

---

# HOW TO DEAL WITH ENCAPSULATION IN ASPECT-ORIENTATION

Lodewijk M.J. Bergmans (lbergmans@acm.org)

Mehmet Aksit (aksit@cs.utwente.nl)

<http://trese.cs.utwente.nl>

TRESE GROUP – UNIVERSITY OF TWENTE, P.O. BOX 217, 7500 AE, ENSCHEDE, THE NETHERLANDS

---

**Abstract—Encapsulation has received considerable interest in the context of object-oriented programming. In particular the encapsulation of objects that are composed through inheritance has been discussed repeatedly [Snyder 86, Micallef 88]. Aspect-oriented composition has similar considerations. However, most approaches to aspect-orientation or multi-dimensional composition of concerns do not enforce encapsulation of the implementation details of concerns. This might even lead to the impression that aspect-orientation conflicts with encapsulation. We claim that aspect-orientation and encapsulation are orthogonal, and that it is possible, and desirable, to enforce strong encapsulation in the presence of aspects. We illustrate this by giving an example and showing a (proposed) solution based on the composition filters model.**

## 1. INTRODUCTION & PROBLEM STATEMENT

A general concern in the encapsulation discussion is that revealing implementation details to clients and/or re-users of a module is bad, in particular from a maintainability perspective. We claim that encapsulation is (even more) desirable in the case of crosscutting specifications. Consider the following example:

Assume that in a corporate environment, a set of objects is communicating with each other within a distributed heterogeneous system. Here, heterogeneity means that objects may be implemented in different languages and/or running on machines with different operation systems or hardware.

In such environments, it is generally agreed that revealing internal details to *clients* is a bad thing, because this makes clients vulnerable to changes in the implementation. In general, by using approaches like COM and CORBA, the implementation details and even the implementation language and platform can be hidden. An Interface Definition Language (IDL) is used to describe the externally visible interface in a language-independent manner.

Now assume that due to changes in the business requirements, some of the objects in the system have to be extended with a simple security mechanism. When a secure object receives a message, it verifies whether the sender object is in its access-list or not. If the sender object is not in the list, the message is rejected. Here, *security* is a crosscutting aspect since it has to be implemented in multiple objects. These objects may be instances of different classes, but not all instances of a class may require security.

Although aspect-oriented languages simplify the task of implementing crosscutting concerns, additional complexities may arise. Consider the example:

- The implementers of the application might not have been aware of the desired security extension. This means the existing modules in the system could not have been prepared for such an extension.
- The application will likely be in continuous evolution. This means that the implementation of the modules may be changed or new modules may be introduced. If the security concern depends on the implementation details of the modules, changing the implementation of modules may require changes to the implementation of the security concern. Since in the future the security concern may also evolve, and/or new crosscutting concerns may be introduced, having dependencies between the crosscutting concerns and implementation of modules considerably increase the complexity of the system.
- In the corporate system example, the software modules may be implemented in different programming languages. The designers of the crosscutting concern, which is security in this case, have to consider the heterogeneity of the environment.

## 2. REQUIREMENTS

To avoid the problems introduced in the previous section, we think that the following two basic requirements must be fulfilled:

- *implementation encapsulation*: Aspect specifications must only refer to interfaces. The implementation of objects may change for all kinds of reasons and therefore

aspects must only refer to the interface of an object but not to its implementation.

- *language encapsulation*: Aspect specifications must be independent of the implementation language of the base level. Since many distributed systems are heterogeneous, having aspects specified in a language independent way has obvious advantages.

A rigid implementation of these two requirements might hinder the evolution of software systems. The following requirements relate to flexibility of aspect specifications:

- *object-based*: Objects are run-time entities and therefore run-time adaptability requires reasoning about objects. Aspect specifications, therefore, must be able to refer to the objects in the system. On the other hand, classes are generally the stable elements in the software system, and therefore they may be referred to in aspect specifications as well. In this case, it is necessary to have a flexible binding of aspects to instances and/or classes. For example, the security concern in our example case may be applied to a set of objects belonging to different classes and/or class hierarchies.
- *open-ended*: Aspect specifications must allow introducing new objects and classes, and deriving new aspects from the existing ones.

