

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 N. 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.

Keywords

Component-based S.E, Software Evolution, Customization

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 [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 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 [6], simultaneously tailoring multiple components and their interactions between each other. Using wrappers has three main benefits. First, wrappers operate at the instance level. This is utterly useful in distributed applications, where a component instance may have several remote clients, each with different customization needs. With wrapping each client-specific customization simply corresponds to another wrapper instance. Second, decorators support modular customization of existing applications. Code needed for implementing an extension is completely encapsulated into the wrapper. Third, it offers an easy programming model for customization of applications. Reflective techniques such as Meta-Object Protocols (MOPs)[11] 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.

However, using object-based 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 [7]. In this paper, we first illustrate our research goals by an example application. Then, we analyze the object identity problems and propose a component-based wrapping model called *Lasagne* solving these problems. The novelty of this model is that it allows

selective integration of extensions on a per collaboration basis, enabling easy client-specific customization. Finally we demonstrate Lasagne by applying it to the example application.

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

A *component (definition)* is a coherent unit of one or more classes. Each component has an explicit documented *component type*. The type of a component is explicitly specified as the set of interfaces its classes provide (its *service interfaces*) and the set of interfaces its classes depend 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. These objects cannot be shared between component instances.

A *connection* connects a dependency interface of one component instance with a type-equivalent service interface of another component instance. A connection may be implemented as a simple language pointer, or as 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 tree, of which the nodes are component instances and the edges (represented by the curly arrows) 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 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.

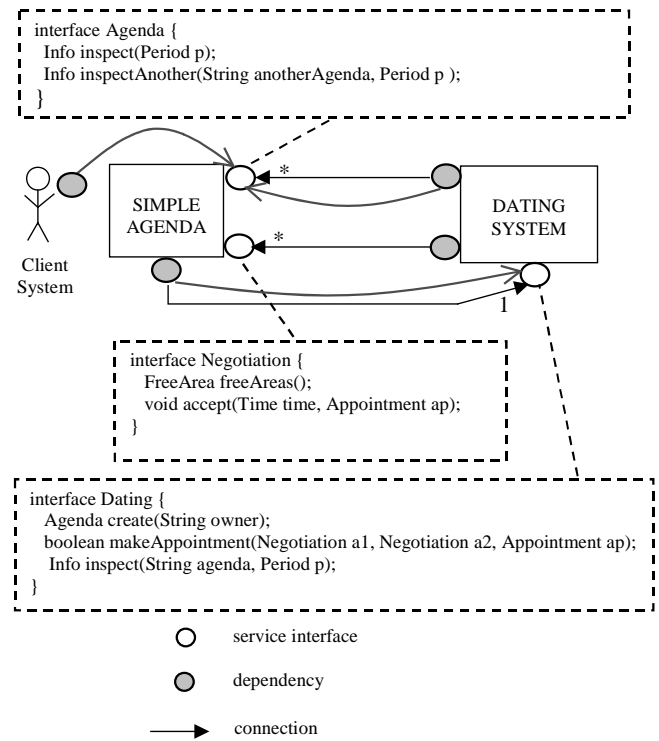


Figure 1: Minimal core of dating system

New Requirements

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 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.

Challenges

Many challenges are involved for reaching a solution that copes with the above thoroughly. We list what are – in our opinion – the challenges to be tackled and present these challenges in an order of most importance with respect to the subject of this paper.

Dynamic and Selective Combination of Extensions

The above requirements illustrate that for different collaborations a selective subset of extensions should be applied. 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”. These examples clearly demonstrate the need for a *selective* and *dynamic* combination of extensions on a *per collaboration basis*. Which subset of extensions must be combined together is dependent on *contextual information* related to that collaboration, such as the subnet-address from which the collaboration was initiated by a client request.

