JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 11
Special issue: OOPS track at SAC 2004, Nicosia/Cyprus

Integration of Independently Developed Components through Aliased Multi-Object Type Widening

Bo Nørregaard Jørgensen, University of Southern Denmark, Denmark

Abstract

The aim of component-based software development is to assemble applications from existing components, writing as little extra code as possible. For programmers, the assembling of applications from existing components should increase reuse, thus allowing them to concentrate on value-added tasks and to produce high-quality software within a shorter time. For users, component-based software development promises tailor made functionality from the customization of ready-made components, and that at a lower cost than applications developed from scratch. However, this ideal scenario has yet to become reality! Today, the majority of all applications are still developed from scratch, and there are still relatively few ready-made components that can be easily reused in the construction of new applications. In this paper, we argue that the present situation is primary caused by the conventional object-oriented programming languages in which we try to assemble components. When Assembling independently developed components in a conventional object-oriented programming language, it leads to a number of complex integration problems. We describe these problems in turn before we discuss how the new language features of Lasagne/J, an extension of the Java programming language, can be used to tackle them.

1 INTRODUCTION

The object-oriented programming languages and development environments of today rely on components to be well integrated and do not handle the integration of independently developed components well. Even slight inconsistencies between components can lead to integration problems that are difficult if not impossible to handle satisfactorily with the reuse mechanisms available in current object-oriented languages. Inconsistencies between components are to be expected in a world where components and application frameworks are designed and developed by independent organizations. The large number of development organizations makes perfect coordination of components practically impossible. However, the current state of affairs is unacceptable because it diminishes the pervasiveness of component-based development due to the unnecessary high cost of reuse caused by the various implementation techniques required for fitting existing components for use in new contexts. It is generally acknowledged that the effort necessary for reusing a component should be smaller than the effort required for creating an equivalent component from scratch. Components developed by a third party will not become a serious alternative to in-house development before programming languages and development environments can seamlessly support the integration of such components. Rethinking and enhancing the programming constructs of current object-oriented languages is therefore a necessity for enabling the success of component-based development.

The present situation for component-based development is characterized by the following dominating factors:

- Components are written in a statically typed, class-based object-oriented programming language.
- Components are developed and maintained by independent organizations.
- Extending the behavior of a system is ideally done by adding new components.

Each of these factors has effect on the development of components. The implications of these factors can be characterized as:

- The internal structure of a component is defined by a set of related classes. Objects of these classes collaborate to implement the services offered by the component.
- Most components have to be adapted before they can be reused in settings different from those in which they were originally developed.
- The source code of existing components cannot or should not be changed.
- The granularity of a component can vary from the size of a simple GUI widget to a full-sized application.

Having identified the dominating factors characterizing component-based development and clarified their implications, we can now turn our attention to the integration problems they cause. We will do so in section 2. In section 3, we show how to tackle these problems using Lasagne/J, an extension of Java^{TM} that supports the extension of collaborating objects through Aliased Multi-Object Type Widening. Related work is discussed in section 4 and section 5 addresses challenges left for future work. Finally, section 6 summarizes the main contributions of this paper.

2 PROBLEMS, CAUSES AND SOLUTIONS

In this section we summarize problems that component-based development must face today, due to the present plane of development of languages and tools. As part of discussing each problem, we briefly outline necessary properties of a solution. We are not the first to recognize and acknowledge the importance of these problems. Existing literature contains numerous examples of the problems in various forms, along with different proposals for solving them. However, the problems have largely been addressed independently of each other, so none of the proposed solutions address all of them in a unified fashion. New solutions which can address a broader range of the problems, ideally all of them, are therefore needed. We return to our proposal for such a solution in section 3.

Multiple representations of real-world entities. When assembling an application from independently developed components, it may occur that different components contain individual representations for the same real-world entities. As argued in [Ossher92] and [Mattsson99] this is a problem because equivalent representations have to be coordinated in the composed application. One practical problem is that objects of equivalent classes must be interchangeable between components. Components must be able to exchange objects that represent the same real-world entity. However, the exchange of objects across component boundaries is forbidden by the type system of the implementation language if the exchanged objects belong to different class hierarchies, i.e. they do not share a common supertype. A solution to the problem has to compensate for the missing shared superclass and the possibility of slightly semantic differences between the representations. Appropriate compensation could be provided by mechanisms which support introduction of superclasses into existing class hierarchies and adaptation of behavior in affected classes.

Extending shared components. A shared component is typically a component which controls a unique resource. In some situations simultaneous clients may have different expectations to the behavior of a shared component. One client could for instance require a newer version of the component than the one required by another client. If the newer version of the component is appropriate for the client requesting the older version, we can simply replace the old version with the newer one. Often the situation is more complex; simultaneous clients may have individual conflicting requirements for the component's behavior. For shared stateless components the problem can be solved by maintaining different versions of the component for each client. However, this approach does not work for components which contain state. Here duplication will result in inconsistencies for the part of the state that is common to all versions of the component. As argued in [Meijer02] conventional programming languages lack adequate support for expressing and managing multiple versions of the same component, especially in the case where different versions of the same component cannot co-exist. The absence of adequate support leaves us in a situation where the requirements of one client may exclude the presence of another. A solution to the problem must allow developers to control the visibility of the modifications made to a component, so that modifications are only visible to the client who requested them.

Interface mismatch. Due to their independent conception it is very unlikely that independently developed components will have exactly the same interface structure and behavior [Smith98]. In order to avoid interface incompatibilities, different component vendors must agree upon both structure and behavioral semantics of shared interfaces. To some extend this may be achievable for domain specific components, by complying with an external specification, but it is generally not the case for general purpose components. Clients will therefore not be able to substitute one component for another without compensating for the differences between the two. Interface incompatibilities typically manifest themselves as typing conflicts caused by the implementation language, when trying to combine independently developed components, as discussed by [Hölzle93] and [Mattsson99]. A solution should allow developers to express how the behaviors of different abstractions relate to each other. One example of this train of thought is a form of polymorphism, called correspondence polymorphism [Rinat96]. In correspondence polymorphism a *correspondence relation* declares how some methods and attributes of one type correspond to other methods and attributes within another type.

