

Dynamic and Selective Combination of Extensions in Component-Based Applications

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 is required in many application areas. We present a (distributed) application 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. In accordance to the separation of concerns principle, each extension is implemented as a layer of mixin-like wrappers. Each wrapper incrementally adds behavior and state to a core component instance from the outside, without modifying the component's implementation. The novelty of this work is that the composition logic, responsible for integrating extensions into the core system, is externalized from the code of clients, core system and extensions. Clients (end users, system integrators) can customize this composition logic on a per collaboration basis by 'attaching' high-level interpretable extension identifiers to their interactions with the core system.

1. Introduction

A component framework defines a semi-complete software architecture that has been carefully 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 normally not the most variable elements of a software architecture – the interactions between components are [2]. 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 system as a selective and dynamic combination of one or more *extensions* to a minimal core system. Each extension is implemented as a layer of mixin-like wrappers [4], simultaneously tailoring multiple components and their interactions between each other.

This work is inspired by the wrapper-based design patterns Decorator (refines the implementation of existing operations) [6] and Role Object (adds new operations) [3] which already support incremental extensions of a core component with multiple, independent client-specific views. These design patterns enable run-time customization since they operate at the instance level. On the negative side however, wrappers suffer from object identity problems in class-based object-oriented programming [8]. This leads to severe scalability and consistency problems when using wrappers for system-wide interaction refinements.

In this paper, we first illustrate our research goals by an example application. Then, we analyze the problems with object identity and propose a dynamic customization model called *Lasagne* 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. Finally we demonstrate *Lasagne* by applying it to the example application.

2. Analysis of challenges

Consider 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 shown in Figure 1.

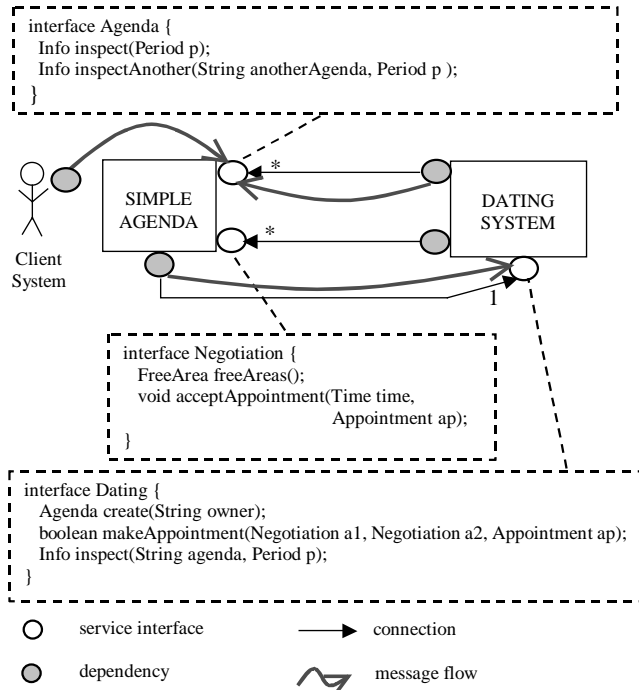


Figure 1: Minimal core of dating system

Before going into detail on this example, we first introduce our terminology related to component-based development from a programming and run-time point of view.

A *component* is a coherent unit of one or more classes. Each component has an explicit documented *component type*. The type of a component is defined by the set of interfaces it provides (its *service interfaces*) and the set of interfaces it depends on (its *dependencies*). For example the type of the SimpleAgenda component consists of the tuple (services = {Agenda, Negotiation}, dependencies = {Dating}).

Component instances are created dynamically. Just like normal objects, the constructor of the component definition is the first operation that is executed. A component instance encapsulates a set of objects.

A *connection* connects a dependency of one component instance with a type-equivalent service interface of another component instance. A connection may be implemented by a simple language pointer, or by a remote reference with the support of an Object Request Broker (ORB).

Component instances can only interact with each other through the interfaces, defined by their component type.

Component interaction happens by sending messages (invoking interface operations). Different models for message passing exist such as asynchronous, synchronous, and implicit invocation.

We call a component interaction between a client system 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 tree 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. 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.

2.1. Three challenges for customization

Suppose the dating system is to be deployed in a new client environment that has a list of additional requirements with regard to 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 also 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.

Many challenges are involved for reaching a solution that will cope with the above thoroughly.