Interaction Refinement and Type Widening

In component-based development, customizability is achieved by replacing components. Components are however not normally the most variable elements of a software architecture – the interactions between components and the type of components are [2]. For almost all extensions of the above example, introducing these extensions involves *refining the interaction behavior* of core components. An interaction refinement may require that additional component instances with new types are introduced into the system, or even widening the type of core component instances (i.e. adding a new interface/dependency to a component). For example, to refine the collaboration “inspect my agenda” with authentication it involves creating a security component instance which implements an appropriate authentication algorithm, then widening the type of the SimpleAgenda component instances with a dependency to the Authentication service interface of this security component instance, and finally refining the interaction behavior of the SimpleAgenda component instances such that the authentication check is performed before executing the method Agenda:inspect().

Non-invasive Change and Modularity

Extensions should be incrementally added without modifying the source code of the core application as well as the client code. The extension itself must be implemented as a coherent module that can be incrementally added to the

core application.

3 TURNING SPAGHETTI INTO LASAGNE

We implement an extension as a coherent module of mixin-like wrappers[3]. We program a wrapper more or less as a component-based decorator [6] that may attach additional interfaces, state and refined (interaction) behavior to a dynamically bound inner component instance. An extension may consist of several wrappers that are logically united by a *declarative binding*.

An extension is incrementally added (not yet integrated) to the core system by decorating each of its wrappers around the appropriate core component instances. For example (see Figure 2), adding the authorization extension involves wrapping the DatingSystem component with an AuthorisationCheckWrapper instance that performs an authorization check, using an access rights database. When access is denied, an AuthorisationException is thrown. This makes necessary that also each SimpleAgenda instance is wrapped with an AuthorisationExceptionHandler wrapper instance, which handles the authorization exception in a manner most suitable for that specific agenda component instance.

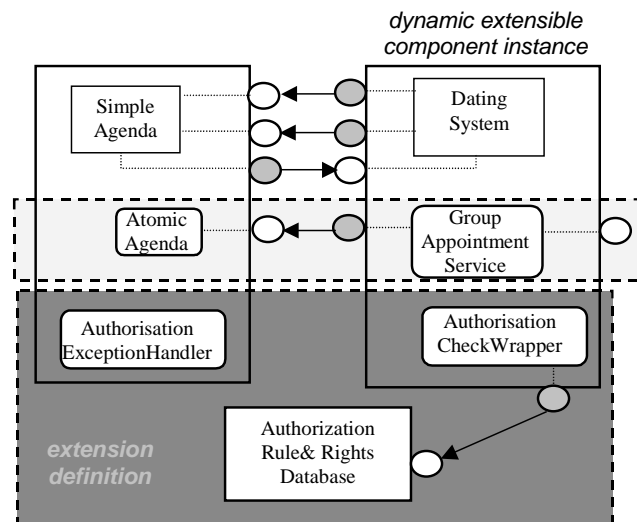


Figure 2: building & deploying extensions

Dynamic composition is enabled with wrappers since a wrapper’s reference to its inner component can be dynamically changed, which may be another wrapper again. This property makes them appropriate for defining extensions that can be uniformly composed with other extensions by statically chaining wrappers conjunctively, as shown in the left part of Figure 3.

Problems with the Static Wrapping Model in Object-oriented Programming

Selective combination of extensions could be realized by constructing several disjunctive wrapper chains per component instance. Each wrapper chain would then be

presented to the outside world as a separate *view* on the component instance. However this approach is completely not feasible using class-based object-oriented programming. This is because one cannot rely on the object identity of wrapped component instances anymore, resulting in two different problems:

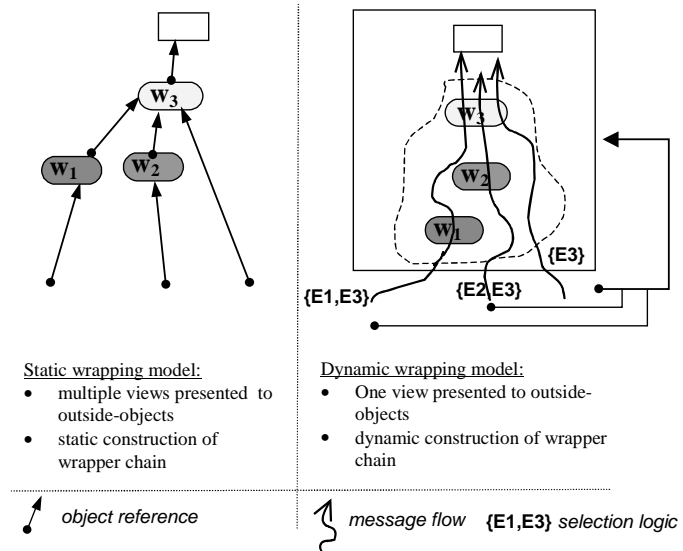


Figure 3: Static vs. Dynamic wrapping

The Spaghetti Problem

From an object identity point of view, each decorator has an identity of his own [6]. This means that N different views on the same component instance result in N different object identities presented to the outside environment. Outside objects must know about these objects, since for every method invoked on the component instance, the outside objects must *select* through which view the method invocation should go through.

Managing this is a complicated task, with many pitfalls. For example, when passing a component reference from out-side object A to object B, it is not clear how the view(s) in which B is interested can be passed – and in one atomic operation. Another problem is when an already operational component instance needs to be decorated with a new view; here, it is not known in the general case which other outside objects have references to this component instance and which of these objects need to have access to the newly created view. Furthermore, in our case the problems are aggravated, since almost all wrappers implement an interaction refinement, invoking another component instance that may also have been decorated with multiple views. As a consequence, not only outside-objects but also wrappers themselves may need to refer to the views of other component instances. This leads to a vast *spaghetti* of object references. Managing this spaghetti in the light of view updates is of exponential complexity. So this would make using the static wrapping model for our purposes

complicated and unmanageable, even for relatively small examples. To eliminate this problem we somehow have to unify the wrappers with the component instance into one identity. In other words, we have to be able to refine a component instance’s type and behavior without losing the component instance’s identity [7].

The Consistency Management Problem

An extension involves *multiple* wrappers, each to be decorated around *different* (distributed) component instances. As a consequence, we need a mechanism that guarantees a *system-wide* and *consistent* adjustment of the *message flow*, such that the wrappers part of that extension are *all* applied to any relevant collaboration. For instance, in order to apply the authorization extension to a client request for inspecting another agenda, the message flow of this collaboration must go through the AuthorizationExceptionHandler wrapper of the client’s SimpleAgenda instance and through the AuthorizationCheckWrapper instance of the DatingSystem instance.

Building a system-wide coordination mechanism that governs such a consistent integration, without loosing the ability to selectively combine extensions is almost impossible with the static wrapping model in class-based object-oriented programming. This is because the composition logic, responsible for adjusting the message flow through all the wrappers of one extension, would be scattered over and locked within the code of the entire system, and thus difficult to control. The scattering of this composition logic is a direct consequence of the previously identified spaghetti problem.

Note that preserving consistency when customizing multiple component instances at the same time is a problem in the general case, with or without wrappers.

The LASAGNE Model

We propose a new programming model and run-time architecture, called Lasagne (depicted in Figure 4) that deals with the above problems. Lasagne can be implemented on top of a class-based object-oriented programming language with an open implementation. The Lasagne model is founded on four cornerstones:

First, we augment the object-oriented programming model with the notion of *component identity* that unites and hides the separate 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 wrapper references, as illustrated in Figure 4.

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 the extension providing the authorization service, could simply be identified with the string “authorization”. As such, the extension identifier logically relates a number of wrapper implementations into one coherent unit by declarative binding.

