

Consistency Management in the Presence of Simultaneous Client-Specific Views

Eddy Truyen, Wouter Joosen, Pierre Verbaeten
DistriNet, Dept. Computer Science
K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
+32 (0) 16327602
 {eddy, wouter, pv}@cs.kuleuven.ac.be

Abstract

This paper is about client-specific customization of systems that implement an on-line Internet service in the presence of simultaneous client-specific views. The problem is that each client must be able to customize the running system for use in its own context, without impacting the service behavior that is delivered to other clients. To solve this, we propose to customize the system on a per client request basis, where the system itself consists of a stable core and several extensions that are injected into the core as needed. However, this approach brings on its own several consistency management problems that must be dealt with in order to make the approach viable. We give an overview of these problems and present a management architecture that deals with these problems.

1. Introduction

With the Internet and the World Wide Web (WWW) a new trend has come up: *on-line distributed services* that are remotely accessible from everywhere in the world. There is a multitude of examples: e-commerce, reservation systems, banking systems, dating services, compose your own garden and we come build it for you, etc. Furthermore, the general acceptance of web service technology as supporting platform for enterprise application integration has significantly contributed to this trend: a growing number of enterprises are turning their internal business solutions into on-line services. This allows the development of new services by dynamically composing the existing portfolio of services.

This paper is about customization of the (object-oriented) systems implementing an online service. With customization we mean that the functionality of the system can be easily adapted to client-specific needs. There is a huge market benefit in highly customizable services. As such, for on-line services, the complexity of software has

definitely shifted from “construction” to “evolution” [1].

However, the character of the Internet introduces a research problem concerning customization of on-line services: a running system instance (the program of the system in execution) is used *at the same time* by hundreds or even thousands of different client systems, where each client system may have *different - possibly conflicting - customization needs* with respect to the functionality of the service. As a consequence, each client of the system instance must be able to customize the system instance for use in his/her own context, without affecting the service behavior that is delivered to other clients. We call this problem *system customization in the presence of simultaneous client-specific views*.

A naïve solution to solve this problem would be to maintain a pool of ‘clones’ of the same system instance, where all the clones perform actions on common data, but where each clone’s implementation is customized to the needs of a specific client. As such, for every client there would be a separate clone of the system instance. In its most simple form, this solution resembles the architecture of Enterprise Java Beans (EJB) where each client (session) is served by a separate session bean instance. Although session beans have been introduced for other purposes, nothing would prevent us from building an EJB server that maintains a pool of session beans that are each customized to the needs of a specific client. This ‘cloning’ approach is harder to maintain for Internet applications, however, given the potentially large number of simultaneous clients. Moreover, cloning is not sufficient for applications where the needs of an existing client evolve over time.

When a system instance cannot be cloned, we are faced with the problem of customizing its behavior at run-time. To support simultaneous client-specific views, we propose to customize the system instance on a per client request basis. Dynamic and client-specific injection (or ejection) of add-on features provides the necessary flexibility to support such a by-request customization. How does this work? We start out with a minimal *core system* whose functionality is

‘stable’ in the sense that it is invariant for all clients. Finding the stable core is not easy and requires a lot of knowledge of the application domain. However, we assume that a stable core system can be found. Furthermore there is a set of *extensions* where each extension implements an additional feature or new business rule to the core. Then we present the customization process of the system as a *run-time* and *selective combination* of extensions to the stable core *on a per client request basis*. As such, each client request enforces a by-request customization of the system where the system itself consists of a stable core and one or more extensions that are injected into the core as needed. Which extensions are injected in the core depends on the needs of the client that is currently making the request.

We have already presented elsewhere [25] the Lasagne component model that supports a run-time and selective combination of extensions on per client request basis.

1.1. Problem statement

Although a by-request customization seems to be the only possibility to support customization in the presence of simultaneous client-specific views, this approach presents several consistency management problems that must be dealt with in order to make the approach viable. In section 3 of this paper, we discuss the consistency management problems and define a management architecture on top of the Lasagne component model that solves these problems. In section 2 we give an overview of the Lasagne component model explaining only these features that are necessary to understand the issues addressed in this paper. We discuss related work in section 4 and conclude in section 5.

2. The Lasagne Model

In this section we first describe the software engineering goals that motivate the Lasagne model. Then we give a short overview of Lasagne.

