

Evolution of collective object behavior in presence of simultaneous client-specific views

Bo Nørregaard Jørgensen^{*}, Eddy Truyen[^]

^{*}The Maersk Mc-Kinney Moller Institute for Production Technology,
University of Southern Denmark, Odense Campus,
DK-5230 Odense M, Denmark.
bnj@mip.sdu.dk

[^]Computer Science Department, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven Belgium
Eddy.Truyen@cs.kuleuven.ac.be

Abstract. When different clients, each with their own individual customization requirements, use the same system simultaneously, the system must dynamically adapt its behavior on a per client basis. Each non-trivial adaptation of the system's behavior will very likely crosscut the implementation of multiple objects. In this paper we present an extension to the Java programming language that supports the dynamic evolution of collective object behavior in the presence of simultaneous client-specific views. In accordance with the separation of concerns and locality principles, client-specific customization of collective object behavior are organized as layers of mixin-like wrappers. Each layer of wrappers incrementally adds behavior and state to a group of core objects without modifying their respective implementations. Hence, collective object behavior can evolve in an additive and non-invasive way. The extension that we propose provides language constructs for defining, encapsulating and selecting behavioral refinements, and runtime support for transparently integrating them on demand.

1 Introduction

In the recent years the need to support simultaneous client-specific views on a single shared information system has become increasingly important within a growing number of application domains. Within those domains clients tend to divert in their specific expectations for required system behavior. To service a diversity of different simultaneously clients, the running system must be able to dynamically adapt its behavior on a per client basis. Literature reports on a growing number of projects where various aspects of software adaptability to client-specific requirements manifest themselves in form of a variety of language technologies and design patterns for addressing customization, integration, and extensibility problems [1], [2], [3], [6], [7], [8], [9], [10], [11], [12], [15], [16], [17], [19], [20], [21]. When issues related to

customization, integration and extensibility become explicit in a development project it can be seen as a result of a specific view that a particular client has on the system. Whereas present approaches primarily address those issues from the view of a single client, we will, in this paper, focus on the situation where the objects of a system have to be adapted in such a manner that the system can meet the customization requirements of multiple clients which interact with it simultaneously. Clearly the customizations required by the individual clients should not interfere with each other. Each client must be able to rely on the fact that the system will behave as the client expects it to. Any customization of the system's behavior must therefore be isolated from other customizations, but at the same time the customizations must also co-exist at runtime. The solution that we present allows individually developed clients to customize the services of a running system, for use in their own contexts, without affecting the behavior of the same services as they are delivered to other clients.

1.1 Challenges

Designing a language extension, which enables multiple clients to simultaneously customize a service of a running system to fit their specific needs without interfering with customizations imposed on the service by other clients, involves a number of language design challenges that have to be solved. These challenges are related to the fact that the presence of simultaneous client-specific views implies that multiple modifications of a system's services can take place simultaneously while the system is running. Furthermore, each of these modifications must exist in isolation of other modifications. The challenges that we have encountered can be summarized as follows:

1. **Run-time applicable:** Since customizations of a system's services are performed at runtime all adjustments have to be done at the object level. That is, objects belonging to the same class may behave differently when used by different clients.
2. **System-wide refinement:** The required adjustments for customizing a given service must be consistently applied to each object responsible for implementing that particular service.
3. **Non-invasive:** Any customization of a service, requested by a client, cannot require changes in the existing implementations of the objects providing the service, since this would interfere with changes caused by customizations imposed by other clients. Hence, the act of customization must depend on additive adjustments rather than invasive changes.
4. **Extensible:** A system service should be equally customizable regardless of the number of times it has been customized. This implies that an extension¹ implementing a particular customization of a service should itself be open to future evolution. That is, it must be possible to create a new extension of a service as an incremental refinement of a previously created extension.

¹ We use the term extension to refer to the adjustments that have to be performed on a group of objects in order to achieve a particular service customization.

5. **Modular:** It must be possible to reason about a client-specific view in isolation from other client-specific views. This implies that each extension must be encapsulated within an identifiable module.