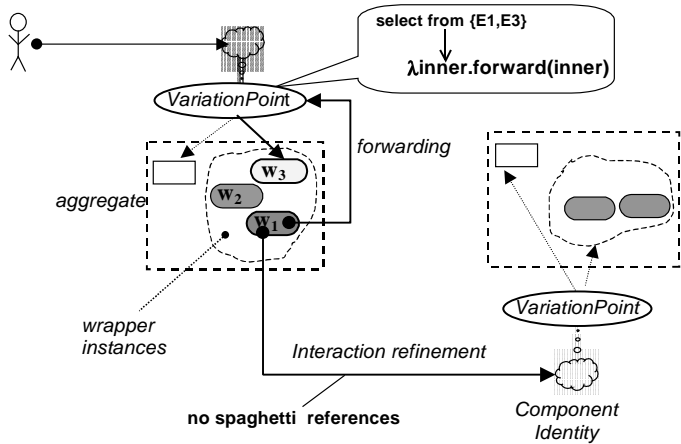


Figure 4 The Lasagne model

Third, we make the specification of wrapper composition logic external from the code of the core system, extensions and clients. Clients (end users, but also system integrators) can dynamically “attach” a subset of extension identifiers to relevant collaborations at any execution point in the core system, indicating which extensions should be applied for that collaboration. This delivers the mechanism to express selective combination of extensions on a per collaboration basis. In the remainder of this paper we refer to such an attached subset of extension identifiers as the *composition policy* of the collaboration. (i.e. the subset of extension identifiers specifies the preferred *composition* of extensions according to a client-specific *policy*.) Composition policies may be declaratively specified, but can also be dynamically built and attached, based on contextual information of the currently ongoing collaboration. Once the composition policy is attached to a collaboration, it is automatically propagated through the system as the execution of its collaboration advances.

Fourth, the wrapping model supports a *dynamic* construction of the wrapper chain by adjusting message flow - instead of selecting between statically constructed 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 wrappers by inspecting its composition policy”. In order to implement this dynamic chain construction, we introduce an additional abstraction, called *variation point* (see Figure 4), between the component identity on the one hand, and the aggregate of core instance and wrapper instances on the other hand. The variation point is a generic “wrapper-inner” construct that concatenates wrapper instances through dynamic evaluation of the inner parameter (similar to the λ -calculus). It is only when a wrapper *forwards* to inner, that an effective instance is dynamically selected from the aggregate for subsequent inclusion in the wrapper chain. This dynamic selection process is implemented by the variation point and is driven by the composition policy of the currently ongoing collaboration. Only these wrapper instances whose extension identifier is listed in the composition policy will be selected for inclusion in the wrapper chain. The reader is deferred to [8, 25] for implementation details of the variation point construct.

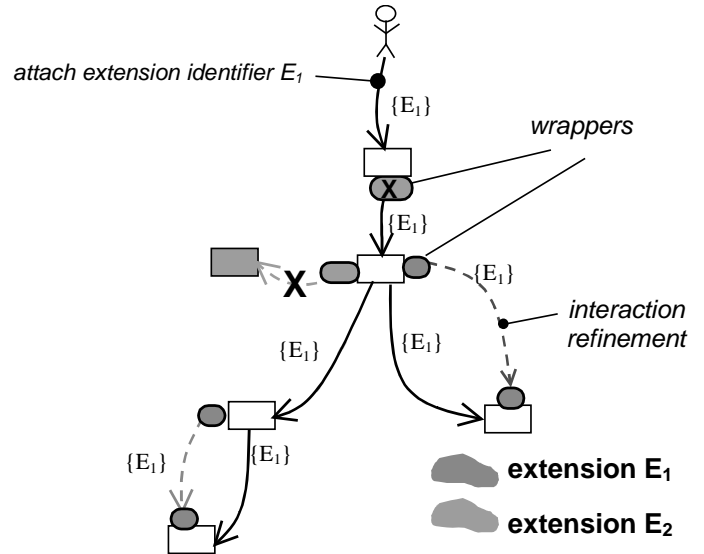


Figure 5: Dynamic and selective composition of extensions

Figure 5 illustrates the customization model from a message flow perspective. Once a composition policy is attached to a collaboration, it travels together with the collaboration from one variation point to another. This provides the mechanism for a system-wide and consistent adjustment of the message flow, solving the above mentioned consistency management problem, without losing the ability to selectively combine extensions. The composition logic travels (in the form of a composition policy) together with the collaboration, rather than being locked up and scattered across the application code.