2.1. Motivation behind Lasagne

In current object-oriented systems, making even minor extensions to software systems requires invasive changes to existing code and is complicated by the lack of locality. This makes extensions tedious to program, makes it difficult to ensure consistency and makes it extremely difficult to plug and unplug extensions at run-time. Therefore, a pre-requisite for the above customization process to work is that software systems have a modular structure such that extensions are compositional with this structure. In this respect, Lasagne is highly inspired by aspect-oriented programming (AOP) [11] research that works on providing linguistic mechanisms for programming extensions as modular units that would

otherwise ‘crosscut’ through multiple parts of existing code.

The Lasagne model defines a **non-invasive, aspect-oriented, per-instance, per-collaboration** component extension mechanism; the idea behind this mechanism is changing the behavior of the system without changing the code of core components, but instead encapsulating them within **wrappers** [6]. In turn, wrappers can be encapsulated in other wrappers if needed.

Wrappers are useful for customizing systems that implement an on-line service: an on-line component instance may have several remote clients during its lifetime, where each client needs different – possibly overlapping – subsets of features to be injected in the core system; with wrappers, each client-specific feature simply corresponds to another wrapper around the core component instance. Furthermore, the wrapper-based approach enables unanticipated, run-time, instance-level customization, joint use of different add-on features (either by disjunctive or conjunctive wrapping) [12].

Since wrappers operate at the instance-level, injection of extensions into a running system instance becomes feasible. This allows the behavior of the system to be adapted to dynamically evolving customization needs without shutting down the system. For certain application domains, such as the banking and telecommunication sector, this is crucial because these applications need to provide 7x24 hours availability.

Wrappers support a simple form of aspect-oriented programming. A crosscutting extension can be implemented in a modular way as a set of wrappers that work together at multiple core components. Each wrapper can be integrated around its respective core component without requiring invasive change in the code of the core component.

Selection of extensions happens per client request. Since a client request typically triggers a collaboration (a graph of messages) between multiple component instances of the core system, special attention must be paid to maintaining consistency: once an extension is selected, the decision must be coordinated so that the wrappers of that extension are dynamically invoked at the appropriate execution points of this collaboration.

However, the usefulness of wrappers in class-based object-oriented programming languages is limited by the underlying object model. The most important problem is that wrappers cannot be transparently injected between a client and server object: the client object has to temporarily discard its reference to the server object and use a reference to the wrapper object instead [9]. Of course, this switching between object references becomes very complex and difficult to maintain for large systems, especially when dynamically injecting/ejecting crosscutting extensions [25].

2.2. An overview of Lasagne

Lasagne defines a wrapper programming model and associated component model that realizes the above software engineering goals without needing any switching of object references. It can be implemented on top of any programming language or middleware platform with an open implementation.

We explain Lasagne by an example. Consider a distributed dating system, which manages a set of electronic agendas on behalf of a set of clients. The system implementation consists of two *components* as shown in Figure 1. The system only implements a stable core.

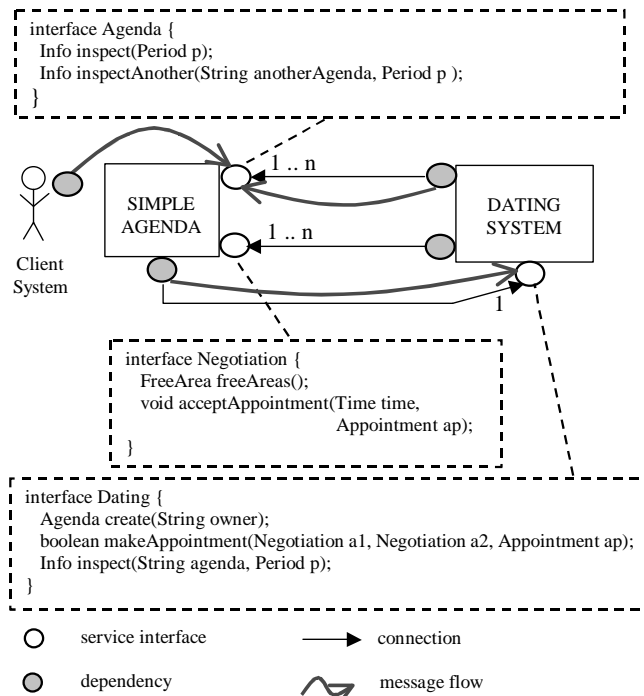


Figure 1: Minimal core of dating system

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 agendas.