(1) **Client-specific and modular customization.** A minimal requirement is to support different client-specific views on the core system [3][5][6][23]. Integrating the above extensions should after all not affect other client applications that are not interested in these extensions.

Furthermore, according to the separation of concerns principle each extension must be implemented as a separate module that can be dynamically added and removed without invasive change in the code of clients and core system [9][13][21][23][1][6].

(2) **System-wide and consistent interaction refinement.** Customizing the core system with a single extension goes beyond refining individual components, but involves a *system-wide* refinement of multiple core components at the same time. This is because integration of an extension often involves refining the *interactions* between multiple core components. For example (see Figure 2) the implementation of the atomic commit protocol for the group appointment service introduces a new slice of interaction behavior that crosscuts the entire core system. Refining the collaboration “inspect my agenda” with authorization involves applying an authorization check before processing the service request; here the exception of denied access propagates back to the service requestor and must be dealt there appropriately. Such system-wide refinements should of course be performed as a *consistent* and *atomic* (all or nothing) operation.

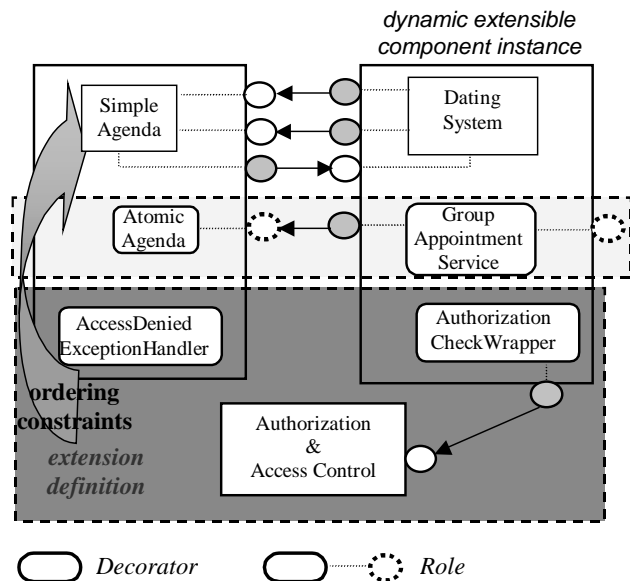


Figure 2: Wrappers and Extensions

(3) **Context-sensitive and dynamic combination of extensions.** The above example also illustrates the need for a dynamic and selective combination of extensions on a per collaboration basis. Which subset of extensions must be combined together is dependent on *contextual properties* [5] of the currently ongoing collaboration between client and core system, such as the particular service operation invoked (e.g. authorization checks should only be applied when inspecting another agenda), implicit client preferences dynamically selected per client request (e.g. the

client’s choice of appointment strategy: leisure or business) and the message flow history of the collaboration (e.g. the subnet location of the client; a client should only be authenticated when he/she issues requests from a remote subnet). Contextual properties change over time during the execution of the client programs (e.g. in the course of consecutively invoked service operations). In the presence of system-wide extensions, such a contextual change then requires run-time changes all over the core system to accommodate the evolved needs of the situation (e.g. in a mobile environment clients can move at runtime from one subnet location to another, possibly requiring to dynamically include/exclude the authentication extension to/from the core system).

3. Turning spaghetti into Lasagne

We implement an extension as a coherent module of mixin-like wrappers that work together (e.g. at multiple core components, at client and server side, etc.) to provide a refined or new core service. Wrappers support modular customization of existing applications. Code needed for implementing an extension can be completely encapsulated into one or more wrappers. The wrapper-based design patterns Decorator [6] and Role Object [3] already support client-specific and dynamic combination of extensions on a per component (interaction) basis.

A decorator supports incremental behavior refinement of existing operations. Since a decorator conforms to the service interfaces of its wrapped core component, it can be *uniformly composed* with decorators from other extensions by chaining them conjunctively.

A role object dynamically attaches new service interfaces to a core component, dynamically *widening its component type*. Multiple role objects can be composed without chaining. A role behavior can however be refined by a chain of decorators wrapping the role interface.

3.1. Problems with disjunctive wrapper chaining in object-oriented programming

Client-specific and selective combination of extensions could then be supported by constructing several disjunctive wrapper chains, as shown in the left part of Figure 3. However disjunctive wrapping in class-based object-oriented programming suffers heavily from *scalability* and *consistency* problems when considering selective combination of *system-wide extensions*. We shortly discuss the two problems here.