### 3. BACKGROUND: CHARACTERISTICS OF COMPOSITION FILTERS

We have no space in this paper to fully explain the composition-filters model in detail. Instead, we highlight a number of principles, and introduce details as they are relevant to this paper.

The composition-filters approach aims to enhance the expression power and maintainability of objects. It does so by adding a modular part to objects, in which the *filters* intercept and manipulate incoming and outgoing messages. Figure 1 illustrates this, adopting UML notation as far as it is applicable.

Figure 1 shows a single class (stereotyped as a `<<CF class>>`), which contains a nested object. The object has stereotype `<<implementation>>`; this is the 'core' object that is enhanced with filters. Within the dashed box, a number of filters are shown: on the left side the *input filters* that deal with incoming messages, and on the right side the *output filters* dealing with outgoing messages. The triangles pointing up or down indicate the roles of input filters respectively output filters, and have no additional semantic meaning. As indicated by the two dotted boxes, filters can be grouped (such a group is called a *filtermodule* since it is an encapsulated unit of reuse).

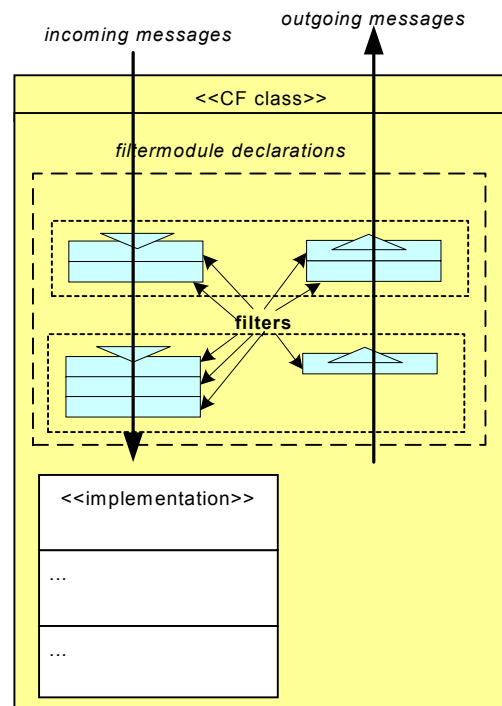


Figure 1 The basic structure of a composition filters object.

The Composition Filters model is based on the following principles:

1. There are a number of pre-defined filter classes, each responsible for expressing a certain aspect. Programmers may introduce new filter classes, provided these fulfil a number of requirements.
2. Instances of a filter class can be created and attached to a class defined in various languages such as Java [Wichman 99], Smalltalk [Koopmans 95] and C++ [Glandrup 95]. Although for reasons of understandability, correctness and optimization this is not encouraged, this may occur, and change, dynamically (at run-time).
3. A filter instance is initialized using a filter expression. A standard filter expression syntax and semantics are available for all filters. This is a declarative specification, in the sense that it does not make any assumptions about how the specification is to be implemented<sup>1</sup>.
4. A message manipulation operation by a filter may change the *explicit* and *implicit* attributes of the received message. The explicit attributes are the receiver object, the message selector and the arguments of the message. The implicit attributes include the sender and server object of the message, and other attributes that can be introduced by filters or application programmers.
5. A filter specification refers to the parameters of the received messages only. It does not make any assumption about other filters. However, a filter may refer to the state of its object, as made accessible and abstracted through the conditions of the object.
6. A filter expression consists of a set of filter elements. These elements and/or filters themselves can be com-

<sup>1</sup> A filter and its parts can be implemented in various ways, for example, as run-time objects by adopting message reflection (e.g. in [Koopmans 95, Glandrup 95]), or as in-lined code, by adopting compilation and optimisation techniques (e.g. in [Wichman 99]).

posed using logical operators such as conditional-OR, conditional-AND, and exclusion. In the composition filters syntax, the character “,” implements a conditional-OR operation, which means that if the expression on the left-hand-side cannot match, then the expression on the right-hand-side will be evaluated. A conditional-AND operation can be implemented by cascading filters, using the “;” sign in the filter definition language.

