# Language Constructs for Context-oriented Programming

## An Overview of ContextL

Pascal Costanza
Vrije Universiteit Brussel
Programming Technology Lab
B-1050 Brussels, Belgium
pascal.costanza@vub.ac.be

Robert Hirschfeld
DoCoMo Communications Laboratories Europe
Future Networking Lab
D-80687 Munich, Germany
hirschfeld@acm.org

## ABSTRACT

ContextL is an extension to the Common Lisp Object System that allows for Context-oriented Programming. It provides means to associate partial class and method definitions with *layers* and to activate and deactivate such layers in the control flow of a running program. When a layer is activated, the partial definitions become part of the program until this layer is deactivated. This has the effect that the behavior of a program can be modified according to the context of its use without the need to mention such context dependencies in the affected base program. We illustrate these ideas by providing different UI views on the same object while, at the same time, keeping the conceptual simplicity of object-oriented programming that objects know by themselves how to behave, in our case how to display themselves. These seemingly contradictory goals can be achieved by separating class definitions into distinct layers instead of factoring out the display code into different classes.

## Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Object-oriented Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

## General Terms

Languages

## Keywords

Context-oriented programming, layers, views, dynamic scope

## 1. INTRODUCTION

Before we explore ContextL and its features in the following sections, let us first go back and recall how objects are introduced to novices.

### 1.1 Back to the Roots

Among prominent domains to motivate object-oriented programming, graphics and people are the most widely used examples. A reason for their popularity is the fact that one can easily introduce the idea that objects know how to behave, that is how to react to messages. Often *draw* or *display* messages are used for pedagogical purposes because they allow demonstrating immediately visible effects on a computer screen. So, for example, *rectangles* and *persons* can be introduced as follows, using a Java-like syntax.

```
class Rectangle {
  int x, y, width, height;
  void draw() { ... }
}

class Person {
  String name, address, city, zipCode;
  void display() { ... }
}
```

However, when programs become more complex, the code for displaying objects is usually *not* contained in the classes to be displayed because there is a need to have different views on the same objects, often at the same time. Therefore, such code is separated into *view* objects that need to be notified of changes to *model* objects (such as instances of `Rectangle` or `Person`), leading to variants of the well-known *Model-View-Controller* (MVC) framework originally introduced with Smalltalk [22]. Unfortunately, this distribution of responsibilities that conceptually belong to a single object complicates the original simplicity of the object-oriented paradigm. For this reason, some more recent object systems like Self and Squeak have even changed their frameworks for presenting objects on the screen back to the original idea that objects maintain their own knowledge about how to display themselves (Morphic, [28]). However, with that they lose the desired property to offer different views of the same objects.[1] ContextL provides an alternative approach that both keeps the conceptual simplicity that all of an object's

[1]Squeak allows us to select whether we want to use Morphic or MVC for the presentation of objects, but this does not change the main argument of our introduction: The choice is still between either associating the display behavior with the classes to be displayed, or being able to create different views on the same objects, but not both at the same time.

behavior is indeed associated with that object and still allows an object to be viewed in different ways depending on the context.

### 1.2 Context-oriented Programming

ContextL is one of the first programming language extensions that explicitly supports a programming style that we call *Context-oriented Programming.*[2] It is an extension of the Common Lisp Object System (CLOS, [2]), but the features we describe are conceptually independent of that particular object model.[3]

In the subsequent sections, we use the introductory *person* class as an example. However, we are convinced that the desire to have different views with different behavior on the same objects is a recurring theme in other scenarios that range from personalization, adaptation to different kinds of devices (desktop, mobile, etc.), diverging business rules for varying provider-customer relations up to enforcing different security policies for different users [17].

From a programmer's point of view, the goal of Context-oriented Programming is to avoid having to spread context-dependent behavior throughout a program. Currently, the only way to introduce context-dependent behavior into a program is either by inserting `if` statements everywhere that check for the context in which a program is running, violating one of the fundamental principles of object-oriented programming, namely to avoid `if` statements for achieving polymorphic behavior, or else by factoring out the context-dependent behavior into separate objects that can be substituted according to the context in which a program is used. Both approaches lead to unnecessarily complicated code that is hard to comprehend and even harder to maintain. Furthermore, they can only be applied for context-dependent behavior that are anticipated in the software development process. The MVC framework is an example of such a distribution of behavior over different objects and requires anticipation of notifying observers of (relevant!) state changes in the model objects. See for example the discussion of the Observer pattern in [10] for an overview of the complexities that are involved in such an approach.