Now suppose that clients can selectively inject the following extensions into this core dating system:

- A service for making group appointments between more than two agendas in an atomic fashion. As a consequence, agendas must be extended with atomic commit behavior.

- When making an appointment between agendas, different strategies for searching a period of time may apply. Which strategy must be used, depends on the kind of the appointment requested, for example whether it is meant for business or for leisure. The dating system must therefore be able to dynamically select between different appointment strategies.
- E-mail notification: when making a new appointment, some clients may want to send an appropriate e-mail to all the persons involved.
- Access control: Some clients want to control access to their agenda, because they don’t want an unauthorized person to inspect or make appointments in their agenda.

To realize dynamic and selective combination of extensions, Lasagne introduces the following concepts and mechanisms:

First, we observe that composition is ideally specified in terms of extensions instead of wrappers, entities that are too fine-grained. 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. Wrapper definitions are specified to be a member of an extension by a declarative binding to the unique extension identifier. Extension identifiers should be managed in a hierarchical namespace. For the sake of simplicity we introduce for the above extensions the dummy extension identifiers “group”, “leisure”, “business”, “access” and “email”

Second, a *composition policy* of a client request is the subset of extensions that must be applied for that client request. The composition policy is externalized from the code of the system implementation, so that it can be controlled in one place: a composition policy is specified as a set of extension identifiers and *travels* as meta-data with the message flow of its collaboration: it is automatically propagated through the system as the execution of its collaboration advances. As such, the decision as to which extensions to use for a specific client request travels as a first-class entity together with the message flow of the collaboration, rather than being locked up and scattered across the implementation of the core system (without Lasagne, the core implementation must do switching of object references to select between extensions)

Third, 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. Interceptors can indicate that an extension should be executed when a programmer-defined expression is true. Due to the propagating nature of composition policies, an

interceptor encapsulates a client-specific customization of an *entire* collaboration with the guarantee of *system-wide consistency*.

Lasagne employs a *dynamic wrapper mechanism* that dynamically concatenates wrappers into a wrapper chain on a per message basis. To realize this dynamic chain construction a special method combination mechanism, called a *variation point*, must be implemented by the underlying middleware platform or programming language. When a method of a component instance is invoked, the variation intercepts this message, interprets the composition policy of the message and will only redirect this message to 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.

2.2.1. The extension programming model

Figure 2 shows some of the example extensions as integrated in the core dating system. We implement each 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.

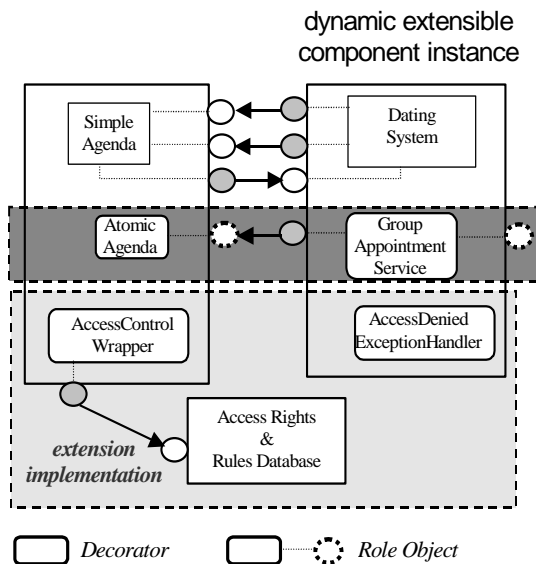


Figure 2 Some of the extensions integrated

A wrapper can *specialize* a core component by reusing its state, by adding new state components or by constraining the set of behaviors of its operations. To do this, the wrapper can perform additional actions before or after forwarding to a dynamically bound inner component. This is similar to the “Decorator” design pattern[6]. However, the inner component is denoted by a special keyword “inner” (see code example of the group appointment extension below). 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 composition time [4]. It is exactly this feature of the wrapper programming model that corresponds with the dynamic construction of wrapper chains by the variation point.

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

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

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

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

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

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

A wrapper can also *extend the interface* of a core component. In this case the state of the core component is unchanged, but the wrapper can include new operations not found in the parent and new parameters for operations that exist at the parent. This is similar to the “Role Object” design pattern [3]. Outside instances that want to use the newly added interface need to have a reference to the role instance. The Role Object design pattern solves this by passing a specification object to the core component instance, and the role instance that matches the specification is returned [3]. Lasagne adopts the same mechanism but passes naturally the extension identifier of the wrapper instance as specification object by means of a special language primitive `getRole()`.

