `MakeCodeFromText` constructs instruction lists from Java text.

Also the entire component system becomes an ordinary Java class, in which composers are static methods. In a composer $(L \rightarrow R, P)$, the graph rewrite tasks are implemented by matching and manipulating class-graph objects. The left-hand side L is implemented with a set of tests on the parameter-components and their relations. The right-hand side R turns into meta-statements which allocate and link meta-objects. The meta-program P consists of applications of elementary meta-operators and translates directly to a sequence of intercessory method calls.

With composers as methods, users may write programs of composer applications (Figure 4). Suppose that three classes `Boss`, `Assistant`, and `Secretary` are given, each of them with a `doit` and `init` method. Before a composer can

```
class ComponentSystem {
  public static void ClassGraph parser() {..};
  public static void void prettyPrint(ClassGraph c) {..};
  public static void JavaBeansEventConnector(EventSource:Class, EventListener: Class) {
    /* L: MATCH whether the source and the listener are related by a relation listener */
    if (!member(EventListener,EventSource->listener)) return;
        /* no application possible */

    /* R: REWRITING: Create meta-objects and meta-object-relations */
    Class EventAdaptor = new Class("EventAdaptor");
    Method receiveEvent= new Method("receiveEvent",MakeCodeFromText("listener->doit()"));
    Method addListener = new Method("addListener",MakeCodeFromText(
                                    "for (o = first(EventSource.adaptors);
                                          o != NULL;
                                          o = next(EventSource.adaptors,i))
                                             o.receiveEvent();"));
    addMethod(EventAdaptor,receiveEvent);

    /* P: MODIFY existing components */
    prefix(findMethod(EventListener,"init"),MakeCodeFromText("source.addListener()"));
    addMethod(EventSource,addListener);
    initialize(EventSource.adaptors);
  }
}
```

Figure 3 A component system as a Java-style class

be applied to a class-graph, a parser has to translate some components from program text to a class-graph. Then the composers (e.g. the EVENTADAPTOR- and EVENTNOTIFICATION-connectors) can be applied to the components. Finally, a pretty-printer has to generate Java code which contains the final layout of the classes. Hence complex applications can be plugged together with several calls to composer methods.

```
public static void CreateApplication() {
  ComponentSystem cs;
  ClassGraph classgraph = cs.parser();
  Boss boss             = findClass("Boss");
  Assistant assistant   = findClass("Assistant");
  Secretary secretary   = findClass("Secretary");

  /* Compose the classes */
  cs.JavaBeansEventConnector(boss,assistant);
  cs.EventNotificationConnector(boss,secretary);

  cs.prettyPrint(classgraph);
}
```

Figure 4 Composer applications as ordinary method applications

Since the composer extends the `doit`- and `init`-methods of the classes appropriately, event communication is introduced automatically by the composer application. Furthermore, since the composer only adds event-firing calls, and these are independent of the old code, the parameter-components are extended *transparently*. This illustrates the power of our approach: com-

ponents may be programmed independent of their context and, if the added code does not conflict with old code, the components are embedded into the context transparently.

When the meta-operators of the composition language are used carefully, components can be composed transparently and consistently. The extended version of this paper [Ass97] defines a criterion which checks that changes of parameter-components do not disturb the rest of the system. Whether new code is independent of old code can be checked by program analysis methods, e.g. program slicing. Then, although code and data of components are modified, old use-contexts never need to be changed. This is a major step forward towards general software composition, since it leads to *grey-box reuse*, i.e. reuse that extends components *transparently* to old use-contexts.

Of course a full-fledged component system would offer a library of composers. Users may use inheritance or even composition to extend them: since composers are components, they can be composed themselves. Composers will be designed along several design dimensions: Which parts of which parameter-components are coupled to others (data-flow dimension)? How complex are links? Which parameter-component executes when (control-flow dimension)? How tight are parameter-components coupled (integration dimension)? Programming composers spans up a large design space of composers, leaving all freedom for users to adapt compositions to their applications.

Also, the approach can be extended to a dynamic scenario. If the pretty-printing step is substituted by a code generation step (e.g. to Java bytecode) the generated classes can be loaded dynamically. Additionally, if bytecode can be read and meta-programmed – which is no problem in Java since type information is attached to each class file – the scenario becomes completely dynamic: compositions can be applied during the run-time of a system, the resulting classes are compiled, type-checked, and re-loaded again. Hence *incremental meta-programming* paves the way for incremental dynamic evolution of architectures.