We want different views on the same object or the same set of objects. With Context-oriented Programming, instead of separating out the display into different classes, we separate out class definitions into separate layers. Depending on the context of use, we can then select different layers for further program execution. As is shown in this paper, this avoids the complexities of distributing behavior across objects as described above. The principal notion of such layers has been suggested before ([1, 27], cf. the section on related work in this paper) but in ContextL, we have combined this idea with the notion of dynamically scoped layer activation (see below), and we are convinced that this combination results in a viable approach for expressing context-dependent behavior.

[2]Thanks to Wolfgang De Meuter for suggesting this term.

[3]For example, we are also working on similar extensions to Smalltalk and Tweak called *ContextS* and *ContextT* respectively.

## 2. CONTEXTL

In the following sections, we introduce the most significant constructs of ContextL by using the introductory *person* class as an example. In order to define that class and enable the subsequent use of the ContextL features, we have to define the class as a *layered class.*[4]

```
(define-layered-class person ()
  ((name :initarg :name
         :accessor person-name)))
```

This code snippet defines the class `person` with no superclasses and one slot (field) `name` that can be initialized with `:name` and accessed with `person-name`.[5]

### 2.1 Layers

Layers are the essential extension introduced by ContextL that all subsequent features of ContextL are based on. Layers can be created by the `deflayer` macro, just like this.

```
(deflayer employment-layer)
```

Layers basically consist of only a name and no further properties of their own. However, other constructs of ContextL can explicitly refer to such layers and add definitions to them accordingly. There is a predefined layer named `t` that denotes the root or default layer that all definitions are automatically placed in when they do not explicitly name a layer. For example, our definition of class `person` (see above) is implicitly placed in the root layer.[6]

Layers can be activated in the dynamic scope of a program.

```
(with-active-layers (employment-layer)
  ... contained code ...)
```

Dynamically scoped layer activation has the effect that the layer is only active during execution of the *contained code*, including all the code that the *contained code* calls directly or indirectly. When the control flow returns from the dynamically scoped layer activation, the layer is deactivated again. Layer activation can be nested, which means that a layer can be activated when it is already active. However, this effectively means that a layer is always active only once, i.e. nested layer activations are just ignored. This also means that on return from a dynamically scoped layer activation, a layer's activity state actually depends on whether it was already active before or not. In other words, dynamically scoped layer activation obeys a stack-like discipline.

[4]In fact, `layered-class` is a CLOS metaclass, but we hide this with the `define-layered-class` macro to simplify the presentation in this paper.

[5]In Java parlance, the `:initarg` keyword implicitly creates a parameter for the `person` constructor, and the `:accessor` keyword implicitly creates getter and setter methods for the respective field.

[6]In Common Lisp, it is common to denote "general", "root" or "default" concepts by `t` or `nil` since they are the canonical truth and false values in that language.

Furthermore in multithreaded Common Lisp implementations, dynamically scoped layer activation only activates layers for the currently running thread. If a layer is not active in other threads, it will remain so unless it is incidentally also activated in some of them.

## 2.2 Layered Classes

We have already seen `define-layered-class` being used for the definition of the `person` class. We can confine the definition of a class to a specific layer.

```
(define-layered-class employer
  :in-layer employment-layer ()
  ((name :initarg :name
         :layered-accessor employer-name)))
```

This confinement does not have a useful effect yet: The class can still be instantiated from anywhere. However, placing a class definition in a specific layer becomes interesting when we use layers to add to the definition of a class that is already defined in other layers.

```
(define-layered-class person
  :in-layer employment-layer ()
  ((employer :initarg :employer
             :layered-accessor person-employer)))
```

Here, the original class `person` is not replaced, the class `person` still has its original slot `name`. It additionally gets the slot `employer` contained in the `employment-layer`. Since the accessors defined for `employer-name` and `person-employer` are declared as `:layered-accessor`, the slots are actually only visible when the `employment-layer` is active (activated by `with-active-layers`). When it is not active, an error is thrown when these accessors are called. This allows us to restrict the view of slots to certain layers.

## 2.3 Layered Functions

So far, we have not yet defined any behavior for the `person` and `employer` classes. In order to simplify the presentation, we do not provide code examples that implement a full-blown graphical display of instances of these classes, but restrict ourselves to textual display. Still, this illustrates the approach how to provide different views with layers. This is achieved by way of a *layered generic function.*[7]

```
(define-layered-function display-object (object))
```

This code defines a function that takes one parameter `object`. In order to make the function perform some useful behavior, we have to define methods on it. Here is the method definition that displays a person object for the root layer.