Architectural style mismatch. Components typically use one of two architectural styles for interacting with other components. The first style of interaction is the explicit invocation style, which is naturally supported by the call idiom used in the majority of all programming languages. The other style of interaction is the *implicit invocation* style [Shaw96]. This style of interaction is probably better known as event-based. Where the explicit invocation style invokes methods directly on other components by calling them, the implicit invocation style invokes methods indirectly by announcing one or more events. Other components register their interest in an event by associating a callback method with it. When the event is announced, the source of the event invokes all of the callback methods that have been associated with the event. The implicit invocation style is more flexible than the explicit invocation style, because it supports multiple receivers for an invocation instead of just one. Furthermore, it supports loose coupling by allowing receivers to be dynamically added and removed. However, the implicit invocation style is not supported by conventional object-oriented programming languages, which means that it has to be simulated by the normal call idiom. The general scheme is that components which listen for events have to implement a listener interface. It is the listener interfaces that associate events with methods. Since the two interaction styles result in fundamentally different code structures, the integration of components that use different styles will either require major re-factoring of existing code or extensive use of new glue code. In a world of independent development, it is most likely that applications will be assembled from components supporting one style or the other. As discussed in [Garlan95] care must be taken when mixing the two styles. To bridge this mismatch in architectural styles, we need language mechanisms which allow components to participate in different styles of interaction simultaneously.

Modification of inter and intra component collaborations. When development takes place in a class-based object-oriented language, any well-formed non-trivial component will consist of a set of related classes. From this set of classes, the component instantiates the objects that form the collaborations responsible for the functionality which the component provides to its clients. Some of these objects also participate in the collaborations that the component has with other components to implement application-level functionality. Hence, any extension of functionality either at the component-level or at the application-level will require modifications of several classes. Likewise, adding a new collaboration will require the modification of several classes in order to adapt these classes to the appropriate collaboration roles. The addition of a new collaboration is often a consequence of integrating a new type of component. Changing the classes of a component to implement an extension ties the original version of the component to its extended version. Often an extension is used to bind a component to a specific usage context, making the component less generic. The irreversible nature of such bindings caused by invasive changes to classes is undesirable, because in general it should be possible to continue the development of a component independently of any extensions made to it. Furthermore, the clarity of the relation between a component and its extensions is damaged, because the code responsible for implementing the extensions is not localized in one place, but instead spread across several classes. This results in tangled code which is difficult to understand and maintain, because it basically suffers from bad modularity. As pointed out in [Mezini00] and [Kiczales97] there is a critical need for programming constructs which can support the separation of base functionality from extended functionality through capturing and encapsulation of distinct and largely independent slides of program behavior into separate modules.

Iterated extension of components. Similar to the creation of applications by assembling independently developed components, one may argue that it should also be possible to create higher-level applications by assembling independently developed applications. Any application can be used as a component in the construction of a higher-level application, if only it complies with the component model of the applied component infrastructure. Hence, at a latter point in its lifecycle an application can itself become a component in the development of an even larger application. Using a development approach that creates new applications from existing applications will cause the classes constituting the components of the applications involved in the assembling process to be iteratively modified. The constituent classes of a component are potentially modified each time the component is integrated into a new context. At the moment wrapping is the most prominent non-invasive extension technique used in component-based development to modify the behavior of existing components. If we choose to ignore the many problems, such as efficiency and the self-problem, which follows from using wrapping in conventional object-oriented programming languages, it turns out that wrapping in general works well as long as a component is only extended by a single client. However, the situation becomes more problematic when subsequent integration iterations extend a component with dependent extensions. Such iterated extension of a component can cause consistency problems if not handled appropriately. Let’s for a moment assume that we have two dependent extensions. To ensure consistency all clients now have to access the extended component through the second extension and the second extension must access the component through the first extension. This implies that all clients of the first extension must have their references to the component replaced with references to the second extension. Hence, as discussed in [Kniesel99] there is a need for the component infrastructure to support dynamic re-wiring of component references. Without support from the component infrastructure, re-wiring of object references makes iterated extension of components very error phone and difficult to manage in systems which continuously undergo evolution. As argued in [Bosch99] an extended component should be just as extensible with the extension as it was without the extension. In general the presence of an extension should not preclude the applicability of another extension, except in the situation where they are semantically incompatible. A solution must allow us to dynamically extend the behavior of objects without the need for changing existing object references.

3 INTEGRATING COMPONENTS USING LASAGNE/J

We will use the development of a University administration system to exemplify how Lasagne/J remedies the integration problems discussed above. But before we do so, we will give a brief introduction to the main ideas behind Lasagne/J.

Lasagne/J in a nutshell

The Lasagne/J language extension facilitates the integration of independently developed components through its ability to dynamically extend the behavior of a group of collaborating objects. Lasagne/J uses generic wrappers to extend the behavior of objects. Generic means that a wrapper can be applied in a type-safe manner to any type that is a subtype of the type which the wrapper is intended to adapt. Each generic wrapper can extend the behavior of exactly one object type. Thus the extension of multiple objects happens through the coordinated use of several wrappers. A generic wrapper extends an object by dynamically widen the object's type. Dynamic widening of object type is achieved by changing the method dispatch process so that method lookup starts at the method table of the outermost wrapper and ends with the object's own method table. In short, type widening supports overriding of existing methods and the addition of new methods. The new method dispatch process establishes a common-self between the object and its wrappers. This implies that *self* within the object refers to the outermost wrapper.

All wrappers which participate in the coordinated extension of a group of collaborating objects are organized in an extension package. Extension packages provide modularization of extensions. The starting points for extending the collaborations of a group of objects are defined by the wrappers in an extension package. More precisely, the types of collaborating objects are dynamically widened by wrappers from an extension package when their collaboration is initiated through an alias of a wrapper type defined in the extension package. Due to the central role of the alias in this process, we refer to this way of widening the types of multiple objects as *Aliased Multi-Object Type Widening*. Aliased Multi-Object Type Widening provides a way of controlling the visibility of extensions to the clients of collaborating objects, because the dynamic widening of object types is done specifically to a particular alias. Hence, different extensions can be made available to simultaneous clients simply by using different aliases. Extensions activated through different aliases are kept isolated from each other so that they only share the state of the extended objects. Any additional state or behavior added by an extension is beyond the reach of other extensions. The interested reader can find more details on the subject in

[Joergensen03]. We will discuss the issue of creating aliases of wrapper types in a latter subsection.

