

# A Dynamic Customization Model for Distributed Component-Based Systems

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*

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*

## Abstract

*Support for dynamic and client-specific customization of distributed services is required in many application areas. We present a distributed service 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 decorator-like wrappers.*

*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 (end users, system integrators) can customize this composition logic dynamically on a per interaction basis by attaching extension identifiers to their interactions with the core system.*

## 1. Introduction

A component framework defines a semi-complete software architecture that has carefully been parameterized with respect to the components that represent the most variable elements of the domain. A component-based application can thus be instantiated from the component framework by simply providing the specific components needed for the particular application. Customization of the application is then supported by replacing components with more suitable ones. The problem however is that components are not normally the most variable elements of a software architecture – the

interactions between components are [1]. As such, the customization process goes beyond replacing individual components, but involves simultaneously refining the interaction behavior of multiple components without breaking consistency. In this paper, we present the customization process of a component-based application as a selective combination of one or more *extensions* into a minimal core system. Each extension is implemented as a layer of decorator-like wrappers [4], simultaneously tailoring multiple core components and their interactions between each other.

Decorators operate at the instance level. This is utterly useful for customizing distributed applications: a component instance may have during its lifetime several new remote clients, each with different customization needs; with decorators each client-specific customization simply corresponds to another decorator instance around the component instance. However, using decorators to customize applications on a large scale has not yet found much success in class-based object-oriented programming, due to problems with object identity [5].

In this paper, we first illustrate our research goals and identify challenges by an example application. Then, we give an overview of the customization model *Lasagne*, which defines a language-independent blueprint for a dynamic customization process of component-based (distributed) systems with decorator-like wrappers. In section 4 we analyze the object identity problem with decorators and in section 5 we present a dynamic wrapping model solving these problems. The novelty of this model is that it allows selective combination of extensions on a per collaboration basis, enabling easy client-specific customization.

## 2. An example

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” (not shown in Figure 1) 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.

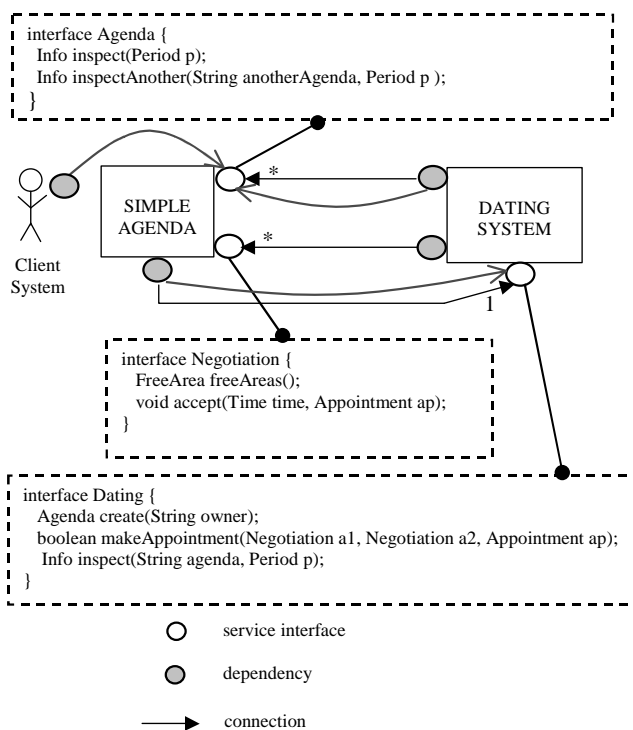


Figure 1: Minimal core of dating system

## 2.1. Challenges for a customization architecture

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 his agenda 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.

Many challenges are involved in the process of customizing the dating system with these extensions:

(1) *Run-time customization*. In order to provide 7x24h availability of the dating system, these extensions should ideally be integrated into the dating system at run time. In order to allow extensions after the dating system becomes operational, the customization process must operate at the component instance level. As a consequence, customization mechanisms that require access to source code of component definitions are not acceptable.

(2) *Modular customization*. According to the separation of concerns principle, each extension must be implemented as a separate module that can be incrementally added to and removed from the core application.