[7]CLOS is based on the notion of generic functions that associates methods with functions instead of classes. This means that in CLOS, classes only define state but not behavior. Layered functions are actually instances of the generic function class `layered-function`, but again this is hidden by the `define-layered-function` macro.

```
(define-layered-method display-object
  :in-layer t ((object person))
  (format t "Person~%")
  (format t "Name: ~A~%" (person-name object)))
```

Such a method definition takes an additional `:in-layer` declaration. Here, the layer is `t` that represents the root layer. The method's only argument `object` is specialized on the class `person`.[8]

Here are two other method definitions for `display-object` that are placed in the `employment-layer`.

```
(define-layered-method display-object
  :in-layer employment-layer
  ((object employer))
  (format t "Employer~%")
  (format t "Name: ~A~%" (employer-name object)))

(define-layered-method display-object
  :in-layer employment-layer :after
  ((object person))
  (display-object (person-employer object)))
```

The first method, with `object` specialized on `employer`, is similar to the previous method for `person` but additionally restricted to the `employment-layer`. The second method adds behavior to the previous method for `person` that is to be executed `:after` the previous one has been executed. The fact that the `:in-layer` declaration is bound to the `employment-layer` has the effect that this `:after` method is only executed when the `employment-layer` is active but not otherwise. The difference can be noticed in the following transcript.

```
> (defvar *vub*
    (make-instance 'employer
      :name "Vrije Universiteit Brussel"))

> (defvar *pascal*
    (make-instance 'person
      :name "Pascal Costanza"
      :employer *vub*))

> (display-object *pascal*)

Person
Name: Pascal Costanza

> (with-active-layers (employment-layer)
    (display-object *pascal*))

Person
Name: Pascal Costanza
Employer
Name: Vrije Universiteit Brussel
```

[8]`format` is a Common Lisp function for formatted output, similar to `printf` in C. `(format t "... control string ..." ...)` means to print the control string to standard output (represented by `t`), by optionally taking further parameters that are spliced into the control string.

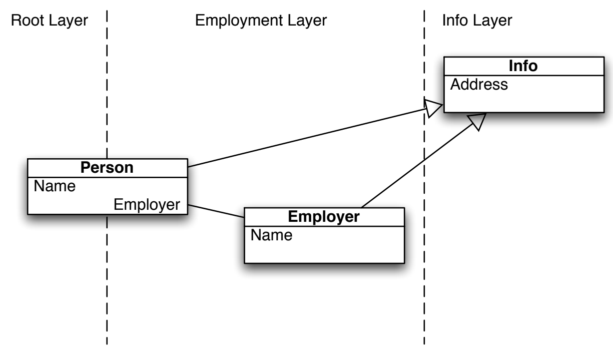

**Figure 1: The class *Person* has a field *Name* in the Root Layer and a field *Employer* in the Employment Layer. The classes *Person* and *Employer* inherit from the class *Address* in the Info Layer.**

Please note again that layer activation is confined to the currently running thread. This means that the latter output that contains the employment information does not show in other threads unless `employment-layer` is incidentally also active in some of them.

To make things more interesting, we add another layer that adds address information to both persons and employers.

```
(deflayer info-layer)

(define-layered-class info-mixin
  :in-layer info-layer ()
  ((address :initarg :address
            :layered-accessor address)))

(define-layered-method display-object
  :in-layer info-layer :after ((object info-mixin))
  (format t "Address: ~A~%" (address object)))

(define-layered-class person
  :in-layer info-layer (info-mixin)
  ())

(define-layered-class employer
  :in-layer info-layer (info-mixin)
  ())
```

Here, we define a mixin class `info-mixin` that defines one slot `address`, and a method for `display-object` that prints the address information. Then, `info-mixin` is added as a superclass to the classes `person` and `employer`.[9] See Figure 1 for the resulting class hierarchy.

[9]CLOS supports multiple inheritance, so we can arbitrarily add superclasses to a given class in different layers.

Here is a transcript that shows a use of the `info-layer`.

```
> (defvar *docomo*
    (make-instance 'employer
      :name "DoCoMo Euro-Labs"
      :address "Munich"))

> (defvar *robert*
    (make-instance 'person
      :name "Robert Hirschfeld"
      :employer *docomo*
      :address "Ilmenau"))

> (with-active-layers (employment-layer)
    (display-object *robert*))

Person
Name: Robert Hirschfeld
Employer
Name: DoCoMo Euro-Labs

> (with-active-layers (employment-layer info-layer)
    (display-object *robert*))

Person
Name: Robert Hirschfeld
Address: Ilmenau
Employer
Name: DoCoMo Euro-Labs
Address: Munich
```

Activation of the `employment-layer` results in display of the `employer` slot of the `person` object, as before. Note that additional activation of the `info-layer` activates display of the address information for *both* the `person` and the `employer` instance.

Note that the `:in-layer` declaration for layered methods is optional. If it is omitted, that declaration is implicitly bound to the root layer. Further note that the `:in-layer` declaration has to occur before any method qualifiers, like `:before`, `:after` or `:around`.