1.2 Organization of this paper

The remainder of the paper is organized as follows: Section 2 discusses the conceptual model behind our approach. In section 3 we explain our extension to Java. The implementation of the language runtime is outlined in section 4. Section 5 positions our work in relation to previous work. Finally, we conclude in section 6.

2 Conceptual overview of our approach

Central to the way that we approach the evolution of collective object behavior in the presence of simultaneous client-specific views is a model based on the concepts of layers and wrappers. In our model a client-specific view defines a perspective on the entire collective object behavior of the system from the context of the client. Conceptually these individual client-specific views can be considered as points of view through different layers on an underlying group of objects whose collective behavior provides the different services of the system. These underlying objects offer the domain specific functionality and the related attributes that together define the core system. Each client-specific view may just involve a single layer or it may require the participation of multiple layers. Consequently, the same underlying group of objects has different manifestations from within different client-specific views. Realization of a client-specific view as a layer that overlies collective object behavior is very intuitive, since the evolution of a particular service, will most likely involve all the objects that are responsible for implementing that service. That is, the implementation of the collaboration that realizes the service is scattered across this group of objects. Hence, any refinement of such a service will crosscut the code of all of those objects. Such crosscutting refinement of collective object behavior is encapsulated within layers. Consequently, a client-specific view and its respective layers will crosscut the class hierarchy of the core system, affecting the behavior of several objects.

A layer is implemented as a set of wrappers, which have to be applied to the group of objects that realize a particular collective behavior. A layer thereby defines a semantic unit that refines the functionality of the core system. A particular client-specific view can therefore be seen as a combination of the core system with a specific layer. The concept of layers provides the necessary modularity that enables us to reason about each client-specific view in isolation.

A wrapper defines the individual adjustments for an object that has to be modified before it can participate in a client-specific view. It is the use of wrappers that makes our approach runtime applicable and non-invasive. Each wrapper modifies the behavior of exactly one object. Taken together the wrappers within a layer collectively define a crosscutting refinement of the class hierarchies in the core

system. On the request of clients the wrappers belonging to the requested layers are wrapped around the objects within the core system. It is this dynamic integration of wrappers that creates the different client-specific views on the system at runtime.

Extensibility is achieved by allowing layers to be superimposed on one another. This enables the system's behavior to evolve in an incremental additive fashion. Superimposition of layers is based on the idea that a layer added on top of another layer will refine the behavior of the layer below it in an incremental fashion. This means that a layer is a refinement of another layer, if its wrappers refine the behavior of the wrappers contained in the overlaid layer. This superimposition of layers on top of one another arranges layers into a refinement hierarchy that is considered to be complementary to the normal inheritance hierarchy.

We have previously presented our model, named Lasagne [21], as a runtime-architecture that features client-specific customization of component-based systems. In the next section we will explain how the same ideas have been used to create a language extension to Java that supports the same purpose.

3 Language level wrappers in Java

In this section we give an informal specification for our extension of the Java programming language. Our extension is strict in the sense that any existing Java application can be extended without modifying its source code.

Before we proceed, we will take a moment to introduce our terminology: We will refer to a wrapped object as a *wrappee*. The product of combining a wrapper and a wrappee is called an *aggregate*. An object reference's declared type is referred to as its *static type*. The type of the actually referenced object is called the object reference's *dynamic type*. Likewise, we also distinguish between the *static* and the *dynamic wrappee type*. The static type of the wrappee is the class that appears in the wrapper definition. The dynamic type is the actual type of the wrapped object. A method's signature consists of a name and zero or more formal parameters [4].

3.1 Defining wrappers

We introduce two new keywords to the Java programming language for defining wrappers. The keyword `wraps` is used to associate a wrapper definition with a class definition and the keyword `wrapper` is introduced to distinguish the definition of a wrapper from the definition of a class. To some extent the definition of wrappers are related to the definitions of classes, they both define behavior and state, but since we regard them as separate language concepts we chose not to treat wrappers as special cases of classes.

Wrappers can be defined for concrete, abstract and final classes, and applied to instances of those classes as well as instances of their respective subclasses. The later is a consequence of fulfilling the *genericity* requirement [2], which states that wrappers must be applicable to any subtype of the static wrappee type. In the first place it may seem strange that we allow wrapping of final classes, but we chose to

allow wrapping of final classes since we consider wrapping to be complementary to inheritance and therefore not conflicting with the fact that a class is made final to prevent it from being modified.

