Reflective Layer Activation in ContextL

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

Robert Hirschfeld
Hasso-Plattner-Institut
Universität Potsdam
D-14482 Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Expressing layer dependencies in Context-oriented Programming is cumbersome because until now no facility has been introduced to control the activation and deactivation of layers. This paper presents a novel reflective interface that provides such control without compromising efficiency. This allows expressing complex application-defined dependencies between layers where the activation or deactivation of a layer requires the activation or deactivation of another one. The activation or deactivation of specific layers can also be prohibited based on application-defined conditions.

Categories and Subject Descriptors

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

Keywords

Context-oriented Programming, dynamic layer activation, layer combination, software composition

1. INTRODUCTION

In Context-oriented Programming, programs consist of partial class and method definitions that can be freely selected and combined at runtime to enable programs to change their behavior according to their context of use. In [4], we have introduced this idea and have presented the programming language ContextL which is among the first language extensions that explicitly realize this vision. As a motivating example in that paper, we have illustrated an alternative implementation of the model-view-controller framework that avoids any secondary non-domain classes and thus increases understandability and flexibility at the same time.

Context-oriented Programming encourages continually changing behavior of programs and employs repeated changes to class and method definitions at runtime. We have illustrated this in [5] by demonstrating a context-oriented implementation of a figure editor, an example typically used to motivate aspect-oriented programming [20]. We have also discussed an efficient implementation strategy in that paper, and have used the figure editor example as a benchmark to show that a program with repeated activations and deactivations of layers is about as efficient as one without.

Until now, a facility for coordinating and controlling the activation and deactivation of layers is lacking. For example, it is not possible to prohibit the activation of an inactive layer, or vice versa the deactivation of an active layer. It is also not possible to express dependencies between layers, for example that the activation or deactivation of one layer requires the activation or deactivation of another.

The contribution of this paper is the introduction of a novel reflective facility in ContextL that allows expressing such dependencies, the illustration of this facility with some example scenarios, and the discussion of its interface that balances flexibility and efficiency.

2. FEATURE DEPENDENCIES

To illustrate some possible dependencies between layers in Context-oriented Programming, we sketch a number of services that may be provided on mobile devices, such as PDAs or smartphones, and that may depend on each other. Apart from regular applications, like word processors, spread sheets or calculators, such a device may provide the features to make phone calls, connect to a wireless network, use a short-messaging service (SMS), and so on. Some of these features depend on dynamic properties, like the presence of a wireless network card, the availability of actual network connectivity, and a contract between the user and a network operator. Furthermore, users need to pay for many of these features. For example, regular phone calls are billed for using different tariff plans based on time, volume, flat rates, and so on. Accordingly, there should be means to control the activation and deactivation of such features: A user may be interested in blocking the activation of certain premium features, or switching from one tariff plan to another without the risk of two tariff plans being active at the same time, and a service operator may be interested in ensuring that the activation of certain features makes the selection and activation of a tariff plan mandatory.

2.1 ContextL in a Nutshell

Layers in Context-oriented Programming are a good match to represent these features whose activation state can only be determined dynamically. In the following, we base our discussion on ContextL, one of the first programming lan-

guage extensions that explicitly support a context-oriented programming style [4]. It is an extension to the Common Lisp Object System (CLOS, [2]), which in turn is based on the notion of generic functions instead of the more widespread class-based object model. However, the context-oriented features of ContextL are conceptually independent of the CLOS object model, and a mapping of ContextL features to a hypothetical Java-style language extension called *ContextJ* has been described in [5].[1]

Layers are the essential extension provided by ContextL on which all subsequent features of ContextL are based. Layers can be defined with the `deflayer` construct, for example like this.

```
(deflayer phone-call-layer)
```

Layers have a name, and partial class and method definitions can be added to them. There exists a predefined root or default layer that all definitions are automatically placed in when they do not explicitly name a different layer.