4 APPLYING LASAGNE TO THE DATING SYSTEM

As proof of concept, we have implemented Lasagne on top of the programming language Correlate[18], developed at DistriNet labs. Correlate is a Java-based concurrent object-

oriented language with a meta-level architecture for building distributed applications. Correlate's MOP provides support for controlling object identity and the interaction behavior of objects.

The Extension Programming and Deployment Model

The previous section has only described the essential concepts of the Lasagne model. In this section we will illustrate Lasagne on the dating system example (see section 2) while filling in yet left-open issues of Lasagne such as the extension programming model, valid composition of extensions and component type widening. The next section describes a mechanism for attaching composition policies to collaborations.

Defining Extension Identifiers

Each extension must first be uniquely identified with an appropriate extension identifier. Extension identifiers should be managed in a hierarchical namespace. For the sake of simplicity, we introduce for the dating system example the dummy extension identifiers "group", "leisure", "business", "authentication" and "authorization".

Documenting Wrappers

According to the decorator pattern [6], there is a dependency between wrapper and decorated core component at the interface level. In Lasagne, we deal naturally with this dependency at the component type level. We specify the 'wrap capabilities' in a Correlate deployment descriptor [19] associated with the wrapper.

Below is given the wrapper descriptor for the AtomicAgenda wrapper. First, we specify the minimal component type to which the wrapper can be applied. Secondly, we specify whether a wrapper adds new services or dependencies to the wrapped component type. Finally, the extension identifier to which the wrapper logically belongs is also specified.

```
WrapperDescriptor AtomicAgenda {
  property String [] wrappedServices() {return new String[] {
    "examples.agenda.Negotiation";
  }

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

  property String[] newDependencies() {return null;}

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

Forwarding to the "inner" Parameter (No Delegation)

When programming a wrapper, the inner component of the wrapper is denoted by the keyword "inner" (see code example of the extension "authentication" below). The "inner" parameter is dynamically bound at run-time,

somewhat similar to the "super" parameter of mixin classes, which is bound at class instantiation time [3]. It is exactly this feature of the wrapper-programming model that corresponds with the dynamic construction of the wrapper chain.

```
package examples.agenda.extensions.authentication;

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

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

package examples.agenda.extensions.authentication;

public class ClientAuthenticationHandler implements ClientInterface {

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

Wrappers perform additional actions before or after forwarding (and not delegation) to "inner". The difference between forwarding and delegation is the binding of the *self* parameter in the inner instance, when called through the wrapper. With delegation, the self parameter is bound to the wrapper instance, with forwarding it is bound to the inner instance [10]. The Lasagne wrapping model does *not* need to rely on delegation, since all instances in the Lasagne space (component as well as wrapper instances) per definition only invoke interface methods on their explicitly specified dependencies (and the "inner" instance in case of wrappers) and thus never on their self parameter.

Programming with Contextual Information

A second distinctive feature of the wrapper programming model is that wrappers can inspect and manipulate the contextual information, historically related to the currently ongoing collaboration. The wrapper has access to this contextual information, since it is provided as an implicit argument with every interface method invocation. This is utterly useful for providing wrappers with necessary information about the client application or environment of

the core system. For example implementing the AuthenticationWrapper (see code example above) 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, which is to be wrapped around a component instance representing the client. Note that the latter wrapper also deals with possibly thrown RuntimeExceptions in a client-specific way.

Component Type Widening

Component type widening consists of incrementing a core component instance with a new service interface/dependency (and associated behavior). Incremental type widening is simply performed by decorating the component instance with a wrapper that implements that new interface (or uses the new dependency). For example, the GroupDating and AtomicAgenda wrappers illustrate this.

