

# Customization of On-line Services with Simultaneous Client-Specific Views

Eddy Truyen            Bart Vanhaute  
Wouter Joosen        Pierre Verbaeten  
*DistriNet, Dept. Computer Science*  
*K.U.Leuven*  
*Celestijnenlaan 200A*  
*3001 Leuven, Belgium*  
*+32 (0) 16327602*

[\[eddy, bartvh, wouter, pv\]@cs.kuleuven.ac.be](mailto:{eddy, bartvh, wouter, pv}@cs.kuleuven.ac.be)

Bo Nørregaard Jørgensen  
*Maersk Institute for Production Technology*  
*University of Southern Denmark*  
*Odense Campus*  
*DK-5230 Odense M, Denmark*  
*(+45) 6550 3545*  
[bnj@mip.sdu.dk](mailto:bnj@mip.sdu.dk)

## 1. Introduction

With the Internet success, a new trend has come up: on-line distributed services. Business companies (acting as client systems of these services) that are able to integrate such services effectively within their internal workflow will become leaders in their business markets. This however introduces the situation where *multiple* independent client systems are *simultaneously* accessing the same on-line service instance. This introduces a new research challenge concerning client-specific customization of an on-line service: different client systems may have different – possibly conflicting - customization needs with respect to the functional and non-functional features of the on-line service. The problem with this is that all the different client-specific views must be simultaneously imposed on the same distributed service *instance*; thus after the distributed service has become operational.

We propose a dynamic customization model, *Lasagne*, that supports such client-specific combination of features at the instance level. In *Lasagne* a distributed service is structured as consisting of a minimal functional core – implemented as a component-based system, and an unbound set of potential extensions that can be selectively integrated within this core functionality. An extension to this core may be a new service, due to new requirements of end users. Another important category of extensions we consider, are non-functional services such as authentication, which typically introduce interaction refinements at the application level. Each extension is implemented as a layer of mixin-like wrappers. Wrappers are utterly useful for customizing on-line distributed services, since wrappers operate at the instance level, enabling runtime customization. The novelty of this work is that the composition logic, responsible for integrating extensions into the core system, is completely separated from the code of the core system, extensions and clients as well. Clients can customize this composition logic dynamically on a per

client request basis by attaching extension identifiers to their interactions with the core system, enabling easy-client-specific customization.

## 2. An Example

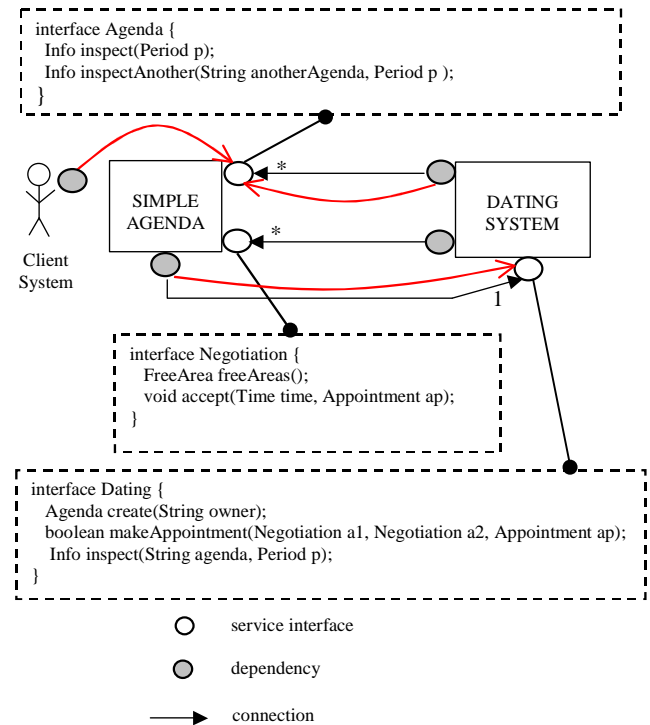


Figure 1: Minimal core of dating system

Suppose a distributed dating system, which manages a set of electronic agendas on behalf of a set of persons. The system implementation consists of two *components* as indicated in Figure 1. We call a component interaction between a client and the core system a *client request*. A

client request initiates a *collaboration* between the component instances of the core system. A collaboration within the system can then be represented by a directed graph, of which the nodes are component instances and the edges (represented by the curly arrows in Figure 1) represent the *message flow* of subsequent component interactions. The root of the graph is the client request, uniquely identifying the entire collaboration within the core system.