For example, consider the following interface in ContextL for making phone calls.

```
(define-layered-function start-phone-call (number))
(define-layered-function end-phone-call ())
```

This defines two generic functions, one taking a phone number as a parameter and the other not taking any parameters. A default implementation for these as yet abstract functions can be placed in the root layer.

```
(define-layered-method start-phone-call (number)
  (error "Phone calls inactive on this device."))

(define-layered-method end-phone-call ()
  (error "Phone calls inactive on this device."))
```

Only if the `phone-call-layer` is active, a user can actually make phone calls.

```
(define-layered-method start-phone-call
  :in-layer phone-call-layer (number)
  ... actual implementation ...)

(define-layered-method end-phone-call
  :in-layer phone-call-layer ()
  ... actual implementation ...)
```

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

```
(with-active-layers (phone-call-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. 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 at a particular point in time, so nested layer activations are just ignored. This also means that on return from a dynamically scoped layer activation, a layer's activity state depends on whether it was already active before or not. In other words, dynamically scoped layer activation obeys a stack-like discipline.

Likewise, layers can be deactivated with a similar construct `with-inactive-layers` that ensures that a layer is not active during the execution of some contained code, and that has no effect when that layer is already inactive. Again, on return from a layer deactivation, a layer's activity state depends on whether it was active before or not.

Furthermore in multithreaded Common Lisp implementations, dynamically scoped layer activation and deactivation only activates and deactivates layers for the currently running thread. If a layer is active or inactive in some other thread, it will remain so unless it is incidentally also activated or deactivated in that thread.

Multiple layers can contribute to the same layered functions. For example, a metering layer can determine the cost of a phone call.

```
(deflayer phone-tariff-a)

(define-layered-method start-phone-call
  :in-layer phone-tariff-a :after (number)
  ... record start time ...)

(define-layered-method end-phone-call
  :in-layer phone-tariff-a :after (number)
  ... record end time & determine cost a ...)



(deflayer phone-tariff-b)

(define-layered-method start-phone-call
  :in-layer phone-tariff-b :after (number)
  ... record start time ...)

(define-layered-method end-phone-call
  :in-layer phone-tariff-b :after (number)
  ... record end time & determine cost b ...)
```

We can now recast the desire to control feature activation in more technical terms: The unconditional layer activation and deactivation constructs could be guarded by appropriate `if` statements that check whether a layer must be activated or deactivated, and/or whether other layers should be activated and/or deactivated as a consequence. While technically possible, this has negative implications for maintainability, understandability, etc., by scattering these checks throughout a program.

2.2 Layer Inheritance

A straigtforward extension to ContextL is to allow for inheritance between layers, similar to class inheritance in object-oriented programming. For instance, the latter two layers `phone-tariff-a` and `phone-tariff-b` have duplicate code, and it is possible to share such code.

```
(deflayer phone-tariff)
(define-layered-method start-phone-call
  :in-layer phone-tariff :after (number)
  ... record start time ...)

(deflayer phone-tariff-a (phone-tariff))
(deflayer phone-tariff-b (phone-tariff))
```

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

In this way, the code for `start-phone-call` does not need to be replicated. Only the code for `end-phone-call` containing details of the actual tariff plan must be provided separately. Note that the activation of an actual tariff, for example `phone-tariff-a`, implies that `phone-tariff` is also active.

Unfortunately, layer inheritance is not sufficient for expressing more interesting layer dependencies. For example, activation of the `phone-call-layer` may require application of one of the phone tariffs, without being specific about which actual one to apply.

3. REFLECTIVE LAYER ACTIVATION

Computational reflection provides programs with the ability to inspect and change data and functions at the meta-level that represent and execute them [17]. The ability to inspect the program state is called *introspection* and the ability to change the behavior is called *intercession*. A metaobject protocol (MOP) organizes the meta-level entities such that applications can extend them in an object-oriented style [11]. For example, there exists a specification of a metaobject protocol for the Common Lisp Object System (CLOS MOP) that is expressed in terms of CLOS itself, that is in terms of classes, generic functions and methods [10].

3.1 Intercession of Layer Activations

In terms of computational reflection, intercession of layer activation and deactivation is needed to be able to control them. In ContextL, this is achieved by implementing layer activation and deactivation through the functions `adjoin-layer-using-class` and `remove-layer-using-class`, following the CLOS MOP approach of using (meta)classes to determine the applicability of methods to meta-level entities. Additionally, these two functions are themselves layered such that layer activation and deactivation can be controlled by other layers. The interface of these functions is as follows.