However, when incrementing a component instance with a new service, another issue arises: outside instances that depend on this new service need to have access to the wrapper instance, implementing the service. For instance, the GroupDating wrapper depends on the Atomic interface, and the new client of the core system may depend on the GroupService interface. As a consequence, the identity of the service widening wrapper instance must nonetheless be visible through the encapsulating curtain of the component identity.

```
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
    .....
}
```

We deal with this by providing a generic *type widening* operation (see code example of GroupDating wrapper). This operation simply returns a connection to the service-widening wrapper instance of the component instance in question. The appropriate wrapper instance can easily be retrieved based on its extension identifier.

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

public class GroupDating implements GroupService {

    public void makeGroupAppointment(Negotiation[] a, Appointment app) {
        FreeAreas[] frees = fetchFreeAreas(a);
        Time time = match(frees);
        for (i=0; i < a.length, i++) {
            Atomic atomAg = (Atomic)widenType(a[i], "group");
            if (!atomAg.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);
            }
        }
    }
}
```

Specifying Valid Composition of Extensions

We must also specify which wrappers must be decorated around which components without having semantic conflicts between extensions. Semantic conflicts potentially arise when different independent extensions are composed together that are not orthogonal to each other due to hidden ordering dependencies.

Specifying a separate deployment descriptor on a per component basis, called *ComponentDescriptor*, deals with the above. 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. Although ordering alone provides no sufficient solution to independent extensibility [23,10], the partial order helps to alleviate the problem of overlapping 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 InterceptorDescriptor[] beforeInterceptor() {
        return new InterceptorDescriptor[] {
            new InterceptorDescriptor("examples.agenda.Agenda", new AuthenticationSelector()),
            new InterceptorDescriptor("examples.agenda.Agenda", new AuthorizationSelector())
        }
    }

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

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

    property String[] dependencyAdapters() {return null;}
}
}
```

Sometimes it may be necessary to specify the “decorators”

property on a more fine-grained scale, such as per interface method, due to different orderings for different methods [16]. 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 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.

Building and Attaching Composition Policies

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*. Interceptors intercept collaborations and attach a client-specific composition policy to them. As explained above, composition policies deliver the mechanism for *expressing* selective combination of extensions on a per collaboration basis. Remember also that a composition policy travels together with its collaboration from variation point to variation point, which automatically interpret the composition policy. As such, interceptors *encapsulate* a *client-specific* customization of the *whole* core system, with the guarantee of *system-wide consistency*.

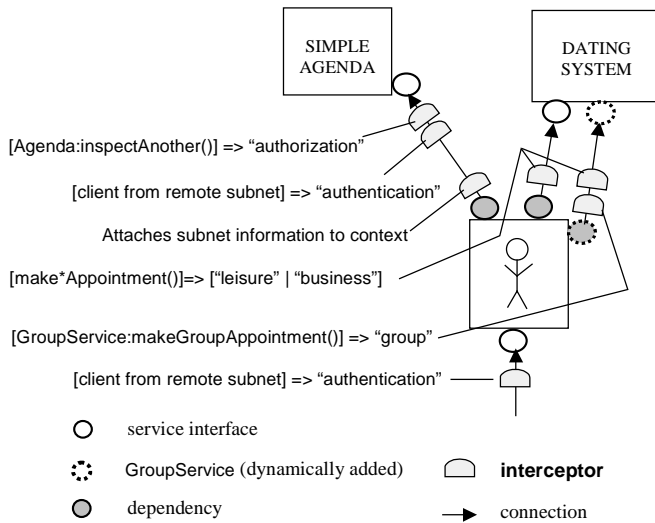


Figure 6: Deploying interceptors

Figure 6 shows a straightforward scenario for customizing the dating system with the described extensions. In this scenario, one has defined a separate interceptor for each one of the extensions. Interceptors are deployed around appropriate component instances (which are often the client instances and front-ends of the core system). They are

attached to a specific service or dependency of the co-located component instance. 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. A component interaction is intercepted as a reified object [11] that has slots for the composition policy and the contextual information (e.g. client’s digital signature, subnet address from which client request is originating) related to the currently ongoing collaboration. The composition policy and the contextual information of the currently ongoing collaboration propagate from one component interaction to the next as the collaboration advances.