Implementation wise Lasagne/J is based on a mix of source code transformation and load-time byte code rewriting, using the structural reflection tool Javassist [Chiba00]. The use of load-time structural reflection enables the extension of existing systems without requiring access to the source code of their components. Source code transformation is only needed to initiate Aliased Multi-Object Type Widening from within new clients of existing components.

Integrating class hierarchies with overlapping abstractions

Let's assume that our University administration system started out as a simple system only responsible for registering students as they enroll. The primary class in such a system will be that of a Student. Clearly this system will turn out to be too limited, sooner or later the system will have to be extended to meet the evolving expectations of its users. A natural extension of the system would be to include semester course management. Typical classes in a semester course management system are Course, CourseOffering, Student, Schedule, etc. According to our basic observations for component-based software development listed in section 1, the administration system will be extended through integration with a component that provides the additional functionality of a semester course management system. Common to the class hierarchies of the administration system and the semester course management system is the abstraction of a Student (see figure 1.). Coincidently the abstractions shared the same class name, but since they are the products of independent development they will be defined in different class hierarchies. They are therefore not interchangeable due to the typing conflict caused by a statically typed, class-based programming language. One of the classes has to be adapted before it can cross the typing barrier between the two class hierarchies. If we assume that it is the administration system that is responsible for creating Student objects we will have to adapt these objects before they can be used by the semester course management system. Hence, we have to extend the type of a Student object without losing its identity.

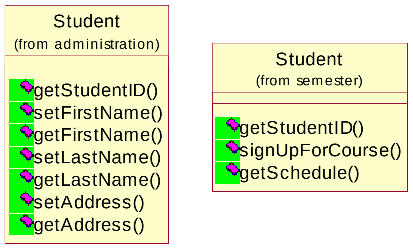

Figure 1 Component dependent representations of the same real-world entity

In Lasagne/J objects are adapted using generic wrappers. The generic wrapper in figure 2 allows Student objects created by the administration system to cross the typing barrier between the two class hierarchies and to be used as Student objects in the component providing the semester course management functionality.

```
wrapper StudentTypeAdaptor
   extends org.lasagnej.examples.university.semester.Student
   wraps org.lasagnej.examples.university.administration.Student
{}
```

Figure 2 Simple type adaptation wrapper

Here we introduce two new keywords to the Java programming language for defining wrappers. The keyword `wrapper` is introduced to distinguish the definition of a wrapper from a class definition and the keyword `wraps` is used to associate a wrapper definition with a class definition (i.e., the static wrappee type). To some extent the definition of wrappers are related to the definition of classes, since they can define both state and behavior. Wrappers can be defined for concrete and abstract classes, and applied to instances of those classes as well as instances of their respective subclasses. The latter is a consequence of fulfilling the *genericity* requirement [Büchi00], which states that a wrapper must be applicable to any subtype of the static wrappee type to be generic. When used together the `extends` clause and the `wraps` clause declare that the aggregate which results from applying a `StudentTypeAdaptor` wrapper to an instance of `org.lasagnej.examples.university.administration.Student` is a subtype of the class `org.lasagnej.examples.university.semester.Student` and the static wrapper type `StudentTypeAdaptor`. Thus the aggregate can now be used where an instance of the class `org.lasagnej.examples.university.semester.Student` is expected. With respect to method lookup and dispatch the combination of the `extends` clause with the `wraps` clause has type adaptor semantics. That is, if the same method signature appears in both the extended class and the static wrappee type the method call is delegated to the wrappee, unless it is overridden by the wrapper. Wrappers can only override public methods. This includes both the public methods of the static wrappee type and the public methods of the extended class. A public method that is not overridden by a wrapper is transparently accessible through the wrapper. Hence, all the methods of the two Student classes listed in figure 1 are accessible through the wrapper `StudentTypeAdaptor`. If the `extends` clause is absent the `wraps` clause declares that the aggregate, which results from combining a wrapper with an instance of the static wrappee type, is a subtype not only of the static wrappee type but also the dynamic wrappee type. However, since the dynamic wrappee type is not known until runtime, it is only the methods of the static wrappee type that can be overridden by the wrapper.

As mentioned earlier wrappers that belong to the same system extension are placed in the same package, a so-called extension package. This package cannot be the same as the package in which the classes that they wrap are defined. This separation of wrappers from the classes that they wrap is required in order to encapsulate extensions into identifiable modules. Consequently, a wrapper is always defined within another package than the package of the class that it wraps. At this point, in our running example, we have one extension package containing a single wrapper, i.e., `StudentTypeAdaptor`, since this is all that is required for performing the integration.

We can now use the Façade design pattern [Gamma95a] to write a class `ServiceFacade` that integrates the administration system with the semester course management system. The class `ServiceFacade` implements the two use-cases *Sign Up For Course* and *Commit Schedule*. The source code of the class `ServiceFacade` is listed in figure 3.

```
 1. public class ServiceFacade {
 2.     public void
 3.     enrollInCourse(String identification, String courseName)
 4.     {
 5.         CourseOffering courseOffering =
 6.         SemesterCourseOffering.getNextSemester().
 7.         getCourseOffering(courseName);
 8.         Student student = Administration.getStudent
 9.             (new StudentID(identification));
10.         StudentTypeAdaptor adaptedStudent =
11.             (StudentTypeAdaptor)student;
12.         adaptedStudent.signUpForCourse(courseOffering);
13.     }
14.
15.     public void commitSchedule(String identification)
16.     {
17.         Student student = Administration.getStudent
18.             (new StudentID(identification));
19.         StudentTypeAdaptor adaptedStudent =
20.             (StudentTypeAdaptor)student;
21.         adaptedStudent.getSchedule().commit();
22.     }
23. }
```

Figure 3. The class ServiceFacade defines the point of integration

The integration of the two components happens through the cast statements in line 11 and line 20. These are not normal cast statements; they are *constructive downcast* statements. We have extended the cast mechanism of the Java language to support *constructive downcast*. Constructive downcast refers to the dynamic extension of an object by casting the object reference that refers to the object into the type of a wrapper belonging to the desired extension. We call this cast constructive because the referenced object dynamically becomes a type of the requested wrapper type when it is accessed through the cast object reference. A wrapper can only be used to change the type of a referenced object if the wrapper wraps the static type of the object reference referring to the object. The effect of a constructive downcast is not limited to the referenced object; it also affects all objects with which the referenced object collaborates. Hence, a constructive downcast designates the point in an object collaboration where the Aliased Multi-Object Type Widening process is initiated. Constructive downcast statements inform the Lasagne/J runtime from which point on in the invocation flow it must start to automatically apply wrappers to objects participating in the object collaboration. The cast statements in line 11 and 20 widen an `org.lasagnej.examples.university.admi` `nistration.Student` into an `org.lasagnej.examples.university.semester.Student` thereby allowing Student objects from the administration system to be used in the semester course management system.