```
(define-layered-function
  adjoin-layer-using-class (layer active-layers))

(define-layered-function
  remove-layer-using-class (layer active-layers))
```

Both functions take a representation of the layer that is about to be activated or deactivated respectively, and a representation of the currently active layers. Whenever a layer is activated via `with-active-layers` (or other layer activation constructs), ContextL calls `adjoin-layer-using-class` to determine the effective ordered set of layers to be active. Its default implementation returns a set according to the semantics of `with-active-layers` in Sect. 2.1. Likewise, `with-inactive-layers` calls `remove-layer-using-class` to determine the effective ordered set of layers to be active, and its default implementation returns a set according to the semantics of `with-inactive-layers` in Sect. 2.1. There is no corresponding call for implicit deactivations/activations of layers at the end of `with-active-layer`/`with-inactive-layer` blocks.

Note that these functions do not cause layer activation/deactivation themselves, but only return effective ordered sets of layers to be active, without any side effects. In this way, ContextL provides a clear separation between determining effective sets of layers and their actual activation.

3.2 Meta-Level Representations

In the following, we discuss meta-level representations of layers and sets of active layers.

Layers At the meta level, a layer is represented as a CLOS metaobject. By default, layer metaobjects are instances of the metaclass `standard-layer-class`. ContextL applications can define their own subclasses of `standard-layer-class`. For example, the `phone-call-layer` can be an instance of the application-defined `managed-layer-class`.

```
(defclass managed-layer-class
  (standard-layer-class) ;; super class
  ()) ;; no slot/field definitions

(deflayer phone-call-layer () ;; no super layers
  () ;; no layer-specific slots/fields
  (:metaclass managed-layer-class))
```

Blocking layer activation This `managed-layer-class` allows restricting methods on `adjoin-layer-using-class` and `remove-layer-using-class` to managed layers, as in the following two method definitions.

```
(deflayer block-managed-layers)
(define-layered-method
  adjoin-layer-using-class
  :in-layer block-managed-layers :before
  ((layer managed-layer-class) active-layers)
  (error "Layer is blocked!"))

(deflayer interactive-managed-layers)
(define-layered-method
  adjoin-layer-using-class
  :in-layer interactive-managed-layers :before
  ((layer managed-layer-class) active-layers)
  (if (not (ask-user "Activate layer ... ?"))
    (error "Layer is blocked.")))
```

An application can now choose to either block managed layers completely, or interactively ask the user to allow or disallow activation of managed layers, by enabling one of these two layers. Here, applications can use the function `layer-name` to determine the name of a layer metaobject and thus present more useful information to the user. More introspective data can be provided in subclasses of `standard-layer-class`.

Active layers The representation of the ordered set of currently active layers is not further specified. However, the function `layer-active-p` can be used to determine whether a given layer is a member of a given set of active layers or not. For example, we can now introduce a layer metaclass `tariff-base-layer-class` that we can use to define the base class for tariffs.

```
(defclass tariff-base-layer-class
  (standard-layer-class) ;; super class
  ()) ;; no slots/fields

(deflayer phone-tariff () ;; no super layers
  () ;; no layer-specific slots/fields
  (:metaclass tariff-base-layer-class))
```

Now we can define a method on `adjoin-layer-using-class` that triggers the selection of an actual tariff whenever the `phone-tariff` layer is activated.

```
(define-layered-method
  adjoin-layer-using-class
  ((layer tariff-base-layer-class) active-layers)
  (if (layer-active-p 'phone-tariff active-layers)
    active-layers
    (let ((tariff (ask-user "Select tariff ...")))
      (adjoin-layer tariff active-layers))))
```

In this case, the new method overrides the default layer activation behavior, so the base layer `phone-tariff` is actually never directly activated itself. Instead, it is first checked whether a phone tariff is already active, and if that is the case, all further activation is skipped. Since all actual phone tariffs inherit from `phone-tariff`, the test whether some phone tariff is active can indeed be performed on the base `phone-tariff` layer.

The function `adjoin-layer` used in this example takes a layer name and a set of active layers, and in turn calls `adjoin-layer-using-class` with the layer represented as a metaobject. So here it is important that the actual tariffs are not instances of `tariff-base-layer-class`. Not providing a `:metaclass` in the definition of those layers is sufficient because the metaclass of a layer is not inherited from any of its super layers, but is always `standard-layer-class` by default unless explicitly specified otherwise.

Layers requiring other layers It is now possible to define a method for activation of the `phone-layer` that also activates `phone-tariff`. The selection and activation of an actual phone tariff is thus implicitly triggered.