A component instance in the core system participates in one or more collaborations with other components. For example the curly arrows in Figure 1 show the collaboration for the client request “inspect another agenda”. A second collaboration “make appointment” consists of the dating system component instance, which coordinates the creation of an appointment between two agenda instances, by searching for a point of time that is marked as free area in both agenda’s.

Suppose one or more specific clients want to use the services of the operational dating system, but require several additional functional and non-functional extensions for the dating system:

- A service for making group appointments between more than two agendas in an atomic fashion must be available. As a consequence, agendas must be extended with atomic commit behavior.
- Different rules apply for making an appointment, whether it is meant for business or leisure. Therefore the dating system should simultaneously support two different appointment strategies for making appointments.
- Authentication: Clients must be authenticated, but a client should only be authenticated if he/she connects remotely to the dating system from a distinct subnet.
- Authorization: Different clients have different access rights for the agenda of another client. Authorization should only be applied for client requests inspecting another agenda.

Of course, the above extensions should exclusively be applied on behalf of the specific clients who have need of these customizations. The extensions must not be applied for requests from the other clients.

### 3. Overview of Lasagne

Lasagne defines a platform-independent architecture for client-specific customization of component-based systems using wrappers. It can be implemented on top of a language or middleware platform with an open implementation. We implemented Lasagne on top of the concurrent object-oriented language Correlate [14] (using Correlate’s Meta-Object Protocol) and on top of Java (using load-time reflection [Renaud Pawlak; personal communication, 4]).

We distinguish between three different phases in the Lasagne customization process: the phase of implementing an extension using a wrapper-programming model, deployment/weaving of one or more extensions into the core system, and selective combination of extensions per client request.

#### 3.1. The extension programming model

We implement an extension as a coherent module of mixin-like wrappers. We program a wrapper more or less as a “component-oriented” decorator[6] that attaches additional state and refined (interaction) behavior to a dynamically bound inner component instance. Another interesting wrapper-based design pattern is the Role Object [2] that attaches new service interfaces to a component instance. An extension may consist of several wrapper definitions, each one to be wrapped around a different point in the core system (see deployment of extensions). Below is given the implementation of the extension for the group appointment service. It consists of two wrapper definitions, which cooperate to implement the service with atomic commit behavior.

```
package examples.agenda.extensions.group;
//hybrid between role and decorator
public class AtomicAgenda implements Negotiation, Atomic {

    public boolean canCommit(Time time) {
        if (!isLocked(time)) {
            lock(time); return true;
        } else return false;
    }

    public void abort(Time time) {
        releaseLock(time);
    }

    public void acceptAppointment(Time time, Appointment app, Context context) {
        inner.acceptAppointment(time, app);
        releaseLock(time);
    }

    public FreeArea getFreeAreas(Context context) {
        return inner.getFreeAreas();
    }

    protected void releaseLock(Time time) {...} //internal
    protected boolean isLocked(Time time) {...} //internal
}

package examples.agenda.extensions.group;
//hybrid between role and decorator
public class GroupAppointmentService implements GroupService {

    public void makeGroupAppointment(Negotiation[] a, Appointment app) {
        FreeAreas[] frees = fetchFreeAreas(a);
        Time time = match(frees);
        for (i=0; i < a.length, i++) {
            If (!(Atomic)(a[i].getRole("group")).canCommit(time))
                //abort by calling abort(time) on all agenda's
        }
        if (allCommitted) {
            for (i=0; i < a.length, i++)
                a[i].acceptAppointment(time, app);
        }
    }
}
```

### 3.2. Deployment of extensions

At deployment time we specify for each extension around which core components each of its wrapper definitions must be wrapped. To avoid semantic conflicts, we may specify partial order constraints between extensions in the way their wrappers must be chained together.