## 2.4 Special Slots

ContextL provides the `dletf` framework for binding values to arbitrary places, again with dynamic scope. In Common Lisp parlance, a place denotes a conceptual location to which values can be stored and read, like for example slots of an object or components of an array. A slot for a layered class can be declared to be "special", which means that such a slot can be bound to a new value in the control flow of a program. This allows us to provide a more generic display functionality. We define the following layer.

```
(deflayer generic-display-layer)

(define-layered-class displayed-slots-mixin
  :in-layer generic-display-layer ()
  ((displayed-slots :special t
                    :initform '()
                    :accessor displayed-slots)))
```

```
(define-layered-class person
  :in-layer generic-display-layer
  (displayed-slots-mixin)
  ())

(define-layered-class employer
  :in-layer generic-display-layer
  (displayed-slots-mixin)
  ())

(defgeneric generic-display (object))

(defmethod generic-display (object)
  (format t "~A" object))

(defmethod generic-display
  ((object displayed-slots-mixin))
  (let ((slots (displayed-slots object)))
    (if slots
        (dolist (slot slots)
          (format t "~&~A: " slot)
          (generic-display
            (slot-value object slot)))
      (format t "No slots to display.~%"))))
```

The (plain) generic function `generic-display` takes an object as a parameter, and if it is an instance of `generic-display-mixin` it will print the slots that are associated with the special slot `displayed-slots`. The latter is introduced into persons and employers by way of the mixin class `generic-display-mixin`. Since it is a special slot, the actual slots to be displayed can be selected on a case-by-case basis via the `dletf` operator. The slot values are retrieved by the standard CLOS function `slot-value` that takes an object and a slot name as parameters. Here is a transcript that uses these definitions.

```
> (with-active-layers (generic-display-layer)
    (dletf (((displayed-slots *robert*)
                                  '(name employer))
            ((displayed-slots *docomo*)
                                  '(name address)))
      (generic-display *robert*)))

Name: Robert Hirschfeld
Employer:
Name: DoCoMo Euro-Labs
Address: Munich
```

The effect of `dletf` is similar to the effect of rebinding a dynamically scoped "special" variable: All forms that refer to the respective `displayed-slots` slot also see the new values in the dynamic extent of the `dletf` form. It is important to note that `dletf` does not introduce new access functions, like `displayed-slots` in the example above, that shadow the previously visible accessors, but that the change to slot values is indeed performed on the actual slots of the `*robert*` and `*docomo*` objects. This means that if those objects are, or have been, passed around to other functions, the new slot values will be visible during the extent of the `dletf` form, no matter how they are accessed.

Again, ContextL ensures that a new binding created via `dletf` is confined to the currently executing thread. For our example, this means that the specific set of slots to be displayed is different from other threads unless the same set of slots is incidentally also selected in some of them.

## 2.5 Layered Slots

In the previous sections, we have illustrated how to display objects in varying ways while keeping the conceptual simplicity that all presentation code is associated with the classes to be displayed. However, the simplification to just textual output misses an important feature that is typical for graphical user interfaces: Whenever the value for a slot of a displayed object is changed, the graphical representation on the screen is automatically updated in order to always present the current state of objects. We need a way to be notified of such changes, and we do this by mimicking the way slot access is implemented in CLOS.

According to the CLOS Metaobject Protocol (MOP, [19]) specification, all slot accesses, including those of the automatically generated accessors, go through the function `slot-value` that takes an object and a slot name as parameters. In turn, that function calls the generic function `slot-value-using-class` that takes the class of the object, the object itself and a representation of the slot as parameters. That scheme enables the definition of one's own slot access behavior by way of providing a method for `slot-value-using-class`, like this:[10]

```
(defmethod slot-value-using-class
  ((class persistent-class) object slot)
  (... access a database ...))
```

In ContextL, slots can be declared to be `:layered` which has the effect that slot accesses go through similar `slot-value-using-layer` functions. Here is a sketch of a `presentation-layer` that takes advantage of this protocol.

```
(deflayer presentation-layer)

(define-layered-class view-mixin
  :in-layer presentation-layer ()
  ((active-views :accessor active-views
                 :initform '())))

(define-layered-method setf-slot-value-using-layer
  :in-layer presentation-layer :after
  (value class (object view-mixin) slot)
  (... notify active views for 'object' ...))

(define-layered-class person
  :in-layer presentation-layer (view-mixin)
  ((name :layered t)
   (employer :layered t)))
```

[10]The call of `slot-value-using-class` is only the specified effect of accessing slots. For efficiency reasons, CLOS implementations take care of bypassing the slot access protocol when it will provably result in the standard behavior.

```
(define-layered-class employer
  :in-layer presentation-layer (view-mixin)
  ((name :layered t)))

(define-layered-class info-mixin
  :in-layer presentation-layer (view-mixin)
  ((address :layered t)))
```