Iterated extension of shared base components

Returning to our University scenario we will assume that the University's management has decided that the extended administration system has proven so successful that it should be extended to also handle the management of tuition fees. Similar to the previous extension this extension will also be based on integration with components developed by third parties. For this purpose a sales management component and an accounting component is purchased from two independent component vendors. Before integration can take place, we have to inspect the class hierarchies of the four components involved, i.e., administration system, semester course management, sales management, and accounting, in order to identify potential overlapping classes. Figure 4 shows two classes within the sales management and accounting components, which may overlap with the class `Student` in the administration system and the class `Student` in the semester course management system.

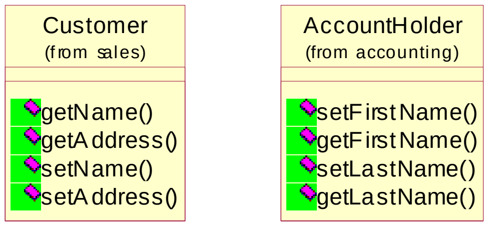

Figure 4 Roles applicable to Student objects

As part of the integration effort, it is discovered that Student objects within the context of sales management must take the place of Customer objects. Now by comparing figure 1 with figure 4 it becomes clear that the interfaces of `org.lasagnej.examples.university.administration.Student` and `org.lasagnej.examples.university.sales.Customer` must be bridged before Student objects can be used in the place of Customer objects. The class `org.lasagnej.examples.university.administration.Student` deals with a student's name and address as String objects whereas the class `org.lasagnej.examples.university.sales.Customer` uses Name and Address objects. Hence, we need to define a type adaptation wrapper that allows Student objects to behave as Customer objects, see figure 5.

```
Wrapper StudentCustomerAdaptor
   extends org.lasagnej.examples.university.sales.Customer
   wraps org.lasagnej.examples.university.administration.Student
{
   public Name getName() {
      return new
      Name(inner.getLastName(),inner.getFirstName());
   }
   public Address getAddress() {
      return new Address(inner.getAddress());
   }
}
```

Figure 5 Bridging Customer objects

The wrapper `StudentCustomerAdaptor` overrides the necessary methods in class `Customer` for bridging the class `Student`. Within the methods of a wrapper, the keyword `inner` refers to the wrappee. It can be thought of and treated as an implicitly declared and initialized final instance field. Hence, the keyword `inner` defines a unique reference to the wrappee within the scope of the wrapper. Similar to the way subclasses use the keyword `super` to call overridden methods of their super class, the keyword `inner` can be used by wrappers to call overridden methods of the wrappee. A wrapper can choose to conceal the behavior of the wrappee by not forwarding an invocation to `inner`. Optionally it can redirect an invocation by invoking another method on the wrappee. Our running example will also require a type adaptation wrapper for making Student objects interchangeable with Account holder objects. However, we choose to exclude the definition of this wrapper since it follows the definition of the wrapper `StudentCustomerAdaptor`. Making objects interchangeable using type adaptation wrappers is however not enough to enhance the extended administration system with support for handling tuition fees. We also have to extend the behavior of the use-case *Commit Schedule*, and furthermore we have to add a new use-case *Get Tuition Fee*. The implementations of these use-cases require a number of behavior adapting and extending wrappers, one of which is a wrapper for the class `ServiceFacade`. Figure 6 shows the wrapper for extending the class `ServiceFacade`.

```
wrapper TuitionServiceFacade wraps ServiceFacade {
    public void commitSchedule(String identification) {
    inner.commitSchedule(identification);
         Student student = Administration.getStudent
              (new StudentID(identification));
         Order order = OrderSystem.createOrder();
         order.setCustomer((StudentCustomerAdaptor)student);
         for (Iterator it = student.getSchedule()
              .getCourses().iterator();it.hasNext();) {
                   order.addItem(new Item
                       ((CourseOfferingProductAdaptor)it.next())
              }
13.             order.close();
14.         }
15.
16.         public int getTuitionFee(String identification) {
17.             Student student = Administration.getStudent
18.                 (new StudentID(identification));
19.             TuitionStudent tuitionStudent =
20.                 (TuitionStudent)student;
21.             return tuitionStudent.getTuitionFee();
22.         }
23.     }
```

Figure 6 Tuition extension for the class `ServiceFacade`

To implement the new use-case *Get Tuition Fee*, we need a number of wrappers for extending the original behavior of the classes in the semester course management system. These wrappers are shown in the sequence diagram in figure 7. To better illustrate the Aliased Multi-Object Type Widening process, we have changed the UML so that it allows us to express wrapping of objects. The meaning of the compartment text is changed from `<object>:<class>` to `<wrapper>:<class>`. Iteration is shown by preceding a method call with a *.

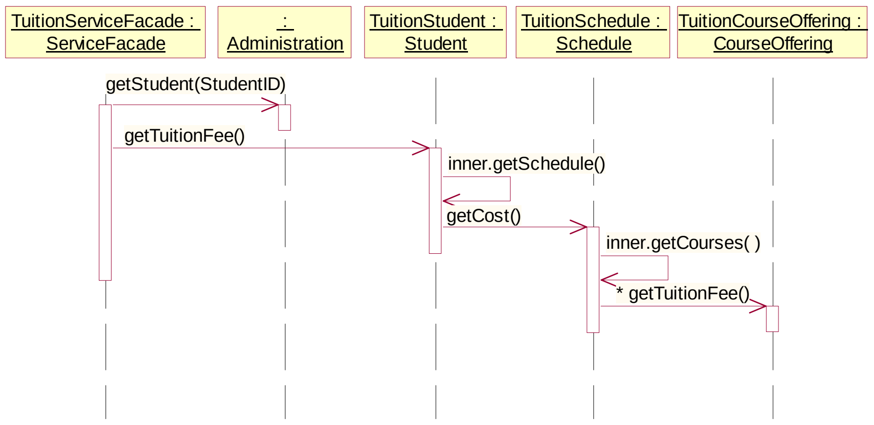

Figure 7 Object collaboration implementing the use-case *Get Tuition Fee*

As we can see from figure 7 the introduction of a new use-case results in the creation of a relatively large number of wrappers. In the specific case, we had to create wrappers for all the classes in the semester course management system. The Aliased Multi-Object Type Widening of the classes `Student`, `Schedule`, and `Course Offering` are initiated by the constructive downcast in line 20. Extending the system to handle tuition fees implies that invoices have to be issued to students. Figure 8 shows how Aliased Multi-Object Type Widening modifies the objects implementing the use-case *Commit Schedule* to call the necessary methods in the sales management component and in the accounting component. The wrappers `TuitionServiceFacade` and `TuitionOrder` integrate the sales management component and the accounting component.