7. Each filter expression specifies a single concern, which is then mapped upon one or more messages that are executed by a method of some object (in particular the object itself). This implements the specification of crosscutting concerns, although with a scope that is restricted to the local object and the objects that is explicitly delegated to.
8. Superimposition of filters upon groups of objects can be used to express concerns that crosscut multiple classes. Superimposition does not break the encapsulation of objects, but only relies on public interfaces.
9. Typing is based on signatures that are derived for each object from its filter specification. For type checking purposes, the filter interface definition language may require additional type declarations, e.g. of objects that are reused.

In the following section, we show how the composition filters model can be used to model the example problem we introduced in section 2.

## 4. IMPLEMENTING THE EXAMPLE USING COMPOSITION FILTERS

We will first explain how the example can be modeled with composition filters, and then discuss to what extent this example fulfills the requirements from section 2.

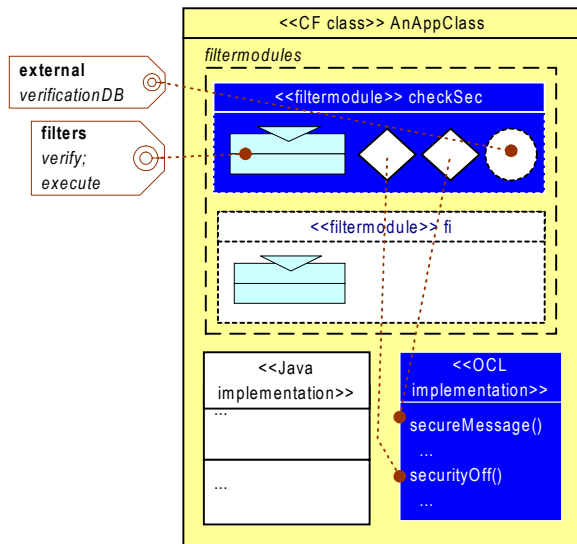


Figure 2. An application class after superimposition.

In Figure 2, part of the solution is shown schematically: this diagram shows an example application class, including the superimposed security concern (that is, as this class would look after the weaving process). The <<Java implementation>> and <<filtermodule>> fi parts, both colored white, are already available before superimposing the security concern. The <<filtermodule>> checkSec and <<OCL implementation>>, both shaded

in a dark color, are the superimposed parts that together ensure that security is checked.

The filtermodule contains a single filter, labeled verify, of type Error, which verifies whether the sender of the message has the proper access rights for each invocation. To realize this filter specification, two *conditions* are required; secureMessage, and securityOff. The conditions are Boolean, side-effect free, expressions which are implemented in the <<OCL implementation>> part that has been superimposed as well. This implementation part will be explained later in this section. Finally, the filtermodule contains an *external*; this is a reference by name (verificationDB) to a shared object<sup>2</sup> that contains the access lists.

The following code defines the filtermodule checkSec that is shown in Figure 2:

```
filtermodule checkSec begin // the crosscutting behavior
  externals
    verificationDB:SecurityEnforce;
  conditions
    secureMessage(), securityOff();
  inputfilters
    verify: Error = { securityOff()=>*, secureMessage() => * };
end filtermodule;
```

As a general rule, the code in the filtermodules follows the UML conventions whenever applicable. We will explain this code starting with the declaration of the inputfilter at the end. This inputfilter, named verify, is an instance of filter class Error, initialized with the *filter expression* between curly brackets. *Error* filters have the following semantics: if a message arrives at the filter, and it matches the filter expression, the message may just continue to the next filter. If the message cannot match the filter expression, an exception will be raised.

The matching process takes place by considering each element in the filter expression, in this case separated by the comma operator. The comma operator defines a conditional OR between the two elements; if the message can be matched by the first element, it matches the entire filter expression (with no need to match subsequent filter elements), otherwise an attempt will be made to match the following filter element with the message.