```
(define-layered-method
  adjoin-layer-using-class
  ((layer managed-layer-class) active-layers)
  (adjoin-layer
    'phone-tariff
    (call-next-layered-method layer active-layers)))
```

This definition performs a super call via `call-next-layered-method` and additionally activates the `phone-tariff` layer.

Mutually exclusive layers Another layer metaclass for actual phone tariffs allows us to switch between different tariffs.

```
(defclass tariff-layer-class (standard-layer-class)
  ())

(deflayer phone-tariff-a (phone-tariff)
  ()
  (:metaclass tariff-layer-class))

(deflayer phone-tariff-b (phone-tariff)
  ()
  (:metaclass tariff-layer-class))
```

Now we can define a method on `adjoin-layer-using-class` that ensures that a switch to a different phone tariff deactivates any other tariffs currently active.

```
(define-layered-method
  adjoin-layer-using-class
  ((layer tariff-layer-class) active-layers)
  (call-next-layered-method
    layer
    (remove-layer 'phone-tariff active-layers)))
```

This definition simply performs a super call and modifies the set of active layers such that any layers derived from `phone-tariff` are deactivated.[2] In another implementation, attempts to switch tariffs could as well be simply rejected.

Likewise, we can control the deactivation of tariffs.

```
(define-layered-method
  remove-layer-using-class
  ((layer tariff-layer-class) active-layers)
  (let ((new-tariff (ask-user "Select tariff ...")))
    (adjoin-layer
      new-tariff
      (call-next-layered-method
        layer active-layers))))
```

Again, a deactivation of tariffs could be rejected in an alternative implementation.

To summarize, we have illustrated the following cases of how to control activation/deactivation of layers:

- Conditional or unconditional blocking of layers.
- Activation of a layer requires activation of another.
- Activation of a layer requires deactivation of another.

It should be clear by now that similar effects can be achieved for layer deactivation as well (that is, blocking of layer deactivation and layer deactivation requiring activation/deactivation of other layers).

4. EFFICIENCY CONSIDERATIONS

In [5], we have presented an implementation strategy for ContextL, and a benchmark in which a layer is repeatedly activated and deactivated again illustrates its efficiency. Compared to an execution of the same benchmark without any layer activation or deactivation, the overhead in runtime ranges from very moderate 2.72% to still reasonable 12.36% across four different Common Lisp implementations. In two other Common Lisp implementations, the runs with layer activations and deactivations are actually 6.13% and 7.29% *faster* than the ones without, indicating that factors beyond layer activation and deactivation play a more important role for the overall performance.

This efficiency is achieved because multiple active layers are always represented by exactly one metaobject – the set of active layers mentioned in the previous section. In our implementation described in [5], computational overhead occurs exclusively on the first activation/deactivation of a previously unused combination of layers and on the first message send in a previously unused combination of methods. After that, both lookups of layer combinations and method dispatches take advantage of highly efficient caches.

The reflective extension of ContextL discussed in this paper so far seems to inhibit the first optimization that speeds up the lookup of layer combinations: Each layer activation and deactivation has to go through the functions `adjoin-layer-using-class` and `remove-layer-using-class` respectively. Indeed, caching the results of these function calls seems to contradict their intention in some cases. For example, the user should be asked each time an attempt is made to activate, say, the `phone-call-layer` in the previous section when interactive blocking of layers is asked for, while

[2]This invocation of `remove-layer` internally calls `remove-layer-using-class`, taking further specializations into account.

Implementation	Without Layers	With Layers	Overhead
Allegro CL 8.0	2.544 secs	2.650 secs	4.17% slower
CMUCL 19c	0.77 secs	0.744 secs	3.49% *faster*
LispWorks 4.4.6	3.128 secs	3.2374 secs	3.50% slower
MCL 5.1	2.187 secs	2.4358 secs	11.38% slower
OpenMCL 1.0	2.3788 secs	2.5938 secs	9.04% slower
SBCL 0.9.16	0.9138 secs	0.8708 secs	4.94% *faster*

Figure 1: The results of running the example from [5] in the new version of ContextL.

the unconditional blocking of such managed layers could still be cached as before. This illustrates why the results of `adjoin-layer-using-class` and `remove-layer-using-class` can only be cached according to *domain-specific* criteria.

To resolve these conflicting requirements, we slightly change the interface of these two reflective functions and require them to return two values: The first value represents the ordered set of all new active layers and the second return value is either true, indicating that this new combination of layers can be cached and reused for the same set of active layers, or otherwise it is false, indicating that the result must not be cached but that `adjoin-layer-using-class` or `remove-layer-using-class` must be called again for the same set of active layers. The default implementations of these two functions always return true as the second return value because the default semantics always allow caching, but application-defined methods may freely choose to return either true or false.

This means that, for example, the two variations of layer blocking can be implemented as follows.[3]