Interceptors 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 determines whether authentication should be applied or not. For this example, the system integrator has decided that only client requests for inspecting another agenda that originate from a remote subnet should be authenticated.

```
package client.customization;

/** [Agenda.inspectAnother() and (client from remote subnet)] => "authentication" */
public class AuthenticationSelector implements Interceptor {

    public void manipulate(Interaction interaction) {
        /**identification of the collaboration**/
        String methodDef = getMethodDef(interaction);
        if (methodDef.equals("examples.agenda.Agenda:inspectAnother(.."))
        {
            /**Updating the composition policy of the collaboration based on:
            -> contextual information
            -> client-specific information **/
            CompositionPolicy policy = interaction.getPolicy();
            Context context = interaction.getContext();
            if ( distinctSubnet(context.getAttribute("clientHost"),localHost()))
                policy.addExtensionIdentifier("authentication");
        }
    }
}
```

The specification and attachment of composition policies can easily be automated by appropriate tool support. Such a tool can be considered as a generic interceptor with a graphical user interface for building and attaching composition policies.

5 RELATED WORK

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

Distributed Component Systems

OMG CORBA, Enterprise Java Beans, (EJB) and Microsoft’s component technologies (DCOM, COM+)[22] offer infrastructure for composing non-functional services (e.g. transactions, security) with a core application. They however do not address the challenges, described in section 2. Integration of non-functional services with the core

application either requires invasive change in the client application (CORBA), or sacrifices client-side customizability in favor of transparency towards the application code (EJB containers, COM+ interception).

Dynamic Behavior Composition

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 combinations of multiple extensions simultaneously.

Mezini [13] presented an object model that does well support dynamic composition of object behavior without name collisions. Similar to Lasagne it introduces an additional abstraction layer, called combination layer, between objects and classes. However, there is no support mentioned for specifying behavior composition on a per collaboration-basis.

Composition Filters [1] compose behavior on a per object interaction basis. Composition Filters however does not have any support for consistent and selective integration on a per collaboration basis; the composition logic enforcing the system-wide integration of a specific extension for a specific collaboration is scattered across multiple composition filters (namely, the CF's attached to the objects that are involved in that collaboration). In Lasagne, the composition logic is completely encapsulated within the composition policy of the collaboration.

Advanced Separation of Concerns

State-of-the-art separation of concerns techniques such as Aspect-oriented Programming[9], HyperSpaces [24], Mixer Layers[21], and Pluggable Composite Adapters[14] allow extension of a core applications with a new aspect/feature/layer/abstract collaboration, by simultaneously refining state and behavior at multiple points in the application in a non-invasive manner. These approaches however operate at the class-level, while we compose extensions at the instance-level. This makes Lasagne more flexible regarding the composition of multiple extensions; Lasagne enables selective integration of extensions at runtime and can define a valid order on the extensions at a very fine-grained scale. However, our approach suffers from a *run-time performance penalty*. Therefore we think that Lasagne is better suited for client-specific integration of (non-functional) extensions at the more *coarse-grained* architectural level of a core system; while the above class-based techniques are better suited for providing separation of concerns at the component

implementation level.

A-TOS [16] is an aspect-oriented reflective middleware for distributed programming. It provides so called aspect components that integrate system-wide properties such as security and authentication within an application in a consistent manner. Its goals are thus very similar to ours. A-TOS operates also at the instance level and provides some support for valid ordering of aspects. A-TOS however uses an event-based technique to integrate aspects non-invasively with the application and has no explicit support for collaborations. A-TOS is furthermore tightly linked to the TOS programming language, making A-TOS language dependent.

Connectors and Configuration Languages

In Regis [12] and ArchStudio[15], 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.

Wrapping Support in Programming Languages

Researchers at the university of Bonn have developed a class-based language called LAVA[5, 10] that separates the notions of object comparison and object reference, potentially providing a solution for the spaghetti problem.