The first filter element, “securityOff() => \*” means the following; if the condition, securityOff(), evaluates to TRUE, the message will be matched with the expression on the right side of the ‘=>’ operator, otherwise there is no match. In this case on the right side there is only a wildcard “\*”, which causes any message to match. In other words, whenever the condition securityOff() is valid, all messages will match; this implies that no further security check will be performed.

In case the condition securityOff() is not valid<sup>3</sup>, it will be attempted to match the message at the second filter element, “secureMessage() => \*”. This element has a similar structure; any message will match, if, and only if, the condition secureMessage() is valid. When the message does match, it may proceed to the next filter (in this case defined by the next filtermodule), otherwise an exception will be raised (because

<sup>2</sup> This declaration is required to verify type-safety and completeness when superimposing filtermodules.

<sup>3</sup> The same condition could be expressed as “not securityOn()”, in case one considers that more natural.

the message was not allowed to pass the filter due to security restrictions).

We will now look at the implementation of the conditions:

#### implementation in OCL begin

**condition** securityOff():

verificationDB.secureObjects -> not exists(v | v = self );

**condition** secureMessage():

verificationDB.access

-> select( to | to.identity = message.receiver )

-> exists( from | from.identity = message.sender);

**end implementation**

All the code fragments that we show are part of a single declaration of the concern named SecurityEnforce. A concern declaration contains parts that are language-independent (especially the filtermodules and the superimposition specifications), and implementation parts that need to be defined in some executable language (e.g. Java or C++). The above code fragment implements the two conditions with OCL (this is the *Object Constraint Language* that is part of the UML specification [UML 00]). We will not go into the details of OCL, but briefly explain what the meaning is of the OCL expressions that we use.

The first condition, securityOff(), is implemented by the expression “verificationDB.secureObjects -> not exists(v | v = self);”. The left part returns the variable secureObjects, which is defined as a part of the shared object verificationDB. The OCL function exists() tests whether the current object is in the list of secure objects; if it is, the object is a ‘secure object’, and the condition returns FALSE. If the object is not in the list, securityOff() returns TRUE.

The second condition is slightly more complex: it looks in the access list of the verificationDB object to find an element that corresponds to both the sender and the receiver of the message<sup>4</sup>. If such an element is available, the message is secure, otherwise the message should not be allowed.

We have now explained how application classes can be extended to deal with the security requirements we posed by superimposing filtermodules. In the following section we will explain how the superimposition can be expressed.

## 5. EXPRESSING CROSSCUTTING CONCERNS

In the previous section, it was shown how the security concern can be expressed modularly using composition filters. In this section, we show how the security behavior can be specified as a single, crosscutting, concern. Figure 3 gives an overview of the elements in the SecurityEnforce concern. The figure shows one instance, named verificationDB, of type SecurityEnforce. This instance contains the shared access information about all objects in the application, for example the conditions secureMessage() and securityOff() that were discussed in the previous section, use this shared information.