So the heart of being notified about changes to slots is by defining a method on the layered slot writer `setf-slot-value-using-layer` and activating the layered slot access protocol for the slots of interest by way of the `:layered` keyword.[11] Whenever a view is created for a given object (not shown here), it must add itself with a collection of active views for that object which is stored in the slot `active-views` for that object. Again, all code that deals with display of objects can be associated with the classes to be displayed.[12]

This approach finally needs a construct for globally activating a layer because updates of graphical representations should always be performed. This can be achieved by the following simple statement.

```
(ensure-active-layer 'presentation-layer)
```

## 2.6 Summary

ContextL provides the following features:

- *Layers* are the basic construct that enables grouping class and method definitions and activating them in some dynamic scope of a program.
- *Layered classes* are classes that can have partial definitions in different layers. The slots of layered classes can additionally be declared to be...
  - ...*layered*: Such slots are accessed through the layered slot access protocol.
  - ...*special*: Such slots can have different bindings in different threads at the same time.[13]

  Furthermore, the accessor for a slot can be a *layered accessor* which means that such a slot can only be accessed when the respective layer is active in which the corresponding class is defined.
- *Layered functions* are functions that can have different methods associated with them for different layers. Methods that are defined in a given layer are only ever executed when the respective layer is active. Layered accessors and the layered slot access functions are actually layered functions.[14]

All these new constructs are integrated with the existing Common Lisp Object System, and this is achieved by implementing ContextL purely on top of the CLOS Metaobject Protocol. It is currently supported in seven major Common Lisp implementations and can be downloaded from http://common-lisp.net/project/closer. This paper does not provide any details on how ContextL is implemented but this will be reported in a future publication. Note that the handling of multiple inheritance and multiple dispatch is reused from CLOS and not reimplemented in ContextL.[15]

Only classes and functions that are explicitly declared to be layered can partake in layered activations of new partial definitions. This may be regarded as a restriction of ContextL because one has to anticipate which classes and functions should be amenable to context-specific changes. However on the other hand, there would also arise a need to protect some classes and functions from context-specific changes for safety reasons, were layered classes and functions to be the default when nothing else is explicitly specified. So this effectively boils down to the question whether context dependency for classes and functions should be the default or not. We prefer a programmer's explicit decision in that regard, and this also better fits the design decisions inherent to the Common Lisp Object System. Note that this also requires very little anticipation when compared to the more cumbersome alternative approaches described in the introduction to this paper. Furthermore, it is clear that one cannot expect full unanticipated software evolution anyway but that the inner workings of a program need careful design so that it becomes adaptable to changing requirements from the outside. This has already been discovered and described as the notion of Open Implementations in [20].

[11]Slot definitions mentioned in the partial class definitions here are merged with slot definitions of other layers when the respective slot definitions have the same slot name, just as is the case for merging slot definitions across superclasses in plain CLOS.

[12]The presentation of layered slots is heavily simplified for this paper, but the essential characteristics are kept. The differences to the actual implementation in ContextL are as follows: The general slot-writer is actually called `(setf slot-value-using-layer)`. The parameter list is also slightly different in that it additionally contains the first-class function to perform the actual slot write access. This slot writer function is passed for technical reasons, in order to ensure that `:before` methods on `(setf slot-value-using-layer)` are actually performed before the write access, that in turn is performed by the primary method on `(setf slot-value-using-layer)`, and `:after` methods are accordingly performed afterwards. The other layered slot accessors have similarly extended definitions. Layered slot accessors are not described in further detail in this paper. Fully functional demo code will be provided on the ContextL website.

[13]Special slots have already been described in [8, 9].

[14]Precursors of layered functions are *dynamically scoped functions* [7] and *special functions* [8] but they are fundamentally different in that they cannot be grouped into layers and must be defined one at a time.

[15]For those who are well-versed in CLOS: With regard to multiple inheritance, the superclasses of the root layer definition of a given class always come before the superclasses of any other layer. The order of the other layers is randomly chosen. If one requires a specific ordering of layers, this can be achieved by way of layer inheritance, similar to class inheritance but not further described in this paper. With regard to multiple dispatch, the layer declaration for a method is indeed a parameter to an internal secondary generic function and passed a representation of currently active layers. The layer argument has least precedence in the argument precedence order, so that methods defined in different layers can be interweaved. The ordering of methods of different layers is determined by layer activation such that methods of more recently activated layers have more precedence than methods of less recent layers.