3.3 Composers encapsulate and configure

Until now only composers have been investigated which couple components (i.e. connectors). Of course a composer may also abstract its parameter-components into a new component (*encapsulation*). Such hierarchical composition of subsystems is desired, since a subsystem hides unnecessary details to the outer world. This means in our example `CreateApplication` that after the two event connectors have been applied, a third composer should be called which encapsulates the three classes `Boss`, `Assistant`, and `Secretary` into one component. Such a composer should be formed according to an *encapsulating design pattern*, such as *Facade* [GHJV94]. It would create at least one new class for encapsulation and would link all parameter-components to it.

Configuring a system means to choose one of several variants for some of its parts. Since composers can be programmed, they can be built in variants. When a composer couples transparently, it can be exchanged to its variant without changing the coupled components. Then configuration amounts to selection of composers. Hence meta-programming composers allow to configure a software system orthogonally in two dimensions: both components and composers can be varied independently.

4 RELATED WORK

Architectural styles [GS93] describe component systems that allow only a certain kind of composers. For instance, in *implicit-invocation systems* only event-based composers are allowed while in *procedure-call systems* only composers are allowed that introduce method call links. In *repository systems* composers couple components tightly and synchronize them by wrapping slot access methods with synchronization protocols. In *pipe-filter systems* composers introduce unidirectional flow of work packages between components. Additionally, when the architecture is described by composer applications, exchange of composers change the architecture. Exchanging one composer style to another changes the architectural style of a system. For instance, it can be easily imagined that a procedure-call system can be turned into an implicit-invocation system: call-composers need to be exchanged to their event-based cousins.

Adaptive programming (ADP) [LSLX94] uses graph rewriting on the class-graph and method aggregation. First ADP computes a set of classes to which new methods are added, evaluating a path expression on the class-graph. This corresponds to the evaluation of a Datalog procedure on the class-graph, or an edge-addition graph rewrite system [Ass94]. In a second step these classes are extended by new methods which are created from a code specification. In essence, ADP is just a form of static meta-programming. It can be regarded as a powerful super-composer which connects a set of classes with an mixed-in algorithm. Because the set of classes is computed from a Datalog procedure, the ADP-operator can do more than a composer, which can match only a fixed number of meta-objects in the left-hand side of its graph rewrite rule.

Aspect oriented programming (AOP) divides programs into *component parts* and *aspects* [Kic96]. Aspects are merged into the components, just as in our approach composers extend components with context-related code. However, AOP relies on a particular aspect language, which describes the coupling, and an aspect weaver, which performs the coupling. Hence for each class of applications new aspect languages and weavers have to be developed. Our approach is simpler, as it only relies on static meta-programming. As in AOP, the composition process can be expanded to code.

Context relations allow to adapt objects to their context at allocation time [SPL96]. This is similar to the exchange of superclasses at allocation

time [Wec97]. However, adapting components and exchanging superclasses is just a special case of meta-programmed composition at allocation time.

Composition-filters [Ber94] [ABV92] and the *layered object model* [Bos95] represent context-related actions of a class by *filters* or *layers* that encapsulate it. Each message that arrives at a class has to cross this set of filters which modify it. However, composing filters (i.e. wrapping code around methods and objects) is a simple meta-operation. [Bos97] details this for context-related adaptation. He argues that components need to be adapted flexibly with *superimposition* which is a modification of the component by means of a new layer around it. Bosch mentions the idea that adaptation can be meta-programming, but does not elaborate on this. Meta-programming is more powerful than layering: it can change components deep inside their implementation while layering can only wrap components.

In his thesis [Zim97], Zimmer develops the idea to use design patterns as transformation operators on the class-graph. Zimmer defines a language in which all actions a design pattern involves can be described systematically (pattern matching on the class-graph, transformations of methods, etc.). Although Zimmer did not recognize that his language uses static meta-programming this provided one of the starting points for our work.

Code generation from design patterns has been attempted only recently [BFVY96]. Design patterns are described in the form of [GHJV94], with an additional description in a special language COGENT. This macro-based language is expanded by the `perl` interpreter to C++ code. Since the items of COGENT are classes, this approach is static meta-programming, although it has not been described as such. In our work, code generation from design patterns results naturally, since composed classes can be compiled.