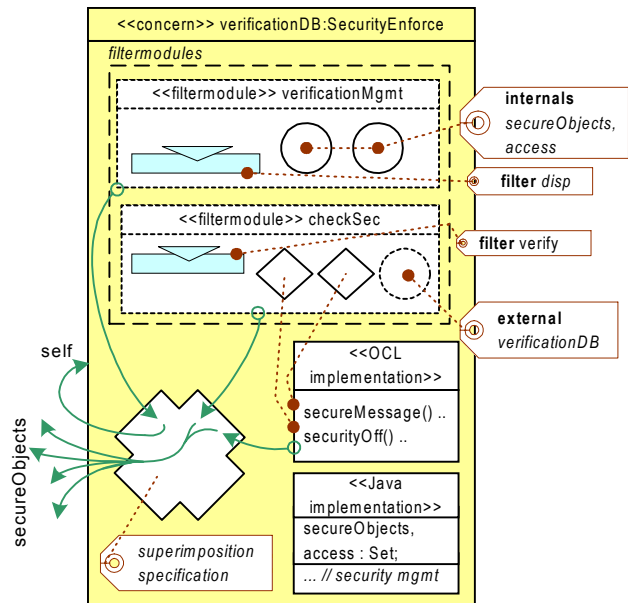


Figure 3. The crosscutting concern SecurityEnforce

The concern consists of several parts: first, at the top of the picture, two filtermodules; verificationMgmt, which defines the interface and behavior for the shared verification database, and checkSec, which is the filtermodule to be superimposed upon the application classes, as discussed in the previous section. Second, the concern contains (shown in the bottom at the right side) two implementation parts that implement part of the behavior, respectively expressed in OCL and Java. Third, the concern contains a superimposition specification: this is a mapping that connects elements from the current or reused concerns to one or more other concern instances. The superimposition specification is shown as a large cross-shape.

We will now discuss the details of this concern: in Appendix A the full source code of concern SecurityEnforce is shown, we will discuss most of this piecewise in the text below. The following code sample shows the structure of the verificationMgmt filtermodule:

```
filtermodule verificationMgmt begin // this is the central DB
  internals
    secureObjects, access : Set;
  methods
    // several methods for configuring security
  inputfilters
    disp : Dispatch = {inner.*};
end filtermodule verificationMgmt;
```

This filtermodule defines the interface for the –shared– object that is used to store and retrieve access control information. It declares two *internals*; i.e. objects that are created and encapsulated within the filtermodule: secureObjects and access. These are both instances of Set. Its usage was shown in the explanation of the conditions securityOff and secureMessage. The filtermodule verificationMgmt defines a number of methods (not shown in detail here) for manipulating the access control information, such as enabling/disabling security verification for certain objects and adding/removing permissions for sending messages between objects.

This filtermodule has a single inputfilter, labeled disp, which has the filter type Dispatch. All messages that can be matched by the filter expression will cause a dispatch of the message to its current target. All messages that cannot be matched will

<sup>4</sup> We assume that a pseudovalue message is available, with fields referring to respectively the *sender* and the *receiver* of the current message.

continue to a subsequent filter; raising an exception if there is none available. In the case of the `disp` filter, the filter expression “`inner.*`” means that all messages that are supported by ‘`inner`’, can be dispatched. The pseudo-variable ‘`inner`’ refers to the implementation of the current concern, only the methods that are defined in the `methods` section of the filtermodule will be matched and dispatched by this filter expression.

The concern contains two implementation parts: an OCL implementation of two conditions that we described earlier, and a Java implementation (not shown here) that implements all methods defined in the `verificationMgmt` filtermodule.

Finally, the *superimposition* part specifies how filtermodules, variables, conditions and methods are to be mapped upon one or more concerns. A superimposition specification consists of two parts: the first part, starting with the keyword `selectors`<sup>5</sup>, defines sets of objects (concern instances). Each of these sets represents a group of joinpoints that together form one crosscut<sup>6</sup>. These selectors are used in the second part to define the superimposition of condition and method implementations, variables, and filtermodules.

In the example code below, the superimposition specification of the concern `SecurityEnforce` is shown. It defines two selectors, one condition superimposition and two filtermodule superimpositions:

```

superimposition begin
selectors
  classes:= Set(Class1, Class2, ClassN);
  secureObjects:=
    system-> select( o|classes-> exists( v| o.oclIsTypeOf(v) );
conditions
  secureObjects<-{self::secureMessage(),
    self::securityOff() };
filtermodules
  self <- self::verificationMgmt;
  secureObjects <- self::checkSec;
end superimposition;

```

The first selector, named `classes`, defines a set of classes, for the purpose of the example simply labeled `Class1`, `Class2` and `ClassN`. Selectors are defined using a subset of the UML OCL standard and define a set of classes and/or instances.