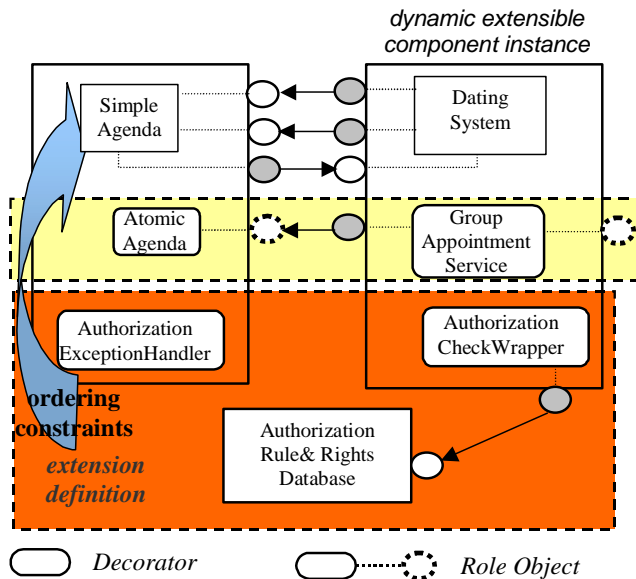


Figure 2: Implementing and deploying extensions

### 3.3. A model of client-specific combination

Finally at run-time, after the core system becomes operational, extensions are selectively integrated into the core system on a per client-request basis.

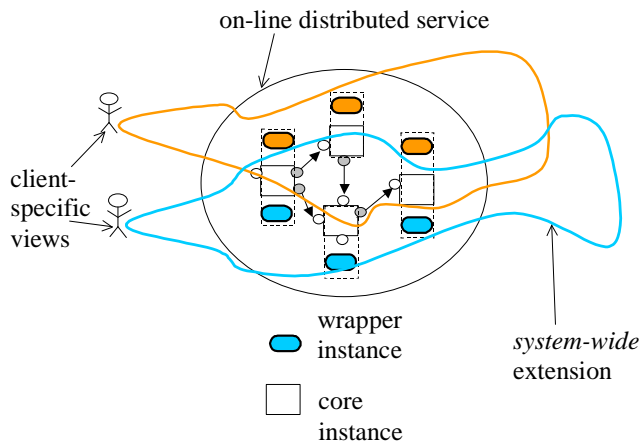


Figure 3 Multiple, simultaneous client views

An extension involves *multiple* wrappers, each to be decorated around *different* (distributed) component instances. As a consequence, since a client request initiates a collaboration between the component instances of the core system, we need a mechanism that guarantees a *system-wide* and *consistent* adjustment of the collaboration's *message flow*, such that the wrappers part of that extension are *all* applied to any relevant collaboration.

The Lasagne model copes with this, without losing the ability to manage multiple, simultaneous client-specific views. The Lasagne model is founded on four conceptual cornerstones:

First, we augment the object-oriented programming model with the notion of *component identity* that unites and hides the separate object identities of the component instance and its decorating wrapper instances. When a wrapper instance is decorated around a core component instance, the wrapper instance takes on the component identity. The object identity of the wrapper instance (defined by the hosting programming language platform) is only apparent within the boundaries of the component. As such, we are able to wrap a component instance without losing the component instance's identity[8].

Second, we observe that composition is ideally specified in terms of extensions instead of wrappers, which represent too fine-grained entities. It is really the extension as a whole that clients want to select or unselect for their collaborations with the core system. Therefore, we introduce the notion of an *extension identifier*, which is an interpretable, high-level name uniquely identifying the extension. For example for the dating system example we introduce the dummy extension identifiers "group", "leisure", "business", "authent" and "authoriz" (see Figure 5). Wrapper definitions are then specified to be member of an extension by declarative binding to the unique extension identifier.

Third, we make the wrapper composition logic external from the code of the core system, extensions and clients by encapsulating it in a *composition policy*. A composition policy specifies *the subset of extension identifiers* that must be applied *for a specific collaboration* between client and core system. A composition policy *travels* together with the message flow of its collaboration: it is automatically propagated through the system as the execution of its collaboration advances. As such, the wrapper composition logic travels (in the form of a composition policy) together with the collaboration's message flow, rather than being locked up and scattered across the code of subsequent component interactions.

Fourth, a composition policy is incrementally defined at run-time by some *interceptors*. An interceptor intercepts incoming or outgoing messages of a specific component instance and may update their associated composition policy by attaching/discarding extension identifiers.

Interceptors typically implement a client-specific strategy that determines which extension identifiers are attached to which collaborations.