Büchi et al. [4] proposes a language construct for strongly typed class-based programming languages, called generic wrappers, that offers the possibility to wrap objects with the guarantee of type transparency. They further analyze the problems related to the spaghetti problem, without however providing a general solution to it.

6 CONCLUSION

Lasagne is a language-independent customization architecture for component-based (distributed) system. It supports a client-specific, non-invasive and consistent integration of system-wide extensions into a component-based application.

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), 1993, Springer-Verlag, 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.
 3. G. Bracha and W. Cook, "Mixin-based inheritance", in Proceeding of OOPSLA/ECOOP '90, October 1990, 25(1), pp. 303-311.
 4. M. Büchi, W. Weck, "Generic Wrappers", in Proceedings of ECOOP'2000, June 2000, Springer Verlag LNCS 1850, pp. 201-225.
 5. P. Costanza, "Separation of Object Identity Concerns", position paper for the ECOOP'2000 workshop on Aspects and Dimension of Concerns,
 6. E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0201633612.
 7. U. Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages, in Proceedings of ECOOP' 93, 1993, Springer-Verlag LNCS.
 8. B. N. Jørgensen, E. Truyen, F. Matthijs, W. Joosen, "Customization of Object Request Brokers by Application Specific Policies", in Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000), 2000 Springer-Verlag, LNCS 1795, pp. 144-163.
 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, Springer-Verlag LNCS 1241, pp. 220-242.
 10. G. Kniessel, "Type-Safe Delegation for Run-Time Component Adaptation", In Proceedings of ECOOP'99, June '99, Springer LNCS 1628, pp.351-366.
 11. 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.
 12. J. Magee, N. Dulay, and J.Kramer, "Regis: A Constructive Development Environment For Distributed Programs", in Distributed Systems Engineering Journal 1, 5 (1994).
 13. M. Mezini, "Dynamic Object Evolution without Name Collisions", in Proceedings of the ECOOP'97 Conference.
 14. M. Mezini, L. Seiter, K. Lieberherr, "Component Integration with Pluggable Composite Adapters", in Software Architectures and Component Technology: State of the Art in Research and Practice, M. Aksit (ed), 2000, Kluwer Academic Publishers.
 15. P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-based Run-time Software Evolution", in Proceedings of ICSE'98, 1998, IEEE, pp. 177-186.
 16. K. Ostermann, G. Kniessel, "Independent Extensibility – an open challenge for AspectJ and Hyper/J", position paper for the ECOOP'2000 workshop on Aspects and Dimension of Concerns.
 17. R. Pawlak, L. Duchien, G. Florin, L. Martelli, L. Seinturier, "Distributed Separation of Concerns with Aspect Components", in Proceedings of TOOLS Europe'2000, June 2000, IEEE, pp. 276-287.
 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, July 1999, Springer-Verlag LNCS 1616, pp. 74-92.
 20. L. Seiter, J. Palsberg, and K. Lieberherr, "Evolution of Object Behavior using Context Relations. In IEEE Transactions on Software Engineering, 1998, 24(1), pp. 79-92.
 21. Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", in Proceedings of the ECOOP'98 Conference, July 1998, Springer-Verlag LNCS 1445, pp. 550-570.
 22. C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, ISBN 0-201-17888-5.
 23. C. Szyperski, "Independently extensible systems – software engineering potential and challenges", in Proceedings of the 19th Australian Computer Science Conference, 1996, Australian Computer Science Communications 18(1), pp. 203-212.
 24. P. Tarr, H. Ossher, W. Harrison, S. Sutton Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns. in Proceedings of ICSE'99, 1999, ACM Press, pp. 107-119.
 25. E. Truyen, B. N. Joergensen, W. Joosen, "Customization of Component-Based Object Request Brokers through Dynamic Configuration", in Proceedings of TOOLS Europe'2000, June 2000, IEEE, pp. 181-194.