3.1.1. The spaghetti problem. The scalability problems arise from problems with object identity [8]: each disjunctive wrapper chain has an object identity of its own. Outside objects must maintain the references to these

different chains, since for every method invoked on the component instance, the outside objects must *select* 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 wrapper that implements an interaction refinement invokes another component instance that may also have been wrapped with wrappers from the same or other extensions. As a consequence, not only outside objects but also wrappers themselves may need to refer to the different wrapper chains of other component instances. This leads to a vast ‘spaghetti’ of object references. Managing this spaghetti in the presence of wrap updates is of exponential complexity.

To eliminate this problem we have to be able to refine a component instance’s type and behavior without losing the component instance’s identity [8]. In other words, we somehow have to unify the component instance and its wrappers into one logical identity.

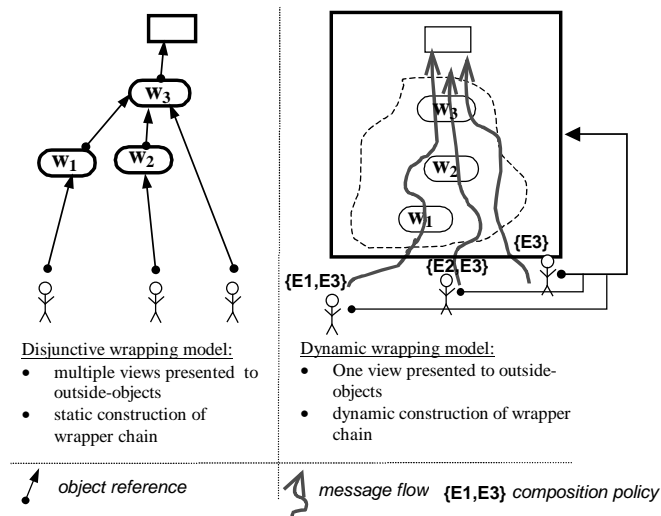


Figure 3: Disjunctive vs. Dynamic wrapping

3.1.2. The consistency management problem. The analysis of the second challenge in section 2 suggests that integration of an extension involves customization of an entire collaboration between multiple core component instances. For instance, in order to apply the authorization extension to a client request for inspecting another agenda (see Figure 2), the message flow of this collaboration must be redirected through the `AccessDeniedExceptionHandler` wrapper of the client’s `SimpleAgenda` instance *and* through the `AuthorizationCheckWrapper` instance of the `DatingSystem` instance. However, the existing wrapper-based design patterns only support customization of one single component (interaction) at a time. This difference in scale causes problems with keeping consistency in the presence of dynamic recombination of extensions: wrapper composition logic, responsible for adjusting a

collaboration’s message flow through all the wrappers of a selected extension is scattered across the code of consecutive component interactions of the collaboration, thus is difficult to update consistently. Furthermore, context-sensitive recombination of extensions (challenge 3) is even impossible: since class-based object-oriented programming does not provide a component interaction mechanism with propagation of contextual properties, wrapper composition logic inside the core system can simply not be context-sensitive. Finally, the problem of consistency management becomes even worse in the presence of partial order constraints between extensions (e.g. when one wrapper must be invoked before another) [15]. Partial order constraints must be enforced to cope with semantic conflicts that may arise when different extensions are composed together that are not orthogonal to each other due to implicit ordering dependencies.

To cope with these management problems we propose to specify the wrapper composition logic in one place, externally from the client and core system programs. A generic dispatch mechanism in the runtime component model is needed that interprets this externally specified composition logic, taking into account any additionally specified ordering constraints.

3.2. The Lasagne model

Lasagne defines a platform-independent architecture for dynamic customization of component-based systems using wrappers, coping with the above problems. It can be implemented on top of any programming language with an open implementation. 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. As such, component identity provides the uniform mechanism by which an outside instance can refer to the “aggregate” of a component instance and its decorating wrapper instances, using one single reference. As such, multiple disjunctive wrapper chains on a component instance are not apparent anymore from an outside point of view, taking away the spaghetti of wrapper references. When a wrapper instance is decorated around a core component instance, the wrapper instance 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 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 the extension providing the authorization service could simply be identified with the string “authorization”. 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. Interceptors can also inspect *contextual properties* (see section 2.1) 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. Due to the propagating nature of composition policies as well as contextual properties, an interceptor encapsulates a context-specific customization of an *entire* collaboration with the guarantee of *system-wide consistency*.