The underlying code example illustrates a simple implementation of an interceptor that selects the “leisure” extension when the client makes an appointment.

```
package client.customization;

/** [(make*Appointment)] =>["leisure"] */
public class AppointmentStrategySelector implements Interceptor {

    public void manipulate(Interaction interaction) {
        /**identification of the collaboration*/
        String methodDef = getMethodDef(interaction);
        if (methodDef.endsWith("Appointment"))
        {
            /**Updating the composition policy of the collaboration based on
            contextual properties (cfr. section 2.1) */
            CompositionPolicy policy = interaction.getPolicy();
            policy.addExtensionIdentifier("leisure");
        }
    }
}
```

Interceptors can also inspect *contextual properties*[4] to determine whether an extension identifier must be attached or not. A contextual property is visible as a <name, value> pair that has been previously attached to the collaboration by another interceptor of the calling context. Contextual properties also travel with their collaboration. The underlying code example illustrates a simple implementation of an interceptor that determines whether an authentication check should be applied or not. For this example, the on-line service administrator has decided that only client requests for making an appointment, which originate from a remote subnet, should be authenticated.

```
package servicefrontend.customization;

/** [(make*Appointment) and (client from remote subnet)] => "authent" */
public class AuthenticationSelector implements Interceptor {

    public void manipulate(Interaction interaction) {
        /**identification of the collaboration*/
        String methodDef = getMethodDef(interaction);
        if (methodDef.endsWith("Appointment"))
        {
            /**Updating the composition policy of the collaboration based on
            contextual properties (cfr. section 2.1) */
            CompositionPolicy policy = interaction.getPolicy();
            Context context = interaction.getContext();
            if (distinctSubnet(context.getProperty("clientHost")))
                policy.addExtensionIdentifier("authent");
        }
    }
}
```

Underlying these four concepts, the run-time model of component must support a *dynamic* construction of the wrapper chain by adjusting message flow. This releases client objects from the complex task of selecting between disjunctive wrapper chains. The intuitive difference in chaining is shown in Figure 4: it is the currently ongoing collaboration itself that searches its way through the appropriate wrapper instances based on inspection of its

composition policy. To realize this dynamic chain construction a generic dispatch mechanism, called *variation point*, is introduced between the component identity, and the aggregate of core and wrapper instances. The variation point of a component instance interprets the composition policy of each incoming message and will only redirect this message through those wrapper instances whose extension identifier is listed in the composition policy. The reader is deferred to [18] for implementation details of the variation point construct. If you want to implement the variation point on top of an existing programming language or middleware platform, it must be possible to redesign their internal message dispatch mechanism.

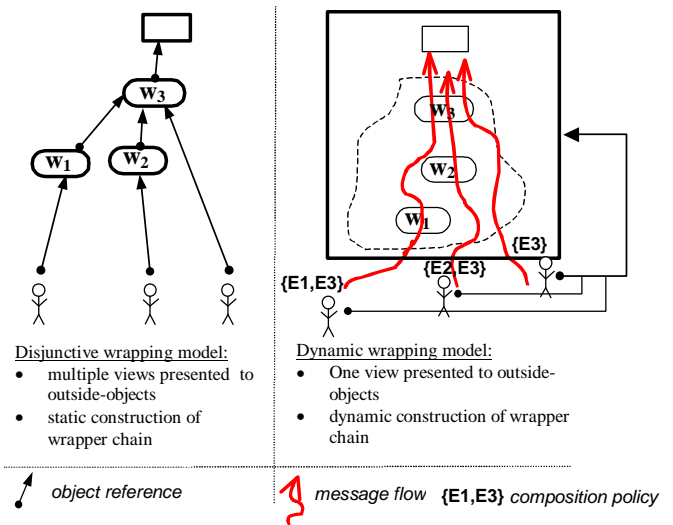


Figure 4 Disjunctive vs. Dynamic Wrapping

Figure 5 presents a message sequence diagram illustrating the process of selective combination with some of the sample extensions. We have defined a separate interceptor for each one of the extensions. Interceptor A dynamically selects an appropriate strategy (“leisure” or “business”) when making an appointment. Interceptor B attaches the “group” extension identifier for incoming service calls on the GroupService interface. Interceptors can also inspect contextual properties of the collaboration. For example interceptor C will only select “authent” for incoming client requests originating from a remote subnet location. As shown in Figure 5, the composition policy (indicated by {...}) determines the way message flow is adjusted within the system. A composition policy may also maintain dynamic state about the applicability of extensions. For example in Figure 5 once a client request is authenticated, it is not necessary anymore to perform the same authentication check later again, thus the composition policy will discard the extension identifier “authent”.