Figure 1 shows a simple wrapper definition, which states that an instance of the wrapper `X` wraps an instance of the class `A`. The `wraps` clause also declares that the aggregate, which results from combining the wrapper `X` with an instance of the class `A`, is a subtype of not only 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.

```
wrapper X wraps A {  
    ...  
}
```

Figure 1 Syntax for defining a wrapper.

Wrappers are only allowed to wrap classes that are declared public. Hence, it is only the top-level classes of a package that can be subject to wrapping. All classes having package scope are considered to be implementation details of the package. This is comparable to inheritance where the extension of a class or an interface, which belong to a different package, requires that they are accessible outside their respective packages, i.e. that they are declared public.

A wrapper can specialize and extend the behavior of another wrapper by wrapping it. Wrapper specialization will be discussed further in section 3.4.

3.1.1 Overriding of instance methods

Similar to class-based inheritance certain rules must be followed to guarantee type and semantic soundness when wrappers override instance methods. For instance, to guarantee type soundness the return type of the overriding method has to be the same as that of the overridden method, the overriding method must be at least as accessible, the overriding method may not allow additional types of exceptions to be thrown, and class methods cannot be overridden. To also guarantee semantic soundness, the overriding method must be a behavioral refinement of the overridden method [5]. The first set of typing rules can be checked at compile-time, whereas the rule for semantic soundness is the responsibility of the programmer.

The statement; at least as accessible, implies that wrappers can only override public methods, due to the accessibility restrictions of protected and private methods. It is both the public methods declared by the class and inherited public methods that can be overridden. A public method that is not overridden by the wrapper is transparently accessible through the wrapper.

Differentiating on accessibility has the consequence that any method defined by a wrapper that coincidentally happens to have the same signature as a protected or private method of the static wrappee type or any subtype thereof, are completely unrelated – the wrapper method does not override the method in the wrappee. This resembles the overriding rules for classes and subclasses in Java. If the method in the superclass is inaccessible to the subclass the method defined by the subclass does not override the method in the superclass. We will return to the issue of method lookup

and dispatch for un-related methods in section 3.5.1. For now, the standard case is sufficient for the following discussions.

```
class A {
    public void m() {}
}

wrapper X wraps A {
    public void m() {}
}
```

Figure 2 Overriding an instance method

Figure 2 shows the standard case where a method `m` in class `A` is overridden by the method `m` in the wrapper `X`. An invocation of `m` on an aggregate of wrapper `X` and class `A` through an object reference of at least type `A` or `X` results in the execution of the method `X.m()`. In section 3.1.2 we will see how the method `A.m()` can be invoked from within `X.m()` using an invocation expression similar to the use of `super` in subclass relations.

Similar to the case with final classes it may happen that a method overridden by a wrapper eventually will be declared final by a subclass of the wrappee. In this case the wrapper still wraps the overridden final method. Hence, the only implication of making the method final is that no new subclass will be allowed to override the method.

3.1.2 Referencing the wrappee

Within constructors and instance methods of a wrapper, the keyword `inner` references 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 method of their superclass, the keyword `inner` can be used by wrappers to call overridden methods of the wrappee. This is illustrated in Figure 3 where the method `m` declared by class `A` is invoked from the wrapper `X` by the invocation statement `inner.m()`. 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.

```
class A {
    public void m() {...}
}

class B extends A {
    public void m() {...;super.m();...}
}

wrapper X wraps A {
    public void m() {
        ...
        inner.m();
        ...
    }
}
```

Figure 3 Invoking overridden methods in the superclass using `inner` and `super`

The actual method invoked through the `inner` reference depends on the dynamic type of the wrappee. This is exemplified in Figure 3 where a public method `m` declared by class `A` is both overridden by the wrapper `X` and the subclass `B`. An external invocation of `m` on an aggregate consisting of an instance of the wrapper `X`

and an instance of the subclass `B`, through the use of the `inner` reference, results in the method `X.m()` being executed first and then the method `B.m()`. The method `A.m()` will only be executed if it is called through the `super` reference from within the method `B.m()`.