(3) *System-wide and consistent integration of extensions*. Customizing the core system with a single extension goes beyond refining individual components, but involves a *system-wide* refinement of multiple core component instances at the same time. For example, integrating the authorization extension for the “inspect another agenda” collaboration involves refining the `DatingSystem` component with an authorization check using a third-party authorization service. When access is denied, the authorization service typically throws an exception, which must be handled robustly too. This requires an additional refinement, namely an appropriate exception handler around the component instance, representing the service requestor. Such a system-wide refinement process should of course be performed in a *controlled* and *consistent* (all or nothing) fashion.

(4) *Client-specific integration of extensions*. 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 other clients.

(5) *Context-specific and dynamic combination of extensions*. Often there is a need for a dynamic and

selective combination of extensions on a per collaboration basis. For example, the collaborations “inspect my agenda” and “inspect another agenda” should only be authenticated when the request originates from a distinct subnet, and authorization should only be applied for the collaboration “inspecting another agenda”. Dynamic combination of extensions is also required. For instance, the extension providing the group appointment service and an appropriate appointment strategy must be together applied for the collaboration “make group appointment”. Which subset of extensions must be combined together is dependent on *contextual information* historically related to the collaboration’s message flow in the client and core system (e.g. the subnet-address at which the collaboration “inspect my agenda” was initiated by a client request determines whether an authentication check should be applied or not; e.g. the client’s choice of appointment strategy: leisure or business).

(6) *Scalability of customization process.* The process of selective combination and system-wide integration of extensions must remain manageable as the number of core component instances and the number of candidate extensions increases.

(7) *Ease of use.* The success of the customization architecture depends on whether the extension programming, deployment and integration processes are easy to understand by normal software developers. The customization process as a whole should be intuitive for end-users. Reflective techniques such as Meta-Object Protocols (MOPs)[6] have known uses for customization, but they are complex and thus difficult to use for application programmers who are typically no experts in meta-programming.

### 3. High-level overview of Lasagne

In this paper we present our customization model Lasagne. Lasagne defines a platform-independent blueprint for a dynamic customization process of component-based (distributed) systems. Lasagne is a language-independent model that can be implemented on top of a class-based object-oriented programming language with an open implementation. We have implemented Lasagne on top of the concurrent object-oriented language Correlate [7] (using Correlate’s Meta-Object Protocol) and on top of Java (using load-time reflection [Renaud Pawlak; personal communication, 3]).

Lasagne supports a selective, non-invasive and consistent integration of system-wide extensions into a component-based application. We distinguish between three different phases in the Lasagne customization process: the phase of implementing an extension using a specific 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 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 method definitions 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).

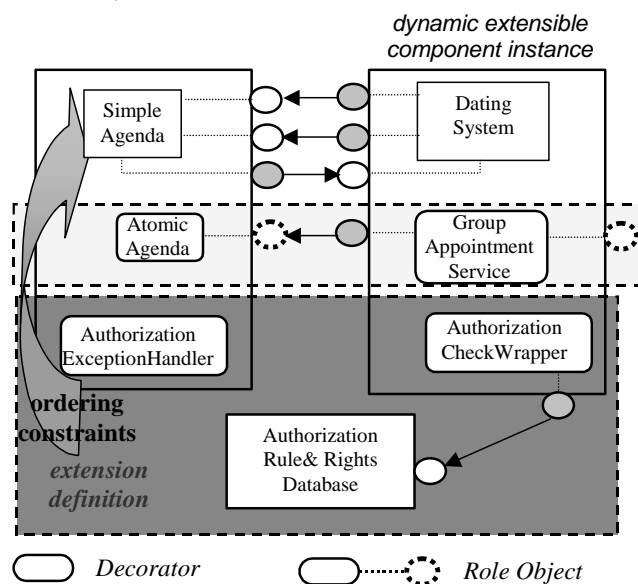


Figure 2: Extensions

#### 3.2. Deployment of extensions

At deployment time an extension definition is made available for possible later integration into the core system, by uploading the code of its wrapper definitions into a global code repository.