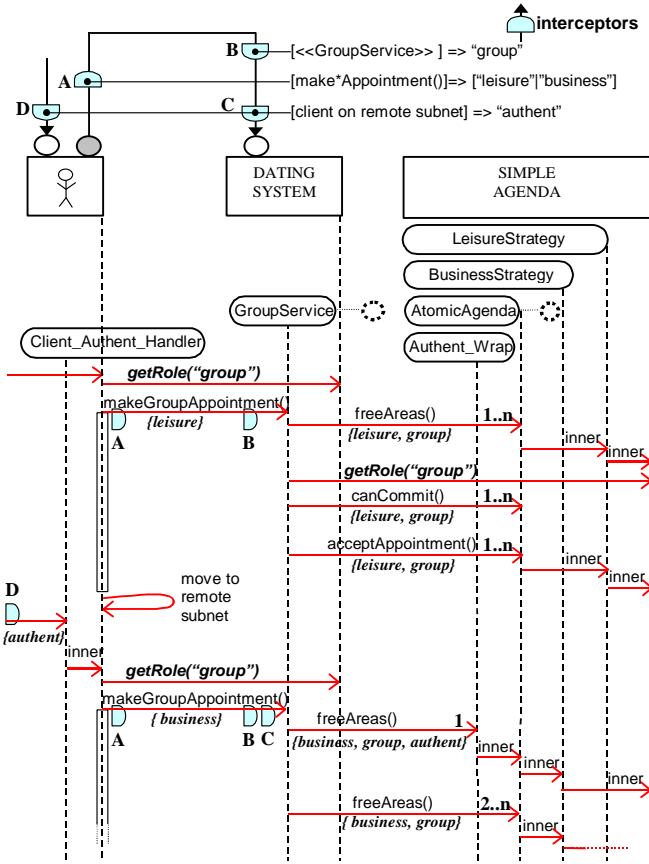


Figure 5: Message Sequence Diagram

Figure 5 also illustrates that due to the propagating nature of composition policies a change in an interceptor has a *system-wide effect* on the composition of wrappers in all the parts of the system that are invoked within the message flow after the interceptor is applied. As such, interceptors encapsulate a *context-specific* customization of the *whole* core system, with the guarantee of *consistency*.

## 4. Related work

### 4.1. Advanced Separation of Concerns

Advanced separation of concerns (ASOC) techniques such as Aspect-oriented Programming [9], Hyperspaces [17], Mixin Layers [16], and Adaptive Plug and Play Components [12] allow extension of a core application with a new aspect/subject/layer/collaboration, by simultaneously refining state and behavior at multiple points in the application in a non-invasive manner. These approaches however mainly operate at the class-level, while we compose extensions at the instance-level, enabling run-time customization. Lasagne also supports customization with multiple, independent client-specific views on the core

system. This is useful for customizing distributed systems, since a distributed service may have, during its lifetime, several remote clients, each with different customization needs. This feature is not really well supported in the above class-based composition techniques. However, Lasagne suffers from a run-time performance overhead. Therefore we think that Lasagne is better suited for client-specific integration of extensions at the more *coarse-grained* architectural level of a (distributed) system (thus macroscopic customization), while the above class-based techniques are better suited for providing separation of concerns at the component implementation level (thus microscopic customization). In addition to this, the wrapper-based approach by itself has a number of known limitations, which cause most problems at the microscopic level: It cannot achieve call-site composition non-invasively, wrappers can only access members that are visible in the public interface of a component; some of the existing ASOC tools don't have these limitations. A particular interesting problem with wrappers is the *object schizophrenia* problem: calls by a core component on self/this cannot be wrapped non-invasively. This is because there is no support for *delegation* in class-based programming languages [10]. Delegation means that the self parameter is bound to the wrapper chain through which the service call, in process by the core instance, was received. Lasagne can simulate delegation quite elegantly though by rebinding the self parameter to the component identity and the variation point will automatically redirect every self call through the appropriate wrapper chain conforming the current composition policy.

Composition Filters [1] composes non-functional aspects on a per object interaction basis. Composition Filters however does not have any support for consistent and system-wide refinement; the composition logic enforcing the integration of a system-wide extension is scattered across multiple object interactions, thus difficult to update consistently in one atomic action. In Lasagne, the composition logic is completely encapsulated within the composition policy of the currently ongoing collaboration.

Aspect Components [13] is an aspect-oriented reflective middleware for distributed programming. It provides components that integrate system-wide properties such as distribution and authentication within a core application in a non-invasive manner. Aspect Components are also implemented as a layer of mixin-like wrappers. Its goal is thus very similar to ours. Aspect Components has however no explicit support for selective combination on a per collaboration basis. This could of course be added, as we proved recently by implementing Lasagne on top of the Java implementation of Aspect Components (JAC).