## 3. RELATED WORK

### 3.1 Multiple Inheritance

The basic notion that different partial classes can contribute to a compound final class was one of the original motivations for Flavors, the first object system for a Lisp dialect and one of the precursors of CLOS [5, 30]. In order to achieve combinations of features, Flavors originated multiple inheritance so that a class can combine the functionality of several other classes by just inheriting from all of them at the same time. As a motivating example, Howard Cannon's original proposal uses a graphical `WINDOW` class that is then further extended to a `WINDOW-WITH-BORDER`, `WINDOW-WITH-LABEL`, `WINDOW-WITH-LABEL-AND-BORDER`, and so forth. It is clear that this leads to a combinatorial explosion of possibilities. ContextL avoids this combinatorial explosion by deferring the combination of different layers to runtime.

### 3.2 Aspect-oriented Programming

Aspect-oriented programming [24] is an umbrella term for a family of approaches that support modularization of crosscutting concerns. A crosscutting concern is some functionality that should be exhibited by a program but that does not fit well with its dominant decomposition (usually into classes), like for example logging or notification. The basic concepts that are implicitly or explicitly common to all aspect-oriented approaches are: join-points – the points in the execution of a program where additional behavior can be woven in; pointcuts – expressions that define sets of such join-points; advice – the code that is to be woven in before, after or around the join-points of a given pointcut.

Although Context-oriented Programming seems to have an aspect-oriented feel to a certain degree, ContextL does not qualify as aspect-oriented. The main reason is that ContextL does not provide a notion of join-points or pointcuts. The `:before`, `:after` and `:around` methods of CLOS contribute to the definition of a generic function but not to some additional behavior for a pointcut, unless one understands a CLOS generic function as a degenerated form of join-points and pointcuts. Furthermore, the layers in our motivating example do not modularize a crosscutting concern – the root layer already contains code for presenting objects on the screen, and the different layers merely define variants of object presentation. This is fundamentally different from the goal of aspect-oriented programming to achieve a complete separation of crosscutting concerns.

Nevertheless, there exist a number of aspect-oriented approaches that allow for dynamic aspect weaving, most notably Steamloom [3] and our AspectL [8] and AspectS [15]. These approaches share the notion that aspects should, or at least can, be activated in the dynamic scope of a program.[16] Apparently, dynamic layer activation as in ContextL and aspect-oriented modularization are independent concepts that may even turn out to be orthogonal. This needs to be explored further in the future.

See [3] for a discussion of other approaches to dynamic aspect weaving.

[16]See [16] for a paper that shows how this can be achieved in AspectS.

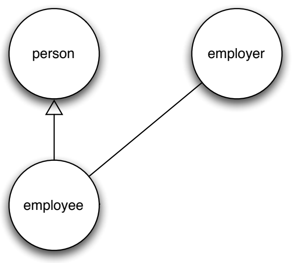

**Figure 2: For some contexts, a `person` object can be wrapped by an `employee` role that refers to an `employer` object.**

### 3.3 Subject-oriented Programming

Subject-oriented Programming [14] is one of the precursors of Aspect-oriented Programming. The idea is that a program should appear to have different class hierarchies from different perspectives. The goal of Subject-oriented Programming is to overcome the so-called "tyranny of the dominant decomposition." This is fundamentally different from our approach: In ContextL, the structure of the different layers must match each other. Partial class and method definitions in one layer can only augment existing definitions in other layers if all these definitions refer to the ("globally") same layered classes and layered functions.

### 3.4 Delegation

Delegation, as defined in [23], is an inheritance mechanism on the object level as opposed to inheritance at the class level as provided by most mainstream programming languages. It allows objects to refer to other superobjects that they can implicitly send messages to when they do not know how to respond to a message by themselves. This is similar to message forwarding, as used in many design patterns, but with the difference that messages sent to `this` or `self` in one of the superobjects are evaluated in the context of the receiver of the original message which is currently processed. This leads to a proper form of overriding between objects. Object inheritance has been used as a design principle for several so-called *prototype-based* programming environments, most notably Self [29], and has later been integrated into class-based programing languages [4, 6, 21].

In principle, delegation can be used to achieve context-dependent behavior of *single* objects by wrapping an object with another one that implements the context-specific behavior. So for example, a `person` object could be wrapped by an `employee` role that additionally sends a `display` message to the `employer` object associated with the role when it processes a `display` message itself (see Figure 2). However, when one wants to add the address information to both the `person` and the `employer` object later on and ensure that a single `display` message prints that additional information for both, one is lost: It is impossible to provide a wrapper for the `employer` object only for some contexts but not for others (see Figure 3).

ContextL solves this dilemma by grouping the context-specific behavior into layers.