In section 3.4 we will return to the meaning of the keyword `inner` in the presence of conjunctive wrapping. Conjunctive wrapping refers to the situation where multiple wrappers are placed around each other. In our approach conjunctive wrapping is a result of wrapper specialization.

3.2 Grouping wrappers into extensions

The process of customizing an existing system will typically require the creation of multiple wrappers. Taken together these wrappers form a logical layer that constitutes a well-defined system extension. We have chosen to organize such layers of wrappers in separate packages in order to keep the definition of wrappers strictly separated from the definition of the classes that they wrap. Hence, each layer implementing a system extension is encapsulated within an identifiable module, a so-called extension package. Using separate packages to modularize system extensions implies that wrappers are always defined within other packages than the classes that they wrap. All wrappers within an extension package are accessible from outside the package. This implies that only classes that are accessible outside their respective packages, i.e. public classes, can be subject to wrapping. Without this restriction it would be possible for a wrapper to expose a class that otherwise was inaccessible. Currently, each extension package is only allowed to contain one wrapper per class or any subclass thereof. If an extension package contains more wrappers which all are applicable to a wrappee, it will cause an ambiguous wrapping conflict to occur at runtime, because the dynamic wrapping mechanism will be unable to decide which wrapper to selected. This restriction can be enforced at compile time. Allowing an extension package to contain more than one wrapper per class is a non-trivial issue that we are pursuing further. Besides wrappers an extension package may contain classes and interfaces that are part of the extension's implementation. In the next section we explain how the naming of extension packages is used to coordinate the deployment of wrappers.

3.3 Superimposing extensions

Any extension will typically cut across the entire system by requiring wrappers to be applied to various objects at multiple points in the code. This means that the use of wrappers must be coordinated to ensure that a system extension is consistently deployed. Furthermore an extension must be imposed on the target application in a non-invasive and transparent manner (i.e. without the need for modifying existing source code). In response to these requirements we chose not to allow programmers to directly create individual wrapper instances and apply them to objects. Instead we chose an approach in which wrapping is transparently done by the language runtime on demand. That is, the language runtime is responsible of performing and

coordinating the wrapping of application objects with wrappers from an arbitrary number of extensions. To perform this wrapping the language runtime needs to know from what point on in the invocation flow it must start to apply wrappers to application objects (i.e. activate an extension), and it needs a mechanism to propagate this information along subsequent object invocations in order to correctly wrap all objects that participate in the refined collaboration. Information about active extensions can be propagated by including an invocation context with every invocation. To inform the language runtime about when to activate an extension, we have extended the cast mechanism of the Java language to support *constructive downcasting*. In the context of this paper constructive downcasting refers to the dynamic extension of an object by casting the object reference that references 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 subtype of its current type when it is accessed through the cast object reference. A wrapper can only be used to change the dynamic type of an object reference into a subtype if the wrapper wraps the static type of the object reference.

We will now explain how this semantic extension of the language's cast mechanism allows us to superimpose an arbitrary extension on an existing group of collaborating objects through any object reference referencing an object within that group. There are two different ways to use constructive downcasting to superimpose an extension on a group of objects. We will first discuss the case where an object reference is constructively downcast as part of using it in an invocation expression. Then we will discuss the case where an object reference is constructively downcast as part of an assignment expression.

Figure 4 shows how the object reference `a` is temporally downcast to the subtype `x` as part of using it in an invocation expression. The object reference `a` will only have the type `x` for the duration of the execution of the method `x.m()`. When the method `x.m()` returns, the type of `a` is restored to `A`.

```
A a = new A();
((dk.sdu.mip.ext.X)a).m(); // executes X.m()
a.m(); // executes A.m()
```

Figure 4 Temporal extension activation

From the point where the invocation expression `((dk.sdu.mip.ext.X)a).m()` appears in the invocation flow the language runtime will start to automatically apply wrappers around all subsequently invoked objects whose classes are wrapped by the wrappers defined within the extension package `dk.sdu.mip.ext`. The before mentioned invocation context serves to propagate the name of the activated extension along with subsequent invocations.