The second selector, `secureObjects`, selects from all the objects (instances) in the `system`, those objects whose type corresponds to one of the classes in the selector `classes`. In other words: all instances from one of the classes defined by selector `classes`. The pseudovvariable `system` can always be used to refer to all instances in the system.

The selector `secureObjects` is used in the subsequent block that starts with the keyword `conditions`; this block defines the superimposition of the conditions `secureMessage()` and `securityOff()` upon all the objects denoted by `secureObjects`. This means that all filtermodules of these objects can refer to these two conditions. In other words, superimposition of conditions (and similarly of methods), can be seen as a form of binding of condition respectively method implementations.

The final part of the superimposition specification defines the superimposition of the filtermodule `verificationMgmt` to the current concern (as denoted by ‘`self`’), and filtermodule `checkSec`

to all objects denoted by `secureObjects`. Superimposition of filtermodules means for instance that all messages will also have to go through the input- respectively output filters of the new filtermodule. The ordering of filtermodules corresponds to the order of declaration<sup>7</sup>.

## 6. EVALUATION OF CF APPROACH

We will first discuss how the example we have shown and explained above behaves according to the requirements we proposed in section 2.

- *implementation encapsulation*: in the composition filter model, concerns can only refer to elements that are provided on the *interface* of another concern. If for example state information is required by other objects or by an aspect (e.g. a superimposed filtermodule), this is only possible by accessing it through methods or conditions. In addition, all behavior is expressed as the observation and manipulation of messages that are passing the boundary of objects, independent of the implementation within that boundary.
- *language encapsulation*: a concern specification can be seen as a very powerful interface definition language; since its’ definition is independent of a particular implementation (language) of an object. The syntax for declaring methods and objects is based on the (language-independent) UML notations. The implementation part can be defined in any object-based language. Composition filters have been combined in the past with Smalltalk, C++, Java, and other languages. Because new concerns may need to refer to the internal state of the objects where they are superimposed, it is possible that these concerns define new conditions, expressed within the language independent OCL language (e.g. conditions `securityOff()` and `secureMessages()` ). From such a specification, it is straightforward to generate implementations of the conditions in any programming language.
- *object-based*: because instances from the same class may belong to several different groups and/or categories, it is very likely that two such instances fall into different crosscuts. For example in the application we presented, some objects are secure and others not, regardless of their classes. At the class level, it is not possible to make this distinction.
- *open-ended*: The approach we presented is open-ended in several ways:
  - (a) the crosscut is defined in selector specifications as an associative query expression: this means that newly added classes and/or instances will automatically become part of the crosscut when they satisfy the query expression.
  - (b) concerns can always be refined or extended, including crosscutting concerns (i.e. aspects), since these are also ‘first-class’ concerns.
  - (c) new filter types can be introduced to express additional semantics in filtermodules.
  - (d) the use of the “\*” wildcard in filter expressions allows to prepare for growing interfaces, and propagate these.

<sup>5</sup> Not to be confused with a message selector: a selector is a selection of objects in the system, as

<sup>6</sup> In AspectJ, a similar notion is called pointcut.

<sup>7</sup> It is still a subject of current and future work how the ordering of filters and filtermodules can be handled in an effective way.

We can conclude that the proposed solution, which is based on composition filters, largely satisfies the requirements that were proposed in section 2.

## 7. DISCUSSION & CONCLUSION

In this paper, we investigated the issue of encapsulation in aspect-oriented programming, and showed how the composition filters model deals with this. Much related work, as exemplified by e.g. AspectJ [Kiczales 01] and HyperJ [Ossher 01], has assumed<sup>8</sup> that it may be necessary to allow aspects to refer to implementation details of the base level.

We argue that concerns or classes must encapsulate implementation detail towards other (crosscutting) concerns. One perspective on this issue is to compare refinement of systems through aspect-orientation with refinement through inheritance: also in the latter case, there has always been a ‘need’ to access the implementation details of a superclass in the refined class. At the same time, it has been argued repeatedly [Snyder 86, Micallef 88] that it is important that subclasses are independent of the implementation of their superclasses. The popularity of black-box (aggregation-based) reuse also suggests that strong encapsulation, even with its disadvantages, is more effective as well. Since superimposing a filtermodule upon an existing base-level concern is comparable to extending a concern through subclassing, it is likely that the same reasoning holds.