```
(define-layered-method
  adjoin-layer-using-class
  :in-layer block-managed-layers
  ((layer managed-layer-class) active-layers)
   ;; Just return the already active layers
   ;; and ensure that this result is cached.
  (values active-layers t))

(define-layered-method
  adjoin-layer-using-class
  :in-layer interactive-managed-layers
  ((layer managed-layer-class) active-layers)
  (values
    (if (ask-user "Activate layer ... ?")
      (call-next-layered-method layer active-layers)
      active-layers)
    ;; In both cases, the result may not be cached.
    nil))
```

As a confirmation that this implementation strategy is indeed successful at maintaining efficiency comparable to that already reported before, we have extended the implementation of ContextL with reflective layer activation based on the interface described in this section. We have then run the benchmark from [5] again, and the results of the various runs on different Common Lisp implementations are presented in Fig. 1. As in [5], the entries in Fig. 1 are average measurements of five runs. The platform on which we have executed the benchmark is an Apple Powerbook 1.67 GHz PowerPC G4 running Mac OS X 10.4.7. The overheads are in the same range as the ones reported in [5] based on the previous non-reflective implementation of ContextL, ranging from 3.5% in LispWorks for Macintosh to 11.38% in Macintosh Common Lisp (MCL). The same two implementations as in [5] show the anomaly again that the runs that repeatedly switch layers on and off are actually *faster* than the runs without layers: On CMUCL 19c, the runs without layers are on average 3.49% slower, and on SBCL 0.9.16 they are 4.94% slower. See [5] for more details about this benchmark.

To summarize, reflective layer activation can indeed be implemented without inhibiting performance. Only when methods on `adjoin-layer-using-class` and `remove-layer-using-class` request not to cache their results, an overhead will be repeatedly incurred that depends on the actual computation performed by these methods.

5. DISCUSSION AND FUTURE WORK

Aspect-oriented technologies approaching the context-oriented notion of dynamically scoped activation of partial program definitions are AspectS [7, 8], LasagneJ [21], CaesarJ [12], and Steamloom [3]. They all add constructs for thread-local activation of partial program definitions at the application level. However, CaesarJ does not provide a corresponding thread-local deactivation construct, and LasagneJ restricts the use of thread-local activation to the `main` method of a Java program [13]. Their lack of thread-local deactivation constructs makes `cflow`-style constructs necessary, for example to implement the figure editor example [5]. Here, Context-oriented Programming allows a modular implementation without using AOP-style pointcuts. Global activation/deactivation constructs, like in CaesarJ and ObjectTeams [22] are not sufficient in this regard. Steamloom provides undeployment of thread-local aspects, but cannot thread-locally undeploy a globally active aspect.

Delegation layers, as in the prototype-based languages Slate [15] and Us [18] and also combined into a class-based programming language in [14], are very similar to Context-oriented Programming. As layers in ContextL, delegation layers group behavior for sets of objects [15, 18] or sets of classes [14]. However, the hierarchy of layers is globally fixed in [14]. One can select a layer in which to send a specific message, but all subsequent layers are predetermined by the original configuration of layers. In [15] and [18], the selection and ordering of layers is not fixed but layers can be ar-

[3]Common Lisp provides the `values` construct to return more than one value from a function that can be received from a function call via `multiple-value-bind`. An approximation of the semantics of multiple values is that they are boxed in a compound data structure when returned and unboxed again at the call site, except that a more efficient implementation is possible by internally making use of multiple return slots on the call stack.

bitrarily recombined in the control flow of a program. However, layer selection and combination has to be done manually, there are no dedicated layer activation/deactivation constructs like in ContextL. Providing these constructs as high-level abstractions allows for less straightforward, but more efficient implementation strategies, and adding reflective layer activation as shown in this paper.