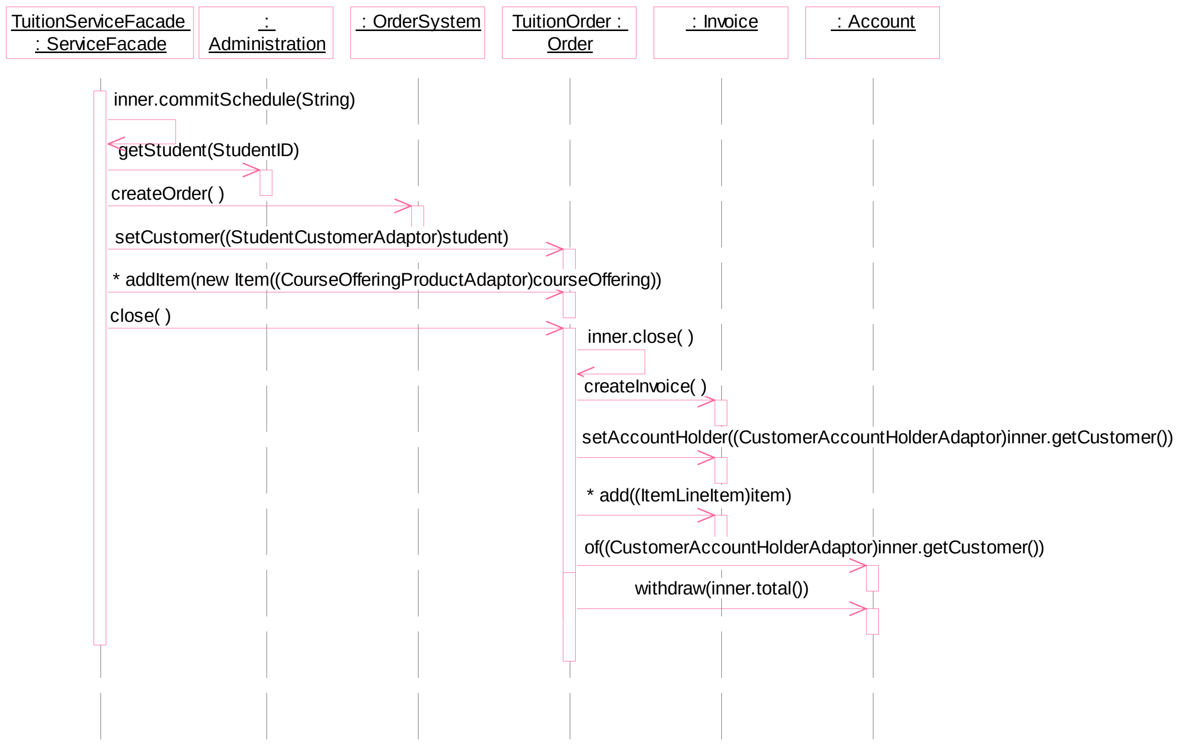
TuitionServiceFacade : ServiceFacade
: Administration
: OrderSystem
TuitionOrder : Order
: Invoice
: Account
inner.commitSchedule(String)
getStudent(StudentID)
createOrder( )
setCustomer((StudentCustomerAdaptor)student)
* addItem(new Item((CourseOfferingProductAdaptor)courseOffering))
close( )
inner.close( )
createInvoice( )
setAccountHolder((CustomerAccountHolderAdaptor)inner.getCustomer())
* add((ItemLineItem)item)
of((CustomerAccountHolderAdaptor)inner.getCustomer())
withdraw(inner.total())

Figure 8 Refinement of the collaboration implementing the use-case *Commit Schedule*

Evolution of business logic through collaboration refinement

Continuing our running example we will assume that the University management has decided that no student will be allowed to sign up for a course if she has a tuition balance from a previous semester. This time no new components are needed. We can implement the decision of the University management by changing the business rule for course enrollment. This can be done through specializing the wrapper `TuitionService Facade` and the wrapper `TuitionStudent` from before. Figure 9 shows the specialization of the wrapper `TuitionServiceFacade`.

```
wrapper TuitionDebitServiceFacade wraps TuitionServiceFacade {
      public void enrollInCourse
             (String identification, String courseName)
      {
             inner.enrollInCourse(identification,courseName);
      }
}
```

Figure 9 Specializing the wrapper for the tuition extension

When the `wraps` clause refers to another wrapper it means that the new wrapper is a specialization of the other wrapper. A specializing wrapper wraps the same static wrappee type as the wrapper that it specializes. This means that the specializing wrapper becomes a subtype of the original wrapper. The specializing wrapper can add methods to and override methods of the original wrapper. Additional methods are accessible through an object reference that is at least of the same type as the specializing wrapper. When a wrapper specializes another wrapper, the use of the keyword `inner` within the outermost wrapper refers to the next inner wrapper. The outermost wrapper can conceal the behavior of the next inner wrapper and the wrappee by not forwarding a call to `inner`. Activation of an extension through constructive downcast of an object to the type of a specializing wrapper will also activate all wrappers within the extension package of the original wrapper. Thus the constructive downcast of a single object will affect all objects participating in the object collaboration in which the constructive downcast happens.