5 CONCLUSION

This work demonstrates that in future component systems application-specific and coupling-specific code can be separated from each other entirely. Powerful composition operators can be developed which introduce coupling code into the application-specific components from outside because they edit and adapt components during composition (*grey-box reuse*). These composers use static meta-programming and coupling design patterns. Whenever a composer is applied, the meta-programming composition creates glue code between components automatically. Additionally, if the meta-programming composition is applied together with dynamic loading, dynamically changing architectures can be constructed easily.

Meta-programming composers generalize architectural description languages to a general composition language. Hence this work lays the foundation for second-generation component systems, in which context-specific aspects will be encapsulated in complex composers while application-specific aspects will be encapsulated in components. Since both components and composers can

be varied orthogonally, reuse will be enhanced enormously. White-box reuse is too difficult and laborious; black-box reuse is too primitive; *grey-box reuse* is the way to go, and meta-programming composers enable grey-box reuse.

REFERENCES

[ABV92] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 372–395, Berlin, Heidelberg, New York, Tokyo, June 1992. Springer-Verlag.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, May 1997. CMU-CS-97-144.

[Ass94] Uwe Assmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *5th Int. Workshop on Graph Grammars and Their Application To Computer Science, Williamsburg*, volume 1073 of *Lecture Notes in Computer Science*, pages 321–335, Heidelberg, November 1994. Springer.

[Ass97] Uwe Assmann. Meta-programming Composers In Second-Generation Component Systems. Technical Report 17, Universität Karlsruhe, September 1997.

[Ber94] Lodewijk M. J. Bergmans. *Composing concurrent objects.* PhD thesis, University of Twente, Enschede, 1994.

[BF96] Judy Bishop and Robert Faria. Connectors in configuration programming languages: are they necessary? In *3rd International Conference on Configurable Distribute systems.* IEEE Press, May 1996.

[BFG94] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical Use of Graph Rewriting. Technical Report Queens University, Kingston, Ontario, November 1994.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[Bos95] Jan Bosch. *The layered object model.* PhD thesis, University Twente, 1995.

[Bos97] Jan Bosch. Adapting object-oriented components. In M. Aksit et al., editor, *ECOOP Workshop on Component Systems*, June 1997.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural

mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[GS93] David Garlan and Mary Shaw. *An Introduction to Software Architecture*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[Jav96] JavaSoft. JavaBeans^{TM}. http://java.sun.com/beans, December 1996. Version 1.00-A.

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4), December 1996.

[KP97] Gregor Kiczales and Andreas Paepcke. Open implementations and metaobject protocols. Technical report, Xerox PARC, 1997.

[LSLX94] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[MDK92] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[Nie95] Oscar Nierstrasz. Research topics in software composition. In *Proceedings, Langages et Modèles à Objets*, pages 193–204, Nancy, October 1995.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[Rie96] Dirk Riehle. The event notification pattern – integrating implicit invocation with object-orientation. *Theory and Practice of Object Systems*, 2(1):43–52, 1996.

[SDK+95] Mary Shaw, Robert DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, pages 314–335, April 1995.

[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions of Software Engineering and Methodology*, 1(3):229–269, July 1992.

[SPL96] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of Object Behavior using Context Relations. In David Garlan, editor, *4th ACM SIGSOFT Symposium on the foundations of Software Engineering*, pages 46–57, October 1996.

[Tic97] Walter F. Tichy. Classification of Design Patterns. Lecture slides, January 1997, Universität Karlsruhe. http://wwwipd.ira.uka.de/~tichy/entwurfsmuster.html.

[VN96] Steven J. Vaughan-Nichols. ActiveX chases Java. *BYTE Magazine*, 21(6):27–27, June 1996.

[Wec97] Wolfgang Weck. Inheritance Using Contracts and Object Composition. In *WCOP Workshop on component-based systems at ECOOP 97*, June 1997.

[Zim97] Walter Zimmer. *Frameworks und Entwurfsmuster.* PhD thesis, Universitt Karlsruhe, February 1997.

BIOGRAPHY

Dr. **Uwe Assmann** did his PhD thesis *Generation of program optimizations with graph rewriting* at University of Karlsruhe in 1995. As a side action he implemented the optimizer generator OPTIMIX. He has also been involved in the design of the compiler component model CoSy which is marketed by ACE b.V. Amsterdam (`http://www.ace.nl`). Currently he works in software architecture, and is interested in mechanisms for flexible composition of software components.