When an object reference is constructively downcast as part of an assignment expression we have to differentiate between the case where the object reference appears as a member variable within a class and the case where it appears as a local variable within the scope of a method. In the case where the object reference is a member variable the extension will be visible for all methods that can access the

variable. In the case where the object reference is a local variable the extension is only visible from within the method.

An extension that is superimposed on an object reference as part of an assignment expression will be active for all invocations made on that object reference. Figure 5 exemplifies this situation.

```
A a = new A();
X x = (dk.sdu.mip.ext.X)a; // activate extension
```

Figure 5 Extension activation

The extension defined in the extension package `dk.sdu.mip.ext` is now active through the use of the object reference `x`. All invocations of `m` on the instance of class `A` made through the object reference `x` will pass through the wrapper `x`. Any invocation of `m` on the object reference `a` will not go through the wrapper `x`.

Multiple extensions can be superimposed by applying constructive downcasting several times to the same object reference.

Moving the responsibility of instantiating wrappers from the programmers to the language runtime eliminates a number of commonly known problems. For instance, we do not have to consider how to replace object references at runtime. Nor do we have to address such issues as whether it should be allowed to replace the wrappee within an aggregate. Another problem that we avoid is the parameterized constructor problem. The parameterized constructor problem refers to the situation where multiple instances of the same wrapper are applied to the same object, but with different arguments for the parameterized constructor. Since wrappers are instantiated by the language's runtime they can only have a non-arg constructor.

An important property of constructive downcasting is that class cast exceptions cannot occur at runtime, because type casting transforms the referenced object into an object of the casting type when accessed through the cast object reference. The compiler can easily check that the casting type indeed does wrap the static type of the object reference.

3.4 Extending wrappers

Similar to classes, wrappers can form specialization hierarchies by extending the behavior of existing wrappers.

```
package dk.sdu.mip.ext2
wrapper Y wraps X {
    public void m() {
        // ...
        inner.m();
        // ...
    }
}
```

Figure 6 Wrapper specialization

In the context of specialization the keyword `wraps` means that one wrapper extends the behavior of another wrapper. In Figure 6 the wrapper `Y` extends the behavior of the wrapper `X`. As a consequence, the wrapper `Y` wraps the same type as the wrapper `X`. This means that the wrapper `Y` becomes a subtype of the wrapper `X`. The extending wrapper is allowed to add methods to the wrapper being extended. These additional methods are accessible through an object reference that is at least of the same type as the extending wrapper.

Wrappers can only extend wrappers that are defined in other extension packages. A wrapper is not allowed to extend a wrapper that is defined within the same extension package as itself. If we allowed a wrapper to extend a wrapper from the same extension package it would result in the existence of two wrappers for the same static wrappee type within the same extension package, which we do not allow accordingly to section 3.2.

When a wrapper extends another wrapper, the use of the keyword `inner` within the outermost wrapper references 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`.

```
((dk.sdu.mip.ext2.Y)a).m(); // activate extension Y and X
```

Figure 7 Activation of multiple extensions

Activation of an extending wrapper will also activate all wrappers that it extends. Figure 7 exemplifies this with the activation of the wrapper `Y`. At runtime the two wrappers `Y` and `X` will conjunctively wrap the wrappee. The wrapper `X` will be the inner wrapper and the wrapper `Y` will be the outer wrapper. Hence an invocation of `m` on the object reference `a` results in the execution sequence `Y.m()`, `X.m()`, and finally `A.m()`.

Extensions that are related due to wrapper specialization are said to be conjunctive and extensions in which no wrappers are related by a `wraps` clause are said to be disjunctive.

3.5 Supplementary issues

A few remaining issues needs to be explained in order to complete the presentation of our language extension.

3.5.1 Method lookup and dispatch for un-related methods

Wrapping an object will in some situations result in an aggregate that contains multiple methods with identical signatures. This happens in the following cases; 1) When a wrapper defines a method with a signature that is identical to the signature of a protected or private method in the static wrappee type. 2) When a subclass of the static wrappee type declares a public method with a signature that is identical to the signature of a method defined by the wrapper. 3) When a wrapper incidentally defines a method with the same signature as a public static method defined by the static wrappee type or any subclass thereof.