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

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

As such, the wrapper programming model of Lasagne combines both the Decorator and the Role Object design pattern. 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 language implementation.

2.2.2. Defining client-specific composition policies with interceptors

As explained above, an interceptor intercepts collaborations and updates their attached composition policy in a client-specific fashion.

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 at a specific service interface.

Interceptors are programmable entities that have a simple interface for intercepting component interactions of their co-located component instance. Interceptors typically implement a client-specific strategy that determines which extension identifiers are attached to which collaborations.

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

```
package client.customization;

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

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

Figure 3 presents a message sequence diagram illustrating a very simple example of the process of client-specific combination of extensions. In the figure there are two clients. An outgoing interceptor is associated with each client that determines whether the client wants to make a business or a leisure appointment. The situation depicted by Figure 3 indicates that one client wants to make a leisure appointment, while the other client wants to make a business appointment.

Figure 3 illustrates that once an interceptor has selected an extension for a collaboration, this decision is propagated with the collaboration’s message flow such that the wrappers of that extension are dynamically invoked at the appropriate execution points of this collaboration. As such, interceptors encapsulate a *client-specific* customization of the core system, with the guarantee of *system-wide consistency*.

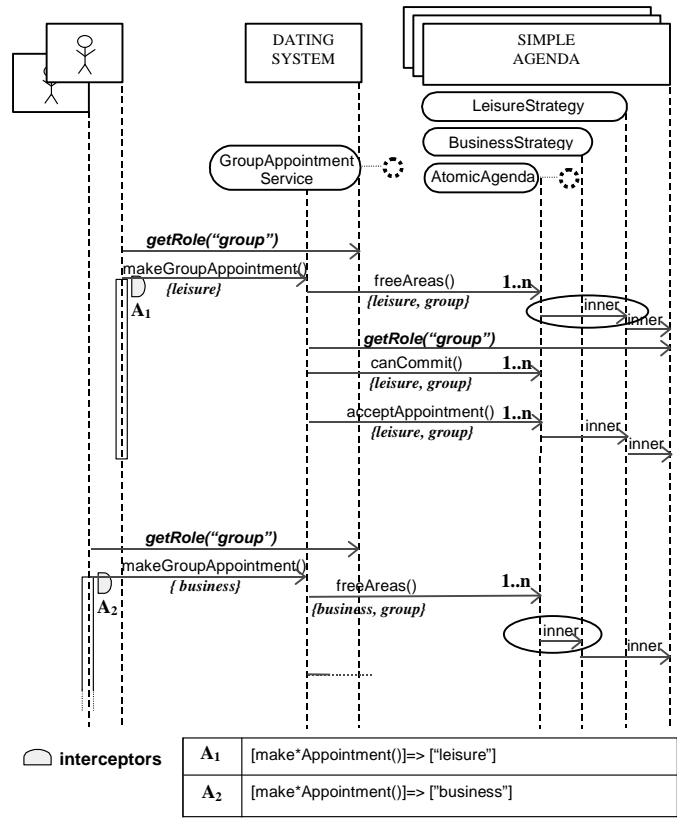


Figure 3 Message Sequence Diagram

Different extensions can of course be combined for the same client request. For example, in Figure 3, both clients want to make a group appointment (but with different appointment strategies). Since the group appointment extension adds role interfaces to the core, references to these roles must be requested via the `getRole()` primitive. When an operation of the `GroupService` role is invoked, the Lasagne run-time architecture automatically attaches the extension identifier “group” to the composition policy of the ongoing collaboration, to assure that further execution of the collaboration includes the group appointment extension.

3. Consistency Management

A system instance is always running inside a context. With respect to a system that implements an on-line service we distinguish between two kinds of context: the *deployment environment* in which the system instance is running, and the *clients* that use the services of that system instance. We introduce the role of *system manager* as the person who installs the system instance in the deployment environment and who makes sure that the system instance

stays operational during its lifetime.

To accommodate client-specific customization, the system manager can inject extensions on behalf of specific clients and Lasagne allows them to selectively integrate extensions on a per client request basis. However, as stated in the introduction, this approach introduces consistency management problems. We distinguish here between three kinds of problems:

Malicious clients. A malicious client should not have the power to corrupt a system instance by injecting an erroneous extension.

Run-time integration of new extensions. Consistency problems may possibly arise when a new client-specific extension must be injected in a running system instance: due to the crosscutting nature of extensions, run-time integration of a new extension requires the wrapping of multiple component instances. To safeguard consistency, this system-wide integration must be performed in a transactional manner. As such, *coordination mechanisms* are needed to preserve the consistency of the system while the system-wide integration of an extension is in progress.

Inconsistent client-specific composition policies. It is important that clients cannot endanger the integrity of the system by specifying illegal or inconsistent composition policies. We distinguish between two kinds of problems:

- *Mandatory extensions:* some extensions (e.g., access control) must be imposed on all clients no matter what. Malicious clients should not have a way to get around such mandatory extensions.
- *Conflicting extensions:* some extensions cannot be applied together for the same client request because they conflict with each other or they just implement the same concern in two different ways (e.g., leisure and business).

It is the responsibility of the system manager to detect such inconsistent client-specific composition policies and deal with them appropriately by either transforming the composition policy to an appropriate one or refusing service to the issuing client.

3.1. System management architecture

We now explain from a high-level point of view, a system management architecture that supports injecting new extensions into a running core system instance. This architecture is built on top of the Lasagne wrapper model.