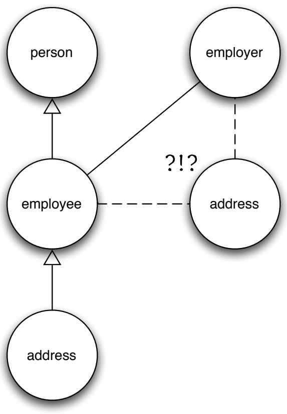

**Figure 3: When one additionally wants to add address information for both `person` and `employer`, one cannot do this in a context-specific way: It must be either added so that all previous contexts are affected, or it cannot be added at all.**

### 3.5 Delegation Layers

Delegation layers, as in [26, 27] and also combined into a class-based programming language in [25], are very similar to our approach. Like in ContextL, delegation layers define layers that group behavior for sets of objects in [27] and for sets of classes in [25]. Unlike in ContextL, the hierarchy of layers is fixed in those approaches. One can select a layer from which to start a specific message send, but all the other layers below are then predetermined by the original configuration of layers. A change in the layer hierarchy has a global effect for all subsequent message sends. In ContextL, the selection and ordering of layers is not fixed but layers can be arbitrarily activated and deactivated in the control flow of a program, leading to an implicit ordering according to the order of layer activations. Furthermore, the selection and ordering of layers can vary across different threads.

### 3.6 Other Related Work

Related work for special slots is discussed in [8, 9]. Related work for special functions, precursors for layered functions, is discussed in [7, 8].

The term Context-oriented Programming has already been used in two contexts. Gassanenko [11, 12] describes an approach to add object-oriented programming concepts to Forth without turning it into an actual object-oriented programming language. Instead, a notion of context is added that essentially boils down to some form of first-class environments [13]. This allows code to behave differently when executed in different environments. The description in Gassanenko's papers is very technical and it is very hard to tell how much overlap, if any, exists with our approach. For example, it is not clear whether Gassanenko's contexts must be fully defined or can be partial and combinable. The examples provided in [12] only cover fully specified, not partial contexts. Furthermore, Gassanenko's contexts seem to cover functions only, neither state nor class definitions, the latter due to the explicit goal not to turn Forth into a fully object-oriented programming language. Therefore, it seems that those contexts are most likely similar to dynamically scoped functions [7], one of our own precursors to ContextL.

Keays and Rakotonirainy [18] use the term Context-oriented Programming for an approach that separates code skeletons from context-filling code stubs that complete the code skeleton to actually perform some behavior. The claimed advantage is that the code stubs can vary depending on the context, for example the device some code runs on. A proof-of-concept implementation in Python and XML is described. Their approach appears to be a reverse macro expansion framework in which code skeletons and code stubs need to be combined at runtime. Furthermore, there is no mention whether different combinations of skeletons and stubs can coexist at the same time. In contrast, ContextL is essentially an extension of an object-oriented approach that does not rely on runtime source code transformation. ContextL's root layer, whose behavior can be altered in other layers, can already be fully operational, and different combinations of different layers can be simultaneously active in multiple threads.

## 4. CONCLUSIONS AND FUTURE WORK

ContextL is our extension to the Common Lisp Object System that allows for Context-oriented Programming. It provides layers, layered classes, layered and special slots with possibly layered accessors, and layered functions. Layered classes, slots and functions can be associated with specific layers, and such layers can be activated in the control flow of a program with dynamic scope, resulting in freely selectable layer combinations. This allows us to associate behavior with classes to which such behavior belongs while keeping the freedom to change the behavior in context-specific ways.

In contrast to the more widely known class-based programming languages, CLOS itself is centered on the generic-function-based approach to object-oriented programming. We are also working on systems implementing the key concepts of Context-oriented Programming in class-based languages, such as ContextS for Smalltalk, ContextT for Tweak, and ContextJ for Java. This will give us the opportunity to understand better the core concepts of Context-oriented Programming, as well as their variations.

Furthermore, we aim to explore the applicability of ContextL in various areas. For example, personalization, internationalization, ambient intelligence, context-sensitive safety and security, and testing frameworks that require simulation environments seem to be obvious candidates.

ContextL already provides a `with-inactive-layers` construct that is similar to `with-active-layers` but ensures that a set of layers is *not* active in the dynamic scope of that construct. Useful applications of `with-inactive-layers` will be described in future publications about ContextL. Possible extensions of the ContextL model include: A notion of dynamic closures that enables capturing the current dynamic class, slot and function bindings into a first-class object and later reactivation of such captured bindings. This would enable inheriting and distributing context-specific layer combinations across several threads. Furthermore, it may be of interest in certain application domains to incorporate layer-specific state as communication channels between multiple concurrent activations of the same layers. There are several possibilities to design language constructs for dealing with such layer-specific state, but we have not yet discovered good examples in which they could actually be useful. Therefore, we have not committed ourselves to one final design, but regard our own activities as work in progress.