To ensure that the right method is executed we need an alternative method lookup and dispatch process. We have chosen to implement a solution in which invocations made through an object reference of the wrapper type will result in the execution of the method defined by the wrapper. And in case an invocation occurs through an object reference that is of the static wrappee type or any subclass thereof, the method defined by the dynamic wrappee type will be executed. The rationale for this choice is that a client who references the aggregate through an object reference of the wrapper type expects the method to behave as defined by the wrapper. Similarly, a client who references the aggregate through an object reference of the static wrappee type expects the method to behave as defined by the static wrappee type or any subclass thereof.

3.5.2 Binding of self in the presence of wrappers

When the wrappee is called through a wrapper the self parameter `this` must either be bound to the wrapper or to the wrappee. By binding the self parameter to the wrapper we will get delegation and by binding it to the wrappee we get consultation. With delegation any self calls on a wrapped method will go through the wrapper whereas the wrapper will be skipped when consultation is chosen. We introduce the method modifier `delegate` to control the selection between delegation and consultation on a per method basis. If we assume that delegation is required in Figure 2 the signature of the method `m` must be changed to `public delegate void m()`. Consequently, unless otherwise specified, consultation is the default choice for all wrapper methods. Alternatively we could have introduced a method modifier, such as `consult`, for designating consultation instead. Leaving delegation as the default choice.

4 Implementing the runtime system

In this section we describe the implementation of the runtime system underlying our language extension. The description is an overview, summarizing main features and applied technologies, rather than an in-depth detailed description of intriguing implementation issues, since the latter is beyond the scope of this paper.

At the conceptual level the design of the runtime system is related to a commonly well known group of design patterns for supporting object extensibility; Decorator [15], Extension Object [16], and Role Object [17]. Because of the close relation, we will adapt the terminology of those patterns in the following discussion. Our design, however, differs from those patterns in more than one way. First of all, we divide an object extension (i.e. a wrapper in our case) into two separate objects instead of one. These two objects are respectively a proxy object and an implementation object. Clients obtain proxy objects from an extensible object (i.e. a wrappee) in order to use specific object extensions. The implementation objects implement the state and behavior of the wrappers. At runtime proxy objects are created specifically for different clients whereas all clients share the same implementation objects. The sharing of implementation objects allows clients to share wrapper state. From the clients' point of view, the proxy is the aggregate. Secondly, the static inheritance relationship, where all extension classes are subclasses of the extensible object's

class, is replaced with a dynamic inheritance relationship where an instance of an extension class (i.e. a proxy object) gets the actual type of the extensible object that it is applied to as its superclass. We had to replace the static inheritance relationship to fulfill the requirement that the aggregate (i.e. the proxy class) is a subtype of the dynamic type of the wrappee. Finally, a new dispatch algorithm is added for all public method of an extensible object. As part of performing method lookup and dispatching the algorithm consults the current invocation context for the extension identifiers of all active extensions. The algorithm executes the implementation objects of active extensions before it executes the wrapped method. Extension identifiers are inserted into the current invocation context by the proxy objects. A proxy object inserts its corresponding extension identifier when it is invoked. We use the Java version of the design pattern Thread Specific Storage [18] to maintain a pool of local variables for each active thread.

To translate the wrapper definitions into Java, we wrote a simple preprocessor. This preprocessor translates each wrapper definition into an interface and the above discussed proxy class and implementation class. The additional interface is needed to represent the type of the wrapper within the source code of clients. It is this interface that is used in the constructive downcast of a wrappee. Both the proxy class and the wrapper implementation class implements the wrapper interface.

The class of a potential wrappee is made extensible when it is first loaded into the JVM. To perform this task, we use Javassist, which is a load-time reflective system for Java developed by Shigeru Chiba [19]. To apply a wrapper to a wrappee we have to replace all constructive downcasts, e.g. cast expressions, with the corresponding `getExtension()` method call on the wrappee. For this purpose we use OpenJava. OpenJava is an extensible language based on Java, developed by Michiaki Tatsubori [20], which allows us to replace a cast expression within the source code with a new expression. In order to work, the replacement of cast expressions requires, that developers use OpenJava to write all new clients. However, from the developers' perspective the only noticeable difference is the change of the file name extension on the source files.