A *configuration manager* manages per extension information items that specify for each of its wrapper definitions around which core components they are to be decorated. For example (see Figure 2), adding the authorization extension involves wrapping the DatingSystem component with an AuthorizationCheckWrapper that performs an authorization check, using an access rights database. Secondly, each SimpleAgenda component instance is to be wrapped with an AuthorizationExceptionHandler wrapper, handling a potentially thrown AuthorizationException.

Each core component instance is watched over by a local *deployment configurator* who manages the “wrapper aggregate” of core component instance and its deployed wrapper definitions. A protocol for accepting new wrappers definitions from the configuration manager is defined. The deployment configurators are also responsible for dealing with semantic conflicts that potentially arise when different independent extensions are composed together that have hidden *ordering* dependencies in the way their wrappers are chained together. Therefore, each deployment configurator defines ordering constraints on how the wrappers must be chained around its local component instance; for example authentication before authorization, authorization before the leisure and business-specific extensions, etc.

### 3.3. Selective combination of extensions

Finally at run-time, after the core system becomes operational, extensions are selectively integrated into the core system on a per client-request basis. Since a client request initiates a collaboration between the component instances of the core system, the message flow of this collaboration must be adjusted to go through all the wrapper instances of the extensions selected for the client request, without violating the specified ordering constraints on chaining. In the next section we show that although statically chaining decorators is a good starting point for achieving this selective combination, it suffers heavily from scalability (challenge 6) and consistency (challenge 3) problems when used on top of traditional class-based object-oriented language platforms, such as Java and C++. In section 5 we then propose a more dynamic wrapping model solving these problems.

## 4. Analysis of the decorator pattern

The decorator pattern copes surprisingly enough with quite a lot of the challenges identified in section 2. First, decorators operate at the instance level (challenge 1). Second, decorators support modular customization of existing applications (challenge 2). Code needed for implementing an extension can be completely encapsulated into one or more decorators. Third, dynamic combination (challenge 5) is enabled with decorators since a decorator’s reference to its inner component can be dynamically changed, which may be another decorator again. This property makes decorators appropriate for defining extensions that can be uniformly composed with other extensions by statically chaining decorators conjunctively. Fourth, also client-specific integration and selective combination of extensions (challenge 4 and 5) could be supported by constructing several disjunctive

wrapper chains, wrapping the same component instance, as shown in the left part of Figure 3. Each wrapper chain would then be presented as a separate view on the component instance. Finally, ease of use (challenge 7) is well supported from a programming perspective, since the decorator is a widely known and well-documented design pattern [4].

Using decorators to customize applications has however not yet found much success in class-based object-oriented programming, due to problems with object identity [5]. In our work, the object identity problem translates itself in *scalability* problems of the customization process (challenge 6): disjunctive wrapping for achieving selective combination of system-wide extensions is complicated and unmanageable, even for small examples. Each disjunctive wrapper chain has an object identity of his own [4]. Outside objects must maintain the object references to these different chains, since for every method invoked on the component instance, the outside objects must *select* (challenge 5) through which wrapper chain the method invocation should go through. Maintaining this indirection is tedious and will lead to severe scalability problems when considering interaction refinement. A decorator that implements an interaction refinement invokes another component instance that may also have been wrapped with decorators from the same or other extensions. As a consequence, not only outside-objects but also decorators themselves may need to refer to the different wrapper chains of another component instance. This leads to a vast *spaghetti* of object references. Managing this spaghetti in the light of wrap updates is of exponential complexity.

This decorator maintenance problem makes it also difficult to realize a consistent integration of system-wide extensions (challenge 3). Building a coordination mechanism that governs a consistent and system-wide integration of the complete set of decorators belonging to one extension, without losing the ability to selectively combine extensions is almost impossible with the static wrapping model. This is because with static wrapping the selection logic, responsible for adjusting the message flow through all the wrappers of one extension, is scattered over and locked within the code of the entire application (the client and the set of core instances involved in the collaboration’s message flow), and thus difficult to control.

To eliminate the above problems we have to unify the decorators with the component instance into one identity. In other words, we have to be able to refine a component instance’s type or behavior without losing the component instance’s identity [5]. Secondly, we argue to specify the wrapper composition logic in one place and externally from the client and core system programs. A generic dispatch mechanism in the programming language implementation is needed that interprets this externally

specified composition logic.