Design philosophy. The basic design philosophy behind the management architecture is that run-time integration of extensions is strictly separated in two consecutive phases:

- (1) *Deployment* of the extensions into the core system. Deployment of an extension means that the extension is being installed in the deployment environment of the system without being “activated”. However, after deployment, the extension is ready to be activated and integrated

with the core system.

- (2) *Dynamic selection* of extensions on a per client request basis with support from the Lasagne wrapping mechanism to integrate the extensions.

Deployment of extensions is the exclusive privilege of the system manager. As such, clients can only select those extensions that have been deployed before by the system manager. If a client wants to have a new feature injected to the system instance, he/she should negotiate this with the system manager.

Deployment of a new extension. We give an overview of the architecture by means of a concrete deployment scenario. Suppose the system manager wants to deploy a new extension to a running system instance. The system manager makes the new extension first available by uploading the implementation of its wrappers into a *global code repository*. Each wrapper implementation is named by a code repository identifier that consists of the extension identifier to which that wrapper implementation belongs followed by a ‘.’ and the fully qualified name (e.g., class name) of the wrapper implementation.

Secondly there is a *configuration management tool* that allows the system manager to deploy the new extension into the running system instance. To do this, the configuration management tool must know about all the core component instances running in that system instance. The configuration management tool allows the system manager to specify *wrapping commands* in an easy way. A wrapping command specifies around which component instances a specific wrapper of the extension must be wrapped. A wrapping command uses code repository identifiers to refer to a specific wrapper implementation. There are three different kinds of wrapping commands. The difference between them lies in the scope at which the wrapping command operates:

- The wrapping command operates at the *component type* level: a wrapper instance must be deployed around all component instances that implement one or more specific interfaces.
- The wrapping command operates at the *component implementation* level: a wrapper instance must be deployed around all the instances of a specific component implementation (archive of a set of classes + version number).
- The wrapping command operates at the component instance level: a wrapper instance must be deployed around one or more specific component instances.

Each core component instance is watched over by a local *component manager* who manages the “wrapper aggregate” of the core component instance and the wrappers that are currently deployed around the core instance. A component manager is implemented as a meta-level object that controls the composition logic of how

wrappers must be composed around its associated core component instance. For example, when different extensions are deployed around the same component instance, there may be constraints on the order in which their respective wrappers must be invoked when processing a message. In the above dating system example, access control should be performed before the group appointment service. It is not yet clear to us who has the responsibility to state these ordering constraints (the system manager, the extension implementer, or a combination of both), but once this responsibility is established, the component manager offers this information to the dynamic wrapper mechanism of Lasagne as a partially ordered set of extension identifiers.

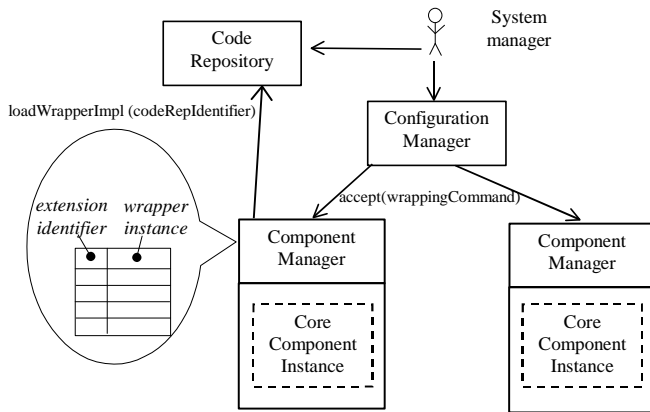


Figure 4 Deployment of extensions

Each component manager has an interface for accepting new wrapping commands from the configuration manager.

The acceptance of a wrapping command eventually results in the construction of a wrapper instance. The system manager can choose to activate the extension in a lazy or strict fashion. With lazy activation, the construction of the wrapper instances is postponed until a client actually wants to use the extension for the first time. With strict activation, the component managers construct the wrapper instances immediately after acceptance of the wrapping commands.

When the wrapper construction is initiated, each component manager loads the wrapper implementations from the code repository by passing the code repository identifier that was specified in the received wrapping command. Once constructed, the wrapper instance is managed in a hash table; the object identity of the wrapper instance is labeled with the extension identifier of that wrapper.

Note that the actual combination and integration of wrappers into the core system is implemented by the dynamic wrapper mechanism of Lasagne.

Deployment-specific interceptors. Besides deploying client-specific extensions, the system manager may also want to deploy one or more extensions on his own behalf. For example, he may want to impose a set of specific business rules. Deployment-specific extensions also involve support for non-functional requirements such as security, and reliability.