Underlying these four concepts, the run-time component model supports a *dynamic* construction of the wrapper chain by adjusting message flow. This releases 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 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 [24] for implementation details of the variation point construct.

4. Lasagne illustrated

Lasagne can be implemented on top of any class-based object-oriented programming language with an open implementation. As proof of concept, we have implemented Lasagne on top of the concurrent object-oriented language Correlate [18], using Correlate’s powerful Meta-Object Protocol (MOP). We recently also implemented Lasagne on top of Java, using Java Aspect Components (JAC) [R. Pawlak, personal communication].

In this section we will illustrate Lasagne on the dating system example (see section 2). Essential to the Lasagne customization process is that it distinguishes between three separate phases: (1) the phase of implementing an extension based on a specific wrapper-based programming model, (2) deployment/weaving of one or more extensions into the core system with specification of ordering constraints on wrapper chaining, and (3) selective combination of extensions on a per collaboration basis.

4.1. The extension implementation model

Two rules of thumb should be followed for implementing an extension. First, an extension may contain *at most one* wrapper definition per core component. Second, an extension must be *self-contained*: it must be closed over the group of wrapper definitions under dependency relations of all kinds. For example, a wrapper definition depending on a role interface of another core component, and the wrapper definition implementing this role interface must belong to the same extension.

An extension is uniquely identified with an appropriate extension identifier. Extension identifiers should be managed in a hierarchical namespace. For the sake of simplicity however we introduce for the dating system example the dummy extension identifiers “group”, “leisure”, “business”, “authent” and “authoriz”.

4.1.1. Wrapper specification. Each wrapper definition must first be specified such that it becomes manageable for later deployment and integration. In the Correlate implementation of Lasagne we specify this information in a Correlate deployment descriptor [19] associated with each wrapper definition. First, a wrapper definition is specified to be member of an extension by declarative binding to the corresponding extension identifier. Second, according to the decorator pattern there is a dependency between decorator and wrapped core component at the interface level [6]. In Lasagne, we deal naturally with this information at the component type level: we simply specify the minimal component type to which the decorator can be applied. Finally, a wrapper can also widen the minimal component type with new services (in case of a role) or new dependencies (in case of an interaction refinement).

```

WrapperDescriptor AtomicAgenda {
  property ExtensionIdentifier extension() {
    return (new ExtensionIdentifier("group"));
  }

  property String [] wrappedServices() {return new String[] {
    "examples.agenda.Negotiation";
  }
  property String[] wrappedDependencies {return null;}

  property String[] newServices() { return new String[] {
    "examples.agenda.extensions.group.Atomic";
  }
  property String[] newDependencies() {return null;}
}

```

4.1.2. Programming wrappers. The wrapper-programming model of Lasagne combines the benefits of both the Decorator and the Role Object design pattern. As such, Lasagne wrappers are programmed in a hybrid fashion. However, the relatively complex object structures required for building up decorators and role objects are not apparent anymore in the programming model of Lasagne, shifting complexity from programmer to programming language implementation.

A *decorator wrapper* performs additional actions before or after forwarding to a dynamically bound inner component, which is denoted by the “inner” parameter (see code example of the extension “authent” below).

```

package examples.agenda.extensions.authentication;

public class AuthenticationWrapper implements Agenda {
  AuthenticationServer aServer=NameServer.lookup("AuthenticationPKIServer");

  protected void authenticate(Context context) //internal method
    throws AuthenticationException {
    Signature signature = context.getProperty("signature");
    Certificate certificate = context.getProperty("certificate");
    aServer.authenticate(signature, certificate);
  }

  public Info inspect( Period period, Context context)
  {
    authenticate(context);
    return inner.inspect(period);
  }

  public Info inspectAnother (String another, Period period, Context context)
  {
    authenticate(context);
    return inner.inspectAnother(another, period);
  }
}

```

The inner parameter of a wrapper is bound at run-time, somewhat similar to the “super” parameter of mixin-based inheritance, which is bound at class instantiation time [4]. It is exactly this feature of the wrapper-programming model that corresponds with the dynamic construction of the wrapper chain: in the run-time component model the inner parameter of a wrapper instance points to the encapsulating component identity. The component identity encapsulates the aggregate of core component instance and all its deployed wrappers. Then (see Figure 4), when a wrapper forwards to inner, the variation point dynamically evaluates inner (i.e. the component identity) to one of the instances

from this aggregate by interpreting the composition policy of the currently ongoing collaboration.