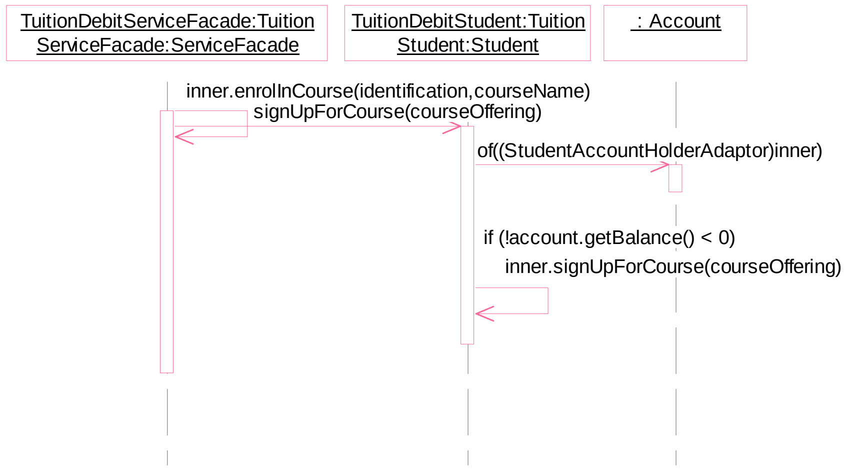
TuitionDebitServiceFacade:Tuition ServiceFacade:ServiceFacade
TuitionDebitStudent:Tuition Student:Student
: Account
inner.enrollInCourse(identification,courseName)
signUpForCourse(courseOffering)
of((StudentAccountHolderAdaptor)inner)
if (!account.getBalance() < 0)
inner.signUpForCourse(courseOffering)

Figure 10 Extended object collaboration for changing the business logic of the use-case *Enroll In Course*

The sequence diagram in figure 10 shows how a type adaptation wrapper is used to adapt Student objects so that they can be used as AccountHolder objects in the context of the accounting component. A comparison of the class `Student` and the class `AccountHolder` in figure 11 shows that a simple type adaptation wrapper will do since the interface of the class `Student` matches the interface of the class `AccountHolder` - no adaptation of methods are required. Methods invoked on an alias of type `AccountHolder` are automatically dispatched to the corresponding methods on a `Student` object. All interactions with the system that go through the new wrapper `TuitionDebitServiceFacade` will be subject to the *no tuition balance* rule.

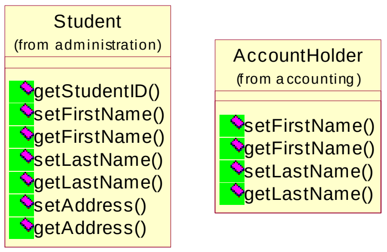

Figure 11 Incidentally structural and semantic compatibility between independent developed classes

Bridging architectural mismatch in interaction styles

To exemplify how Lasagne/J supports the assembling of applications from components that use different architectural styles of interaction, we will extend our running example with a web content management module for publishing semester related information. One of the things that the university's administration wants to publish on the University's home page is late cancellation of courses. The development department purchased a new web content management (WCM) component especially for this purpose. However, it soon turns out that the WCM component is implemented using the Observer design pattern [Gamma95b], whereas all other components in the system use the call idiom. The WCM component uses an interface and two classes to implements event-based interaction. The interface `Observer` declares the callback method `update(Object c)` for content change events. The class `WebPublisher` implements the interface `Observer` and provides the concrete implementation for the callback method `update(Object c)`. Finally, the abstract class `Subject` is a utility class that provides methods for attaching and notifying observers. In order to work with the WCM component, components that cause content changes have to extend the class `Subject`. In our case, such a component is the class `SemesterCourseOffering` in the semester course management system. Hence, the class `SemesterCourseOffering` has to notify the class `WebPublisher` when a course is removed; that is, each time its method `removeCourseOffering()` is invoked. Without Lasagne/J this would clearly be a problem since we don't want to change the source code of the class `Semester CourseOffering` to invoke the method `update(Object c)` on the class `WebPublisher`. However, with Lasagne/J the integration can simply be achieved by defining a wrapper that extends the class `Subject` and wraps the class `SemesterCourseOffering`, see figure 12.

```
wrapper CancellationSubject extends Subject wraps SemesterCourseOffering
{
        public adherent void removeCourseOffering(CourseOffering course)
        {
                inner.removeCourseOffering(course);
                notifyObservers(course);
        }
}
```

Figure 12 Type widening of the class SemesterCourseOffering

Using the wrapper `CancellationSubject` defined in figure 12 we can attach an instance of the class `WebPublisher` to an instance of the class `SemesterCourse Offering` by the combined constructive downcast and invocation statement `((CancellationSubject)sco).attach(new WebPublisher())` where the variable `sco` refers to the semester that we want to monitor. All invocations of the method `removeCourseOffering()` will now cause the method `update(Object c)` to be invoked as well. Thus, whenever a course is removed from a semester the WCM component will be notified. The presence of the method modifier `adherent` declares that the method `removeCourseOffering()` in the wrapper `Cancellation Subject` has to be executed for all subsequent invocations of the method `removeCourseOffering()` on instances of the class `SemesterCourseOffering`. Hence, the presence of the method modifier `adherent` does not only affect those invocations made through the alias `sco` but also those made through other aliases. A wrapper method declared as `adherent` sticks, so to speak, to the wrappee after the first time the wrappee has been touched by an extension. Invoking the method `attach(WebPublisher wp)` touched the `SemesterCourseOffering` object with the web publishing extension.

4 RELATED WORK

In this section, we focus on work that possess properties which we believe are essential for any solution which aims at eliminating the integration problems that developers face when assembling independently developed components. The first property is the ability to support multiple simultaneous representations for the same objects while preserving identity and state. Representation can refer to both type and behavior. The second property is that of facilitating coordinated extension of object collaborations.

Different approaches for supporting multiple representations for the same object, by dynamically changing the class of the object, have been discussed in [Drossopoulou01], [Serrano99] and [Malabarba00]. However, these approaches are limited to re-classifications within the same class hierarchy in order to prevent run-time type errors. Furthermore, the different representations cannot exist simultaneously, that is, an object can only belong to one class at a time. Thus, none of these approaches meet the need for re-classification across class hierarchies, nor do they meet the need for supporting multiple simultaneous representations.

Support for multiple simultaneous representations is discussed in [Bertino95]. Their approach introduces the idea that an object can have multiple most specific classes. Class selection is based on the static type of the object reference through which the object is accessed. This allows for multiple simultaneous representations of the same object. However, their approach cannot be used as an integration tool for classes belonging to disjoint class hierarchies, because it requires the set of most specific classes to have a common superclass.