The current version of the pre-processor does not perform syntax analysis of the parsed wrappers definitions. This means that syntax errors are first caught at a later stage by the OpenJava compiler. Syntax analysis should be included in the preprocessor to ease the job of the developers.

5 Related work

State-of-the-art separation of concerns techniques such as Aspect-oriented Programming [6], Hyperspaces [7], Mixin Layers [8], and Adaptive Plug and Play Components [12] allow extension of a core application with a new aspect/subject/layer/collaboration, by simultaneously refining state and behavior at multiple points in the application in a non-invasive manner. These approaches however mainly operate at the class-level, while we impose extensions at the instance-level, enabling run-time customization. Our approach also supports customization with multiple, independent context-specific views on the core system.

This is useful for customizing distributed systems, since a distributed service may have, during its lifetime, several remote clients, each with different customization needs. This feature is not really well supported in the above class-based extension techniques.

Composition Filters [9] composes non-functional aspects on a per object interaction basis. Composition Filters however does not have any support for consistent and crosscutting refinement; the composition logic enforcing the integration of an extension that crosscuts the system is scattered across multiple object interactions, thus difficult to update consistently in one atomic action. In Our approach, the composition logic is completely encapsulated within the invocation context propagating along the currently ongoing collaboration. However, since the Composition Filter model is very generic, the necessary coordination mechanisms can be added. This can be done by delegating incoming messages to a meta-object that implements the necessary coordination mechanisms. Linda Seiter et al. [10] 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 invocation context of our approach. However, context relations have overriding semantics and do not allow selective combination of extensions.

Mira Mezini [11] presented the object model Rondo that does well support dynamic composition of object behavior without name collisions. However, there is no support mentioned for specifying behavior composition on a per collaboration basis. Martin Büchi et al. [2] proposed Generic Wrappers to customize components developed by different teams. Generic wrappers are special classes that wrap instances of a given reference type (i.e. interface or class). Their approach performs wrapping on a single object basis whereas our approach works on the basis of collaborating objects.

The work of Klaus Ostermann [14] is especially related to our work because it addresses both the problem of simultaneously extending the behavior of a group of collaborating objects in a transparent manner while keeping the original behavior intact, and the problem of extending a collaboration that has already been extended. He proposes delegation layers, a result of combining delegation with virtual classes, as a mean to define functionality that affects the behavior of a set of different classes whose objects take part in the same collaboration. A delegation layer is organized as virtual classes nested within an outer class. A collaboration is now extended by overriding the nested virtual classes by concrete classes in a class that extends the outer class. Two collaborations are composed by composing their respective outer classes by means of delegation. This provides a simple, yet powerful mechanism to composing crosscutting collaborations at run-time. Since all application development must express object collaboration in terms of virtual nested class the evolution of an existing code base is excluded. Our approach allows the evolution of existing code bases since we superimpose behavioral changes onto the group of effected objects.

6 Conclusion

The contribution of our work is a novel extension to the Java programming language that allows simultaneous clients to dynamically extend the collective behavior of a group of objects in an incremental manner that is additive rather than invasive. This establishes the foundation for performing context-sensitive stepwise program refinement at runtime. Furthermore we have identified a number of language design challenges that we believe are fundamental to the evolution of collective object behavior in presence of simultaneous client-specific views.

Our preliminary experiments with the language extension and our previous experiences within the field of dynamic component adaptation [21], [22], [23], [24] reinforce our belief that new language-based technologies will eventually lead to flexible software systems that are better engineered. However, real-life pilot projects are required to validate whether our language extension has the right mix of language features and whether developers will actually use it as their applications are evolving.

A particular interesting issue that remains open for future research is what effect, if any at all, the proposed language extension will have on the software development process.