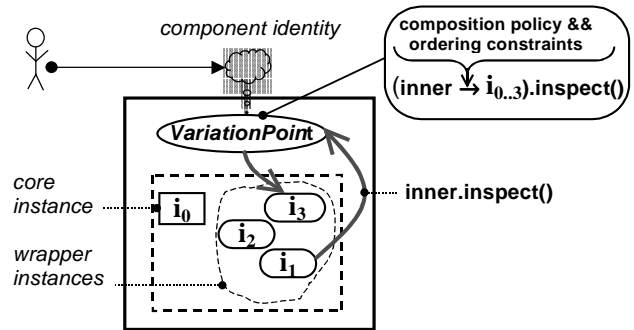


Figure 4 Dynamic wrapper chaining

Wrappers have also access to the *contextual properties* of the currently ongoing collaboration. The variation point provides by default the contextual properties as an additional argument of type Context with every method invoked on a wrapper. This is utterly useful for providing wrappers with necessary information about the calling client. For example implementing the AuthenticationWrapper relies on client-specific information such as the client’s digital signature and certificate. These client-specific items were previously attached to the collaboration by the ClientAuthenticationHandler wrapper (see code below), which is to be wrapped around a component instance representing the client.

```

package examples.agenda.extensions.authentication;

public class ClientAuthenticationHandler implements ClientInterface {

  public void clientOperation(Context context) {
    context.addProperty("signature", sign);
    context.addProperty("certificate", cert);
    try {
      inner.clientOperation();
    }
    catch(AuthenticationException ex) {
      //client-specific exception handling
    }
  }
}

```

In case authentication fails the AuthenticationWrapper throws an AuthenticationException (extends RuntimeException). This requires wrapping existing clients with an appropriate exception handler (see code ClientAuthenticationHandler). Note that although this compiles, it is a naïve solution: throwing an undeclared exception violates the contract between client and core service and may leave the calling client instance in an inconsistent and incomplete state.

A *role (wrapper)* instance attaches a new *service interface* to a core component instance (e.g. see code of AtomicAgenda wrapper below). However, role requestors (outside instances that depend on this role of the component instance) need to have access to such a role instance. The

Role Object design pattern solves this by passing a specification object to the core component instance to request a role from it; the role instance that matches the specification is returned [3]. Lasagne adopts the same mechanism but passes naturally the extension identifier of the role instance as specification object. This provides a scalable solution under provision of the second rule of thumb, which requires that a role definition and dependent role requestor must belong to the same extension identifier. In the Correlate implementation of Lasagne a primitive *getRole()* is defined on the variation point for requesting a role (see code of GroupAppointmentService below).

```
package examples.agenda.extensions.group;

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;

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 (!(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);
            }
        }
    }
}
```

4.2. 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. In the Correlate implementation of Lasagne we deal with both issues by specifying a separate deployment descriptor on a per component basis, called *ComponentDescriptor*. For

example, the component descriptor of SimpleAgenda is shown below. The “decorators” property lists the wrapper definitions that should be decorated around SimpleAgenda instances. The list defines a partial order on the wrappers, for example authentication before authorization. Sometimes it may be necessary to specify the ordering constraints on a more fine-grained scale, such as per interface method due to different orderings for different methods. For example, one might have a core component with methods start() and end() as its service interface and two wrappers: the one decorates the component such that a new method openTheDoor() should be invoked before start() and closeTheDoor() before end(); the other one decorates the component such that a new method goInside() should be invoked before start() and goOutside() before end(). Clearly the only sensible sequence of events is openTheDoor(), goInside(), start() and goOutside(), closeTheDoor(), end() which corresponds to different orderings for different methods [16]. To support such fine-grained ordering constraints, we simply specify the decorator property on a per method basis by means of a method property [19].