We have shown how a crosscutting concern can be expressed in the CF model, while retaining strong encapsulation. Its important characteristics are:

- The aspect code is kept completely separate from the base level by operating upon messages.
- The aspects can refer only to the declared interface of the base level objects; this interface consists of public methods and conditions that are only available for filters.
- The aspect specifications (i.e. filter specifications) are independent of the language of the base level; this means that a single aspect specification could be applied to a crosscut with objects in several languages.
- The condition implementations that are part of the aspect can be implement in the OCL language, which makes it possible to generate implementations in different languages.

We have constructed a prototype implementation [Salinas 01] that implements the superimposition of crosscutting concerns as a preprocessor for our Java-specific implementation [Wichman 99]. However, this prototype does not implement language heterogeneity or generate condition implementations from OCL specifications.

## REFERENCES

[Glandrup 95] M. Glandrup, *Extending C++ using the concepts of Composition Filters*, MSc. thesis, Dept. of Computer Science, University of Twente, November 1995

[Kiczales 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold, *An Overview of AspectJ*, Proceedings of ECOOP 2001, LNCS 2072, Springer Verlag, 2001

[Koopmans 95] P. Koopmans, *On the design and realization of the Sina compiler*, MSc. thesis, Dept. of Computer Science, University of Twente, August 1995

[Micallef 88] J. Micallef, *Encapsulation, Reusability and Extensibility in Object-Oriented Programming*, JOOP April/May 1988, Vol. 1, No. 1, pp. 12-35

[Ossher 01] Harold Ossher & Peri Tarr, *Multi-Dimensional Separation Of Concerns And The Hyperspace Approach*, Harold Ossher & Peri Tarr, M.Aksit (ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001

[Salinas 01] P. Salinas, *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*, submitted EMOOSE MSc. thesis, Vrije Universiteit Brussel, August 2001

[Snyder 86] A. Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, OOPSLA '86, pp. 38-45

[UML 00] Unified Modeling Language (UML) 1.3 specification, OMG standard *formal/00-03-01*, 2000

[Wichman 99] C. Wichman, *ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language*, MSc. thesis, Dept. of Computer Science, University of Twente, December 1999.

<sup>8</sup> This is actually an assumption from our side, based on the fact that both language explicitly allow aspects to refer to implementation details.

## APPENDIX A: FULL SOURCE CODE OF THE EXAMPLE

Although the source code of the example has been shown throughout the paper, the following code

```

concern SecurityEnforce begin
  filtermodule verificationMgmt begin
    // this defines the access to the central DB
    internals
      secureObjects, access : Set;
    methods
      // several methods for configuring security
    inputfilters
      disp : Dispatch = {inner.*};
  end filtermodule verificationMgmt;
  filtermodule checkSec begin
    // this defines the crosscutting behavior; to be superimposed
    externals
      verificationDB:SecurityEnforce;
    conditions
      secureMessage(), securityOff();
    inputfilters
      verify: Error = { securityOff()=>*, secureMessage() => * };
  end filtermodule;
  superimposition begin //
    selectors
      classes:= Set(Class1, Class2, ClassN);
      secureObjects:=
        system-> select( o|classes-> exists( v| o.oclIsTypeOf(v) );
    conditions
      secureObjects <- { self::secureMessage(),
        self::securityOff } };
    filtermodules
      self <- self::verificationMgmt;
      secureObjects <- self::checkSec;
  end superimposition;
  implementation in OCL begin
    condition securityOff():
      verificationDB.secureObjects -> not exists(v | v = self );
    condition secureMessage():
      verificationDB.access
        -> select( to | to.identity = message.receiver )
        -> exists( from | from.identity = message.sender);
  end implementation
  implementation in java file "..."
    // a java class that implements the verificationMgmt interface
end concern SecurityEnforce

```