Generic wrappers as discussed in [Büchi00] support simultaneous representations for the same object via the use of object composition at runtime. Different clients of an object can extend the object's type and behavior by applying different wrapper objects to it. A notable feature of Generic wrappers is that an existing wrapper object can be wrapped again. Because the wrapping process is type transparent. Thus, it always remains possible to extend an already extended object. However, the approach suffers from the object reference re-wiring problem. A closely related approach is the work on type-safe delegation by [Kniesel99]. Like Generic wrappers, type-safe delegation also suffers from the object reference re-wiring problem.

Correspondence Polymorphism is proposed in [Rinat96] as a means for establishing correspondences between types, none of which is necessarily a subtype of the other. As a result, methods may operate on objects and may receive arguments of types different than the ones originally intended for. This allows programmers to take advantage of similarities between existing types when such are identified and recognized as beneficial.

Subject-oriented programming, as originally introduced by [Harrison93], aims to support unanticipated integration of independently developed applications. The subject-oriented programming model claims that it can achieve this goal by maintaining several class hierarchies, or so-called subjects, over the same set of instantiated objects. The basic idea is that subjects can be combined into new applications using a number of composition rules. The programming environment HyperJ, from IBM Research, adds subject-oriented programming to Java. HyperJ is available, free of charge, from IBM's alphaWorks.

Language constructs for capturing the collaborations among objects have been discussed in [Mezini98] and [Smaragdakis98]. The rationale for capturing object collaborations is that the functionality of an application is not confined in the implementation of a single class but rather scattered across the implementation of several classes. Making collaborations explicit at the language level provides us with a single place for changing the behavior of multiple objects. A common shortcoming of these two approaches is however that the behavior of a collaboration is fixed when the collaboration is first instantiated in a component. Consequently, different clients of a shared component cannot request different specializations of the same collaboration. Approaches which can provide such support have been proposed in [Ostermann02] and [Herrmann03]. These approaches allow different clients to activate different specializations of the same collaboration for a shared component. However, the approach proposed in [Herrmann03] fails short when the extended collaboration has state that must be shared by the specialized collaborations. The problem is the use of inheritance for specialization of collaborations. Inheritance leads to replication of state maintained in the extended (i.e., super) collaboration. The proposal in [Ostermann02] does not suffer from this problem, because it associates a specialized collaboration with its super collaboration via a delegation link. Hence, state maintained in the super collaboration is shared by the specialized collaborations.

A new relation between classes, called a context relation, is proposed in [Seiter98]. Through a context relation a class (i.e., context class) can override the behavior of several classes (i.e., base classes). A context class can either be explicitly bound to a base class or it can be bound to a method invocation. Binding a context class to a method invocation will affect all nested invocations as well. Thus, clients can modify the object collaborations of a component by attaching different context classes when invoking its services.

In Aspect-oriented languages, like AspectJ [Kiczales01], modifications which affect several classes are defined within a separate module called an aspect. At compile time, the compiler weaves these modifications into the sources of the affected classes. Since aspects affect several classes they can be used to modify object collaborations. However, the weaving of aspect code into classes limits the use of AspectJ to extensions that apply for all clients of a component.

We have deliberately excluded the discussion of using object composition in conventional object-oriented languages, because of its well-known shortcomings. The interested reader may refer to [Hölzle93] for a detailed discussion.

5 FUTURE WORK

Of particular interest for future work is the investigation of mechanisms for identifying and handling of semantic incompatibilities between independently developed extensions. Extensions are said to be semantically incompatible if their combined use introduces undesirable application behavior. In most cases we can determine whether two extensions will be semantically incompatible by observing their behavior independently of each other. However, it sometimes happens that semantic incompatibilities first appear after the extensions have been integrated with the application. That is, the mutual presence of the extensions causes the application to behave in ways that could not be predicted simply by observing them independently. The common cause for semantic incompatibility seems to be that the additional application logic introduced by one extension interferes with the application logic introduced by another. The kind of interference which is observed for independently developed extensions can be seen as a variant of the *feature interaction problem*. The feature interaction problem occurs when the addition of a new feature to an application disrupts the existing services offered by the application. Feature interaction has drawn a lot of attention, especially in the telecommunication field where it has been a well acknowledged problem for many years. An overview of the state of art of feature interaction research in telecommunication can be found in [Calder03]. The feature interaction problem can also be observed for plug-ins created by multiple vendors. Plug-ins can interact in ways not envisioned by either vendor, and in ways not predictable by users and system administrators. To facilitate the development of more advanced applications from independently developed extensions it is crucial that we find ways of developing extensions in such a way that they can be seamlessly integrated without too much additional effort for correcting interference related problems.

6 SUMMARY

In this paper we have identified and described the programming problems that we believe are most relevant to programmers in a world where application development becomes more dependent on the integration of independently developed components with existing applications. Furthermore, we have discussed how the new language features of Lasagne/J, an extension to the Java programming language, remedy many of those problems. Evidence for the feasibility of our approach is given through a non-trivial integration example. The integration example shows that the proposed language features provide a unifying solution to the identified problems. To summarize:

Multiple representations of real-world entities: Objects belonging to different class hierarchies can be made interchangeable by applying wrappers that dynamically alternate their types to the types expected by the receiving class hierarchy.

Extending shared objects: The visibility of wrappers is alias specific. This means that clients independently of each other can simultaneously apply different extensions to the same component.

Interface mismatch: Wrappers can adapt and extend the interface of an object. This allows developers to deal with both structural and behavioral integration problems.

Architectural style mismatch: The use of wrappers which contain so-called *adherent* methods allows developers to introduce a component using event-based interaction into an architecture based on the call idiom.

Modification of intra and inter object collaborations: Aliased Multi-Object Type Widening facilitates coordinated modification of object collaborations.

Iterated extension of components: Repeated constructive downcast of the alias through which an object collaboration is initiated will extend the objects participating in the collaboration with wrappers from more extension packages.

Through experimenting with Lasagne/J we have gained confidence that language features like those proposed in this paper will help to narrow the current gap between object-oriented programming languages and the needs of component-based software development. However, in the balance of this paper, we chose to focus on how the proposed language features help to address those problems, instead of their implementation. An implementation of Lasagne/J is available and interested readers may download it from its project web site www.lasagnej.org.