Last but not least, it has been a surprise to us that we have apparently found a way to implement the functionality described in this paper without any considerable performance overhead, largely by discovering representations of layers that trigger the advanced optimizations already there in modern CLOS implementations. However, we first need to carry out several benchmarks before reporting on this in more detail. Some performance considerations related to special slots are already discussed in [9].

ContextL can be downloaded from the ContextL section at http://common-lisp.net/project/closer.

## 5. REFERENCES

[1] Daniel Bobrow and Ira Goldstein. Representing Design Alternatives. Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior. Amsterdam, July 1980.

[2] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, David Moon. Common Lisp Object System Specification. Lisp and Symbolic Computation 1, 3-4 (January 1989), 245-394.

[3] Christoph Bockisch, Michael Haupt, Mira Mezini, Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. AOSD 2004, Proceedings, ACM Press.

[4] Martin Büchi and Wolfgang Weck. Generic Wrappers. ECOOP 2000, Proceedings, Springer LNCS.

[5] Howard Cannon. Flavors – A Non-Hierarchical Approach to Object-oriented Programming. Unpublished draft, 1979, 1992, 2003.

[6] Pascal Costanza, Günter Kniesel, Armin Cremers. Lava – Spracherweiterungen für Delegation in Java. JIT '99 – Java-Informations-Tage 1999. Springer, Informatik Aktuell, 1999.

[7] Pascal Costanza. Dynamically Scoped Functions as the Essence of AOP. ECOOP 2003 Workshop on Object-oriented Language Engineering for the Post-Java Era, Darmstadt, Germany, July 22, 2003. ACM Sigplan Notices 38, 8 (August 2003).

[8] Pascal Costanza. A Short Overview of AspectL. European Interactive Workshop on Aspects in Software (EIWAS'04), Berlin, Germany, September 23-24.

[9] Pascal Costanza. How to Make Lisp More Special. International Lisp Conference 2005, Stanford. Proceedings.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[11] Michael Gassanenko. Context-oriented Programming: Evolution of Vocabularies. Proceedings of the euroFORTH'93 Conference. Marianske Lazne, Czech Republic.

[12] Michael Gassenenko. Context-oriented Programming. euroFORTH'98, Schloss Dagstuhl, Germany.

[13] David Gelernter, Suresh Jagannathan, Thomas London. Environments as First Class Objects. POPL '87, Proceedings.

[14] William Harrison and Harold Ossher. Subject-oriented Programming – A Critique of Pure Objects. OOPSLA '93, Proceedings, ACM Press.

[15] Robert Hirschfeld. AspectS – Aspect-oriented Programming with Squeak. In M. Aksit, M. Mezini, R. Unland (eds.). Objects, Components, Architectures, Services, and Applications for a Networked World. Springer LNCS 2003, 2003.

[16] Robert Hirschfeld and Pascal Costanza. Extending Advice Activation in AspectS, European Interactive Workshop on Aspects in Software (EIWAS 2005), Brussels, Belgium, September 2005.

[17] Robert Hirschfeld, Katsuya Kawamura, Hendrik Berndt. Dynamic Service Adaptation for Runtime System Extensions. Wireless On-Demand Network Systems, First IFIP TC6 Working Conference, WONS 2004, Proceedings, Springer LNCS 2928.

[18] Roger Keays and Andry Rakotonirainy. Context-oriented Programming. International Workshop on Data Engineering for Wireless and Mobile Access, San Diego, USA, 2003. ACM Press.

[19] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[20] Gregor Kiczales. Towards a New Model of Abstraction in Software Engineering. Proceedings of the International Workshop on Reflection and Meta-Level Architectures, 1992.

[21] Günter Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. ECOOP '99, Proceedings, Springer LNCS 1628.

[22] Glenn Krasner and Stephen Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-oriented Programming 1, 3 (August/September 1988).

[23] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems. OOPSLA '86, Proceedings.

[24] Hidehiko Masuhara and Gregor Kiczales. Modeling Crosscutting in Aspect-oriented Mechanisms. ECOOP 2003, Proceedings, Springer LNCS.

[25] Klaus Ostermann. Dynamically Composable Collaborations with Delegation Layers. ECOOP 2002, Proceedings, Springer LNCS.

[26] Lee Salzman and Jonathan Aldrich. Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model. ECOOP 2005, Proceedings, LNCS.

[27] Randall Smith and David Ungar. A Simple and Unifying Approach to Subjective Objects. Theory and Practice of Object Systems, 2, 3 1996.

[28] Randall Smith, John Maloney, David Ungar. The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility. OOPSLA '95 Conference Proceedings, Austin, Texas, October 1995.

[29] David Ungar and Randall Smith. Self: The Power of Simplicity. OOPSLA '87, Proceedings.

[30] Daniel Weinreb and David Moon. Flavors: Message Passing in the Lisp Machine. AI Memo 602, Massachusetts Institute of Technology, 1980.