The variation point uses these ordering constraints to determine which wrapper must be chained next. Note that ordering alone provides no sufficient solution against semantic conflicts. Conflicting situations may still arise due to many other kinds of hidden dependencies of extensions.

```
ComponentDescriptor SimpleAgenda {

    property String[] services() {return new String[] {
        "examples.agenda.Agenda",
        "examples.agenda.Negotiation"};
    }

    property String[] dependencies() {return new String[] {
        "examples.agenda.Dating"};
    }

    property String[] decorators() {return new String[] {
        "examples.agenda.extensions.authentication.AuthenticationWrapper",
        "examples.agenda.extensions.authorization.AccessDeniedExceptionHandler",
        "examples.agenda.extensions.group.AtomicAgenda",
        "examples.agenda.extensions.strategy.LeisureAgenda",
        "examples.agenda.extensions.strategy.BusinessAgenda"};
    }

    methodproperty String[] decorators() {...}

    property InterceptorDescriptor[] beforeInterceptor() {
        return new InterceptorDescriptor[] {
            new InterceptorDescriptor("examples.agenda.Agenda", new AuthenticationSelector()),
            new InterceptorDescriptor("examples.agenda.Agenda", new AuthorizationSelector())
        };
    }

    property InterceptorDescriptor[] afterInterceptor() {return null;}
}

}
```

4.3. Selective combination per collaboration

It is important to realize that the above component descriptor only specifies the points in the core application where wrapper instances must be deployed, without however turning the switch adjusting the message flow to go through these wrappers. The “turning of the switch” is under control of so-called *interceptors*. As explained in

section 3.2, an interceptor intercepts collaborations and updates their attached composition policy in a context-specific fashion. An interceptor can only select between extensions that have been deployed before.

Interceptors are deployed around appropriate component instances (which are often the client instances and front-ends of the core system). They intercept either outgoing messages from a specific dependency or incoming message on a specific service interface. This mapping between services/dependencies and interceptors is specified in the component descriptors too (see example descriptor above).

Interceptors are programmable entities that have a simple interface for intercepting component interactions of their co-located component instance. In the Correlate implementation of Lasagne, a component interaction is intercepted as a reified object that has data slots for propagating the composition policy and the contextual properties (e.g. the subnet address from which a client request originates, the client’s digital signature) with it. 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 system integrator has decided that only client requests for making an appointment, which originate from a remote subnet, should be authenticated.

```
package client.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"),localHost()))
                policy.addExtensionIdentifier("authent");
        }
    }
}
```

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”.

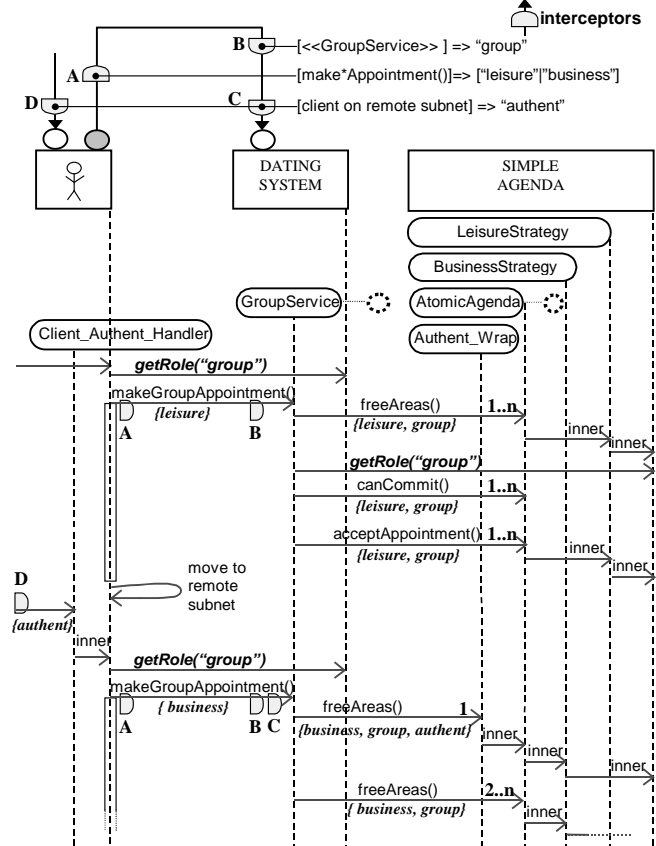


Figure 5: Message Sequence Diagram

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*.

5. Related work

The discussion on related work is organized according to the related fields that our work touches.