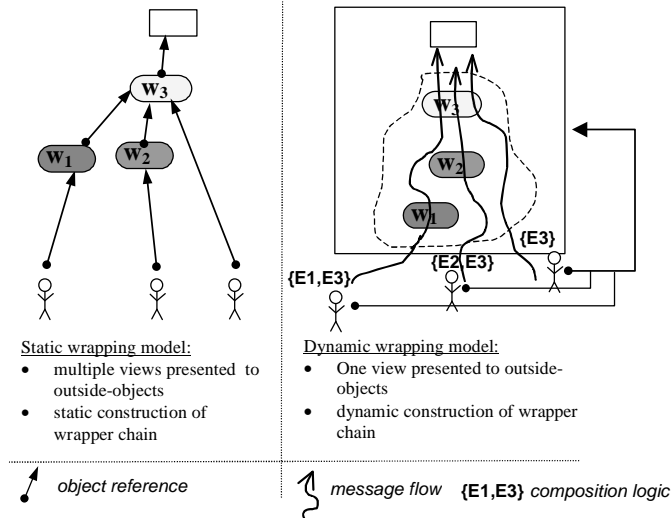


Figure 3: Static vs. Dynamic wrapping

## 5. A model of selective combination

By now it must be clear that the decorator pattern alone cannot provide a satisfactory fundament for the Lasagne customization process. In this section, we present a wrapping model of *selective combination of extensions on a per collaboration basis*. The novelty of this wrapping model is that the wrapper composition logic is encapsulated in a *composition policy* that is incrementally defined at run-time by *interceptors*. Such composition policy determines the way service requests flow within the system. We will discuss this now into detail and show how the scalability and atomicity problems are taken care of.

Lasagne is founded on five cornerstones. First, we introduce the notion of *component identity* that unites and hides the separate object identities of the component instance and its decorating wrapper instances. As such, component identity provides the uniform mechanism by which an outside instance can refer to the aggregate of component instance and decorating wrappers, using one single reference. As such, multiple disjunctive views on a component instance are not apparent anymore from a programmer’s point of view, taking away the spaghetti of references to wrapper chains. When a wrapper instance is decorated around a core component instance, the wrapper takes on the component identity of the aggregate. The object identity of the wrapper instance (defined by the hosting programming language platform) is only apparent within the boundaries of the aggregate.

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 within the core system. Therefore, we introduce the notion of an *extension identifier*, which is a string-based, 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 4).

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* (the subset of extension identifiers specifies the preferred *composition* of extensions according to a context-specific *policy*). A composition policy has a *propagating nature*: it travels together with the message flow of its associated collaboration.

Fourth, at any point in the message flow the composition policy can be updated by so called *interceptors*. That is, an interceptor intercepts incoming or outgoing messages of a specific component instance and dynamically updates the related composition policy. Interceptors typically implement a *client-specific* strategy that determines which extension identifiers are attached to which collaborations. In Figure 4, two interceptors intercept outgoing client requests. The one selects an appropriate appointment strategy (“leisure” or “business”) for making an extension. The other interceptor attaches the “group” extension identifier when the client invokes the dating system to make a group appointment. Interceptors can also use contextual information historically related to the collaboration. For example in Figure 4 a client is only authenticated when he makes a service request from a remote subnet location.

Fifth, the wrapping model supports a *dynamic* construction of the wrapper chain by adjusting message flow – releasing outside objects from the complex task of selecting between disjunctive wrapper chains. The intuitive difference in chaining is shown in Figure 3: it is the currently ongoing collaboration itself that searches its way through the appropriate wrapper chain based on inspection of its composition policy. To realize this dynamic chain construction a generic dispatch mechanism, called *variation point*, is introduced in the run-time model of a component instance. The variation point of a component instance interprets the composition policy of each incoming message and will only thread the message through those wrapper instances whose extension identifier is listed in the composition policy. The reader is deferred to [8] for implementation details of the variation point construct.