Of course, the system manager wants to have full control on the selection logic of these deployment-specific extensions without interference from clients. Lasagne naturally supports this by interceptors that are placed in the deployment environment before the front-ends of the system instance. To make a clear difference, we call these interceptors deployment-specific interceptors, while interceptors at the client side are called client-specific interceptors. Deployment-specific interceptors impose the deployment-specific extensions by selecting the appropriate extension identifiers and attaching them to the composition policy of every incoming client request. A client-specific interceptor cannot override a deployment-specific interceptor due to the direction of control transfer from client to system instance. Thus, deployment-specific interceptors have precedence over client-specific interceptors.

A deployment-specific interceptor can of course also be used to dynamically control the composition policy of ongoing collaborations. This allows for example adapting the system instance to changing circumstances in the deployment environment (e.g., a drop in the availability of computational resources) by switching between non-functional extensions at run-time. Different extensions can implement the same non-functional requirement, but each extension is optimized for different circumstances in the deployment environment. If the system instance must be adaptable to changing circumstances in the deployment environment, the system manager must deploy the alternative extensions with lazy activation. A deployment-specific interceptor can then be programmed that monitors the circumstances in the deployment environment and dynamically selects the extension that is best suited to the current circumstances.

3.2. Dealing with the management problems

In this section we show how the management architecture deals with the three kinds of consistency management problems.

Malicious clients. After the deployment of a new extension is finished, the system manager publishes the extension identifier (plus a documentation of the feature implemented by this extension) as part of the service interface of the system. As such, clients can only select those extensions that have been previously deployed by the system manager. Provided that access to the configuration management tool

is checked, it becomes more difficult for a malicious client to inject a corrupting extension to the core system.

Coordination support for integrating new extensions. In a software system that is developed with a traditional programming language, complex coordination protocols are needed to preserve the consistency of the system while the run-time integration of an extension is in progress. These coordination protocols are extremely complex, however, especially in distributed systems [5].

With the above management architecture built on top of the Lasagne component model, the required coordination mechanisms can be kept very simple. This is a result of the strict separation between deployment and the dynamic selection of extensions (integration = deployment + selection):

During deployment of a new extension, it is only required that the wrapping commands for that extension are *all* correctly accepted by the component managers. This can easily be supported with a simple 2-phase commit protocol between the configuration management tool and the component managers. It is important to realize that the “acceptance of the wrapper commands” does *not* involve the construction of the wrapper instances nor turning the switch to actually integrate these wrapper instances into the core system.

Instead, the “turning of the switch” is under control of interceptors that dynamically select extensions by manipulating the composition policy of ongoing collaborations. The coordination mechanism that takes care of preserving the consistency while turning the switch is very simple:

- A client can only select those extensions whose extension identifiers are published by the system manager, who in turn has already made sure that these extensions were properly deployed into the system.
- Once an extension is selected for a client request (i.e. it is attached to the composition policy of the client request’s collaboration), its consistent integration is automatically taken care of by the Lasagne component model, using the automatic propagation of composition policies.

Inconsistent composition policies. To deal with inconsistent client-specific composition policies, the system manager has to program a deployment-specific interceptor that detects and overrides inconsistent composition policies issued by clients. This is sufficient since deployment-specific interceptors have precedence over client-specific interceptors.

For example, the code example below shows such a deployment-specific interceptor for the dating system example of section 2.2. The interceptor first imposes access control as a mandatory extension and, secondly, it filters out composition policies that apply the “leisure” and the

“business” extensions at the same time, which is a conflict because these extensions implement the same concern.