### 4.2. Dynamic behavior composition

Linda Seiter et al. [15] proposed a *context relation* to

dynamically modify a group of base classes. A context class contains several method updates for several base classes. A context object may be dynamically attached to a base object, or it may be attached to a collaboration, in which case it is implicitly attached to the set of base objects involved in that collaboration. This makes the underlying mechanism behind context relations very similar to the traveling composition policies of Lasagne (solving the consistency management problem). However, context relations have overriding semantics and do not allow selective combination of extensions.

Mira Mezini [11] presented the object model Rondo that does well support dynamic composition of object behavior without name collisions. However, there is no support mentioned for specifying behavior composition on a per collaboration basis. Research is however ongoing to make her work about composing collaborations [12] more dynamic [7].

## 5. Concluding remarks

In this paper we have presented the customization model Lasagne. Lasagne defines an architecture for dynamic customization of component-based systems using wrappers. Lasagne supports a client-specific, non-invasive and consistent integration of system-wide extensions on a per collaboration basis. We have implemented Lasagne on top of the languages Correlate and Java.

Future work involves real-world evaluation of the usability and scalability of the Lasagne model. Some tooling should be provided to semi-automate the Lasagne customization process. Furthermore we are investigating how the Lasagne model can be implemented on top of component technologies such as EJB and dynamic service architectures such as Jini. Here we will also look into the possibility of load-time combination of extensions at client-side stubs, which is flexible enough at client-side but more cost-efficient than run-time combination.

## 6. Acknowledgments

This research was supported by a grant from the Flemish Institute for the advancement of scientific-technological research in the industry (IWT).

## 7. References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, "Abstracting Object-Interactions Using Composition-Filters", in *Object-Based Distributed Processing*, R. Guerraoui, O. Nierstrasz and M. Riveill (eds), Springer-Verlag, 1993, pp. 152-184.  
 [2] D. Bäumer, D. Riehle, W. Siberski and M. Wulf, "Role Object". In *Pattern Languages of Program Design 4*,

N. Harisson (ed.), Addison-Wesley, 2000.  
 [3] G. Bracha and W. Cook, "Mixin-based inheritance", in *Proceeding of OOPSLA/ECOOP '90*, October 1990.  
 [4] J. Brichau, W. De Meuter and K. De Volder, "Jumping Aspects", position paper for *the ECOOP'2000 Workshop on Aspects and Dimensions of Concerns (ADC'2000)*, C. V. Lopes (ed.), 2000.  
 [5] S. Chiba, "Load-Time Structural Reflection in Java", in *Proceedings of ECOOP'2000*, 2000, Springer-Verlag LNCS 1850.  
 [6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, pp. 175-184.  
 [7] S. Hermann and M. Mezini, "PIROL, A Case-Study for Multi-Dimensional Separation of Concerns in Software Engineering Environments", in *Proceedings of OOPSLA'2000*, October 2000.  
 [8] U. Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages, in *Proceedings of ECOOP'93*, 1993.  
 [9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwan, "Aspect-Oriented Programming", in *Proceedings of ECOOP'97*, June 1997.  
 [10] G. Kniessel, "Type-Safe Delegation for Run-Time Component Adaptation", In *Proceedings of ECOOP'99*, June 1999.  
 [11] M. Mezini, "Dynamic Object Evolution without Name Collisions", in *Proceedings of ECOOP'97*, 1997.  
 [12] M. Mezini and K. Lieberherr, "Adaptive Plug and Play Components for Evolutionary Software Development", in *Proceedings of OOPSLA'98*, 1998.  
 [13] R. Pawlak, L. Duchien, G. Florin, L. Martelli, L. Seinturier, "Distributed Separation of Concerns with Aspect Components", in *Proceedings of TOOLS Europe'2000*, IEEE press, June 2000.  
 [14] B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects*, Phd KULeuven, 1999, ISBN 90-5682-194-6.  
 [15] L. Seiter, J. Palsberg, and K. Lieberherr, "Evolution of Object Behavior using Context Relations. In *IEEE Transactions on Software Engineering*, 24(1), 1998.  
 [16] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", in *Proceedings of ECOOP'98*, 1998.  
 [17] P. Tarr, H. Ossher, W. Harrison, S. Sutton Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", in *Proceedings of ICSE'99*, 1999.  
 [18] E. Truyen, B. N. Jørgensen, W. Joosen, "Customization of Component-Based Object Request Brokers through Dynamic Configuration", in *Proceedings of TOOLS Europe'2000*, IEEE press, June 2000.