Aspect-oriented approaches can express conditions under which a given aspect is applicable or not, but these conditions can typically only be expressed at the aspect level and thus have a static, fixed nature [19]. Reflective layer activation allows checking for more flexible conditions while keeping the advantage of being able to activate/deactivate layers in a straightforward way anywhere in a program. We have shown several examples in this paper to illustrate this flexibility. In spite of this flexibility, we have also been able to design an interface for reflective layer activation that allows for an efficient implementation. Depending on domain-specific criteria, application-defined extensions of layer activation/deactivation can indicate whether the resulting layer combination may be cached or not, thus taking advantage of the efficient implementation strategy discussed in [5] when applicable, and only incurring a runtime overhead when necessary.

The major downside of reflective layer activation is that the implementation of reflective extensions is complex: A programmer has to define layer metaclasses, decide which layers are instances of which layer metaclasses, define methods on `adjoin-layer-using-class` and `remove-layer-using-class` correctly, and understand the interactions between different such methods. What is needed in the long run is a declarative interface at a higher level of abstraction to express rules for layer activation/deactivation and layer dependencies.

Context-oriented Programming is related to the mixin layers approach [16] of feature-oriented programming [1], and an established approach for expressing feature dependencies in that field are feature diagrams [6, 9]. We are currently working on how to adapt them to express dynamic dependencies between context-oriented layers.

Acknowledgements

We thank Nick Bourner, Johan Brichau, Drew Crampsie, Brecht Desmet, Attila Lendvai, and Igor Plekhov for fruitful discussions and valuable contributions.

6. REFERENCES

[1] D. Batory and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, June 2004.

[2] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, D. Moon. Common Lisp Object System Specification. *Lisp and Symbolic Computation* 1, 3-4 (January 1989), 245-394.

[3] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann. Virtual Machine Support for Dynamic Join Points. *AOSD 2004*, Proceedings, ACM Press.

[4] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming. *ACM Dynamic Languages Symposium 2005*. Proceedings, ACM Press.

[5] P. Costanza, R. Hirschfeld, and W. De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. *Joint Modular Languages Conference 2006*, Proceedings, Springer LNCS.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[7] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. *Objects, Compononts, Architectures, Services, and Applications for a Networked World*, Springer LNCS, 2003.

[8] R. Hirschfeld and P. Costanza. Extending Advice Activation in AspectS. EIWAS 2005.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[10] G. Kiczales, J. des Rivières, D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[11] P. Maes. *Computational Reflection*. Ph.D. thesis, Vrije Universiteit Brussel, 1987.

[12] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. *AOSD 2003*. Proceedings, ACM Press.

[13] A. Moors, J. Smans, E. Truyen, F. Piessens, W. Joosen. Safe language support for feature composition through feature-based dispatch. 2nd Workshop on Managing Variabilities Consistently in Design and Code, OOPSLA 2005.

[14] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. *ECOOP 2002*, Proceedings, Springer LNCS.

[15] L. Salzman and J. Aldrich. Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model. *ECOOP 2005*, Proceedings, Springer LNCS.

[16] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Design. *ACM Transactions on Software Engineering and Methodology*, March 2002.

[17] B. Smith. *Procedural Reflection in Programming Languages*. Ph.D. thesis, Massachusetts Institute of Technology, 1982.

[18] R. Smith and D. Ungar. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems*, 2, 3, 1996.

[19] E. Tanter. On Dynamically-Scoped Crosscutting Mechanisms. EWAS 2006.

[20] P. Tarr, M. D'Hondt, L. Bergmans, C. Lopes. Workshop an Aspects and Dimensions of Concerns: Requirements on, and Challenge Problems For, Advanced Separation of Concerns. *ECOOP 2000 Workshops*, Proceedings, Springer LNCS.

[21] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, B.N. Jorgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. *ICSE 2001*, Proceedings.

[22] M. Veit and S. Herrman. Model-View-Controller and ObjectTeams: A Perfect Match of Paradigms. *AOSD 2003*, Proceedings, ACM Press.