REFERENCES

[Bertino95] Bertino E., Guerrini G.: Objects with Multiple Most Specific Classes. In: proceedings of ECOOP'95. Lecture Notes in Computer Science, Vol. 952. Springer-Verlag, (1995) pp. 102-126

[Bosch99] Bosch J.: Superimposition: A Component Adaptation Technique. In: Information and Software Technology. No 41. (1999) pp. 257-273

[Büchi00] Büchi M., Weck W.: Generic Wrappers. In: proceedings of ECOOP 2000. Lecture Notes in Computer Science, Vol. 1850. Springer-Verlag, (2000) pp. 201-225

[Calder03] Calder M., Kolberg M., Magill E.H., Reiff-Marganiec S.: Feature interaction: a critical review and considered forecast. In: Computer Networks - The International Journal of Computer and Telecommunications Networking, Vol. 41 (1). Elsevier (2003) pp. 115-141

[Chiba00] Chiba S.: Load-time Structural Reflection in Java. In: proceedings of ECOOP 2000. Lecture Notes in Computer Science, Vol. 1850. Springer-Verlag, (2000) pp. 313-336

[Drossopoulou01] Drossopoulou S., Damiani F., Dezani-Ciancaglini M.: *Fickle:* Dynamic Object Re-classification. In: proceedings of ECOOP'01. Lecture Notes in Computer Science, Vol. 2072. Springer-Verlag, (2001) pp. 130-149

[Gamma95a] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley (1995) pp. 185-194

[Gamma95b] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley (1995) pp. 293-304

[Garlan95] Garlan D.,Allen R., Ockerbloom J.: Architectural Mismatch or Why it's hard to build systems out of existing parts. In: proceedings of the 17th International Conference on Software Engineering. ACM (1995) pp. 179-185

[Herrmann03] Herrmann S.: Object Teams: Improving Modularity for Crosscutting Collaborations. In: proceedings of NetObjectDays. Lecture Notes in Computer Science, Vol. 2591. Springer-Verlag, (2003) pp. 248-264

[Hölzle93] Hölzle U.: Integrating Independently-Developed Components in Object-Oriented Languages. In: proceedings of ECOOP '93. Lecture Notes in Computer Science, Vol. 707. Springer-Verlag, (1993) pp. 36-56

[Harrison93] Harrison W., Ossher H.: Subject-Oriented Programming (A Critique of Pure Objects). In: proceedings of OOPSLA'93. SIGPLAN Notices, Vol 28 (10). ACM (1993) pp. 411-428

[Joergensen03] Jørgensen B.N., Truyen E.: Evolution of Collective Object Behavior in Presence of Simultaneous Client-Specific Views. In: proceedings of OOIS03. Lecture Notes in Computer Science, Vol. 2817. Springer-Verlag, (2003) pp. 18-32

[Kiczales97] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C.V., Loingtier J., Irwan J.: Aspect-Oriented Programming. In: proceedings of ECOOP'97. Lecture Notes in Computer Science, Vol. 1241. Springer-Verlag, (1997) pp. 220-242

[Kiczales01] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold G. W.: An Overview of AspectJ. In: proceedings of ECOOP'01. Lecture Notes in Computer Science, Vol. 2072. Springer-Verlag, (2001) pp. 327-353

[Kniesel99]Kniesel G.: Type-Safe Delegation for Run-Time Component Adaptation. In: proceedings of ECOOP'99. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, (1999) pp. 351-366

[Malabarba00] Malabarba S., Pandey R., Gragg J., Barr E., and Barnes F.: Runtime Support for Type-Safe Dynamic Java Classes. In: proceedings of ECOOP 2000. Lecture Notes in Computer Science, Vol. 1850. Springer-Verlag, (2000) pp. 337-361

[Mattsson99] Mattsson M., Bosch J.: Composition Problems, Causes, & Solutions.: In: M.E. Fayad, D.C. Schmidt, R.E. Johnson (Eds.): Building Application Frameworks: Object Oriented Foundations of Framework Design. Wiley & Sons, (1999) pp. 467-487

[Meijer02] Meijer E., Szyperski C.: Overcoming Independent Extensibility Challenges. Erik Meijer and Clemens Szyperski. Communications of the ACM, Vol. 45, No. 10. ACM Press (2002) pp. 41–44

[Mezini98] Mezini M., Lieberherr K.: Adaptive Plug and Play Components for Evolutionary Software Development. In: proceedings of OOPSLA’98. Sigplan Notices, Vol. 33, No. 10. ACM Press (1998) pp. 97-116

[Mezini00] Mezini M., L. Seiter and K. Lieberherr.: Component Integration with Pluggable Composite Adapters. In: M. Aksit (Eds.): Software Architectures and Component Technology: State of the Art in Research and Practice. Kluwer Academic Publishers, (2000).

[Ossher92] Ossher H., Harrison W.: Combination of inheritance hierarchies. In: proceedings of OOPSLA'92. ACM SIGPLAN Notices, Vol 27 (10). ACM (1992) pp. 25-40

[Ostermann02] Ostermann K.: Dynamically Composable Collaborations with Delegation Layers. In: proceedings of ECOOP '02. Lecture Notes in Computer Science, Vol. 2374. Springer-Verlag, (2002) pp. 89-110

[Rinat96] Rinat R., Magidor M.: Metaphoric Polymorphisn: Taking Code Reuse One Step Further. In: proceedings of ECOOP’96. Lecture Notes in Computer Science, Vol. 1098. Springer-Verlag, (1996) pp. 449-471

[Seiter98] Seiter L., Palsberg J., Lieberherr K.: Evolution of Object Behavior using Context Relations. In: IEEE Transactions on Software Engineering, Vol. 24(1) (1998) pp. 79-92

[Serrano99] Serrano M.: Wide Classes. In: proceedings of ECOOP'99. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, (1999) pp. 391-415

[Shaw96] Shaw M., Garlan D., Software Architecture: Perspectives on an emerging discipline. Prentice Hall, 1996. ISBN 0-13-182957-2.

[Smaragdakis98] Smaragdakis Y., Batory D.: Implementing Layered Designs with Mixin Layers. In: proceedings of ECOOP'98. Lecture Notes in Computer Science, Vol. 1445. Springer-Verlag, (1998) pp. 550-570

[Smith98] Smith G., Gough J., and Szyperski C.: Conciliation: The Adaption of Independently Developed Components. In: proceedings of PDCN'98. (1998) pp. 31-38

About the author

Bo Nørregaard Jørgensen is associate professor in Software Engineering at the Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark. His current research areas include reflective middleware systems, component-based software development and the design of programming language technologies for dynamic adaptation. Bo can be reached at bnj@mip.sdu.dk.