```
public class PolicyEnforcer implements Interceptor {  
  
    public void manipulate(Interaction interaction) throws InconsistentPolicy {  
        CompositionPolicy policy = interaction.getPolicy();  
        policy.mergeExtension("access");  
        validate(policy);  
    }  
  
    private void validate(CompositionPolicy policy) throws InconsistentPolicy {  
        if ((policy.containsExtension("leisure") &&  
            policy.containsExtension("business"))  
            throw new InconsistentPolicy("Extensions <leisure> and <business>  
implement the same concern");  
    }  
}
```

4. Related Work

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

Programming Mechanisms. Recent work by Klaus Ostermann proposes an object-oriented programming mechanism, called delegation layers [17], that supports on-the-fly run-time combination of behavioral extensions to a set of local objects, with the guarantee of consistency. A strong point of delegation layers, in comparison with Lasagne, is that it generalizes existing programming language features like inheritance and virtual classes, instead of extending the language with new mechanisms. Delegation layers does not allow behavioral extensions to random sets of existing objects, meaning that these objects must be predefined as part of an outer object.

Advanced separation of concerns techniques such as Aspect-oriented Programming [11], Hyperspaces [23], Mixin Layers [22], Adaptive Plug and Play Components [15] allow refinement of a core system with a crosscutting extension, by refining state and behavior at multiple points in the application in a non-invasive and modular way. One of the strong arguments in favor of these techniques is that they make it easier to preserve the consistency of the system. The implementation of a crosscutting extension can now be programmed in a well-localized, modular manner. However, these approaches are mainly based on static composition of implementation units, limiting the integration of extensions to compile- or load-time. Lasagne on the contrary integrates crosscutting extensions at the object level, enabling run-time integration and per instance customization. The price to pay for this gain of flexibility is that the Lasagne model introduces several new consistency management problems. However, the proposed management architecture in section 3.1 deals with these problems effectively.

Composition Filters [2] and Aspect Components [19][20] support dynamic combination of aspects on a per

object interaction basis. However, these models do not provide coordination mechanisms to preserve consistency in the presence of dynamic combination of system-wide extensions on a per collaboration basis. Since the underlying architectures of these models are very generic, however, the necessary coordination mechanisms can be added of course. For example we implemented Lasagne on top of the Java implementation of Aspect Components (JAC) [20]. With regard to the composition filters model, incoming messages can be delegated to a meta-object that implements the necessary coordination mechanisms.

Linda Seiter et al. [21] 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 propagating composition policies of Lasagne. However, context relations have overriding semantics and do not allow selective combination of extensions.

Mira Mezini [14] presented the object model Rondo that does support dynamic composition of object behavior well without name collisions. However, there is no support mentioned for specifying behavior composition on a per collaboration basis. Research is ongoing, however, to make her co-work on composing collaborations [15] more dynamic [8] [16].

Dynamic software architecture. In Regis [13] and ArchStudio [18], systems are constructed from a number of components and connectors that encapsulate the interactions between these components. The emphasis of this work is on the description, reconfiguration and evolution of the system'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 this work 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.

Work from Andrade et al. [1] defines mathematical semantics (corresponding tool support is also implemented [7]) that integrates the notion of superimposition [10] with dynamic software architectures. Superimposition is a basic mechanism for adapting black box components (and thus is somewhat similar to the wrapper-based approach). Andrade

et al. take superimposition beyond component adaptation and adopts it as the basic structuring principle that underlies component interconnection via architectural connectors. That is to say, superimposition is taken as a mechanism by which more global interaction mechanisms can be used to *coordinate the joint behavior of several components* of the system. Adapting multiple components in a consistent way becomes a particular case of this coordination mechanism. As such, we feel that this work shares a common ground with Lasagne and may provide a formal framework in which we can express the semantics of Lasagne. However, this is subject of further research.