5.1. Advanced separation of concerns

State-of-the-art separation of concerns techniques such as Aspect-oriented Programming [9], Hyperspaces [23], Mixin Layers [21], and Adaptive Plug and Play Components [13] 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 context-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, while the above class-based techniques are better suited for providing separation of concerns at the component implementation level.

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 [17] 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).

5.2. Dynamic behavior composition

Linda Seiter et al. [20] 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 [12] 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 [13] more dynamic [7].

5.3. Distributed component systems

OMG CORBA, Enterprise Java Beans, (EJB) and Microsoft's component technologies (DCOM, COM+)[22] offer infrastructure for composing non-functional services with a core application. However, none of these infrastructures addresses *all* three challenges of section 2.1.

5.4. Connectors and configuration languages

In Regis [11] and ArchStudio [14], systems are constructed from a number of components and connectors that encapsulate the interactions between these components. The emphasis of these works is on the description, reconfiguration and evolution of the application's architecture. Connectors help to decouple components from one another and the systems use configuration languages to describe the configuration of the components. The run-time structure of the application is altered by applying a program written in the configuration language to the current architecture, thus generating a different arrangement of components and connectors. Our goal differs from these works in that we focus on customization of a system to client-specific needs, instead of coping with run-time software evolution in general. Furthermore, our customization process is based on a system-wide additive refinement of existing components, rather than replacing components with new ones and switching connectors.

6. 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 context-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.

This work is fairly young. No real-world evaluation of the usability and scalability of Lasagne has been performed yet. We believe however that development of interceptors can easily be automated by appropriate tool support. Such a tool can be considered as a generic interceptor with a graphical user interface for updating composition policies.

The wrapper-based approach by itself has a number of known limitations. It cannot achieve call-site composition non-invasively, wrappers can only access members that are visible in the public interface of a component, etc. However, these do not really conflict with the goals of

Lasagne. More importantly 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.

7. Acknowledgments

This research was supported by a grant from the Flemish Institute for the advancement of scientific-technological research in the industry (IWT). We would like to thank the anonymous reviewers for their very useful comments.

8. 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] C. Atkinson, T. Kühne, C. Bunse, "Dimensions of Component Based Development", in *Proceedings of the 4th International Workshop on Component-Oriented Programming*, C. Szyperski (ed.), 1999.
- [3] D. Bäumer, D. Riehle, W. Siberski and M. Wulf, "Role Object". In *Pattern Languages of Program Design 4*, N. Harisson (ed.), Addison-Wesley, 2000.
- [4] G. Bracha and W. Cook, "Mixin-based inheritance", in *Proceeding of OOPSLA/ECOOP '90*, October 1990.
- [5] 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.
- [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] J. Magee, N. Dulay, and J.Kramer, "Regis: A Constructive Development Environment For Distributed Programs", in *Distributed Systems Engineering Journal*, 1(5), 1994.
- [12] M. Mezini, "Dynamic Object Evolution without Name Collisions", in *Proceedings of ECOOP'97*, 1997.
- [13] M. Mezini and K. Lieberherr, "Adaptive Plug and Play Components for Evolutionary Software Development", in *Proceedings of OOPSLA '98*, 1998.
- [14] P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-based Run-time Software Evolution", in *Proceedings of ICSE'98*, 1998.
- [15] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal, Specifying Subject-Oriented Composition, in *Theory and Practice of Object Systems*, 2 (3), Wiley & Sons, 1996.
- [16] K. Ostermann, G. Kniessel, "Independent Extensibility – an open challenge for AspectJ and HyperJ", position paper for the *ECOOP'2000 Workshop on Aspects and Dimension of Concerns*, C. V. Lopes (ed.), 2000.
- [17] 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.
- [18] B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects*, Phd KULeuven, 1999, ISBN 90-5682-194-6.
- [19] B. Robben, B. Vanhaute, W. Joosen, P. Verbaeten, "Non-Functional Policies", in *Proceedings of the Second International Conference on Metalevel Architectures and Reflection*, Springer-Verlag LNCS 1616, July 1999.
- [20] L. Seiter, J. Palsberg, and K. Lieberherr, "Evolution of Object Behavior using Context Relations. In *IEEE Transactions on Software Engineering*, 24(1), 1998.
- [21] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", in *Proceedings of ECOOP'98*, 1998.
- [22] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1996.
- [23] P. Tarr, H. Ossher, W. Harrison, S. Sutton Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", in *Proceedings of ICSE'99*, 1999.
- [24] 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.