References

- [1] Bosch J.: Superimposition: A Component Adaptation Technique. In: Information and Software Technology (1999)
- [2] Büchi M., Weck W.: Generic Wrappers. In: proceedings of ECOOP 2000. Lecture Notes in Computer Science, Vol. 1850. (2000) p. 201 ff.
- [3] Brant J, Foote B., Johnson R.E., Roberts D.: Wrappers to the Rescue. In: Proceedings of ECOOP '98. Lecture Notes in Computer Science, Vol. 1445. Springer-Verlag, (1998) p. 396 ff.
- [4] Gosling J., Joy B., Steele G.: The Java™ Language Specification. Addison Wesley, (1996)
- [5] America P.: Designing an object-oriented programming language with behavioral subtyping. In: Foundations of Object-Oriented Languages. Lecture Notes in Computer Science, Vol. 489. Springer-Verlag, (1991) 60-90
- [6] 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) p. 220 ff.
- [7] Tarr P., Ossher H., Harrison W., Sutton Jr. S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of ICSE'99. (1999)
- [8] 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) p. 550 ff.
- [9] Aksit M., Wakita K., Bosch J., Bergmans L. and Yonezawa A.: Abstracting Object-Interactions Using Composition-Filters. In: Guerraoui R., Nierstrasz O. and Riveill M. (Eds.): Object-Based Distributed Processing. Springer-Verlag, (1993) 152-184
- [10] Seiter L., Palsberg J., Lieberherr K.: Evolution of Object Behavior using Context Relations. In: IEEE Transactions on Software Engineering, Vol. 24(1) (1998) 79-92

- [11] Mezini M.: Dynamic Object Evolution without Name Collisions. In: Proceedings of ECOOP'97. Lecture Notes in Computer Science, Vol. 1241 Springer-Verlag, (1997) p. 190 ff.
- [12] 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) 97-116
- [13] Hermann S., Mezini M.: PIROL, A Case-Study for Multi-Dimensional Separation of Concerns in Software Engineering Environments. In: Proceedings of OOPSLA'2000. Sigplan Notices, Vol. 35, No. 10. (2000)
- [14] Ostermann K.: Dynamically Composable Collaborations with Delegation Layers. In: Proceedings of ECOOP '02. Lecture Notes in Computer Science, Vol. 2374 Springer-Verlag, (2002)
- [15] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley (1995) 175-184.
- [16] Gamma E.: Extension Object. In: Martin R., Riehle D., Bruschmann F. (Eds.): Pattern Languages of Program Design 3. Addison-Wesley (1998) 79-88
- [17] Bäumer D., Riehle D., Siberski W. and Wulf M.: Role Object. In: Harisson N. (Eds.): Pattern Languages of Program Design 4. Addison-Wesley (2000) 15-32
- [18] Schmidt D.C., Harrison T.H., and Pryce N.: Thread-Specific Storage - An Object Behavioral Pattern for Accessing per-Thread State Efficiently. In the C++ Report, SIGS, Vol. 9, No. 10, November/December (1997)
- [19] Chiba S.: Load-time Structural Reflection in Java. In: proceedings of ECOOP 2000, Lecture Notes in Computer Science, Vol. 1850. Springer Verlag, (2000) p. 313 ff.
- [20] Tatsubori M, Chiba S., Killijian V, Itano K.: OpenJava: A Class-Based Macro System for Java. In: Cazzola W, Stroud R.J., Tisato F. (Eds.): Reflection and Software Engineering. Lecture Notes in Computer Science, Vol. 1826. Springer-Verlag, (2000) 117-133
- [21] Truyen E., Vanhaute B., Joosen W., Verbaeten P., Jørgensen B.N.: Dynamic and Selective Combination of Extensions in Component-Based Applications, In: Proceedings of ICSE. (2001) 233-242
- [22] Truyen E., Vanhaute B., Joosen W., Verbaeten P., Joergensen B.N.: A Dynamic Customization Model for Distributed Component-Based Applications. In: Proceedings of DDMA'2001. (2001) 147-152
- [23] Truyen E., Jørgensen B.N., Joosen W.: Customization of Component-based Object Request Brokers through Dynamic Reconfiguration. In: Proceedings of TOOLS EUROPE. IEEE (2000) 181-194
- [24] Jørgensen B.N., Truyen E., Matthijs F., Joosen W.: Customization of Object Request Brokers by Application Specific Policies. In: Proceedings of Middleware 2000. Lecture Notes in Computer Science, Vol. 1795. Springer Verlag, (2000) 144-164