Coordination support in adaptive systems. As stated in section 3, performing a system-wide adaptation at run-time requires coordination mechanisms that preserve the consistency of the system while such an adaptation is in progress. In the context of distributed systems, Chen et al. [5] proposes a graceful adaptation protocol that allows adaptations to be made in a coordinated manner across hosts transparently to the application. This protocol seems more complex than the coordination mechanisms that are used with Lasagne. (see section 3.2 for more details).

The current implementations of Lasagne seem to have a higher run-time performance overhead though. Therefore we think that Lasagne is better suited for adaptation at the *coarse-grained* architectural level of a (distributed) system, while the work from Chen et al. is better suited for providing adaptability at the component implementation level.

5. Conclusion and Discussion

The problem that underlies this work – the need for clients of on-line services to be able to integrate and customize these services for use in their own contexts – is a real one, and it is an important problem to resolve. The proposed solution to this problem – a by-request customization of the system implementing the service, where the system itself consists of a stable core and several extensions that are injected into the core as needed – brings some consistency management problems of its own.

We have presented the component model Lasagne that allows implementing systems that can be dynamically customized on a per client request basis. A system management architecture built on top of Lasagne deals with the mentioned consistency management problems. The design philosophy behind this management architecture is a strict separation between deployment and selection of extensions.

As stated in section 3, some kinds of extensions will inherently conflict with one another. An important issue to resolve in this matter is how does one detect and identify such conflicts? Since extensions are imposed dynamically

in Lasagne, the only way to identify such conflicts is to wait until someone gets a bizarre result at run-time. These "bizarre results" may be context-specific (i.e., they may only occur when some subset of extensions is running at the same time), which means that it will take the very hardest kind of debugging to track them down (essentially, debugging of concurrent systems). While one can make this argument, to some extent, about all software composition approaches, those that include more of a declarative approach have the advantage that they may be amenable to static analysis methods to identify conflicts.

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

6. References

- [1] L. Andrade and J. Fiadeiro, "Coordination: The evolutionary dimension", in Proceedings of TOOLS Europe 2001, W. Pree (ed.), IEEE Computer Society Press 2001, pp. 136-147.
- [2] 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.
- [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] Wen-Ke Chen, M.A. Hiltunen, R. D. Schlichting, "Constructing Adaptive Software in Distributed Systems", in Proceedings of International Conference on Distributed Computing (ICDCS'2001), 2001, pp. 635-643.
- [6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, pp. 175-184.
- [7] J. Gouveia, G. Koutsoukos, L. Andrade and J. Fiadeiro, "Tool support for coordination-based software evolution", in Proc. TOOLS Europe 2001, W. Pree (ed), IEEE Computer Society Press, 2001.
- [8] 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.
- [9] U. Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages, in Proceedings of ECOOP'93, 1993.
- [10] S. Katz, "A Superimposition Control Construct for Distributed Systems", ACM TOPLAS 15, 1993.
- [11] 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.
- [12] G. Kniessel, "Type-Safe Delegation for Run-Time Component Adaptation", In Proceedings of ECOOP'99, June 1999.
- [13] J. Magee, N. Dulay, and J. Kramer, "Regis: A Constructive Development Environment For Distributed Programs", in Distributed Systems Engineering Journal, 1(5), 1994.
- [14] M. Mezini, "Dynamic Object Evolution without Name Collisions", in Proceedings of ECOOP'97, 1997.
- [15] M. Mezini and K. Lieberherr, "Adaptive Plug and Play Components for Evolutionary Software Development", in Proceedings of OOPSLA'98, 1998.
- [16] 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.
- [17] K. Ostermann, "Dynamically Composable Collaborations with Delegation Layers", in Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'2002), 2002.
- [18] P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-based Run-time Software Evolution", in Proceedings of ICSE'98, 1998.
- [19] 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.
- [20] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, "JAC: A Flexible and Efficient Solution for Aspect-Oriented Programming in Java", in Proceedings of Reflection'2001, 2001.
- [21] L. Seiter, J. Palsberg, and K. Lieberherr, "Evolution of Object Behavior using Context Relations. In IEEE Transactions on Software Engineering, 24(1), 1998.
- [22] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", in Proceedings of ECOOP'98, 1998.
- [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.
- [25] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten and B. N. Jørgensen, "Dynamic and Selective Combination of Extensions in Component-based Applications", in Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), 2001.