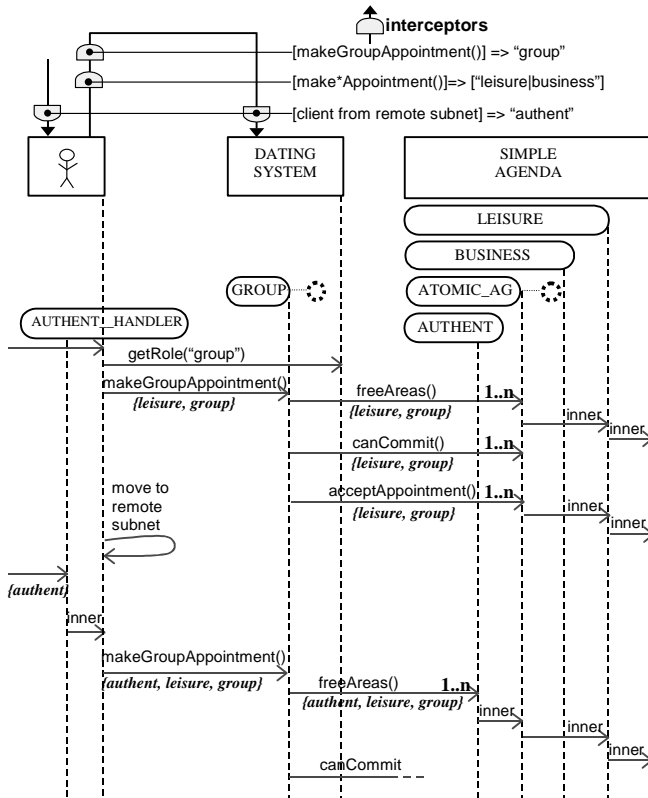


Figure 4 Illustrating Lasagne

Due to the propagating nature of a composition policy, an update in an interceptor triggers a *system-wide effect* on the composition of wrappers in all the parts of the core system that are involved in the control flow after the interceptor is applied. As such, interceptors encapsulate a *client-specific* customization of the *whole* core system, with the guarantee of *consistency*.

## 6. Conclusion

In this paper we have presented our customization model Lasagne. Lasagne defines a blueprint for a dynamic customization process of component-based (distributed) systems. We have implemented Lasagne on top of the language Correlate and Java. Future work involves an investigation of 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, which is less flexible but more cost-efficient than run-time combination. Finally, integration of an integrated security architecture governing who is allowed to do what in the Lasagne customization process (deploying extensions, implementing interceptors), is an open issue.

A parallel study on a design methodology or tool that supports and aligns well with the Lasagne customization process is also of interest to us.

An extended version of this paper, including a section covering related work can be found at <http://www.cs.kuleuven.ac.be/~eddy/research.html>.

## 7. Acknowledgments

This research was supported by a grant from the Flemish Institute for the advancement of scientific-technological research in the industry (IWT) and the A. P. Møller and Chastine Mc-Kinney Møller Foundation.

## 8. References

1. C. Atkinson, T. Kühne, C. Bunse, "Dimensions of Component Based Development", in Proceedings of the 4<sup>th</sup> International Workshop on Component-Oriented Programming.
2. D. Bäumer, D. Riehle, W. Siberski and M. Wulf, "Role Object". In Pattern Languages of Program Design 4 (Ed.: N. Harisson), Addison-Wesley, 2000, pp. 15-32.
3. S. Chiba, "Load-Time Structural Reflection in Java", in Proceedings of ECOOP'2000, 2000, Springer-Verlag LNCS 1850.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0201633612.
5. U. Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages, in Proceedings of ECOOP' 93, 1993, Springer-Verlag LNCS.
6. P. Maes, "Concepts and Experiments in Computational Reflection, in Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), 1987, pp. 147-155.
7. B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects*. Phd KULeuven, 1999. ISBN 90-5682-194-6.
8. E. Truyen, B. N. Jørgensen, W. Joosen, "Customization of Component-Based Object Request Brokers through Dynamic Configuration", in Proceedings of TOOLS Europe'2000, June 2000, IEEE, pp. 181-194.